Introduction to
Software Engineering and C++

COMP 2404 Companion Guide

Christine Laurendeau

School of Computer Science
Carleton University
Ottawa, Ontario

claurend@scs.carleton.ca

To the ones who persist in striving for their better selves,

And the ones who battle trolls for the truth and light of day,

You are the heroes of every story.

Now boot up. Timmy Tortoise needs our help.

Image of Timmy Tortoise: Ai Generated PNGs by Vecteezy
Image of Harold the Hare: Pinterest ai image creator

Contents

Coding Examples

List of Figures

Preface

This text is intended to serve as a companion resource for COMP 2404 (Introduction to Software Engineering), offered by the School of Computer Science at Carleton University in Ottawa, Ontario.

COMP 2404 is not meant as a full software engineering course. While it does provide a general introduction to the software engineering life cycle activities, it focuses mainly on object-oriented design and implementation. A secondary goal of the course is to serve as an introduction to the C++ programming language. This textbook captures the concepts covered in the COMP 2404 lectures, as well as the accompanying programming examples.

COMP 2401 (Introduction to Systems Programming) is the established prerequisite for COMP 2404, as it covers C programming in the Linux environment. Readers are expected to be fully proficient with the concepts covered in COMP 2401. This includes pointers, memory management, and dynamic memory allocation concepts, as well as C syntax and programming in a Linux environment.

Part I introduces the basics of software development in the C++ programming language on a Linux platform. Chapter 1 reviews how program building works in Linux. Chapter 2 discusses the basic C++ language features, including operators, control structures, variables and data types, and functions and their parameters, as well as the concept of C++ references. Chapter 3 introduces the organization and features of basic C++ classes, including constructors and destructors. Chapter 4 covers the memory management concepts of pointers, double pointers, and dynamic memory allocation with C++ objects.

Part II focuses on the design of object-oriented (OO) software. Chapter 5 discusses the basic OO design principles of data abstraction and encapsulation. Chapter 6 introduces the concept of object design categories, which ensure than the objects in an OO design have a focused role in a program. Chapter 7 covers how Unified Modelling Language (UML) diagrams are used to graphically depict classes and the relationships between them.

Part III covers the essential techniques that are common to most OO programming languages, and how to implement these techniques in C++. Chapter 8 discusses the encapsulation of object data and behaviour in C++. Chapter 9 covers the implementation of inheritance hierarchies, and it discusses the unique C++ features of multiple inheritance and the different inheritance types. Chapter 10 illustrates how polymorphism, with virtual and pure virtual functions, is implemented in C++. Chapter 11 discusses some of the well-known OO design patterns. Chapter 12 covers the concept of overloading in C++, including operator overloading. Chapter 13 shows how generic programming in C++ is implemented using function and class templates.

Part IV discusses some of the important C++ language features. Chapter 14 demonstrates the exception handling (EH) mechanism. Chapter 15 covers the basic architecture and components of the C++ standard template library (STL). Chapter 16 reviews the concepts of streams and files, and how these are represented and implemented in C++.

Part I

Basics of C++ development

Chapter 1

Program Building in Linux

Program building is the process of converting the instructions of a program, written in a high-level programming language like C or C++, to a format that is executable by a computer's hardware. Because large programs are normally separated into multiple files, possibly across multiple directories (as folders are called in Unix-type operating systems), program building is more complex than issuing a single compiler command.

It's understood that most modern software development occurs within an integrated development environment (IDE). While an IDE is a great tool for simplifying and optimizing the programmer's task, the reality is that the development of some applications on certain platforms is restricted to a command line interface. It is the position of this author that, if a programmer only learns to code using an IDE, they will face tremendous obstacles (and possibly reduced employability) if one day they are required to work from the command line. If a programmer learns to code from the command line, adapting to the use of an IDE takes a trivial amount of time.

In this chapter, we review how program building works from the command line in a Linux programming environment. All the programming examples in this textbook work with a GNU compiler [1] and execute on an Ubuntu Linux operating system (OS).

1.1. Terminology

We begin by defining some of the terminology associated with program building.

1.1.1. Code

1.1.1.1. What is *source code*:

- A *computer program* is a complete set of instructions, written in a high-level programming language like C, C++, or Java, that work together to provide a specific functionality.
- *Source code* is a generic term for computer program instructions.
- By convention, in Unix systems (including Linux), C++ programs are stored in files with a `.cc` extension. In a Windows operating system, a `.cpp` file extension is often used. Because we are programming in a Linux environment, we use the `.cc` extension in the coding examples of this textbook.
- Source code **cannot** be directly understood by a computer's central processing unit (CPU). The CPU is the hardware "brain" of the computer, and it's in charge of executing programs. It can only process instructions in a much lower-level language called *machine code*.

1.1.1.2. **What is *machine code*:**

- *Machine code* is the low-level equivalent of a program written in a high-level programming language like C++. It consists of program instructions translated to numeric values.
- It can be understood by the CPU, but it is **not** readable by people.
- As a result, it's necessary for source code to be translated to machine code before the CPU can execute the instructions in our C++ programs.

1.1.2. Executables

1.1.2.1. **What is *program building*:**

- *Program building* is the translation of source code written in a high-level language to the corresponding machine code.
- It results in the creation of a *program executable* file from one or more source code files.

1.1.2.2. **What is a *program executable*:**

- A *program executable* is a file that stores the program's machine code instructions.
- A C++ program can be made up of instructions stored into multiple source code files, which must then be translated together into one executable file.
- The end-user runs the program by launching the executable, either by double-clicking on it in a graphical used interface (GUI) based operating system or by entering the executable file name from the command line in a terminal window.

1.1.2.3. **Characteristics of a program executable:**

- Machine code instructions are typically in a format that is both OS- and CPU-dependent. An executable that is created on one platform, for example a specific type or distribution of OS or a specific CPU manufacturer, cannot execute on a different platform. If we want a program to run on multiple platforms, we must build a different executable for each one.
- An executable is made up of machine code from multiple source files. In a large project, source code from many different programmers are combined together to build the executable. Even if a program is contained within a single source file, it typically relies on library code for some of its functionality.
- An executable must contain exactly **one** `main()` function. If it contained more than one, the CPU would not know where to begin executing the program. If it contained no `main()` function, then it would be a library, and not a program executable.

NOTE: It's important to note that the separation of source code into multiple source files is **not** a requirement of any OS or CPU. Rather, it is a **very important** coding convention rooted in the basic principles of correct software engineering. By separating the code into modular components, we ensure that it can be better maintained and extended in the future.

1.2. Compilation and linking

We discuss the different steps of program building in a Linux programming environment. We also provide a short overview of some common errors encountered during program building.

1.2.1. Concepts

1.2.1.1. **Steps in program building:**

- The two main steps for translating a C++ program from source code to an executable:
  - compilation
  - linking
- The programming examples in this textbook use the default GNU compiler provided with the Ubuntu distribution of Linux. The compiler is invoked by launching the `g++` executable from the command line.

1.2.1.2. **What is *compilation*:**

- *Compilation* transforms source code into *object code*, which is an intermediate format.
- Every source code file is compiled into a corresponding object code file.
- There is a one-to-one correspondence between source and object files.

1.2.1.3. **What is *linking*:**

- *Linking* transforms object code into an executable.
- All the object code files required for a program are linked into a single executable.
- There is a many-to-one correspondence between object files and the program executable.
- Linking is the step where library code, which is usually provided as object code, is added to the programmer's object code.

  NOTE: In the context of program building, the use of the term *object* code is **unrelated** to the concepts of classes and objects in OO programming.

1.2.1.4. **Building a single source file:**

- A single C++ source file, for example `p1-hello.cc`, can be compiled and linked into an executable file called `p1` using the command: `g++ -o p1 p1-hello.cc`
- The `g++` compiler uses the same command line arguments as the `gcc` compiler used for C programming in Linux. Important options for both `g++` and `gcc` include:
  - the `-o` option allows the user to indicate the name of an output file for the command
  - the `-c` option stops the program building after the compilation step, as we see below

1.2.1.5. **Building multiple source files:**

- Several steps are required to compile and link multiple source files into an executable.
- Assuming that we have two source files, `file1.cc` and `file2.cc`, the following steps must be taken to both compile and link the source code into an executable:
  - compile `file1.cc` into object code using `g++ -c file1.cc`, which generates `file1.o`
  - compile `file2.cc` into object code using `g++ -c file2.cc`, which generates `file2.o`
  - link `file1.o` and `file2.o` into an executable called `p2` using `g++ -o p2 file1.o file2.o`

Figure-1.1 shows how a program comprised of three source files is compiled into three object files, which are then linked into one executable.

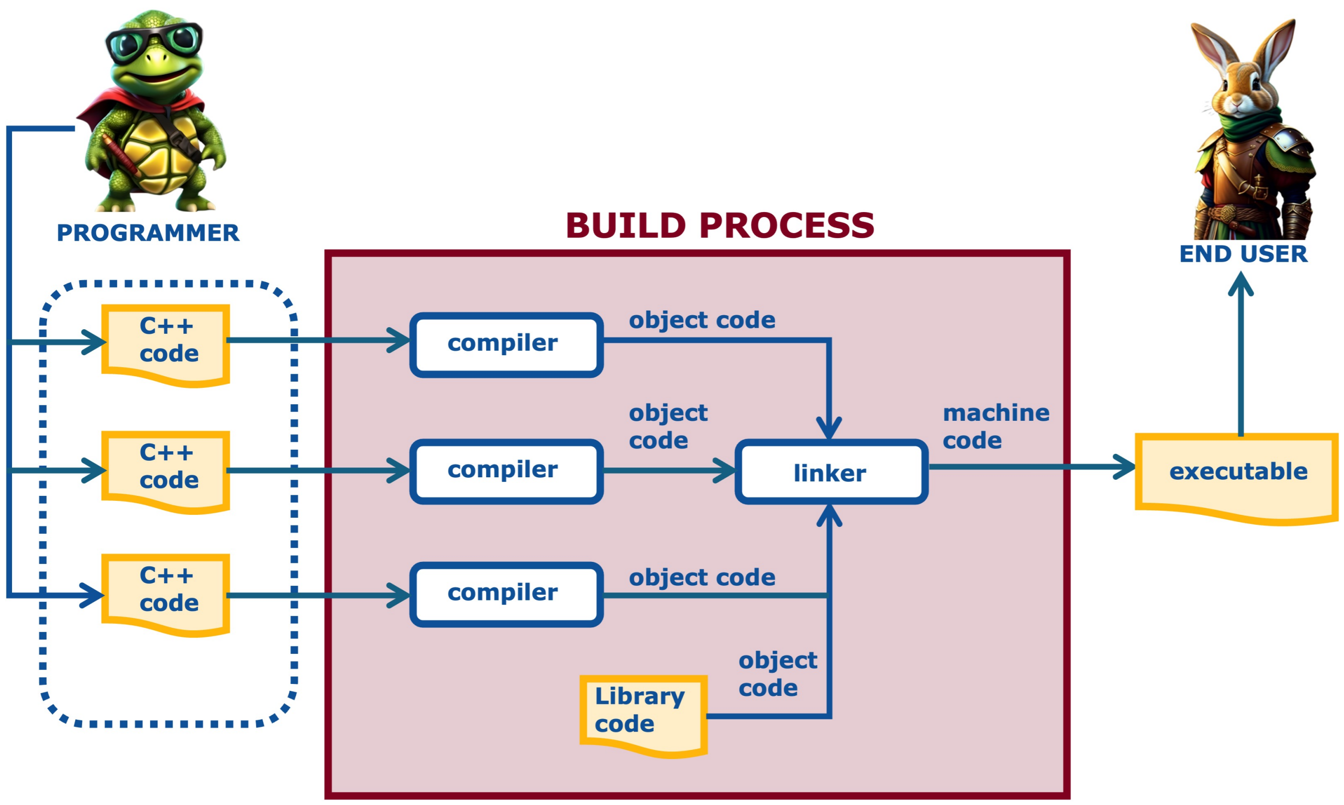
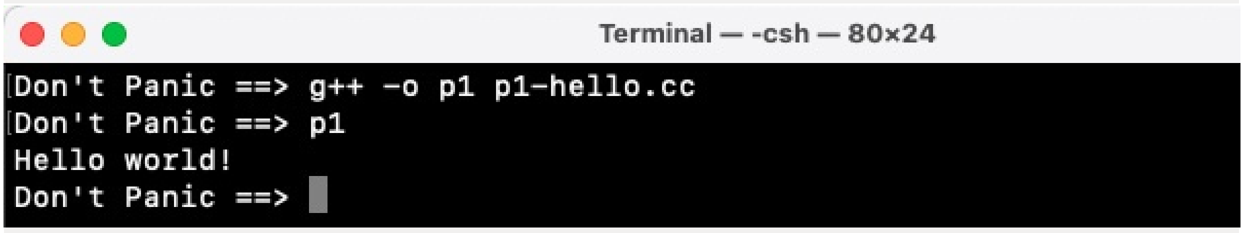


Figure-1.1: Program building

1.2.2. Coding example: Building a program with a single source file

```cpp
/* * * * * * * * * * * * * * * * *
 * Filename:  p1-hello.cc        *
 * * * * * * * * * * * * * * * * */
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello world!" << endl;
  return 0;
}
```



```
Don't Panic ==> g++ -o p1 p1-hello.cc
Don't Panic ==> p1
Hello world!
Don't Panic ==> 
```

Program-1.1: Building a program with a single source file

Program purpose:

- Program-1.1 prints out the string "Hello world!" to the screen.
- Because the program is contained in a single source file, compilation and linking can be combined into a single call to the GNU compiler, as we see in the provided output.

Line 4:

- This line copies the header file for the `iostream` library into our source file. The `iostream` library provides the tools to perform user input/output (I/O).

Line 5:

- This line tells the compiler that our program uses identifiers from a predefined namespace called `std`. Namespaces are essentially a predefined grouping of identifiers, and they are discussed in section 8.5.
- The identifiers contained in `std` include objects and functions necessary to perform user I/O.

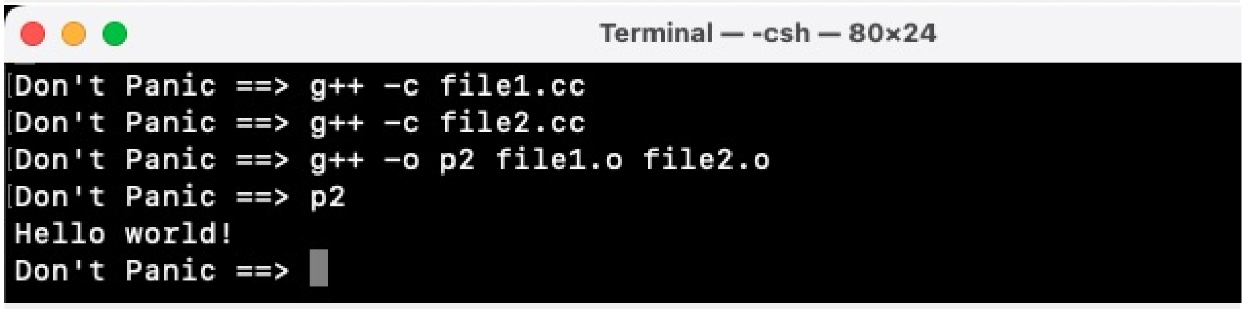
Lines 7-11:

- These lines show the implementation of the `main()` function.
- Line 9 performs the actual output to the screen.
- On line 9, the `cout` object represents the standard output stream. In our coding examples, this object is used for printing information to the screen.
- In the `iostream` library, the `cout` object and the `endl` function are declared inside the `std` namespace. On line 5, the `using` keyword brings into scope everything from the `std` namespace, so we can use `cout` and `endl` on line 9 without specifying that they belong to the namespace. The alternative would be to use the *scope resolution operator* (`::`) to explicitly state this relationship everywhere we use the namespace content (for example, `std::cout` and `std::endl`). The scope resolution operator is discussed in section 2.2.5.
- Line 9 uses the *stream insertion operator* (`<<`) to send the string `"Hello world!"` to the standard output stream object `cout`. The same statement then calls the `endl` function to insert a new line character into the standard output stream. The stream insertion operator is further described in section 2.2.5.
- Line 10 returns the status code zero to the OS, to indicate that the program terminated normally.

1.2.3. Coding example: Building a program with multiple source files

```
/* * * * * * * * * * * * * * * * *
 * Filename:  file1.cc          *
 * * * * * * * * * * * * * * * * */
#include <iostream>
using namespace std;

void foo1();
void foo2();

int main()
{
  foo1();
  foo2();
  return 0;
}

void foo1()
{
  cout << "Hello ";
}

```

```
22 /* * * * * * * * * * * * * * * * *
23  * Filename:  file2.cc        *
24  * * * * * * * * * * * * * * * * */
25 #include <iostream>
26 using namespace std;

28 void foo2()
29 {
30   cout << "world!" << endl;
31 }
```



```
Don't Panic ==> g++ -c file1.cc
Don't Panic ==> g++ -c file2.cc
Don't Panic ==> g++ -o p2 file1.o file2.o
Don't Panic ==> p2
Hello world!
Don't Panic ==> 
```

Program-1.2: Building a program with multiple source files

Program purpose:

- Program-1.2 prints out the words "Hello world!" to the screen, by calling functions that are packaged into two different source files.
- Because the program spans more than one source file, it is standard practice to separate the compilation and linking into different steps, to reduce the amount of recompilation in the event of future code changes.
- First, each source file must be compiled into its own object file, then both object files must be linked into the program executable.
- The provided output shows the three calls to the `g++` compiler: once to compile `file1.cc` to an object file, once to compile `file2.cc` to an object file, and a final time to link both object files into an executable called `p2`.

Lines 4-20:

- These lines show the contents of the `file1.cc` source file, which contains the implementation of the `main()` and `foo1()` functions.
- Lines 7-8 each declare a ***forward reference*** for the functions that `main()` calls, but that are declared either later in the same source file or in a different file.
- Forward references are sometimes necessary to inform the compiler of the existence and prototype of functions that it has not yet encountered as part of the compilation process. Without lines 7-8 to inform the compiler of the `foo1()` and `foo2()` function prototypes, lines 12-13 would not compile.
- Lines 10-15 show the implementation of the `main()` function, which calls the `foo1()` and `foo2()` functions.
- Lines 17-20 show the implementation of the `foo1()` function, which prints the string `"Hello"`.

NOTE: Another way to avoid compilation errors on lines 12-13 would be to position the implementations of `foo1()` and `foo2()` *before* the `main()` function in the same source file. This would be **bad programming style**. Other programmers expect to see the `main()` function positioned first in the file, because it's the most important.

Lines 25-31:

- These lines show the contents of the `file2.cc` source file, which contains the implementation of the `foo2()` function.
- Lines 28-31 show the implementation of `foo2()`, which prints the string `"world!"`.

1.2.4. Makefiles

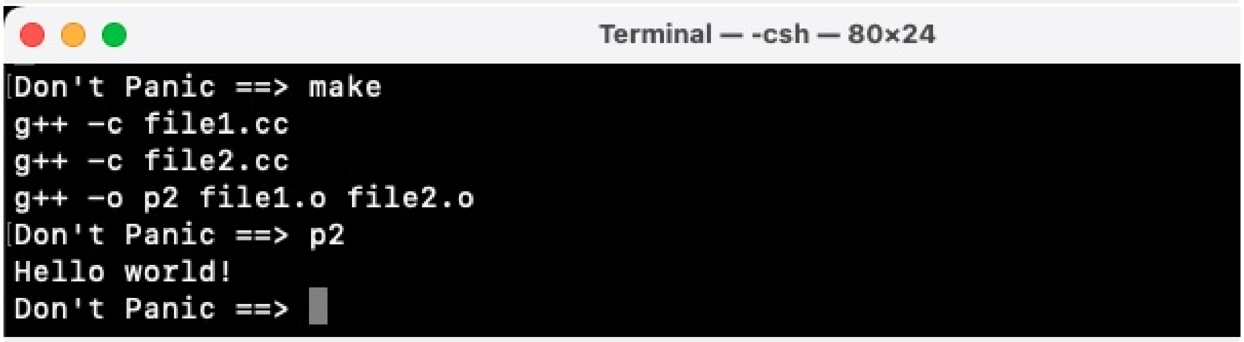
1.2.4.1. What is a *Makefile*:

- A *Makefile* is a simple text file called `Makefile` that contains compilation and linking commands to build a program executable from source code. [2]
- Remember that files in Linux do *not* require any file extensions, and that files names are case-sensitive.
- Makefiles have a very specific syntax that must be followed in order to work.

1.2.4.2. Why use a Makefile:

- A typical C++ program comprises many source files, and manually compiling each one is an onerous and error-prone task. The program building process can be greatly simplified with the use of Makefiles.
- The commands are executed by invoking `make` from the command line in the same directory as the Makefile.
- If correctly laid out, a Makefile can also assist the programmer in managing dependencies between files, for example only recompiling the source files that have changed.
- More information on Makefiles can be found in the Linux OS manual pages (using the `man` command) and in the COMP 2401 course notes.

```
p2:    file1.o file2.o
       g++ -o p2 file1.o file2.o

file1.o:  file1.cc
          g++ -c file1.cc

file2.o:  file2.cc
          g++ -c file2.cc

```



```
Don't Panic ==> make
g++ -c file1.cc
g++ -c file2.cc
g++ -o p2 file1.o file2.o
Don't Panic ==> p2
Hello world!
Don't Panic ==>
```

Program-1.3: Sample Makefile

Program-1.3 illustrates a simplified Makefile associated with Program-1.2.

1.2.5. Build and runtime errors

It’s an unfortunate reality that most of our programs will encounter errors as we try to get them working correctly. It’s important to recognize the categories of errors we may encounter, in order to quickly narrow down the possible causes.

1.2.5.1. **Terminology**:

- The term ***compile time*** generally denotes when a program is in the process of being compiled and linked. At this point, we cannot run the program because it has not yet been translated into machine code. If compile time completes successfully, without errors, the outcome is a program executable that can be launched.
- The term ***runtime*** denotes when a program is running, also called "executing". Program runtime only begins once an executable has been launched.

1.2.5.2. **Types of errors**:

- *Compiler errors* occur during the compilation step, as seen in Figure-1.1.
- *Linker errors* are discovered during the linking step.
- *Runtime errors* happen after a program executable has been created and launched.

1.2.5.3. **Compiler errors**:

- Compiler errors occur in one specific source file, during compilation.
- They are often due to an error in using the C++ language syntax, or in organizing the code.
- The compiler usually reports the source file and line number where the error is found. This information can help in narrowing down the problem.
- Common errors include misspellings of keywords or identifiers; inconsistent data types, where variables are not used as they are declared; missing forward references; using undeclared variables or classes, for example if we forget to include a header file; redeclaration of variables or classes; using library classes where the header file has not been included; and many more.

1.2.5.4. **Linker errors**:

- Linker errors occur after all the source files have been compiled into individual object files, and the linker is trying to connect each function call with the corresponding implementation.
- They are typically easy to fix, because there is ever only one thing wrong: some function is used in the code, but its implementation cannot be found among the object code provided.
- Common errors include forgetting to compile a source file, or forgetting to bring an object file or a library into the linking step.

1.2.5.5. **Runtime errors**:

- Runtime errors occur during execution, *after* a program executable is created and launched.
- Segmentation faults are the most common runtime errors. These occur when our program is attempting to access a memory location that doesn’t belong to our program, i.e. it is outside the program’s virtual memory segments.
- Common errors include dereferencing or accessing a null or garbage pointer; allocating memory for a pointer but not for its data; accessing an array out of bounds; and many more.

Chapter 2

C++ Basic Language Features

In this chapter, we discuss some basic C++ language features, including operators, control structures, variables and data types, user I/O, and functions and their parameters. Many of these features also appear in other programming languages, for example C and Java.

Then, we introduce the concept of C++ references, which provide a new technique, not present in C, for functions to pass parameters by reference. Some general programming conventions for writing Linux-based C++ code are also discussed.

Several resources for the C++ language and the C++ standard library can be found online. [3] [4]

2.1. Principles

We briefly introduce the C++ programming language. Then, we define some basic programming terms that are necessary for understanding the more advanced concepts in later chapters.

2.1.1. Characteristics of C++

2.1.1.1. **Origins of C++**:

- C++ is a general-purpose, object-oriented (OO) programming language, and it is based on the C language.
- It was developed by computer scientist Bjarne Stroustrup [5] starting in 1979, and it has gone through multiple language extensions since then.
- C++ was initially known as "C with classes" or "object-oriented C".

2.1.1.2. **Characteristics of C++**:

- C++ provides all the standard functionality of other OO languages, and more, including static and dynamic polymorphism, inheritance, generic programming using templates, operator overloading, multiple inheritance, and exception handling.
- It also allows calls to low-level C functions and system calls, for example memory management operations and bit-wise operators. Variations of essential C libraries are also available in C++, including sockets and threads.
- C++ is a powerful language that combines both high-level and lower-level features, providing programmers with a wide range of implementation choices.

2.1.2. Terminology

We review some basic terminology relevant to programming in general.

2.1.2.1. What is a *keyword*:

- A ***keyword*** is a sequence of characters that has a special meaning in a programming language. For example, `while`, `int`, and `return` are all keywords in C, C++, and Java.
- A keyword is a *reserved word*, so a programmer cannot use it for another purpose, for example to name a data type, a variable, or a function.
- A list of C++ keywords can be found here.

2.1.2.2. What is an *identifier*:

- An ***identifier*** is a word that is *user-defined*. In this context, the user of a programming language is the programmer. So an identifier is a programmer-defined word in a program.
- For example, the names of variables and functions are considered identifiers.

2.1.2.3. What is a *literal*:

- A ***literal*** is a sequence of characters in a program that is meant to be interpreted literally.
- For example, a string value in double quotes like `"Hello world"` and a specific numeric value like `77` are considered literals.

2.1.2.4. What is an *expression*:

- An ***expression*** is a sequence of variables, operators, constants, and/or function calls.
- For example: `a + b - 3 * c`
- Expressions always "return" a single value. We can also say that an expression *resolves to* or *evaluates to* a returned value.

2.1.2.5. What is a *statement*:

- A ***statement*** is an expression that is terminated by a semi-colon.
- For example: `tmpValue = a + b - 3 * c;`
  - how many operators do we find in the statement above, 3 or 4?
  - it's 4, because the assignment operator (`=`) is also an operator

2.1.2.6. What is a *block*:

- A ***block*** is a sequence of statements positioned between a pair of matching braces `{ }`
- For example:
  - the body of a function or a loop is a block
  - each branch of an `if`-statement is a block
  - we can even have free-floating blocks!
- In a single-statement block, the braces are optional.

NOTE: *Curly brackets* is a colloquial term sometimes used for braces. While every programmer is familiar with the term, it's far from canonical. C/C++ features only two kinds of brackets: square and angle. "Round" brackets are called *parentheses*, and "curly" ones are *braces*.

2.1.2.7. **What is *scope***:

- The ***scope*** of an identifier is the part of a program where the identifier can be used.
- An identifier with ***block scope*** is limited to the inside of a block:
  - for example, a variable declared inside a function or loop has block scope inside that function or loop; it does not exist outside that block
  - a variable with block scope is often called a *local variable*
  - any kind of block can have local variables
- An identifier with ***global scope*** (also called *file scope*) exists outside of any block:
  - for example, a variable declared at file level outside of any block is called a global variable, and it has global scope
  - all functions in the C programming language have global scope

NOTE: It is correct software engineering practice to strictly limit the number of identifiers declared at global scope, in order to best protect our data. The correct number of global variables to use in a program is always **zero**.

2.1.2.8. **What is *storage class***:

- The ***storage class*** of a variable specifies the lifetime of the variable, as well as the area of virtual memory where it is stored.
- The default is *automatic* storage class:
  - automatic variables are stored in the function call stack
  - they are created on declaration, and they disappear when the program exits the block where they are declared
  - examples of automatic variables include local variables and function parameters
- Another common storage class is *static*:
  - static storage class variables are stored in the data segment in global memory
  - they are created when they are *first* declared, and they disappear at the end of the program
  - they "remember" their value until the end of the program, even though their visibility depends on their scope

2.2. Operators

We introduce the terminology associated with operators, including their characteristics and operator categories.

2.2.1. Concepts

2.2.1.1. **What is an *operator***:

- An ***operator*** is a symbol, built into a programming language, that performs a specific task. For example, the `+` operator performs an addition operation with two values.
- An operator takes a predefined number of existing values, for example a variable, a literal, or a function's return value, and it computes a result and returns that result.
- In that sense, operators behave similarly to functions.
- A list of C++ operators can be found here.

2.2.1.2. **What is an *operand***:

- An ***operand*** is a value that an operator works upon.
- If an operator is similar to a function, an operand is like a parameter to that function.
- For example, the addition (`+`) operator takes two operands, adds them together, and returns the result.

2.2.1.3. **What is a *return value***:

- Every operator returns a result.
- This return value may be used as an operand to another operator, if an expression or statement contains multiple operators.

2.2.2. Categories of operators

There are several different categories of operators in C++:

2.2.2.1. **Arithmetic**:

- An arithmetic operator performs a simple arithmetic operation and returns the result.
- For example: addition (`+`), subtraction (`-`), multiplication (`*`), integer division (`/`), modulo (`%`), prefix/postfix increment (`++`), prefix/postfix decrement (`--`) are arithmetic operators.

2.2.2.2. **Relational**:

- A relational operator compares two operands and returns a boolean value.
- For example: equality (`==`), inequality (`!=`), less-than (`<`), greater-than (`>`), less-than-or-equal-to (`<=`), greater-than-or-equal-to (`>=`) are relational operators.

2.2.2.3. **Logical**:

- A logical operator performs an operation on one or two boolean values and returns the resulting boolean value.
- For example: logical AND (`&&`), logical OR (`||`), logical NOT (`!`) are logical operators.

2.2.2.4. **Bitwise**:

- A bitwise operator manipulates a value at the bit-level and returns the result.
- For example: bitwise AND (`&`), bitwise OR (`|`), bitwise NOT (`~`), bitwise XOR (`^`), left-shift (`<<`), right-shift (`>>`) are bitwise operators.

2.2.2.5. **Assignment**:

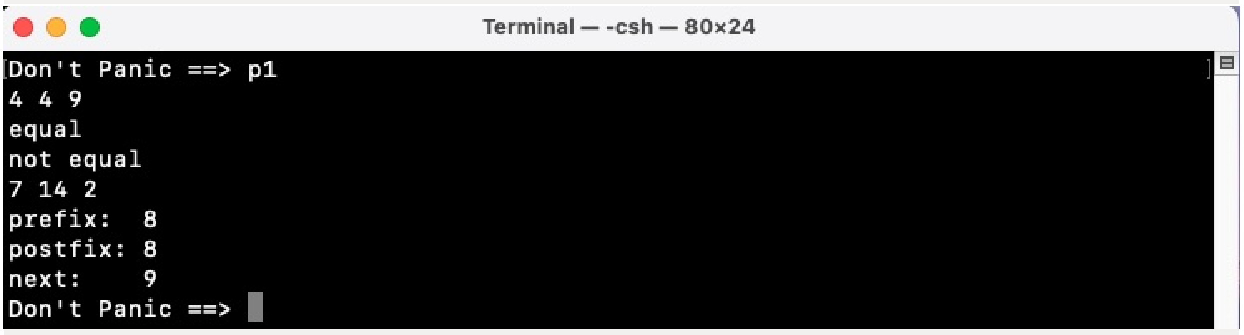
- An assignment operator performs an operation between two operands and stores the result in the left-hand side operand.
- For example: assignment (`=`), addition-assignment (`+=`), subtraction-assignment (`-=`), multiplication-assignment (`*=`), division-assignment (`/=`), modulo-assignment (`%=`) are assignment operators.

2.2.2.6. **Conditional**:

- The conditional operator returns either the second or third operand, depending on whether the first operand evaluates to true or false.
- For example: there is only one conditional (`?:`) operator.

2.2.3. Coding example: Operators

```
 1 int main()
 2 {
 3   int x, y, z;

 5   x = 4;
 6   y = x;

 8   z = y + 2 * x - 3;
 9   cout << x << " " << y << " " << z << endl;

11   if (x == y)
12     cout << "equal" << endl;
13   else
14     cout << "not equal" << endl;

16   cout << ((y == z) ? "equal" : "not equal") << endl;

18   x = y = 7;
19   z -= x;
20   y *= z;
21   cout << x << " " << y << " " << z << endl;

23   cout << "prefix:  " << ++x << endl;
24   cout << "postfix: " << x++ << endl;
25   cout << "next:    " << x   << endl;

27   return(0);
28 }
```



```
Don't Panic ==> p1
4 4 9
equal
not equal
7 14 2
prefix:  8
postfix: 8
next:    9
Don't Panic ==> 
```

Program-2.1: Operators

Program purpose:

- Program-2.1 demonstrates the use of some common operators in C++.

Lines 5-9:

- Line 8 contains four operators. The assignment operator (=) is an operator, just like the others. The only difference is that it has *lower* precedence than the multiplication, addition, and subtraction operators, so it is evaluated last.

Lines 11-14:

- These lines show an `if`-statement that branches on the equality of variables `x` and `y`.
- There no braces around the `if` and `else` blocks because braces are optional for single-statement blocks. If the code is later modified to add more statements to either block, the programmer must add the braces. Otherwise, the program will not behave as expected.

Line 16:

- This line compares two values, like lines 11-14, but it uses the conditional operator (`?:`) instead of an `if`-statement.
- From line 16, we see that the conditional operator takes three operands.
- The first operand, in this case `(y == z)`, must evaluate to a boolean value. If true, the conditional operator returns the second operand `"equal"`, otherwise it returns the third operand `"not equal"`.
- The conditional operator's return value is used as the second operand of the stream insertion operator (<<), and it is printed out using the standard output stream object `cout`.

Lines 18-21:

- Line 18 shows how the assignment operator (=) can be used multiple times in a single expression. Because the *associativity* of assignment operators is right-to-left, the first operator to execute is the right-most one.
- On line 18, the literal `7` is first assigned to variable `y`. That first assignment operation returns the new value of `y`, which then serves as the right-hand-side operand of the left-most assignment operator. So the value of `y` is assigned to variable `x`.

Lines 23-25:

- These lines show the difference between the *prefix* and *postfix* increment operators.
- It's important to note that the behaviour differences can *only* be observed if the increment operator is used in an expression with multiple operators. If the increment is the *only* operator in a statement, then prefix and postfix behaviour appear to be identical (we see in chapter 12 that there are computational performance differences).
- Lines 23-24 show the increment operator in statements with the stream insertion operator.
- On line 23, the prefix operator performs the incrementation of variable `x` by adding one to its current value. The value returned from a prefix operator is the *new value* of the incremented variable. So the initial value of `x` was `7`. Line 23 increments it to `8`, and this new value is returned to serve as the second operand of the << operator. So the value `8` is printed out, as we see from the program output.
- On line 24, the postfix operator performs the incrementation of variable `x`. However, the value returned from a postfix operator is the *original value* of the incremented variable. So the initial value of `x` was `8`. Line 24 increments it to `9`, but the original value `8` is returned to serve as the second operand of the << operator. So the value `8` is printed out, as we see from the program output. But the increment operation did take place on line 24, so when line 25 prints out the current value of `x`, we see that it's the new value `9`.

2.2.4. Characteristics of operators

Every C++ operator has three very important characteristics that determine how an expression is interpreted by the compiler.

2.2.4.1. **What is operator *arity*:**

- The ***arity*** of an operator is the number of operands that it takes.
- An operator is ***unary*** if it takes a single operand; ***binary*** if it takes two operands; and *ternary* if it takes three operands. There is only one ternary operator in C++: the conditional operator.
- Many operators are *overloaded*, so the same operator has different behaviours with different types of operands. We discuss overloading in chapter 12.
- Some operators are overloaded with different arities. For example, the `*` binary operator performs a multiplication. However, as a unary operator, it dereferences a pointer variable.

2.2.4.2. **What is operator *precedence*:**

- The ***precedence*** of an operator decides the order in which operators execute in an expression.
- For example, in Program-2.1, on line 8, multiplication has higher precedence than addition, subtraction, and assignment. So the multiplication operator executes first by multiplying the literal `2` with the value of variable `x`. Then the addition operator adds the multiplication result to the value of `y`. Then the subtraction operator subtracts `3` from the addition result. Finally, the subtraction result is assigned into variable `z`, since assignment operators have lower precedence than most other operators.
- A handy table of operator precedence can be found here.
- A programmer can always use parentheses to ensure that some operators execute first. For example, in Program-2.1, on line 16, parentheses are used around the conditional operator to make sure that its result is computed before the stream insertion operations.

2.2.4.3. **What is operator *associativity*:**

- The ***associativity*** of an operator decides the order in which operators *of the same precedence* execute in an expression. Associativity may be *left-to-right* or *right-to-left*.
- For example, in Program-2.1, on line 8, the addition and subtraction operators have equal precedence. But because they have left-to-right associativity, the left-most operator (addition) executes first, after multiplication which has higher precedence. On line 18, there are two assignment operators, so they have the same precedence. Since assignment has right-to-left associativity, the right-most operator executes first.

2.2.5. Important operators in C++

C++ provides three important operators that do not exist in C.

2.2.5.1. **Stream insertion operator:**

- The *stream insertion operator* (`<<`) is used to add a sequence of bytes to an output stream. The output stream can be one of the standard streams (`cout` for standard output or `cerr` for standard error), or an output file.
- The first operand of the stream insertion operator is the output stream object (for example, `cout`), and the second operand is the data to be output.

2.2.5.2. **Stream extraction operator:**

- The *stream extraction operator* (`>>`) is used to read a sequence of bytes from an input stream. The input stream can be the standard input stream (`cin`), or an input file.
- The first operand of the stream extraction operator is the input stream object (for example, `cin`), and the second operand is the variable into which the data is read.

2.2.5.3. **Scope resolution operator:**

- The ***scope resolution operator*** (`::`) is without question the most powerful operator in C++. Its existence is the reason why multiple inheritance can be implemented in this language, as we see in chapter 9.
- This operator links an identifier to the scope to which it belongs. For example, we may use an identifier defined inside a class somewhere else in the program. With the scope resolution operator, we can indicate that the identifier belongs to a specific class.
- We make extensive use of the scope resolution operator in this textbook once we start working with classes in chapter 3.

2.3. Control structures

We review the ***control structures*** featured in the C++ programming language.

2.3.1. Types of control structures

2.3.1.1. **Conditional:**

- The `if-else` control structure is used to branch between two blocks of code, based on the evaluation of a condition.

2.3.1.2. **Selective:**

- The `switch` control structure is used to choose one of multiple blocks of code for execution, based on the value of a variable.

2.3.1.3. **Iterative:**

- The `for`, `while`, and `do-while` control structures are used for iteration.
- The ***loop header*** is the code between a pair of parentheses that follows the loop keyword. For example, the `while` and `do-while` loop headers evaluate the loop's iteration condition, and the `for`-loop header contains three statements for managing the looping variable and testing the iteration condition.

NOTE: Recursion should *never* be used where iteration is the better choice. Recursion is a powerful tool for solving very specific problems, but it is an inelegant solution to iteration-based problems.

2.3.1.4. **Jump:**

- The `break` and `continue` control structures are used to control the program flow within an iterative control structure.
- The `break` keyword jumps the program control flow out of a loop to the first instruction that follows the loop's closing brace.
- The `continue` keyword jumps the control flow back to the loop header. In a `do-while` or `while` loop, the condition is then evaluated to determine if another iteration will execute. In a `for` loop, the advancing statement executes before the condition is evaluated.

PRO TIP: Programmers should always seek to minimize "indentation creep", where minor decisions regarding code structure can result in too many levels of indentation. Deeply nested statements can result from the inefficient placement of break and return statements, and they invariably have a negative impact on readability. Code should be structured so that most statements are within 2-3 levels of indentation inside a function.

2.3.2. Coding example: Jump control structures

```
 1 bool getArrayData(int* arr, int* num)
 2 {
 3   int newNum;
 4   int currCount = 0;

 6   cout << endl << "Enter the number of elements:  ";
 7   cin >> *num;
 8   if (*num >= MAX_ARR_SIZE) {
 9     cout << "Too many elements" << endl;
10     return false;
11   }

13   while (1) {
14     cout << "Enter the next number: ";
15     cin  >> newNum;

17     if (newNum < 0) {
18       cout << "number cannot be negative" << endl;
19       continue;
20     }

22     arr[currCount] = newNum;
23     ++currCount;

25     if (currCount >= *num) {
26       break;
27     }
28   }
29   return true;
30 }
```

Program-2.2: Jump control structures

Program purpose:

- Program-2.2 implements a `getArrayData()` function that prompts the user to enter a sequence of non-negative integers and stores them in a given integer array.
- The array and number of elements are returned using output parameters, and the function's return value indicates whether the function succeeded or failed in its task.

Lines 6-11:

- These lines prompt the end-user to specify the number of array elements to be entered. If the number exceeds the array's maximum capacity, the function returns a failure flag.

Lines 13-28:

- These lines show the `while` loop that performs the work of reading in and storing array values.
- Line 13 represents the loop header, which specifies the iteration condition, i.e. the condition that must be true for another iteration to execute. When the iteration condition is false, the loop terminates, and the program control flow transfers to the instruction following the loop.
- Lines 14-27 show the *loop body*. This is an example of an *infinite loop*, where the loop header is always true. This technique requires that the body of the loop contains a condition that terminates the loop. In C/C++, the number `0` means false, and all other values are true.
- Lines 14-15 prompt the user to enter the next number to be stored in the array.

Lines 17-20:

- These lines show a correct usage of the `continue` keyword.
- The function only allows positive numbers (or zero) to be added to the array. If the number entered is less than zero, the `continue` statement on line 19 jumps the control flow back to line 13, and the iteration condition is evaluated again. In this case, lines 22-27 are *not* executed for the current iteration.
- It is important to note that the `if`-statement on lines 17-20 *does not require an* `else` *branch!* If the condition on line 17 is true, control jumps to line 13. If it's false, execution resumes on line 22 automatically.
- While the presence of an `else` branch would not change the function's behaviour, it would result in the indentation of lines 22-27 by an additional, unnecessary level, which is bad programming style.

Lines 22-23:

- These lines add the user-entered number to the array and increment the current number of array elements.

Lines 25-27:

- These lines show a correct usage of the `break` keyword.
- If the number of elements specified by the end-user on line 7 has been reached, then the loop must terminate. The `break` statement on line 26 jumps the control flow to line 29, which is the first instruction following the loop's closing brace on line 28.

2.4. Variables and data types

When a program is launched, the operating system (OS) assigns to it four separate areas of virtual memory: the code segment, the data segment, the function call stack, and the heap. The last three are used to store the program's data, in a manner decided by the programmer.

Variables are an essential tool for a program to manipulate data. We can think of a program's memory as a bookshelf and variables as boxes stored in that bookshelf.

2.4.1. Characteristics of variables

The main characteristics of a variable in C/C++ include: a name, a value, a data type, and a location in memory.

2.4.1.1. **Name:**

- A variable's name is the identifier used in a program to store or retrieve a unit of data.
- If we think of a variable as a box in a bookshelf, its name is the label affixed to the box.

2.4.1.2. **Value:**

- A variable's value is the data stored in that variable.
- If the variable is a box, its value is the contents inside the box.

2.4.1.3. **Data type:**

- A variable's data type determines the number of bytes required in memory to store its data.
- If the variable is a box, its data type decides the size of the box.

2.4.1.4. **Location:**

- A variable's location is the address in memory where its first byte is stored. The subsequent bytes of a variable are necessarily contiguous. So given the address of the first byte and the number of bytes that a variable occupies, its contents can be retrieved or updated.
- If the variable is a box, its location is the address in the bookshelf where the box is stored.

2.4.2. Data types

2.4.2.1. **Primitive (also called *built-in*) data type:**

- A ***primitive data type*** is a data type that is built into the programming language.
- In C++, these include `int`, `float`, `double`, `char`, and `bool`.

2.4.2.2. **User-defined data type:**

- In the context of a programming language, the user is the programmer.
- A ***user-defined*** *data type* is a data type that is defined by the programmer.
- In C++, these include classes and `struct`s.

2.4.2.3. **Memory address:**

- In C++, pointer variables are used to store memory addresses as their contents. Pointers are discussed in chapter 4.

2.4.3. Aggregate data types

Aggregate data types in C++ allow for the grouping of data and behaviour. These include C-style `struct`s and classes.

2.4.3.1. **C-style `struct`s:**

- In C++, `struct`s contain variables and functions, just like classes do.
- By default, the variables and functions inside a `struct` are declared with *public* access, so the entire program can view or update the `struct`'s variables and call its functions. This is **bad** software engineering.

2.4.3.2. **Classes:**

- Classes contain variables and functions, as they do in most OO languages.
- By default, the variables and functions inside a class are declared with *private* access, which restricts how the program can use or update them. This is **good** software engineering.
- The use of classes instead of `struct`s upholds the *principle of least privilege*, which is discussed in chapter 5.

PRO TIP: Classes should always be used instead of `struct`s, to ensure that we follow the principles of correct software engineering.

2.5. Standard I/O Streams

Unix-type operating systems, including Linux, conceptualize standard input/output (I/O) as streams, which are essentially sequences of bytes. In C++, the standard I/O streams are represented as instances of classes that are defined in the `iostream` library.

2.5.1. Standard I/O stream objects

2.5.1.1. **Using standard I/O stream objects**:

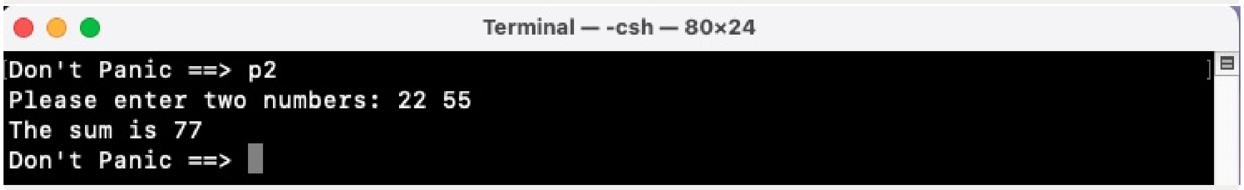
- The standard I/O stream objects are encapsulated within the `std` namespace. To use them in a program, we can use either of the following techniques:
  - precede every reference to a standard I/O stream object with the `std` namespace, followed by the scope resolution operator (`::`); for example, the standard output object `cout` can be referred to as `std::cout`
  - explicitly scope in the entire `std` namespace, with the statement `using namespace std` at the top of a source file; the standard I/O objects can then be used directly

2.5.1.2. **Important stream objects**:

- The two commonly used standard I/O stream objects are `cin` and `cout`.
- There are additional streams related to error reporting and logging, but these are left to explore as an exercise for the reader.
- The `cout` object is an instance of the output stream class `ostream`.
- The `cin` object is an instance of the input stream class `istream`.

2.5.2. Coding example: Standard I/O streams

```cpp
#include <iostream>
using namespace std;

int main()
{
  int x, y, z;

  cout << "Please enter two numbers: ";
  cin  >> x >> y;
  cout << "The sum is " << x+y << endl;

  return(0);
}
```


Terminal — -csh — 80×24

```
Don't Panic ==> p2
Please enter two numbers: 22 55
The sum is 77
Don't Panic ==> 
```

Program-2.3: Standard I/O streams

Program purpose:

- Program-2.3 demonstrates the use of both the stream insertion and stream extraction operators, with the standard I/O stream objects.

Line 2:

- This line shows the scoping in of the entire `std` namespace, so that every use of the identifiers within it does not require the scope resolution operator.

Line 8:

- This line prints out a literal string to the standard output stream, *without* a newline character at the end.
- The printing uses the *stream insertion operator* to send the string to the `cout` stream object.

Line 9:

- This line shows two values being read from the standard input stream and stored into local variables `x` and `y`.
- It uses the *stream extraction operator* to retrieve each value from the `cin` stream object.
- The two occurrences of the stream extraction operator within the same statement is called *cascading*, which is discussed in chapter 12. It is analogous to using any two operators in the same expression, for example the multiple arithmetic operators on line 8 of Program-2.1.

2.6. Functions

Functions play a fundamental role in the correct design of any OO program. We discuss the types of functions in C++, as well as the design terminology around functions and their parameters.

2.6.1. Types of functions

C++ supports two types of ***functions***, which are differentiated by their scope: *global functions* and *member functions*.

2.6.1.1. **What is a *global function*:**

- As the name implies, a *global function* has global scope, and it is defined at file scope, outside of any blocks. It can be called from any function and any class in the program.
- In C, all functions are global.
- The `main()` function is an example of a global function in C++.

2.6.1.2. **What is a *member function*:**

- A *member function* has scope within a single class. Depending on each member function's *access specifier* (public, protected, or private), a member function can be called on an object of the class, or on the class itself in the case of *static member functions*, which are discussed in chapter 8.
- In some OO languages, member functions are called *methods*. That term is **not** standard use in C++, and it will be not be used in this textbook.

PRO TIP: It is considered good practice to minimize the number of global functions in our OO designs, and to use member functions wherever possible.

2.6.2. Characteristics of well-designed functions

Functions that follow a correct design typically have the following characteristics.

2.6.2.1. **Well-designed functions are *modular*:**

- They "receive" data from the calling function, through parameters.
- They do some work and/or compute some result.

- They “give back” results to the calling function. There are two ways to do this: using the function’s return value, which is a very limited technique, or using parameters

2.6.2.2. **Well-designed functions are *single-purpose*:**

- They have a single goal and do one thing only.

2.6.2.3. **Well-designed functions use *abstraction* to hide their functionality:**

- Other functions know *what* the function does, not *how* it does it.
- They know how to call the function, what information to provide through parameters, and what information it returns. They do *not* know the function’s logic or how it’s implemented.

2.6.2.4. **Well-designed functions are *reusable*:**

- They can be reused within the same program.
- A truly well-designed function may be reused in different programs as well.

2.6.3. Return values

2.6.3.1. **Function success or failure:**

- Return values can be used to indicate the success or failure of a function upon its termination.
- They can be used to return results from very simple functions, for example getter functions.
- Return values are a very limited technique, since only a single value can be returned.

2.6.3.2. **Returning data from a function:**

- Actual function results, or any complex data, should be returned using *output parameters*.
- Output parameters allow a function to return *multiple results*, unlike a return value. They are discussed in the next section.

2.7. Function parameters

We review the design and implementation terminology related to function parameters. We also discuss how they are shared between functions.

2.7.1. Calling and called functions

2.7.1.1. **What are *calling and called functions*:**

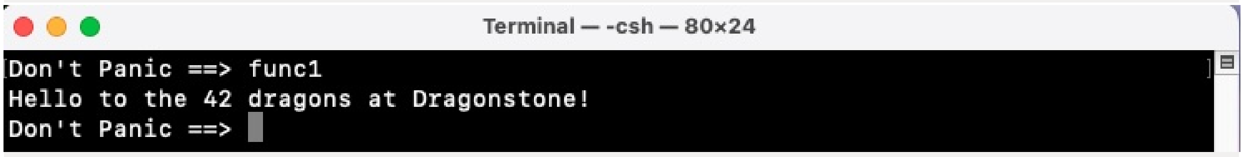
- A ***calling function*** is one that, as part of its instructions, calls another function.
- A ***called function*** is a function that is called by another.

2.7.1.2. **Example:**

- In Program-2.4, function `foo()` calls function `bar()` on line 4.
- For the duration of `bar()`’s execution on lines 8-10, `foo()` is the calling function, and `bar()` is the called function.
- The purpose of function parameters is to transfer information back and forth between the calling function and the called function.
- In Program-2.4, `foo()` tells `bar()` how many dragons are in Dragonstone by passing the value of its local variable `numDragons`, as a parameter to `bar()`.

2.7.2. Coding example: Calling and called functions

```
 1 void foo()
 2 {
 3   int numDragons = 42;
 4   bar(numDragons);
 5 }

 7 void bar(int n)
 8 {
 9   cout << "Hello to the " << n << " dragons at Dragonstone!" << endl;
10 }
```



```
Don't Panic ==> func1
Hello to the 42 dragons at Dragonstone!
Don't Panic ==>
```

Program-2.4: Calling and called functions

2.7.3. Parameter modifiability

2.7.3.1. What is *parameter modifiability*:

- *Parameter modifiability* is a design technique that specifies if the *value* or the *address* of a variable inside the calling function is passed as a parameter into a called function.
- A called function may change the values of variables declared in the calling function, depending on a parameter's modifiability.
- There are two types of parameter modifiability in C++: *pass-by-value* and *pass-by-reference*.

2.7.3.2. Pass-by-value:

- With *pass-by-value*, a parameter value is *copied* from the calling function into a local variable in the called function.
- Any changes that the called function makes to that parameter value are to the local copy *only*. No changes can be made inside the calling function using a pass-by-value parameter.
- For example, on line 4 of Program-2.4, the value of `numDragons`, which is `42`, is passed by value into `bar()` and stored in `bar()`'s local variable `n`. Any changes made to `n` during `bar()`'s execution would be made to the local copy only. In this program, there is no way for `bar()` to change the value of the `numDragons` variable declared in `foo()`.

2.7.3.3. Pass-by-reference:

- With *pass-by-reference*, the *memory address* of a variable is passed from the calling function into a parameter in the called function.
- If a parameter is passed by reference, there is no local copy of its value inside the called function. But the called function may access and change the value inside the calling function by dereferencing the pass-by-reference parameter.
- For example, on line 4 of Program-2.5, the variable `numDragons` is passed by value into the called function `bar()`, and `goneDragons` is passed by reference. The called function contains a local copy of the value `42` in its `n` variable, but local variable `gone` contains the *memory address* of the variable `goneDragons` that's declared in the calling function `foo()`.
- In C++, parameters are passed by reference by using pointers or *references*. C++ references are introduced in the next section.

2.7.3.4. **Best practice**

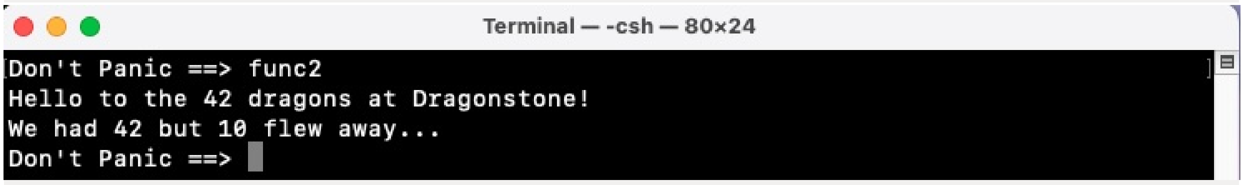
- Except for primitive data type parameters, it is almost always better coding practice to use pass-by-reference instead of pass-by-value.
- The problem with pass-by-value is that it creates *copies* of the data. There are two significant issues with this:
  - *Making copies takes time.* Why waste computational resources making unnecessary copies? There may be little efficiency impact if copies are made only a few times, but what about 1000 times? Or a million times? We should **always** write code with *scalability* in mind. Our design should be able to handle large volumes of data as easily as small amounts.
  - *Making copies may result in an inconsistent state.* Most data should have a single instance only, so that information remains consistent through the different parts of the program.

2.7.3.5. **Beware of creating an *inconsistent state*:**

- An ***inconsistent state*** occurs in a program when it contains contradictory information. For example, one part of the program may have one value for some important data, but another part has a different value for *the same data*.
- This is can be a consequence of inadvertently duplicating data, instead of maintaining multiple references to a single instance. With multiple copies of what is supposed to the *same data*, there is always the risk that each instance ends up with a different value.
- For example, in a university system, there should be only one instance of your student record, with potentially multiple references that same record. That way, if you need to change your home address, the single instance is changed, and the new information becomes available everywhere in the system. But if multiple instances exist, changing your address may update it in some parts of the system but not others. That would make your information inconsistent across the university system, which inevitably leads to errors (and lost mail).

2.7.4. Coding example: Pass-by-value and pass-by-reference

```cpp
void foo()
{
  int numDragons = 42;  int goneDragons;
  bar(numDragons, &goneDragons);
  cout << "We had " << numDragons << " but " << goneDragons
       << " flew away..." << endl;
}

void bar(int n, int* gone)
{
  cout << "Hello to the " << n << " dragons at Dragonstone!" << endl;
  *gone = 10;
}
```



```
Don't Panic ==> func2
Hello to the 42 dragons at Dragonstone!
We had 42 but 10 flew away...
Don't Panic ==>
```

Program-2.5: Pass-by-value and pass-by-reference

2.7.5. Parameter direction

2.7.5.1. **What is *parameter direction*:**

- The ***parameter direction*** is a design technique that specifies if a parameter is *providing information* to the called function, or if it's returning information from the called function to the calling one.
- The parameter direction is specific to each parameter. It is also *unrelated* to whether the parameter is passed by value or by reference.
- There are three types of parameter direction: *input*, *output*, and *input-output*.

NOTE: The parameter direction has *nothing* to do with user I/O.

2.7.5.2. **Input parameter:**

- An ***input parameter*** specifies a value that is required by the called function in order to work.
- The value of an input parameter must be initialized by the calling function.
- If the calling function passes garbage as an input parameter, the called function will fail and possibly crash the program.
- An input parameter may be passed by value or by reference.

2.7.5.3. **Output parameter:**

- An ***output parameter*** is one that is initialized by the called function.
- The calling function must allocate the memory that is initialized by the called function.
- There is no expectation that the output parameter is initialized by the calling function. Any value contained in the referenced variable gets overwritten by the called function.
- Because the called function modifies the output parameter, it is passed by reference.

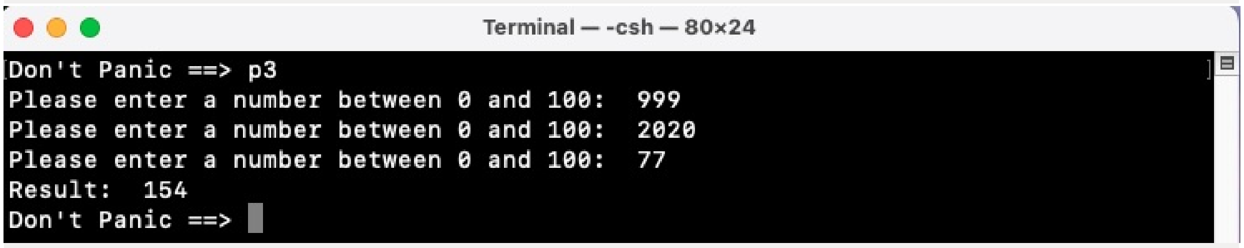
2.7.5.4. **Input-output parameter:**

- An *input-output parameter* is both an input and an output parameter at the same time.
- The value of an input-output parameter must be initialized by the calling function, because it is required by the called function in order to work.
- The input-output parameter value is also modified by the called function during its execution.
- Because the called function modifies the input-output parameter, it is passed by reference.

NOTE: Parameter modifiability and parameter direction are **not** implementation concepts. They are both *design* concepts, and they are programming language-independent. During function design, we decide how each function uses parameters for sharing information. During implementation, the mechanism used for parameter modifiability and direction depends on the programming language.

2.7.6. Coding example: Parameter modifiability and direction

```
 1 int main()
 2 {
 3   bool inputOk = false;
 4   int  num, result;

 6   while (!inputOk) {
 7     cout << "Please enter a number between 0 and 100:  ";
 8     cin >> num;
 9     inputOk = checkNum(num);
10   }

12   doubleNum(num, result);
13   cout<<"Result:  " << result << endl;

15   return 0;
16 }

18 void doubleNum(int n, int& res)
19 {
20   res = n * 2;
21 }

23 bool checkNum(int n)
24 {
25   return (n>=0 && n<=100);
26 }
```



```
Don't Panic ==> p3
Please enter a number between 0 and 100:  999
Please enter a number between 0 and 100:  2020
Please enter a number between 0 and 100:  77
Result:  154
Don't Panic ==> 
```

Program-2.6: Parameter modifiability and direction

Program purpose:

- Program-2.6 prompts the end-user to enter a number and calls the `checkNum()` function to validate that the number entered is between `0` and `100`. If not, the user is prompted again until they enter a valid number. The `doubleNum()` function doubles the value of the entered number, and the result is printed to the screen.

Line 1-16:

- These lines show the implementation of the `main()` function.
- On line 9, `main()` is the calling function and `checkNum()` is the called one. The `num` variable is passed by value as an input parameter into the called function. The return value is used to communicate success or failure back to the calling function.
- On line 12, `main()` is the calling function and `doubleNum()` is the called one. The `num` variable is passed by value as an input parameter into the called function. The `result` variable is passed by reference as an output parameter.

Lines 18-21:

- These lines show the implementation of the `doubleNum()` function.
- The function takes two parameters: `n` is passed by value, and `res` is passed by reference *using a C++ reference and not a pointer*. We discuss C++ references in the next section.
- An existing value is required in parameter `n` on line 20 for `doubleNum()` to do its work. So it has an input direction. Also, the parameter does not get modified by `doubleNum()`, so it has no output direction. Therefore, `n` is an input parameter.
- No pre-existing value is required in parameter `res`, so it has no input direction. The value gets modified on line 20, so it has an output direction. Therefore, `res` is an output parameter.

Lines 23-26:

- These lines show the implementation of the `checkNum()` function.
- The function takes one parameter: `n` is passed by value.
- An existing value is required in parameter `n` on line 25 for `checkNum()` to do its work. So it has an input direction. Also, the parameter does not get modified by `checkNum()`, so it has no output direction. Therefore, `n` is an input parameter.

2.8. References

One of the most important advantages of passing parameters by reference is the ability for a called function to modify data stored in a calling function. This capability is crucial for the correct design of classes and functions in delegating sub-tasks to other parts of the program.

References are a feature of C++ that does not exist in C. They play an essential role in passing parameters by reference.

2.8.1. Characteristics

2.8.1.1. **Passing parameters by reference in C++:**

- In C++, parameters may be passed by reference in one of two ways: using pointers or using references.
- Passing parameters using pointers works the same way as in C. Pointers are very powerful, but they can make the code overly complicated. We review pointers in chapter 4.
- Passing parameters using C++ references is a feature of C++ only. References use simple syntax, but they provide very limited functionality.

2.8.1.2. **What is a *reference*?**

- A ***reference*** is a **binding**, or an *alias*, between an existing variable and a new identifier.
- Declaring and initializing a reference creates a bond between:
  - the reference name as a new identifier, and
  - an existing variable
- The reference bond is *unbreakable* for the scope of the reference identifier. It cannot be reassigned to a different variable.
- The most common usage for references is passing parameters by reference, although they can have other uses.

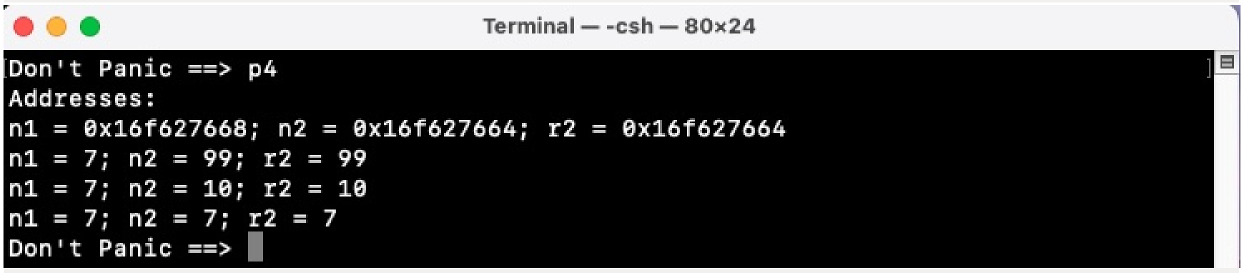
2.8.1.3. **What is a reference *not*?**

- A reference is **not** a separate variable! It does *not* occupy memory. It is simply an alternative name for an existing variable.

- A reference must be *bound* to an existing variable when it is declared. A reference cannot be declared without being initialized at the same time.
- The reference cannot be bound to a different variable during its lifetime.

2.8.2. Coding example: C++ references

```
 1 int main()
 2 {
 3   int n1, n2;

 5 //  int& r1;
 6   int& r2 = n2;

 8   n1 = 7;
 9   n2 = 99;

11   cout <<"Addresses:" << endl;
12   cout<< "n1 = "<<&n1<<"; n2 = "<<&n2<<"; r2 = "<<&r2 << endl;

14   cout<< "n1 = "<<n1<<"; n2 = "<<n2<<"; r2 = "<<r2 << endl;

16   r2 = 10;
17   cout<< "n1 = "<<n1<<"; n2 = "<<n2<<"; r2 = "<<r2 << endl;

19   r2 = n1;
20   cout<< "n1 = "<<n1<<"; n2 = "<<n2<<"; r2 = "<<r2 << endl;

22   return 0;
23 }
```



```
Don't Panic ==> p4
Addresses:
n1 = 0x16f627668; n2 = 0x16f627664; r2 = 0x16f627664
n1 = 7; n2 = 99; r2 = 99
n1 = 7; n2 = 10; r2 = 10
n1 = 7; n2 = 7; r2 = 7
Don't Panic ==>
```

Program-2.7: C++ references

Program purpose:

- Program-2.7 shows an example of how references are declared and how they behave.

Lines 5-6:

- Line 5 declares the identifier `r1` as a reference to an integer, using the data type `int&`, but it does not assign a value to `r1` upon declaration. For this reason, line 5 does *not* compile and is commented out.
- Line 6 shows the correct declaration of identifier `r2` as an integer reference that is bound to variable `n2`. From that line, until the end of `r2`'s scope at the end of the `main()` function, the identifier `r2` is an alias for variable `n2`. It is *not* a separate variable.

NOTE: The use of the ampersand (`&`) to declare a reference is ***UNRELATED*** to the address-of operator used to initialize pointers. We can tell how the `&` symbol is used, based on whether it's found in a variable declaration or in a regular statement with operators.

Line 12:

- Line 12 prints out the addresses of three variables: `n1`, `n2`, and `r2`.
- Because `n2` and `r2` are the *exact same variable*, their addresses are the same, as we see from the program output.
- This line is an example where the ampersand symbol (`&`) is used as the address-of operator, and not in a reference declaration.

Line 14:

- Line 14 prints out the values of `n1`, `n2`, and `r2`.
- Because `n2` and `r2` are the *exact same variable*, their values are the same, as we see from the program output.

Lines 16-17:

- Because `r2` is *not* a separate variable and is simply an alias for `n2`, line 16 sets the value of variable `n2` to the value `10`.
- On line 17, the same value `10` is printed out for both `n2` and `r2`, because they remain the *exact same variable*.

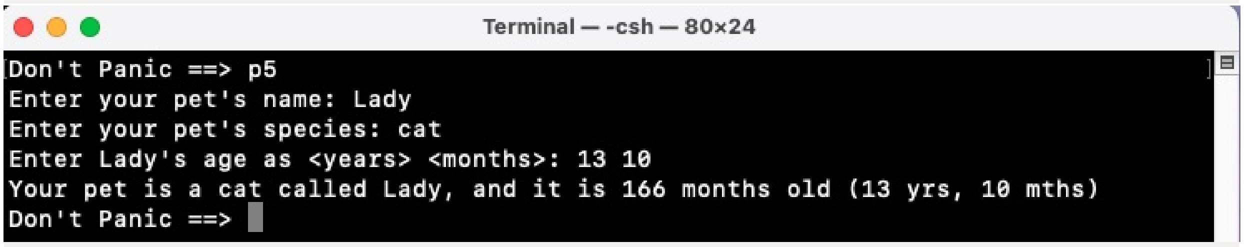
Lines 19-20:

- Again, because `r2` is an alias for `n2`, line 19 sets the value of variable `n2` to the value currently stored in `n1`.
- Once a reference is declared and initialized, it *cannot* be reassigned to a different variable.
- On line 20, the same value `7` is printed out for `n1`, `n2`, and `r2`, as seen in the program output.

NOTE: It would be lovely to choose one favourite pass-by-reference technique (pointers or references) and use it everywhere exclusively. Unfortunately, almost nothing in programming is that simple. Pointers are mandatory for some C++ language features, for example the use of dynamically allocated memory, and references are mandatory for other features like operator overloading.

2.8.3. Coding example: Parameter passing with references

```
int main()
{
  string name, species;
  int    age;

  enterInfo(name, species, age);

  cout << "Your pet is a " << species << " called " << name
       << ", and it is " << age << " months old ("
       << age/12 << " yrs, " << age%12 << " mths)"
       << endl;

  return 0;
}

void enterInfo(string& n, string& s, int& a)
{
  int years, months;

20   cout << "Enter your pet's name: ";
21   cin  >> n;

23   cout << "Enter your pet's species: ";
24   cin  >> s;

26   cout << "Enter " << n << "'s age as <years> <months>: ";
27   cin  >> years >> months;

29   a = years*12 + months;
30 }
```



```
Don't Panic ==> p5
Enter your pet's name: Lady
Enter your pet's species: cat
Enter Lady's age as <years> <months>: 13 10
Your pet is a cat called Lady, and it is 166 months old (13 yrs, 10 mths)
Don't Panic ==> 
```

Program-2.8: Parameter passing with references

Program purpose:

- Program-2.8 demonstrates the use of C++ references to pass parameters by reference.
- The program uses the `enterInfo()` function to prompt the end-user to enter their pet's data. The information is stored in variables declared in `main()`, and they are passed as parameters to `enterInfo()` using references.

Lines 1-14:

- These lines show the implementation of the `main()` function.
- Lines 3-4 allocate the memory for three variables in `main()` to store the pet information (name, species, and age).
- Line 6 calls `enterInfo()` with the three variables passed in using references. It is unclear from line 6 alone that the parameters are declared as references, until we look at the function prototype on line 16.

Lines 16-30:

- These lines show the implementation of the `enterInfo()` function.
- Line 16 reveals that the three parameters are passed as references. So for the scope of the entire `enterInfo()` function on lines 16-30, the following is true:
  - the reference `n` is an alias for the `name` variable declared in `main()`
  - the reference `s` is an alias for the `species` variable declared in `main()`
  - the reference `a` is an alias for the `age` variable declared in `main()`
- At all times, there is a single instance of each of the three variables for the pet name, species and age. There are no duplicates of this information anywhere in the program. It *only* exists in the three variables declared in `main()`.
- Lines 21 and 24 read the name and species information from the end-user. The data is stored directly in the corresponding variables in `main()`, through their parameter references.
- Line 27 reads the age as years and months. Line 29 computes the total number of months and stores the result in the corresponding variable in `main()`, through its parameter reference.
- Lines 21, 24, and 29 show that the data is returned using the three output parameters.

2.9. Programming conventions

Every programming language has its own programming conventions, and they can be different across platforms. Every company and software development team also has its own particular conventions, so it's necessary to constantly adapt our own personal style for each project.

This section discusses the programming conventions we use in this textbook. These are based on traditional C++ conventions for Unix-type operating systems.

2.9.1. Naming conventions

2.9.1.1. **What are *naming conventions*:**

- *Naming conventions* are a *communication tool* between programmers. They are a short-cut to understanding someone else's code.
- If a programmer strays from the usual standards, their code will be very difficult to read and modify, with likely negative consequences for their employability.
- We can never assume that we are the last programmer to work on our code. Realistically, there will always be someone coming along after us, possibly a junior programmer or a coop student, to update or extend our code. We want to make their job easier by following conventions.

2.9.1.2. **Constants:**

- Constant names are usually in all uppercase letters.
- Compound names (names made up of multiple words) can be separated using underscores, for example `MAX_ARR_SIZE`.

2.9.1.3. **Variables:**

- Variables names begin with a lowercase letter.
- Compound names may use *camel-case* or underscores to separate the words, for example `numElements` or `num_elements`.
- *Camel-case* is the practice of writing the first letter of a new word in uppercase, with the other letters in lowercase. The shifts in case go up and down like the bumps on a camel's back.
- Variable names must be descriptive without being overly long. We can use scope as the context, for example inside a `Student` class, we can use `id` instead of `studentId`.
- Single letters can be used as temporary variables, for example `i` or `j` for loop counters.

2.9.1.4. **Data types:**

- Data type names begin with an uppercase letter, for example the `Student` class.
- Compound names may use camel-case or underscores to separate the words, for example the `BookArray` class.

2.9.1.5. **Functions:**

- Functions names, whether they are global or member functions, begin with a lowercase letter, for example a `getName()` member function.
- Compound names may use camel-case or underscores to separate the words.
- Function names must be short and descriptive.

2.9.2. Coding practices

2.9.2.1. **What are *coding practices*:**

- Like naming conventions, *coding practices* are used as a short-cut to understand another programmer's code.
- The coding practices that we follow will vary based on the company and development team where we work, and they will adapt to match each project.

2.9.2.2. **Common coding practices we use:**

- Do design classes that are separated into object categories (entity, control, interface), which are discussed in chapter 6.
- Do **not** use `struct`s, but use classes instead.
- Do **not** use global variables.
- Do **not** use global functions, except for `main()`, unless directed.
- Do **not** pass objects by value, but by reference instead.
- Do reuse the code that you write wherever possible. Do not copy and paste code. Instead, create and call a separate function that contains the common code.
- Do perform all basic error checking.

Chapter 3

Basic C++ Classes

In this chapter, we present the basic features of simple C++ classes and their organization in a Unix-type OS. We discuss the different types of constructors in C++, and we introduce destructors.

3.1. Principles

We review some OO programming terminology in general and the specific terms that apply to C++ in particular.

3.1.1. Terminology

3.1.1.1. **What is a *class*:**

- A *class* in an OO programming language is a programmer-defined data type that aggregates together related data and behaviour.
- Every class contains:
  - data, called *data members* in C++ (sometimes called *instance variables* in some other OO languages), and
  - behaviour, called *member functions* in C++ (*methods* in other OO languages)
- As a data type, a class is a *blueprint* only. It stores no actual data, but it defines what data every instance of the class contains. For example, a `Student` class may specify that all students have a student number and a name.
- A class occupies no memory, and it cannot be used in any operations, except for *static member functions*, which are discussed in chapter 8.

3.1.1.2. **What is an *object*:**

- An *object* is an instance of a defined class.
- While the class specifies the blueprint on which every object is modelled, an object contains specific values for its data.
- An object does occupy memory. Like any variable. it has a name, a value for each of its data members, a data type (which is the class), and a location in memory.
- For example, we can declare an object as a variable with a name of `matilda`, a data type of `Student` class, and a value of `"100567899"` for the student number and `"Matilda"` for the student name.

3.1.2. Class members

3.1.2.1. What is a *class definition*:

- A ***class definition*** defines a new class data type.
- It contains the following information:
  - a class name, and
  - class members
- In a class definition, ***class members*** include:
  - data members (sometimes called instance variables in other OO languages), and
  - member functions (sometimes called methods in other OO languages)
- Every class member has an access specifier (public, protected, or private).

NOTE: Industrial-grade software is typically huge, with potentially millions of lines of code, written by hundreds of developers in multiple releases over many years. It is imperative that the software is written in a way that protects the runtime objects from possible bugs and runtime errors. One important way to do this is with the *principle of least privilege*, where each class member is given the most restrictive access (private or protected) wherever possible.

3.1.2.2. What is an *access specifier*:

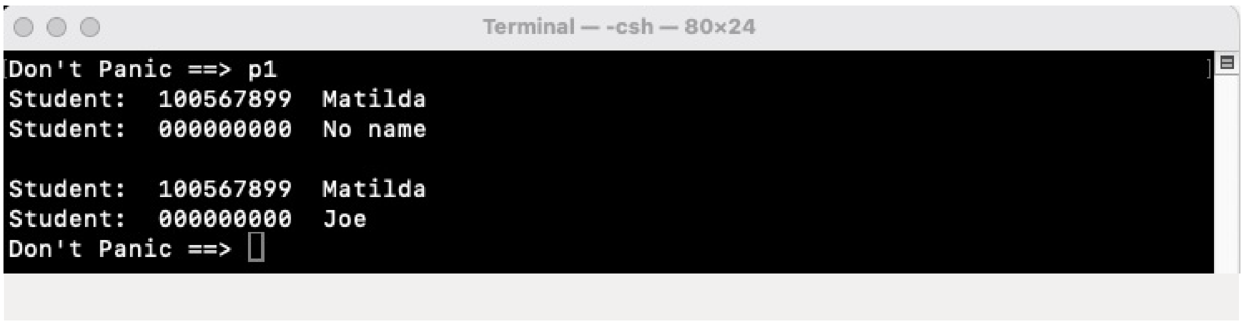
- An *access specifier* defines the level of access to a class member by the rest of the program.
- *Public* access:
  - a *public* class member is visible to all objects and global functions in the program
- *Protected* access:
  - a *protected* class member is visible to objects of other subclass types only (subclasses are sometimes called "child classes")
  - protected access only makes sense when classes are organized in an inheritance hierarchy, which is discussed in chapter 9
  - to global functions and objects of all other classes, a protected member is private
- *Private* access:
  - a *private* class member is not visible to objects of any other class types, or to global functions
  - but in a surprising twist, *C++ objects of the same class* **can** *access each other's private members*; this is necessary for the correct functioning of copy constructors, as we discuss later in this chapter
- Within C++ classes, the *default* access specifier is private, which supports the principles of good software engineering.

3.1.2.3. What is a *function implementation*:

- A ***function implementation*** consists of the sequence of statements that comprise that function, i.e. the statements inside the pair of braces.
- A function implementation is also called the *function body*. These terms apply to both global functions and member functions.
- In a correctly organized C++ program, member function implementations are stored outside the class definition, as we discuss later in this chapter.

3.1.3. Coding example: Simple C++ class

```cpp
class Student
{
  public:
    Student()
    {
      number = "000000000";
      name   = "No name";
    }

    Student(string s1, string s2)
    {
      number = s1;
      name   = s2;
    }

    string getName()
    {
      return name;
    }

    void setName(string n)
    {
      name = n;
    }

    void print()
    {
      cout<<"Student:  "<<number<<"  "<<name<<endl;
    }
  private:
    string number;
    string name;
};

int main()
{
  Student matilda("100567899", "Matilda");
  Student joe;

  matilda.print();
  joe.print();

  joe.setName("Joe");
  matilda.print();
  joe.print();

  return 0;
}
```



```
Don't Panic ==> p1
Student:  100567899  Matilda
Student:  000000000  No name

Student:  100567899  Matilda
Student:  000000000  Joe
Don't Panic ==> 
```

Program-3.1: Simple C++ class

Program purpose:

- Program-3.1 declares a `Student` class with data members and member functions. The `main()` function creates two instances of the `Student` class and calls their member functions.

Lines 1-33:

- These lines show the `Student` class definition.
- The *public* portion of the class definition is found on lines 3-29, and everything declared on those lines is given public access.
- The *private* portion of the class definition is found on lines 30-32, and everything declared on those lines is given private access.
- There are no class members declared with protected access.

Lines 31-32:

- These lines declare the data members of the `Student` class: the `number` data member that stores the student number, and `name` that contains the student's name.
- We use the C++ standard library's `string` class throughout the coding examples in this textbook.
- Every instance of the `Student` class, therefore every `Student` object, has its own values for each data member.
- In keeping with the correct principles of good software engineering, both data members are declared with private access. Because of this, only the member functions that belong to the `Student` class are allowed to access and modify the data members directly.

Lines 4-8:

- The member function defined on these lines is a *constructor* that takes no parameters. In C++, we call this the *default constructor*. It sets all the data members to default values.

Lines 10-14:

- The member function defined on these lines is a *constructor* that takes two parameters. It initializes the data members to specific values that are passed in as parameters.
- In a later section, we present the correct technique for combining multiple constructors into a single one.

Lines 16-24:

- These lines define getter and setter member functions for the class's `name` data member.

Lines 26-29:

- These lines define a member function that prints all the information contained in a `Student` object.

- There is no C++ equivalent to Java's `toString()` member function. However, a better approach for outputting an object directly to a stream is discussed in chapter 12.

Line 37:

- This line declares an instance of the `Student` class, with the variable name `matilda`.
- When an object is declared, the class constructor is called automatically. Because two parameter values are used in the declaration (`"100567899"` and `"Matilda"`), the two-parameter constructor on lines 10-14 is called.
- A constructor is *always* called when an object is created.
- Because we do *not* use the `new` keyword on line 37, the new `Student` object is **not** allocated dynamically. It is allocated statically, so the object resides in the function call stack, inside the stack frame for the `main()` function, and not in the heap.
- Unlike Java, C++ does not limit all objects to dynamic allocation. The language allows the programmer to decide whether an object is allocated statically or dynamically.
- We review memory management concepts in chapter 4, where we discuss statically and dynamically allocated memory, and how C++ manages the memory allocation of objects.

Line 38:

- This line declares an instance of the `Student` class, with the variable name `joe`.
- Because no parameter values are used in the declaration, the default constructor on lines 4-8 is called.

Lines 40-41:

- These lines print out the information in both `Student` objects.
- Line 40 calls the `Student` class's `print()` member function defined on lines 26-29 using the `matilda` object. The student number and name that belong to that object are printed to the screen, as we see in the program output.
- Line 41 calls the `print()` member function using the `joe` object. The student information initialized in the default constructor on lines 4-8 is printed out.
- Because `matilda` and `joe` are *objects*, and not *pointers to objects*, we use the dot operator (`.`) to access their class members, in this case the `print()` member function. In chapter 4, we see how the arrow operator (`->`) is used to access class members using pointers to objects. This is consistent with the C syntax used for accessing the fields of `struct`s.

Lines 43-45:

- These lines change one student's name and print the information in both objects again.
- Line 43 calls the `Student` class's `setName()` member function using the `joe` object. This modifies the value of that object's `name` data member to the given parameter `"Joe"`.
- Lines 44-45 call the `print()` member function again on both `Student` objects, and we see the updated name for the `joe` object in the program output.

3.2. Code organization

In software engineering, a very large program or application is called a ***software system***. When developing a large system, it’s important to consider how to best organize the code into separate source files.

A key goal is to separate the code in ways that minimize the impact of changes on other parts of the system. Reducing the number of files *that require recompilation* after a change is an important technique for achieving this goal.

3.2.1. Class organization

3.2.1.1. **Very small programs**

- Program-3.1 presents two serious issues in terms of code organization.
- First, the member function implementations (the function bodies) are contained *inside the class definition*. While this is standard practice is some other OO programming languages, it fails to minimize program building dependencies between classes.
- Second, the program is contained entirely in a single source file. While this may be acceptable for very small programs, it’s problematic for larger ones.

3.2.1.2. **All other programs**

- In a correctly organized C++ program, each class is separated into *two files*:
  - a *header file* that contains the class definition, and
  - a *source file* that contains the class’s member function implementations
- While this does increase the number of files in the program, the advantages for minimizing recompilation efforts are worthwhile. As we discuss in chapter 5, class independence is an important factor in strengthening resilience to changes, and packaging code correspondingly is a key part of that.

3.2.1.3. **Class header file**

- Each class definition is contained in its own header file.
- A class header file is named after the class itself, followed by the `.h` extension. For example, if we define a `Date` class, the corresponding header file is called `Date.h`
- The header file contains the class definition, including the data member declarations and the member function *prototypes only*. It does **not** contain the implementation (the actual code) for the member functions.

3.2.1.4. **Class source file**

- Each class has its own source file.
- A class source file is named after the class itself, followed by the `.cc` extension. For example, the source file for the `Date` class is called `Date.cc`
- The source file contains the member function implementations.
- If the class declares static storage class data members, they must be initialized in the source file, as we discuss in chapter 8.

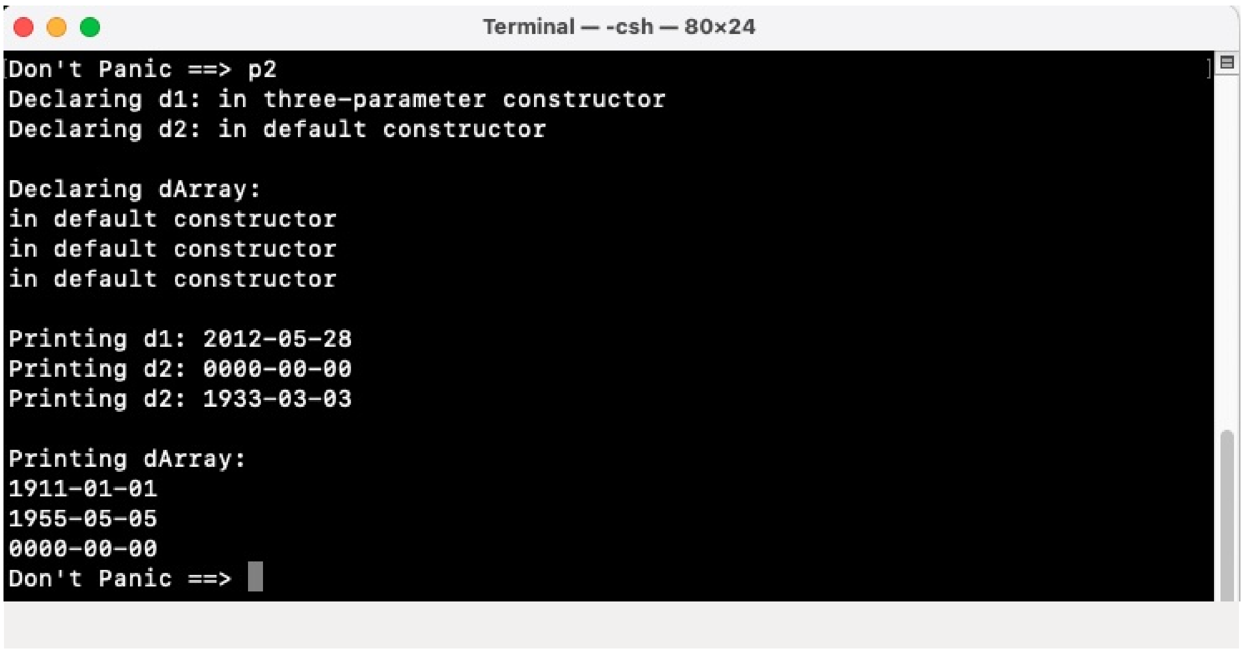
3.2.1.5. **Code organization issues:**

- Large programs may contain dozens or hundreds of classes, all dependent on each other to some extent. If one class requires the contents of another, it must use the `include` preprocessor directive to bring the other class definition into its header file.
- This can potentially cause two issues:
  - *multiple inclusions* of the same class definition inside two or more header files
  - *circular inclusions* between two classes that depend on each other
- Multiple inclusions are prevented with the use of include guards, which are discussed later in this section.
- Forward references are used to address the issue of circular inclusions, as we see in the discussion of Program-8.5, line 4.

3.2.2. Coding example: Code organization

```cpp
/* * * * * * * * * * * * * * * * * *
 * Filename:  Date.h                *
 * * * * * * * * * * * * * * * * * */
class Date
{
  public:
    Date();
    Date(int,int,int);
    void setDate(int,int,int);
    void print();
  private:
    int  day;
    int  month;
    int  year;
    int  lastDayInMonth(int, int);
    bool leapYear(int);
};

/* * * * * * * * * * * * * * * * * *
 * Filename:  Date.cc               *
 * * * * * * * * * * * * * * * * * */
#include <iostream>
#include <iomanip>
using namespace std;
#include "Date.h"

Date::Date()
{
  cout<<"in default constructor"<<endl;
  day = month = year = 0;
}

Date::Date(int d, int m, int y)
{
  cout<<"in three-parameter constructor"<<endl;
  setDate(d, m, y);
}

```

```
void Date::setDate(int d, int m, int y)
{
  year  = ( ( y > 0) ? y : 0 );
  month = ( ( m > 0 && m <= 12 ) ? m : 0 );
  day   = ( ( d > 0 && d <= lastDayInMonth(m, y) ) ? d : 0 );
}

void Date::print()
{
  cout << setfill('0') << setw(4) << year << "-"
       << setfill('0') << setw(2) << month << "-"
       << setfill('0') << setw(2) << day << endl;
}

int  Date::lastDayInMonth(int m, int y) {  /* not shown */  }
bool Date::leapYear(int y)              {  /* not shown */  }

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc             *
 * * * * * * * * * * * * * * * * */
#include "Date.h"
#define MAX_ARR_SIZE  3

int main()
{
  cout << "Declaring d1: ";
  Date d1(28, 5, 2012);

  cout<<"Declaring d2: ";
  Date d2;

  cout << endl << "Declaring dArray: " << endl;
  Date dArray[MAX_ARR_SIZE];

  cout << endl << "Printing d1: ";
  d1.print();

  cout << "Printing d2: ";
  d2.print();
  d2.setDate(3, 3, 1933);
  cout << "Printing d2: ";
  d2.print();

  dArray[0].setDate(1, 1, 1911);
  dArray[1].setDate(5, 5, 1955);

  cout << endl << "Printing dArray: " << endl;
  for (int i=0; i<MAX_ARR_SIZE; ++i) {
    dArray[i].print();
  }

  return 0;
}
```



```
Don't Panic ==> p2
Declaring d1: in three-parameter constructor
Declaring d2: in default constructor

Declaring dArray:
in default constructor
in default constructor
in default constructor

Printing d1: 2012-05-28
Printing d2: 0000-00-00
Printing d2: 1933-03-03

Printing dArray:
1911-01-01
1955-05-05
0000-00-00
Don't Panic ==> 
```

Program-3.2: Code organization

Program purpose:

- Program-3.2 defines a `Date` class with two constructors, a setter, a printing member function, and some helper member functions.
- The `main()` function declares several `Date` objects, including an array of dates, and calls the member functions on these objects.

Lines 4-17:

- These lines are contained within the class header file, and they show the `Date` class definition.
- The class contains three private data members, declared on lines 12-14: the date's day, month, and year.
- The public member functions include two constructors (lines 7-8), a setter (line 9), and a printing function (line 10).
- Private member functions are usually *helper* functions. These are only used within the class's other member functions and cannot be called from outside the class itself.
- Here, the class's private member functions are used for validating the data member values.

Lines 22-54:

- These lines are contained within the class source file, and they show the `Date` class member function implementations.
- By default, all functions in C++ are *global functions*. In order to define the functions in the class source file as *member functions* of the `Date` class, we must the **scope resolution operator** (`::`).
- The scope resolution operator takes the class name as the left-hand side operand and the class member as the right-hand side operand.
- For example:
  - on line 27, the function name `Date::Date()` declares that the default constructor `Date()` is a member function of the `Date` class
  - on line 39, the function name `Date::setDate()` declares that the `setDate()` function is a member of the `Date` class, which allows this function to access the private data members on lines 41-43; if we omitted the `Date::` portion, we would be declaring `setDate()` as a global function, which does *not* have any data members

- The `print()` member function on lines 46-51 uses the `iomanip` library to format the output width and filler character. The reader is encouraged to look into the tools available in this library to assist in producing professional-looking output.

Lines 59-91:

- These lines show the contents of the `main.cc` source file, including the `main()` function.
- Line 65 declares a `Date` object called `d1`, and it calls the three-parameter constructor on lines 33-37 with specific values.
- Line 68 declares another `Date` object called `d2`, and it calls the default constructor on lines 27-31.
- Line 71 declares an array called `dArray` that contains three `Date` objects. The default constructor is called automatically for each of the three `Date` objects when the array is declared.
- Line 74 shows the printing member function called using the `d1` object.
- Lines 77-80 show the setter and printing member functions called using the `d2` object.
- Lines 82-83 show the setter member function called using the first two elements of `dArray`.
- Lines 86-88 loop over `dArray` and call the printing member function using each element.

3.2.3. Best practices for code packaging

3.2.3.1. **Basic principles:**

- Very few professional developers write code directly for the end-users.
- Most professionals write code for *other programmers*. This is different from writing code for end-users, and it's a very important skill to develop.
- A ***class developer*** is a programmer who writes the code for a class. In the coding examples of this textbook, we assume that the class development team consists of the reader and the author working together as student and instructor.
- A ***class user*** is a programmer who uses our class in their code. For example, a class user may create instances of our class in their program, or they may develop new subclasses.

3.2.3.2. **Class interface:**

- In computing, an ***interface*** is a mechanism for interacting with a software system or part of a system. For example, an application programming interface (API) allows code to interact programmatically with an existing system.
- A ***class interface*** consists of the information that a class user requires in order to create and interact with objects of that class. This information includes:
  - the class name
  - its public members (both data members and member functions)
  - sometimes its protected members too
- Some OO languages use the term *interface* to denote a language-specific construct. In this textbook, we use *interface* in strict accordance with its correct meaning in general computing, as defined above.

NOTE: Java provides a language construct that it calls an *interface*. This feature is needed as a workaround to simulate multiple inheritance, which Java does not support. C++ can provide true multiple inheritance because of its scope resolution operator. The Java-specific usage of the term *interface* is **unrelated** to how the term is used in general computing and in this textbook.

3.2.3.3. **Class development:**

- A class user requires two things from a class developer:
  - the class definition, which is contained in the class header file, and
  - the class object code, which is compiled from the class source code
- The class user's source code must use the `include` C++ preprocessor directive to copy the class header file into their own code.
- Then, they must link the class object code with their own object code to create the program executable.
- The class user does *not* require the class source code.

3.2.3.4. **Class header file and include guards:**

- A class header file contains the class definition, including the data members and the member function prototypes.
- The header file must also use ***include guards***. These are very useful in preventing compilation errors due to the same class being redefined, when its header file is included two or more times. This can happen when classes are have circular dependencies on each other.
- Include guards use precompilation directives to make sure that each header file is only included once.
- The following is an example of an include guard for the `Date` class:
  - at the top of the class header file, we use the following two preprocessor directives:

    ```
    #ifndef DATE_H
    #define DATE_H
    ```
  - the `ifndef` directive ("if not defined") checks whether a constant called `DATE_H` has already been defined; if it has not, it means that the header file has not yet been included anywhere
  - if the constant does not yet exist, then the next line defines it, and the rest of the header file is included
  - if the constant does exist, it means that the header file has already been included once during the compilation process; in this case, the preprocessor skips to the end of the file and does *not* include the header file
  - at the bottom of the class header file, we must terminate the preprocessor `ifndef` directive with: `#endif`
- An alternative to include guards in C++ is the use of `#pragma once` at the top of a class header file. However, this directive is neither standard nor universally supported on all compilers. The use of include guards instead makes the code more portable to a wider variety of platforms.

3.2.3.5. **Class source file:**

- A class source file contains all the class's member function implementations.
- Remember! By default, all functions are global. The scope resolution operator must be used in the source file to define a function as belonging to a specific class.
- The source file also contains initialization statements for the static storage class data members, which we discuss in chapter 8.

3.2.3.6. **Program building with class files:**

- Figure-3.1 shows what files are shared by a class developer with the class user.
- The class user requires the class header file in order to access the class definition.
- Once the class user's source code is compiled, the corresponding object file is linked with the class object file to produce the program executable.
- The class developer does *not* provide the source code to the class user.
- Question: If the class source code changes, what is required for the class user to rebuild the program?
- Answer: Only an updated class object file must be provided by the class developer. The class user simply needs to re-link their existing object code with the new class object file. The class user does *not* need to recompile their code.
- Question: If the class header file changes, what is required for the class user to rebuild the program?
- Answer: Both an updated header file and class object file must be provided by the class developer. The class user must recompile their code with the new header file, then re-link their new object code with the updated class object file.
- We can see how changes to the class header file has a greater impact on the class user than simply changing the class source code. For this reason, modifications to class header files should be minimized if possible. Using correct design techniques and good software engineering practices can go a long way in planning for changes in advance and reducing their impact across the system.

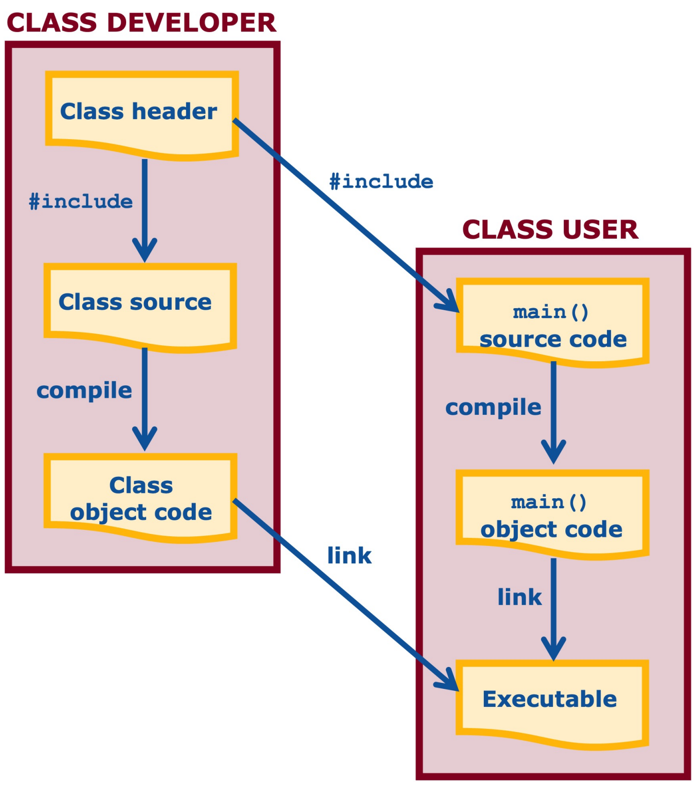


Figure-3.1: Program building with class files

3.3. Default arguments

We discuss the role of default arguments in C++ functions. These allow for greater flexibility in how global and member functions are called.

3.3.1. Concepts

3.3.1.1. What is a *default argument*:

- A *function argument* is a parameter value, from the point of view of a calling function:
  - a calling function sends arguments to a called function
  - the called function receives parameter values from the calling function
- A ***default argument*** is a predefined default value for a parameter. If the calling function sends *no value* for an argument, then the called function uses the default value instead.

3.3.1.2. Characteristics of default arguments:

- A default argument is specified where the function prototype first appears.
- One or more parameters may have default values.
- A function may have a mix of some parameters with default values, and others without.
- To eliminate ambiguity, all parameters with default values must be positioned as the right-most parameters in the function prototype, after the parameters without default values.

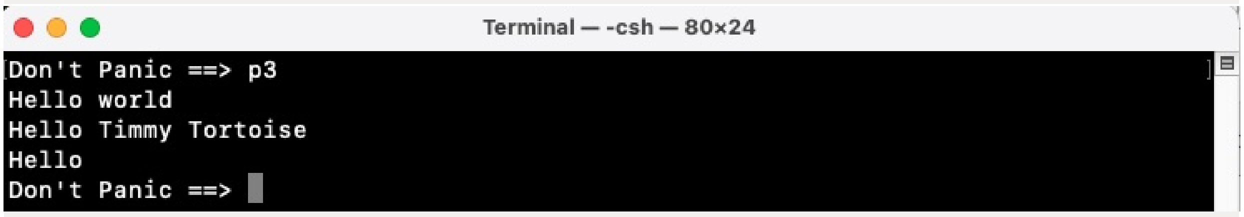
3.3.1.3. Uses of default arguments:

- Default arguments can be used in global functions or in member functions.
- A common usage is to *combine* a default constructor with a multiple-parameter one.

NOTE: A class may only declare **one** default constructor. Declaring default values for all the parameters of a multi-parameter constructor automatically makes it a default constructor.

3.3.2. Coding example: Default arguments

```
void sayHello(string = "Timmy Tortoise");

int main()
{
  sayHello("world");
  sayHello();
  sayHello("");
  return 0;
}
void sayHello(string who)
{
  cout << "Hello " << who << endl;
}
```



```
Terminal — -csh — 80×24
Don't Panic ==> p3
Hello world
Hello Timmy Tortoise
Hello
Don't Panic ==>
```

Program-3.3: Default arguments

Program purpose:

- Program-3.3 declares a default value for the parameter to the `sayHello()` global function. Then the `main()` function calls it in three different ways.

Line 1:

- This line declares `sayHello()` as a global function that takes a single `string` parameter. The function prototype establishes that the parameter has the default value `"Timmy Tortoise"`.

Lines 10-13:

- These lines show the implementation of the `sayHello()` function.
- Line 12 prints out `"Hello"`, followed by the parameter value.

Lines 3-9:

- These lines show the implementation of the `main()` function.
- Line 5 calls `sayHello()` with the parameter value `"world"`. As a result, `"Hello world"` is printed out, as we see from the program output.
- Line 6 calls `sayHello()` with *no parameter value*. If the function did not provide a default value for the parameter on line 1, then line 6 would not compile. But since it does, line 6 calls `sayHello()` with the string `"Timmy Tortoise"` as parameter. As a result, `"Hello Timmy Tortoise"` is printed out.
- Line 7 calls `sayHello()` with an empty string as parameter value. This line does *not* use the default parameter value, because there *is* a value provided. It just happens to be an empty string. We see from the program output that `"Hello "` is printed out.

3.4. Default constructors

Constructors have one basic purpose: to initialize every part of a new object. We know from programming in C that allocated variables are *not* automatically assigned default values at runtime. Instead, they contain whatever garbage was in those memory cells the last time they were used. To make sure that our programs don't crash due to invalid data, it is imperative that we initialize every variable and data member before they are used.

3.4.1. Concepts

3.4.1.1. What is a *default constructor*?

- A *default constructor* is a constructor that takes *no parameters*.
- Equivalently, it is also a constructor whose every parameter is a default argument. If the constructor is called with no parameters, then all the default values are used.

3.4.1.2. Characteristics of a default constructor:

- A default constructor is a member function of a class.
- Only *one* default constructor can exist for each class.
- An empty default constructor is provided automatically by the compiler, if the class developer does not implement any constructors for a class.
- The default constructor can be called *explicitly* or *implicitly*.

3.4.1.3. Explicit and implicit function calls:

- A function is called ***explicitly*** in C++ when the programmer uses syntax that clearly and deliberately calls the function.

- A function is called ***implicitly*** when the programmer uses syntax that causes the compiler to automatically *convert* that syntax into a call to the function. There are several types of syntax in C++ that trigger these automatic conversions to function calls, and they are discussed throughout this textbook.

3.4.1.4. **Uses of default constructors:**

- The job of any constructor is to *initialize all the data members of a new object*.
- A default constructor must initialize all the data members to default values.
- Remember, in many programming environments including most Unix-type systems, all variables contain *garbage* unless they are initialized by the programmer.

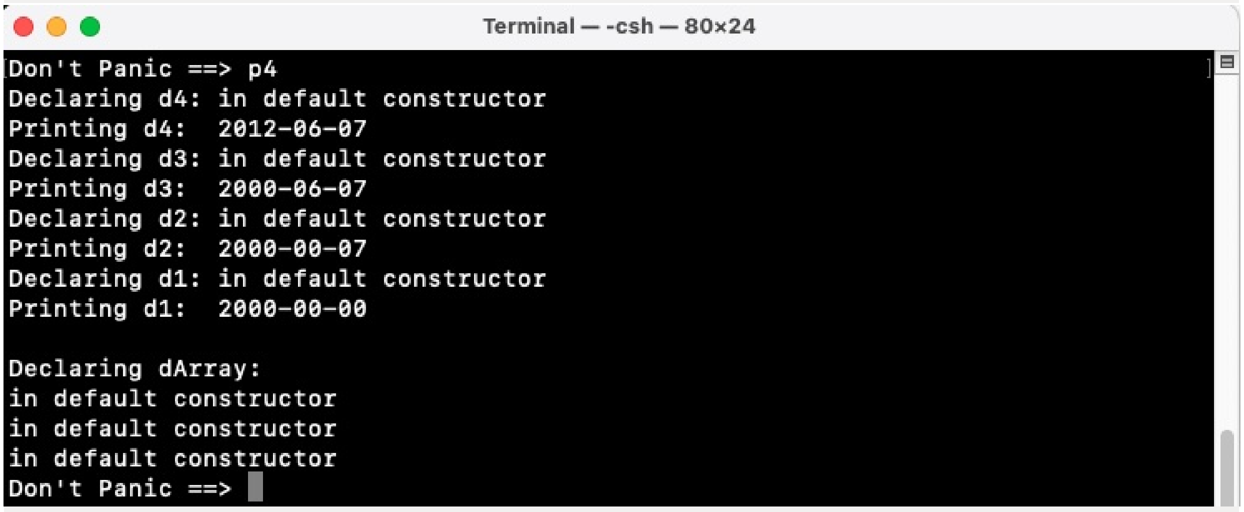
3.4.1.5. **The default constructor is called implicitly:**

- when an object is declared with no parameters specified
- when an array of objects is declared, and the default constructor is called automatically for every object in the array
- when memory for a new object is dynamically allocated; dynamic allocation in C++ is done using the `new` operator, which is discussed in chapter 4

3.4.2. Coding example: Default constructors

```
/* * * * * * * * * * * * * * * * * *
 * Filename:  Date.h               *
 * * * * * * * * * * * * * * * * * */
class Date
{
  public:
//    Date();
    Date(int=0, int=0, int=2000);
    void setDate(int, int, int);
    void print();

  private:
    int  day;
    int  month;
    int  year;
    int  lastDayInMonth(int, int);
    bool leapYear(int);
};

/* * * * * * * * * * * * * * * * * *
 * Filename:  Date.cc              *
 * * * * * * * * * * * * * * * * * */

/*  NO LONGER USED
Date::Date()
{
  cout<<"in default constructor"<<endl;
  day = month = year = 0;
}
*/

```

```
/*  This is now the default constructor  */
Date::Date(int d, int m, int y)
{
  cout<<"in default constructor"<<endl;
  setDate(d, m, y);
}
/*  The rest of the file is not shown.  */

/* * * * * * * * * * * * * * * * * *
 * Filename:  main.cc              *
 * * * * * * * * * * * * * * * * */
#include "Date.h"
#define MAX_ARR_SIZE  3

int main()
{
  cout << "Declaring d4: ";
  Date d4(7,6,2012);
  cout << "Printing d4:  ";
  d4.print();

  cout << "Declaring d3: ";
  Date d3(7,6);
  cout << "Printing d3:  ";
  d3.print();

  cout << "Declaring d2: ";
  Date d2(7);
  cout << "Printing d2:  ";
  d2.print();

  cout << "Declaring d1: ";
  Date d1;
  cout << "Printing d1:  ";
  d1.print();

  cout << endl << "Declaring dArray: " << endl;
  Date dArray[MAX_ARR_SIZE];
  return 0;
}
```



```
Don't Panic ==> p4
Declaring d4: in default constructor
Printing d4:  2012-06-07
Declaring d3: in default constructor
Printing d3:  2000-06-07
Declaring d2: in default constructor
Printing d2:  2000-00-07
Declaring d1: in default constructor
Printing d1:  2000-00-00

Declaring dArray:
in default constructor
in default constructor
in default constructor
Don't Panic ==> 
```

Program-3.4: Default constructors

Program purpose:

- Program-3.4 modifies the `Date` class from Program-3.2 by merging the two constructors into a single default constructor that defines default arguments.
- The `main()` function declares several `Date` objects, including an array of dates, and the program output shows when the default constructor is called.

Lines 7-8:

- In the `Date` class definition, the three-parameter constructor on line 8 has been modified to define default values for all three parameters, which now makes it the default constructor.
- The previous default constructor on line 7 must be removed or commented out, otherwise there would be two default constructors and compilation would fail.
- The default constructor takes as parameters the default values of zero for both day and month and 2000 for the year, as defined on line 8.

Lines 25-37:

- The implementation of the previous default constructor on lines 25-29 is commented out, as it is no longer used.
- The implementation of the three-parameter constructor on lines 33-37 is unchanged from Program-3.2, except for the debugging statement printed out.
- The default parameter values are *not* specified in the member function implementation, as we see on line 33.

Lines 46-71:

- These lines show the implementation of the `main()` function.
- Lines 49, 54, 59, and 64 together declare four `Date` objects called `d4`, `d3`, `d2`, and `d1`, respectively. Line 69 declares an array of three `Date` objects. Each `Date` object created on these lines triggers a call to the `Date` default constructor, with different parameters.
- On line 49, the constructor is called with three parameters for the day, month, and year. The `d4` object is initialized accordingly, as the 7th day of the 6th month, in the year 2012.
- On line 54, the constructor is called with two parameters for the day and month. This results in the default value being used for the right-most parameter, the year. The `d3` object is initialized accordingly, as the 7th day of the 6th month, in the year 2000.
- On line 59, the constructor is called with one parameter for the day. This results in default values being used for the two right-most parameters, the month and year. The `d2` object is initialized accordingly, as the 7th day of month zero, in the year 2000.
- On line 64, the constructor is called with no parameters. This results in default values being used for all three parameters. The `d1` object is initialized accordingly, as day zero of month zero, in the year 2000.
- On line 69, the constructor is called with no parameters for each of the three `Date` objects in the array. This results in default values being used for all three parameters for each object.

3.5. Destructors

We introduce the concept of destructors, which are a key component of C++ programming.

3.5.1. Concepts

3.5.1.1. What is a *destructor*?

- A *destructor* is the mirror image of a constructor. While a constructor initializes a newly created object, a destructor cleans up an object when it is destroyed.
- We note that the term "de-constructor" is incorrect in C++ and should *not* be used.

3.5.1.2. Characteristics of a destructor:

- A destructor is a member function of a class.
- Only *one* destructor can exist for each class, and it takes no parameters.
- An empty destructor is provided automatically by the compiler, if the class developer does not implement one for a class.
- The destructor is **never** called explicitly. It is always called implicitly, at specific points in the program which are discussed in this section.

3.5.1.3. Uses of destructors:

- The job of a destructor is to perform any necessary cleanup when an object is destroyed. It's the responsibility of the class developer to know what cleanup is required for an object and to implement the class destructor accordingly.
- If necessary, the destructor can release resources that are no longer needed after the object is destroyed. For example:
  - a destructor can be used to close any files that the object opened
  - it can be used to deallocate dynamically allocated memory that is contained within the object, for example if the object has a data member that's dynamically allocated
- If a class does not require any cleanup when its objects are destroyed, the empty destructor provided by the compiler is usually sufficient.

NOTE: There is no magic that performs object cleanup on behalf of the class developer. If cleanup is required, it's up to the developer to write the destructor code for their classes. It's generally considered bad practice to place most cleanup code outside the destructor, except where it would result in objects being cleaned up more than once.

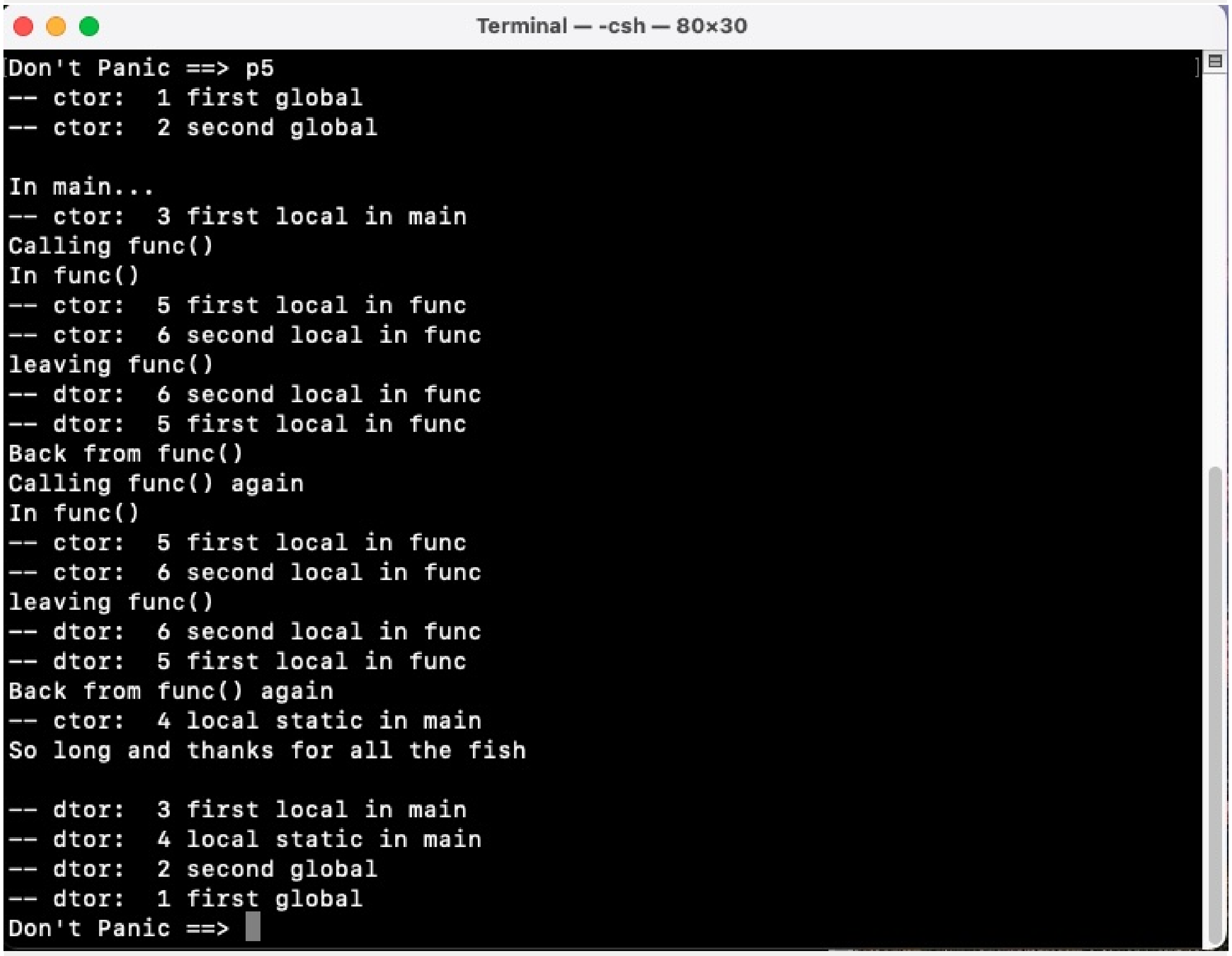
3.5.1.4. When is the destructor called implicitly:

- For block scope objects, the destructor is called automatically when the control flow exits the block where the object is declared.
- Tor global scope objects, the destructor is called when the control flow exits the program.
- For dynamically allocated objects, the destructor executes when the memory is explicitly deallocated with the `delete` operator, which is discussed in chapter 4.
- Destructors are usually called in the reverse order of constructor calls, but not always.

3.5.2. Coding example: Destructors

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Thing.h             *
 * * * * * * * * * * * * * * * * */
class Thing
{
  public:
    Thing(int=0, string="");
    ~Thing();
  private:
    int id;
    string location;
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Thing.cc            *
 * * * * * * * * * * * * * * * * */
Thing::Thing(int i, string s)
{
  id = i;
  location = s;
  cout<<"-- ctor:  "<< id <<" "<< location <<endl;
}

Thing::~Thing()
{
  cout<<"-- dtor:  "<< id <<" "<< location <<endl;
}

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc            *
 * * * * * * * * * * * * * * * * */
void func();

Thing t1(1,"first global");
Thing t2(2,"second global");

int main()
{
  cout<<endl<<"In main..."<<endl;
  Thing t3(3, "first local in main");

  cout<<"Calling func()"<<endl;
  func();
  cout<<"Back from func()"<<endl;

  cout<<"Calling func() again"<<endl;
  func();
  cout<<"Back from func() again"<<endl;

  static Thing t4(4, "local static in main");

  cout<<"So long and thanks for all the fish"<<endl<<endl;
  return 0;
}

```

```cpp
void func()
{
  cout<<"In func()"<<endl;

  Thing t5(5, "first local in func");
  Thing t6(6, "second local in func");

  cout<<"leaving func()"<<endl;
}
```



```
Don't Panic ==> p5
-- ctor:  1 first global
-- ctor:  2 second global

In main...
-- ctor:  3 first local in main
Calling func()
In func()
-- ctor:  5 first local in func
-- ctor:  6 second local in func
leaving func()
-- dtor:  6 second local in func
-- dtor:  5 first local in func
Back from func()
Calling func() again
In func()
-- ctor:  5 first local in func
-- ctor:  6 second local in func
leaving func()
-- dtor:  6 second local in func
-- dtor:  5 first local in func
Back from func() again
-- ctor:  4 local static in main
So long and thanks for all the fish

-- dtor:  3 first local in main
-- dtor:  4 local static in main
-- dtor:  2 second global
-- dtor:  1 first global
Don't Panic ==> 
```

Program-3.5: Destructors

Program purpose:

- Program-3.5 creates multiple instances of the `Thing` class at different scopes (global and local) in the program, using different storage classes.
- The program output shows when each object is created and destroyed, based on the debugging statements in the `Thing` class constructor and destructor.

Lines 4-12:

- These lines show the `Thing` class definition, which does not represent a real-life concept. Its purpose is to show the creation and destruction of objects, based on their scope and their storage class.
- The data members declared on lines 10-11 represent a unique number for the object and a string to show where the object is declared in the program.

Lines 17-27:

- These lines implement the `Thing` class's default constructor and its destructor, which both print out the contents of the data members.

Lines 34-35:

- In the `main.cc` source file, these lines declare objects `t1` and `t2` at global scope. Their constructors are called in the order in which the objects are declared, before the `main()` function begins execution.

Line 40:

- This line declares object `t3`, which has block scope within the `main()` function. Its constructor executes on line 40.

Line 43 and 56-64:

- Line 43 calls the `func()` function for the first time.
- Lines 56-64 show the implementation of `func()`.
- Lines 60-61 declare two objects, `t5` and `t6`, which both have block scope within `func()`. The constructor for each object executes on the line where it is declared.
- On line 64, the `func()` function terminates, and the control flow returns to `main()`.
- Because line 64 represents the end of the `func()` block, the objects declared locally within its scope are destroyed. The destructors for `t5` and `t6` are called at this time, in the reverse order of construction.

Line 47 and 56-64:

- Line 47 calls the `func()` function for the second time.
- Lines 60-61 declare two objects, `t5` and `t6`, *once again*. The constructor for each object executes on the line where it is declared.
- On line 64, the `func()` function terminates, and its locally declared objects are destroyed. The destructors for `t5` and `t6` are called at this time, in the reverse order of construction.

Line 50:

- This line declares object `t4`, which has block scope within the `main()` function.
- However, `t4` has *static* storage class, instead of the default automatic storage class. While this has no effect on when the object constructor is called, it does affect its destruction.
- Remember, static storage class variables are stored in global memory, in the data segment, and not in the function call stack. Because of this, objects with static storage class are *not* destroyed when the control flow exits the scope where they are declared.
- The `t4` constructor executes on line 40, where it is declared.

Line 53:

- On this line, the `main()` function terminates and returns control to the OS. At this point, all the objects still remaining in memory are destroyed.
- First, the locally declared objects are destroyed, starting with the automatic storage class object (`t3`), followed by the static storage class object (`t4`).
- Once the local objects have been cleaned up, the ones with global scope (`t1` and `t2`) are destroyed, in the reverse order of construction.
- We note that the destructors for these objects are called automatically on line 53, after the last printing statement in `main()`, on line 52.

3.5.3. Order of execution

3.5.3.1. Explicit calls to a destructor:

- Destructors should **never** be called explicitly by the programmer.
- Although some compilers do allow for explicit calls to be made, destructors are **always** called implicitly. So an explicit call to a destructor *cannot* prevent an implicit call from taking place at the predetermined time, based on object scope and storage class.
- As a result, if an explicit call is made, it may result in the destructor being called twice, possibly resulting in a segmentation fault if the same memory is deallocated multiple times.

3.5.3.2. Global scope objects:

- For global scope objects, the constructor is called when the object is declared, before the `main()` function executes, in the order of declaration.
- The destructor is called after `main()` terminates, or when the `exit()` system call is invoked, in the reverse order of construction.

3.5.3.3. Local (block) scope objects:

- For local scope objects, the constructor is called when the object is declared, for example in a function, or a loop, or any block.
- The destructor is called when the control flow exits the block where the object is declared.
- If a function is called multiple items and declares local objects, then different objects are created on every call. They are destroyed when the control flow returns to the calling function.

3.5.3.4. In case of abnormal program termination:

- If the program terminates because of a system call to `exit()`:
  - this indicates a controlled failure and causes program termination
  - the destructors for the global scope objects *do* execute
  - the destructors for the local scope objects *do not*
- If the program terminates because of a system call to `abort()`:
  - this indicates a non-recoverable failure and causes immediate program termination
  - *no destructors* execute

3.6. Object duplication

Before we move on to copy constructors, we must discuss some of the concepts related to object duplication in a program. We clarify the distinction between an object variable declaration and initialization, and we introduce terminology around the different ways to make copies of objects.

3.6.1. Declaring objects

3.6.1.1. What is an object *declaration*:

- An object ***declaration*** allocates memory for an object and calls its class constructor.
- For example:
  - given the object declaration: `Student matilda;`
  - this declares a new variable called `matilda` as an object of the `Student` class, and it calls the `Student` class default constructor

3.6.1.2. **What is an object *initialization*:**

- An object ***initialization*** declares an object variable and initializes it using the values from an existing object.
- Because the end goal of an initialization is to copy an existing object into a brand new one, an initialization triggers a call to the copy constructor, as we discuss in the next section.
- For example:
  - given the object initialization: `Student bertha = matilda;`
  - the compiler converts this initialization to its two parts: (1) it declares a new variable called `bertha` as an object of the `Student` class, and (2) it calls the `Student` copy constructor for object `bertha`, with `matilda` as parameter

3.6.1.3. **What is an object *assignment*:**

- An object *assignment* copies one existing object into another existing object. Because an assignment does *not* declare a new variable, no constructors are called.
- For example:
  - given the object assignment: `matilda = bertha;`
  - this copies the data members of the existing object `bertha` into the existing object `matilda`
  - it's important to remember that any data previously stored in the `matilda` object are overwritten by an assignment operation
  - the assignment operation copies data members from one object to another, like the copy constructor does, but it's considered a *different operation* because no new object is created

NOTE: In C++, the equal symbol (=) in an initialization is **not** an assignment operation. It is a construction operation. The two have different meanings, and two different member functions are called.

3.6.2. Depth of object duplication

3.6.2.1. **Depths of copies**

- A key concept when duplicating objects is how deeply the copying should go.
- A *deep copy* duplicates an object to its full depth, whereas a *shallow copy* does not.
- It's important to note that there is no universally good or bad way to create copies. In some situations, a deep copy is required, and in other cases, a shallow copy makes more sense.

3.6.2.2. **What is a *deep copy*:**

- A ***deep copy*** of an object, or a pointer to an object, creates a full duplicate, including copies of all its data members and containee objects, and their containee objects, and so on, all the way to the deepest containee object.
- At every level, if a data member is a primitive data type, then its value is copied.
- If a data member is an object, then that object and all its containees are fully duplicated.
- If the data member is a pointer, then a duplicate of the pointee and all its containees must be allocated and initialized.

3.6.2.3. What is a *shallow copy*:

- A ***shallow copy*** creates a duplicate of an object, only based on the data type of its members.
- If a data member is a primitive data type, then its value is copied.
- If a data member is an object, then that object is copied.
- If the data member is a pointer, then only the pointer itself is copied, and *not* the pointee. This results in two pointers with the same pointee.

NOTE: Shallow copies create havoc when cleaning up objects that have dynamically allocated containees. If there are multiple pointers to a containee, the code must ensure that **only one** container object deallocates that containee. If a container destructor cleans up a containee that is still in use, then all the other pointers to that same containee will be pointing to memory that is no longer allocated. These are called *dangling references*, and they usually result in a segmentation fault and program crash.

3.6.2.4. What is *member-wise assignment*:

- Another term for the shallow copy of an object is ***member-wise assignment***.
- If a class developer does not implement the class's copy constructor or its overloaded assignment operator, the compiler automatically provides an implementation that performs member-wise assignment.
- If the class developer requires a deeper copy for those operations, they must provide their own implementations. Copy constructors are discussed in the next section, and overloaded operators in chapter 12.

3.7. Copy constructors

Copy constructors are a special type of constructor that can be called explicitly, but can also be called implicitly in unexpected ways. We discuss the characteristics and uses of copy constructors and the syntax that can lead to their implicit call.

3.7.1. Concepts

3.7.1.1. Constructors with one parameter

- A constructor that takes *one parameter* is special.
- It is considered by the compiler to be either a *copy constructor* or a *conversion constructor*.
- Conversion constructors are discussed in the next section.

3.7.1.2. What is a *copy constructor*?

- A *copy constructor* is a constructor that takes one parameter that is a reference to an object of the same class.
- A copy constructor cannot take as parameter an object of the same class passed by value, or passed as a pointer.
- Only **one** copy constructor can exist for each class.

3.7.1.3. **Characteristics of a copy constructor:**

- A copy constructor is a member function of a class.
- It takes one parameter: a reference to an object of the same class.
- A copy constructor is provided automatically by the compiler, if the class developer does not implement one:
  - the compiler-provided copy constructor is **not** an empty function, like the provided default constructor and the destructor
  - the provided copy constructor performs member-wise assignment
  - in either a deep copy or a shallow copy of an object, the copy constructor of each of its containee objects is called automatically
- A copy constructor may be called either explicitly or implicitly, depending on the syntax used.

NOTE: It is critical that we understand what syntax triggers a call to the copy constructor, so that we understand why our program behaves the way it does.

3.7.1.4. **Uses of a copy constructor:**

- The job of any constructor is to initialize all the data members of a new object.
- The purpose of a copy constructor is to initialize a new object based on the contents of an existing object of the same class.
- It is used for constructing a new object as a copy of an existing one that's passed in as parameter.

NOTE: Remember that objects have access to *all* the members of another object of the *same class*, even the private ones. When implementing a copy constructor, it is **not** necessary to implement getters to access the members of the parameter object. The copy constructor *already* has direct access to all the members.

3.7.1.5. **Explicit calls to the copy constructor:**

- A copy constructor can be called explicitly, with an existing object of the same class as a constructor parameter.
- For example: `Student bertha(matilda);`

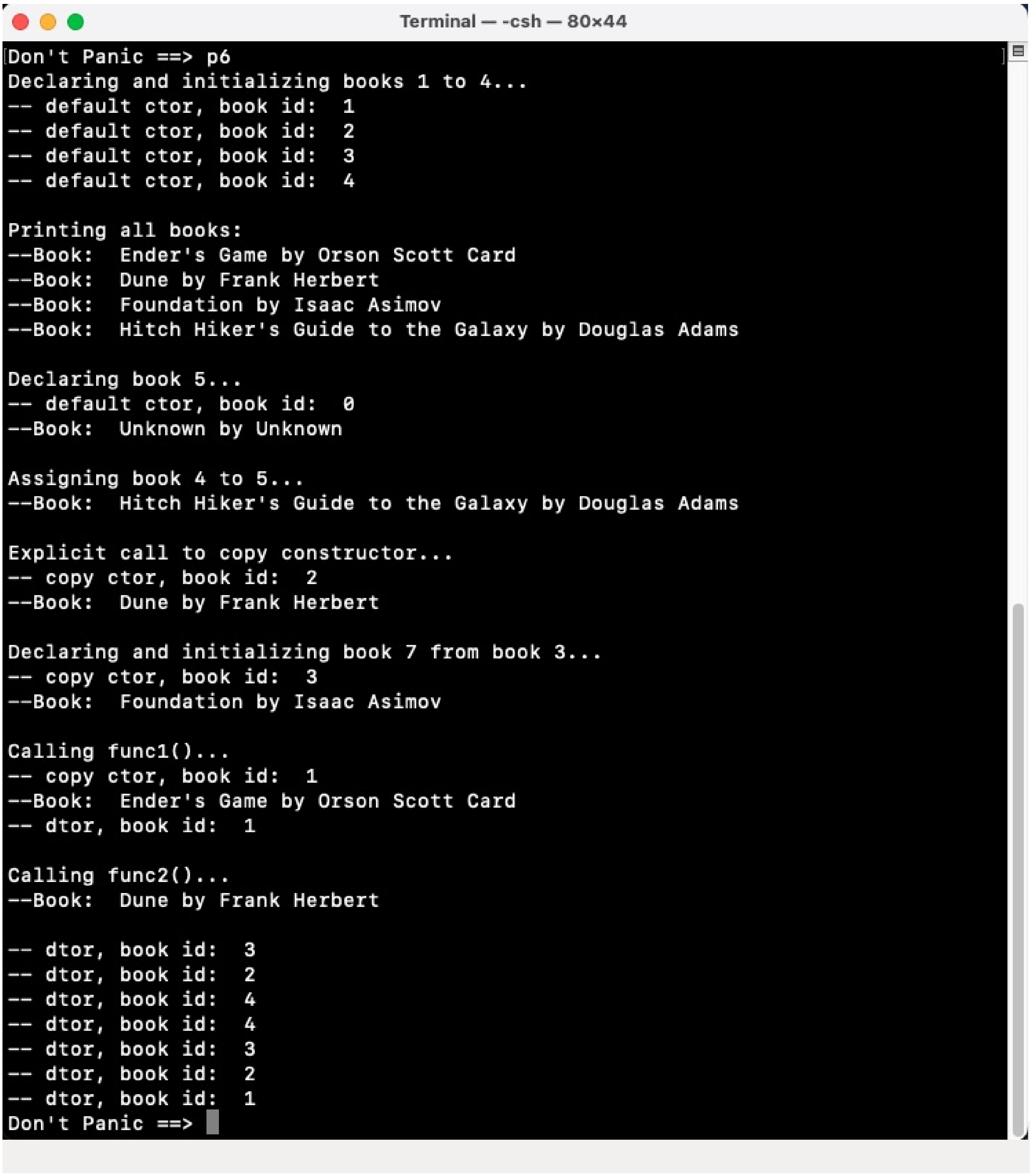
3.7.1.6. **Implicit calls to the copy constructor:**

- The copy constructor is called implicitly on initialization:
  - for example: `Student bertha = matilda;`
- It is also called implicitly when an object is *passed by value* as a function parameter.
- In older compilers, the copy constructor could be called implicitly if an object was returned by value. Newer compilers appear to be better optimized and no longer do this.
- However, we must keep in mind that objects should never be returned or passed as parameter by value, but always by reference instead.

3.7.2. Coding example: Copy constructors

```
/* * * * * * * * * * * * * * * * * *
 * Filename:  Book.h               *
 * * * * * * * * * * * * * * * * */
class Book
{
  public:
    Book(int=0, string="Unknown", string="Unknown");
    Book(const Book&);
    ~Book();
    void print();
  private:
    int id;
    string title;
    string author;
};

/* * * * * * * * * * * * * * * * * *
 * Filename:  Book.cc              *
 * * * * * * * * * * * * * * * * */
Book::Book(int i, string t, string a)
{
  id     = i;
  title  = t;
  author = a;
  cout<<"-- default ctor, book id:  "<< id <<endl;
}

Book::Book(const Book& oldBook)
{
  id     = oldBook.id;
  title  = oldBook.title;
  author = oldBook.author;
  cout<<"-- copy ctor, book id:  "<< id <<endl;
}

Book::~Book()
{
  cout<<"-- dtor, book id:  "<< id <<endl;
}

void Book::print()
{
  cout<<"--Book:  "<< title <<" by "<<author<<endl;
}

```

```cpp
/* * * * * * * * * * * * * * * * * *
 * Filename:  main.cc               *
 * * * * * * * * * * * * * * * * * */
void func1(Book);
void func2(Book&);

int main()
{
  cout<<"Declaring and initializing books 1 to 4..."<<endl;

  Book b1(1, "Ender's Game", "Orson Scott Card");
  Book b2(2, "Dune", "Frank Herbert");
  Book b3(3, "Foundation", "Isaac Asimov");
  Book b4(4, "Hitch Hiker's Guide to the Galaxy", "Douglas Adams");

  cout << endl << "Printing all books:" << endl;
  b1.print();
  b2.print();
  b3.print();
  b4.print();

  cout<<endl<<"Declaring book 5..."<<endl;
  Book b5;
  b5.print();

  cout<<endl<<"Assigning book 4 to 5..."<<endl;
  b5 = b4;
  b5.print();

  cout<<endl<<"Explicit call to copy constructor..."<<endl;
  Book b6(b2);
  b6.print();

  cout<<endl<<"Declaring and initializing book 7 from book 3..."<<endl;
  Book b7 = b3;
  b7.print();

  cout<<endl<<"Calling func1()..."<<endl;
  func1(b1);

  cout<<endl<<"Calling func2()..."<<endl;
  func2(b2);

  cout << endl;
  return 0;
}

void func1(Book b)
{
  b.print();
}

void func2(Book& b)
{
  b.print();
}
```



```
Don't Panic ==> p6
Declaring and initializing books 1 to 4...
-- default ctor, book id:  1
-- default ctor, book id:  2
-- default ctor, book id:  3
-- default ctor, book id:  4

Printing all books:
--Book:  Ender's Game by Orson Scott Card
--Book:  Dune by Frank Herbert
--Book:  Foundation by Isaac Asimov
--Book:  Hitch Hiker's Guide to the Galaxy by Douglas Adams

Declaring book 5...
-- default ctor, book id:  0
--Book:  Unknown by Unknown

Assigning book 4 to 5...
--Book:  Hitch Hiker's Guide to the Galaxy by Douglas Adams

Explicit call to copy constructor...
-- copy ctor, book id:  2
--Book:  Dune by Frank Herbert

Declaring and initializing book 7 from book 3...
-- copy ctor, book id:  3
--Book:  Foundation by Isaac Asimov

Calling func1()...
-- copy ctor, book id:  1
--Book:  Ender's Game by Orson Scott Card
-- dtor, book id:  1

Calling func2()...
--Book:  Dune by Frank Herbert

-- dtor, book id:  3
-- dtor, book id:  2
-- dtor, book id:  4
-- dtor, book id:  4
-- dtor, book id:  3
-- dtor, book id:  2
-- dtor, book id:  1
Don't Panic ==> 
```

Program-3.6: Copy constructors

Program purpose:

- Program-3.6 demonstrates how the copy constructor is called both explicitly and implicitly.
- The program output shows when an object is created using the default constructor, when one is created with the copy constructor, and when one is destroyed, based on the printed debugging statements.

Lines 4-15:

- These lines show the `Book` class definition.
- Lines 12-14 declare three data members: the book id, title, and author.
- Line 8 shows the copy constructor prototype inside the class definition.

Lines 28-34:

- These lines show the implementation of the `Book` class's copy constructor.
- On lines 30-32, the data members of the newly created book are assigned the values from the existing `oldBook` object that's passed in as parameter.
- Because the copy constructor is a member function of the `Book` class, it is permitted to access all the members of another `Book` object like `oldBook`. No getters are necessary.

Lines 56-59:

- These lines declare four `Book` objects as local variables in the `main()` function, and they call the default constructor with different values for each one.

Line 68:

- This line demonstrates a simple object declaration, with a call to the default constructor.

Line 72:

- This line shows an assignment operation, where the values of object `b4` are copied into `b5`.
- No constructor is called on this line.

Line 76:

- This line demonstrates an *explicit* call to the copy constructor.
- A new `Book` object `b6` is allocated and initialized with the values from existing object `b2`.

Line 80:

- This line contains an initialization, which triggers an *implicit* call to the copy constructor.
- A new `Book` object `b7` is allocated and initialized with the values from existing object `b3`.

Lines 84 and 93-96:

- Line 84 demonstrates a call to function `func1()`, with object `b1` as parameter.
- The function prototype on line 93 shows that the parameter is a `Book` object, and not a reference or a pointer, so the object is passed *by value*.
- Because the parameter object is passed by value, line 84 triggers an implicit call to the copy constructor. The parameter object `b` is allocated and initialized using the values from existing object `b1`.
- The program output shows that parameter object `b` is destroyed when `func1()` returns, since the parameter has block scope within the function.

Lines 87 and 98-101:

- Line 87 demonstrates a call to function `func2()`, with object `b2` as parameter.
- The function prototype on line 98 shows that the parameter is a `Book` reference, and not an object, so the object is passed *by reference*.
- Because the parameter object is passed by reference, line 87 does *not* result in the creation of a new object, so no constructors are called. As a `Book` reference, the identifier `b` becomes an alias for the existing object `b2`.

3.8. Conversion constructors

Like copy constructors, conversion constructors are one-parameter constructors that can be called either explicitly or implicitly. We discuss the characteristics and uses of conversion constructors and the syntax that can lead to their implicit call.

3.8.1. Concepts

3.8.1.1. **What is a *conversion constructor*?**

- A *conversion constructor* is a constructor that takes one parameter that is **not** a reference to an object of the same class.
- Any number of conversion constructors can exist for each class, as long as their parameter are of different data types.

3.8.1.2. **Characteristics of a conversion constructor:**

- A conversion constructor is a member function of a class.
- It takes one parameter, of a data type that is *not* a reference to an object of the same class.
- It may be called either explicitly or implicitly, depending on the syntax used.

3.8.1.3. **Uses of conversion constructors:**

- The job of any constructor is to initialize all the data members of a new object.
- The purpose of a conversion constructor is to initialize a new object based on the contents of an existing variable of a *different* data type. This data type could be any primitive or aggregate type, including an object.
- Its usage is highly dependent on the nature of both the class and the parameter data type. Not every class can be (or should be) instantiated using data from a different data type.

3.8.1.4. **Issues with conversion constructors:**

- Unfortunately, many simple syntax errors can accidentally trigger an implicit call to a conversion constructor. For this reason, conversion constructors are considered risky.
- The `explicit` keyword should be used when defining a conversion constructor in its class definition. This keyword *disables all implicit calls* to the conversion constructor and allows only explicit calls.

3.8.1.5. **Explicit calls to the conversion constructor:**

- A conversion constructor can be called explicitly, with an existing variable of a different data type as a constructor parameter.
- For example: `Book b1; Movie m1(b1);`

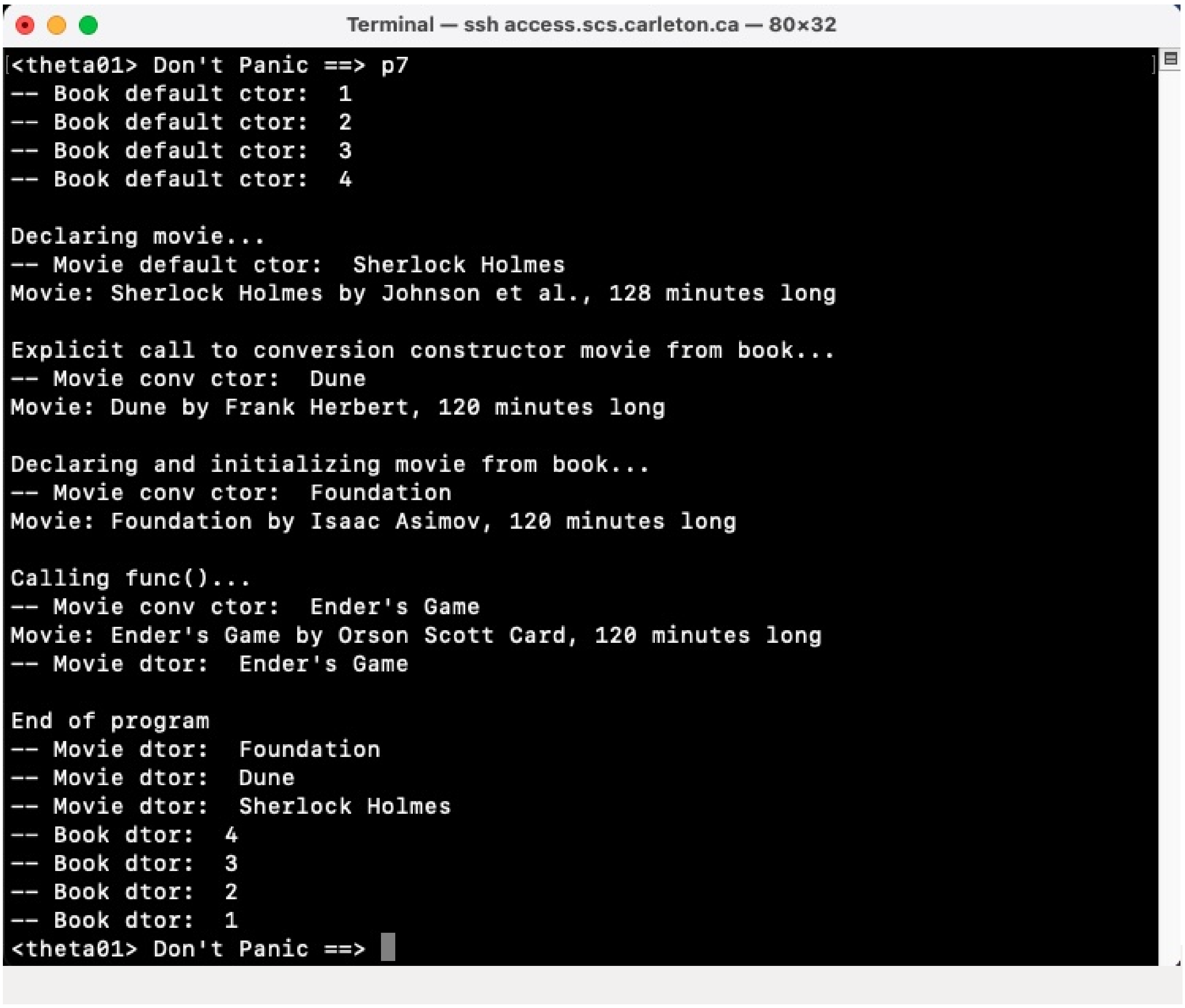
3.8.1.6. **Implicit calls to the conversion constructor:**

- A conversion constructor is called implicitly when the syntax expects an instance of the class, but instead is given a variable of a data type matching the conversion constructor parameter.
- For example, the conversion constructor can be called implicitly on initialization:
  - for example: `Book b1; Movie m1 = b1;`
- It is also called implicitly when a function expects an object of the class as a parameter, but the calling function instead uses a variable of a data type matching the conversion constructor parameter.

3.8.2. Coding example: Conversion constructors

```cpp
/* * * * * * * * * * * * * * * * * *
 * Filename:  Movie.h               *
 * * * * * * * * * * * * * * * * * */
class Movie
{
  public:
    Movie(string="", string="", int=0);
//    explicit Movie(Book&);
    Movie(Book&);
    ~Movie();
    void print();
  private:
    string title;
    string screenwriter;
    int    duration;
};

/* * * * * * * * * * * * * * * * * *
 * Filename:  Movie.cc              *
 * * * * * * * * * * * * * * * * * */
Movie::Movie(string t, string s, int d)
{
  title        = t;
  screenwriter = s;
  duration     = d;
  cout<<"-- default Movie ctor:  "<< title <<endl;
}

Movie::Movie(Book& b)
{
  title        = b.getTitle();
  screenwriter = b.getAuthor();
  duration     = 120;
  cout<<"-- conversion Movie ctor:  "<< title <<endl;
}

Movie::~Movie()
{
  cout<<"-- Movie dtor:  "<< title <<endl;
}

void Movie::print()
{
  cout<<"-- Movie: "<< title <<" lasting "<< duration<< " minutes"<<endl;
}

```

```
/* * * * * * * * * * * * * * * * * *
 * Filename:  main.cc               *
 * * * * * * * * * * * * * * * * * */
void func(Movie);

int main()
{
  Book b1(1, "Ender's Game", "Orson Scott Card");
  Book b2(2, "Dune", "Frank Herbert");
  Book b3(3, "Foundation", "Isaac Asimov");
  Book b4(4, "Hitch Hiker's Guide to the Galaxy", "Douglas Adams");

  cout<<endl<<"Declaring movie..."<<endl;
  Movie m1("Sherlock Holmes", "Johnson et al.", 128);
  m1.print();

  cout<<endl<<"Explicit call to conversion constructor movie from book..."<<endl;
  Movie m2(b2);
  m2.print();

  cout<<endl<<"Declaring and initializing movie from book..."<<endl;
  Movie m3 = b3;
  m3.print();

  cout<<endl<<"Calling func()..."<<endl;
  func(b1);

  cout << endl << "End of program" << endl;
  return 0;
}

void func(Movie m)
{
  m.print();
}
```



```
<theta01> Don't Panic ==> p7
-- Book default ctor:  1
-- Book default ctor:  2
-- Book default ctor:  3
-- Book default ctor:  4

Declaring movie...
-- Movie default ctor:  Sherlock Holmes
Movie: Sherlock Holmes by Johnson et al., 128 minutes long

Explicit call to conversion constructor movie from book...
-- Movie conv ctor:  Dune
Movie: Dune by Frank Herbert, 120 minutes long

Declaring and initializing movie from book...
-- Movie conv ctor:  Foundation
Movie: Foundation by Isaac Asimov, 120 minutes long

Calling func()...
-- Movie conv ctor:  Ender's Game
Movie: Ender's Game by Orson Scott Card, 120 minutes long
-- Movie dtor:  Ender's Game

End of program
-- Movie dtor:  Foundation
-- Movie dtor:  Dune
-- Movie dtor:  Sherlock Holmes
-- Book dtor:  4
-- Book dtor:  3
-- Book dtor:  2
-- Book dtor:  1
<theta01> Don't Panic ==> 
```

Program-3.7: Conversion constructors

Program purpose:

- Program-3.7 demonstrates how a conversion constructor is called both explicitly and implicitly to create a `Movie` object from an existing `Book` object.
- The `Book` class definition and implementation used in this program is identical to Program-3.6, with getter member functions for the data members.
- The program output shows when an object is created using the default constructor, when one is created with the conversion constructor, and when one is destroyed, based on the printed debugging statements.

Lines 4-16:

- These lines show the `Movie` class definition.
- Lines 13-15 declare three data members: a movie title, screenwriter, and duration in minutes.
- Lines 8-9 show two alternative versions of the conversion constructor prototype, one that allows *only* explicit calls, and one that allows both explicit and implicit calls. Both versions take a `Book` reference as parameter, allowing for a new `Movie` object to be initialized based on an existing `Book` object.

- Line 8 is commented out but shows the `explicit` keyword before the conversion constructor prototype. If this line was uncommented, the implicit calls to this constructor would be disallowed and would generate a compilation error. Only the explicit calls would be allowed.
- Line 9 shows the conversion constructor prototype that allows for both explicit and implicit conversions.

Lines 29-35:

- These lines show the implementation of the conversion constructor that initializes a new `Movie` object from an existing `Book` object.
- The data members of the newly created `Movie` object are assigned values from the existing `Book` object on lines 31-33.
- Because the two are different classes, getter member functions are required in the `Book` class for the `Movie` class to access them.

Lines 52-76:

- These lines show the implementation of the `main()` function.
- Line 60 demonstrates a simple `Movie` object declaration, with a call to the default constructor.
- Line 64 shows an explicit call to the conversion constructor, where a new `Movie` object `m2` is allocated and initialized with the values from the existing `Book` object `b2`.
- Line 68 shows an initialization, where a new `Movie` object `m3` is allocated and initialized with the values from an existing `Book` object `b3`. This line triggers an implicit call to the conversion constructor, as we see from the program output.
- Line 72 demonstrates a call to function `func()` implemented on lines 78-81, with `Book` object `b1` as parameter. The function prototype on line 78 shows that the parameter to `func()` is a `Movie` object.
- Because `func()` is expecting a `Movie` object as parameter, but line 72 is providing a `Book` object instead, this triggers an implicit call to the conversion constructor, as we see from the program output. The `Movie` parameter object `m` is allocated and initialized with the values from the existing `Book` object `b1`.

Chapter 4

Memory Management

Unlike programming languages where memory management choices are predetermined, C/C++ allow the programmer full flexibility in deciding how and where to store data.

On one hand, this requires the programmer to possess a deeper understanding of the trade-offs between the different memory management options. On the other hand, it places crucial memory-related design decisions firmly in the hands of those with the best knowledge of program functionality, the programmers.

4.1. Principles

We review the basic principles of virtual memory and the role of the function call stack and heap in memory management.

4.1.1. Virtual memory

4.1.1.1. **What is *virtual memory*:**

- *Virtual memory* is the main memory that the OS allocates to a program when it launches, for the duration of the program's execution.
- It is called *virtual* because it does *not* map to a contiguous area of physical memory. Virtual memory does appear to be contiguous, but it usually corresponds to multiple separate areas of physical memory. The OS is responsible for the runtime mapping of virtual to physical memory.
- When a program begins execution, the OS assigns it four areas of virtual memory:
  - the code segment (also known as the text segment)
  - the data segment
  - the function call stack
  - the heap, which is part of the data segment but is so important that we discuss it separately

4.1.1.2. **Code segment:**

- The code segment contains the individual program instructions.
- Every instruction has its own unique memory address in the code segment.
- The program control flow usually executes instructions sequentially, except in case of jumps.
- Jump instructions transfer control to an instruction at a specific memory address, for example in the implementation of a loop, or a function call or return.

4.1.1.3. **Data segment:**

- The data segment stores variables with global scope, variables with static storage class, and literals.
- It also includes the heap, which is discussed below.

4.1.1.4. **Function call stack:**

- The function call stack stores variables with local scope and automatic storage class.
- But its primary purpose is the management of the function-call-and-return mechanism.

4.1.1.5. **Heap:**

- The heap stores the *dynamically allocated memory*, which is memory that is allocated by a program at runtime.

4.1.2. Function call stack and heap

4.1.2.1. **What is the *function call stack*:**

- The ***function call stack*** manages the function-call-and-return mechanism during program execution.
- Like all stacks in computing, it is a data structure that stores elements in a last-in-first-out (LIFO) manner. Elements are *pushed* onto the top of a stack when added, and *popped* from the top when removed.
- When a function is called, a *stack frame* is created specifically for that function, and it is pushed onto the call stack. When the function returns, its stack frame is popped off.
- The *top* of the call stack, or its latest element, always corresponds to the function currently executing.
- Example: Let's assume that the `main()` function calls function `foo()`, which then calls `bar()`, which is currently executing. In this scenario, the bottom stack frame belongs to `main()`, the middle frame to `foo()`, and the top one to `bar()`. When `bar()` returns, its stack frame is popped off the call stack. Control flow returns to `foo()`, and its stack frame is now at the top of the call stack.

4.1.2.2. **Characteristics of a stack frame:**

- A stack frame belongs to a specific function that has been called and has not yet returned.
- It contain all of the function's local automatic variables, including the function parameters.
- It also contains the memory address of the next instruction in the calling function where the control flow must return when the function terminates.

4.1.2.3. **What is the *heap*:**

- The ***heap*** is the area of program memory used for *dynamic allocation*.
- Types of memory allocation:
  - *static memory allocation* occurs at compile time by simply declaring a variable
  - *dynamic memory allocation* is performed at runtime by using the `new` operator in C++
- Dynamic memory allocation is discussed later in this chapter.

4.2. Pointers

Pointers are a crucial tool for a programmer to fully control how the program memory is used.

4.2.1. Concepts

4.2.1.1. **What is a *pointer*:**

- A ***pointer*** is a variable that stores a memory address as its value.
- A pointer variable's value is the address of the first byte of either:
  - another existing variable, or
  - a block of dynamically allocated memory
- The variable or block that a pointer points to is called the ***pointee***.
- A pointee can be of any data type or any size.

4.2.1.2. **Uses of pointers:**

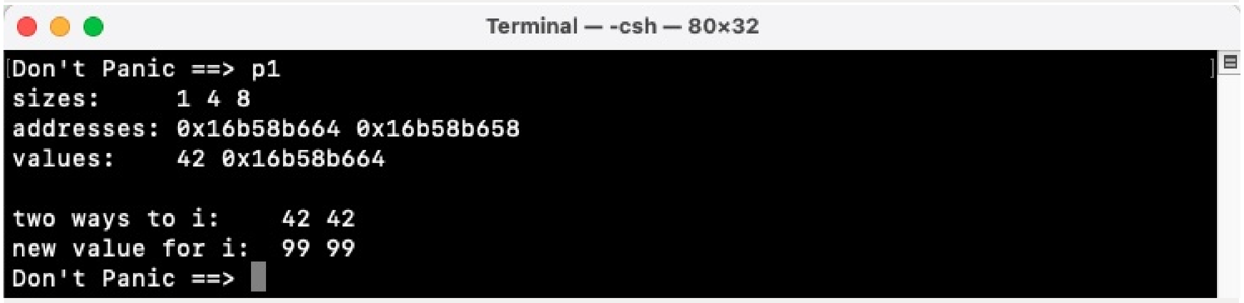
- Pointers promote computational and memory efficiency:
  - whatever the size of a pointee, a pointer to it always occupies a fixed number of bytes
  - because of their fixed size, pointers usually occupy less memory than their pointee
  - this is important in preventing the unnecessary copying of data
- Pointers allow a function to make changes to data that is stored in memory outside of the function's scope:
  - in a correct modular design, helper functions assist in doing portions of the work; in order for different functions to modify the program's data, they must be able to access the data; this can only be done with pointers
  - in a real-life system, there is *one instance* of each unit of data, and multiple functions are passed in pointers to the data in order to access it
- Pointers help to avoid copying data:
  - C++ is notorious for automatically making unnecessary copies of data, either as temporary copies or multiple instances of the same data
  - copies are usually bad! their creation and destruction is a drain on computational resources, and they increase the chances of putting the program in an inconsistent state
- Pointer use is necessary when working with dynamically allocated memory.

4.2.1.3. **Characteristics of pointers:**

- Like all variables, pointers have:
  - a name, since all variables must have a unique identifier
  - a data type, comprised of the data type of the pointee, followed by the asterisk symbol (`*`)
  - a value, which is the memory address of the pointee
  - a location in memory; since a pointer is a variable, it resides in memory at a specific address
  - for example: `char* c;` declares a pointer variable with the name `c`, the data type `char*`, an initial value that is garbage because it has not been initialized, and a location in memory that is chosen by the OS
- If we assume a 64-bit OS, a pointer variable occupies 8 bytes.

4.2.2. Coding example: Pointers

```
 1 int main()
 2 {
 3   char c = 'H';
 4   int  i = 42;
 5   int* iptr;

 7   cout << "sizes:      " << sizeof(c) << " " << sizeof(i) << " " <<
      sizeof(iptr) << endl;

 9   iptr = &i;
10   cout << "addresses: " << &i << " " << &iptr << endl;
11   cout << "values:    " << i  << " " << iptr << endl << endl;

13   cout << "two ways to i:    " << i << " " << *iptr << endl;

15 //  iptr = 99;
16   *iptr = 99;
17   cout << "new value for i:  " << i << " " << *iptr << endl;

19   return 0;
20 }
```



```
Don't Panic ==> p1
sizes:     1 4 8
addresses: 0x16b58b664 0x16b58b658
values:    42 0x16b58b664

two ways to i:    42 42
new value for i:  99 99
Don't Panic ==>
```

Program-4.1: Pointers

Program purpose:

- Program-4.1 demonstrates a simple example of pointer declaration and usage.

Line 5:

- This line shows the declaration of a pointer variable. The variable name is `iptr`, its pointee must be an integer variable, and its initial value is garbage.

Line 7:

- This line prints out the number of bytes occupied by the three variables in this program.
- From the program output, we see that a `char` takes up 1 byte, an integer is 4 bytes, and a pointer is 8 bytes.

Lines 9-11:

- Line 9 assigns a value to the `iptr` variable. This value is the address of the existing integer variable called `i`.
- The ampersand (`&`) on line 9 is the *address-of* operator. It takes one operand, which is an existing variable, and it returns the address in memory where the operand variable is located.
- Line 10 prints out the addresses of the `i` and `iptr` variables. The two addresses are different from each other because each variable has its own location in memory.

- Line 11 prints out the values of the `i` and `iptr` variables. From the program output, we see the value of `i` is 42, and the value of `iptr` is the address of variable `i`.

Line 13:

- This line prints out the value of `i` in two ways: one by directly accessing variable `i`, and the other by following `iptr` to its pointee value.
- The asterisk (`*`) on line 13 is the *dereferencing* operator. It takes one operand, which is a pointer variable, and it returns the value of the operand's pointee variable.

Lines 15-17:

- Line 15 is commented out because it's an error and does not compile on some OSs. It tries to assign a value of `99` to the `iptr` variable, which is a pointer. This would result in address `99` being stored. While this line may or may not be flagged by the compiler, if it was accepted, it would cause a runtime segmentation fault if we used the dereferencing operator to access its pointee. Address `99` is not valid within our programs.
- Line 16 assigns the value `99` to `iptr`'s pointee variable, which is `i`.
- Line 17 prints out the updated value of `i` using both the variable and its dereferenced pointer.

4.2.3. Pointer operators

4.2.3.1. The *address-of* operator:

- The ampersand symbol (`&`) is used as the unary address-of operator to obtain the address of an existing variable.
- This operator takes one operand, which is a variable, and it returns the address in memory where the operand variable is located.
- The ampersand symbol is used as a binary operator for a very different purpose, i.e. the bitwise AND operator.

4.2.3.2. The *dereferencing* operator:

- The asterisk symbol (`*`) is used as the unary dereferencing operator to obtain the value of a pointee variable.
- This operator takes one operand, which is a pointer variable, and it returns the value of the operand's pointee variable.
- The asterisk symbol is used as a binary operator for a very different purpose, i.e. multiplication.
- A very common cause of runtime segmentation faults is the dereferencing of either a null pointer or a garbage pointer:
  - a *null pointer* is one that has been initialized and set to a null (zero) value
  - a *garbage pointer* is one that has *not* been initialized and therefore contains whatever garbage was in the pointer variable's memory cell the last time it was used
- When a null or garbage pointer is dereferenced, the program tries to access the value at that zero or garbage address. Since these are almost never valid addresses in our program, it crashes with a segmentation fault.

PRO TIP: It is crucial to *always* initialize our pointer variables, either with valid address or with null, and then check the pointer variable's value before dereferencing it. We can easily check a pointer to see if it has a null value and choose to not dereference it, but there is no way to test for garbage pointers.

4.2.3.3. **Uses in variable declarations and statements:**

- Both the ampersand and the asterisk symbols have a very different meaning if they are used in a *variable declaration* or in a regular program *statement*.
- In a variable declaration:
  - the ampersand symbol (&) declares a C++ reference to an existing variable
  - the asterisk symbol (*) declares a pointer variable
- In a statement:
  - the ampersand symbol (&) serves as the unary address-of operator
  - the asterisk symbol (*) represents the unary dereferencing operator
- The two uses are *independent* from each other and are **unrelated**.

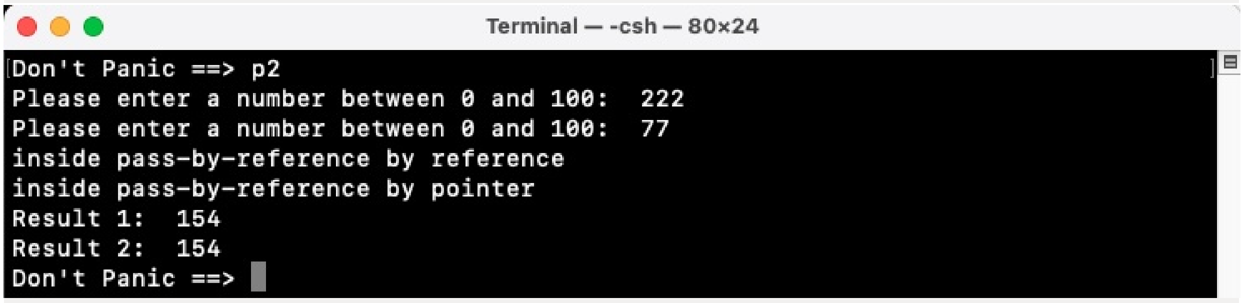
4.2.3.4. **Parameter passing:**

- Pass-by-value makes a copy of a variable value and *does not* use the variable's address.
- Pass-by-reference in C++ may be done in two ways:
  - pass-by-reference by reference: an alias is created inside the called function for an existing variable
  - pass-by-reference by pointer: the address of an existing variable is passed by the calling function to the called function

4.2.4. Coding example: Pass-by-reference using pointers and references

```
int main()
{
  bool inputOk = false;
  int  num, result1, result2;

  while (!inputOk) {
    cout << "Please enter a number between 0 and 100:  ";
    cin >> num;
    inputOk = checkNum(num);
  }

  doubleNum(num, result1);
  doubleNum(num, &result2);
  cout<<"Result 1:  " << result1 << endl;
  cout<<"Result 2:  " << result2 << endl;

  return 0;
}

void doubleNum(int n, int& res)
{
  cout << "inside pass-by-reference by reference" << endl;
  res = n * 2;
}

void doubleNum(int n, int* res)
{
  cout << "inside pass-by-reference by pointer" << endl;
  *res = n * 2;
}

```

```
32 bool checkNum(int n)
33 {
34   return ( n>=0 && n<=100);
35 }
```



```
Don't Panic ==> p2
Please enter a number between 0 and 100:  222
Please enter a number between 0 and 100:  77
inside pass-by-reference by reference
inside pass-by-reference by pointer
Result 1:  154
Result 2:  154
Don't Panic ==> 
```

Program-4.2: Pass-by-reference using pointers and references

Program purpose:

- Program-4.2 is a variation of Program-2.6, but with two versions of the `doubleNum()` function.
- One version of `doubleNum()` passes the result parameter by reference by reference, and the other passes it by reference by pointer.

Lines 20 and 26:

- These lines show the function prototypes of the two versions of the `doubleNum()` function.
- Both versions take the number to be doubled as input parameter and the result as the output parameter. The implementation that begins on line 20 passes the output parameter using a reference, and the one starting on line 26 uses a pointer.

Lines 12-13:

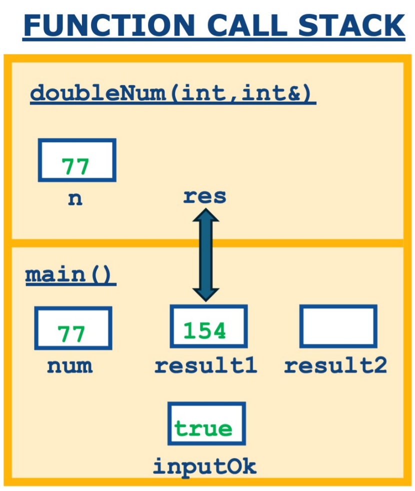
- These lines show the two calls to `doubleNum()`.
- Line 12 passes the result parameter as a reference, so no operator is required for the `result1` variable.
- Line 13 passes the result parameter as a pointer, so the address-of operator must be used to pass the address of the `result2` variable.

Lines 20-24:

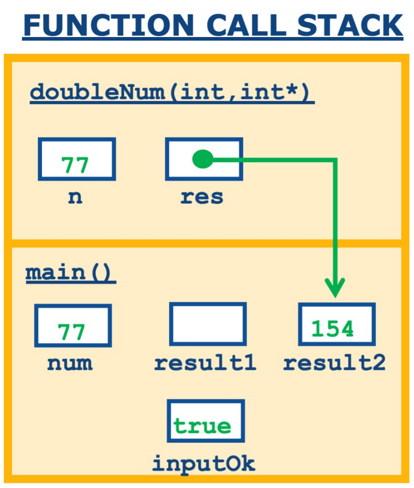
- These lines show the `doubleNum()` function that takes the result parameter as a reference.
- When `main()` calls this function on line 12, the output parameter `res` becomes an alias for the `result1` variable declared in `main()`.
- Line 23 sets the result to the updated value and does *not* need any pointer operators to do so. Because `res` is an alias and not a separate variable, modifying it on line 23 directly modifies the `result1` variable in `main()`.

Lines 26-30:

- These lines implement the `doubleNum()` function that takes the result parameter as a pointer.
- When `main()` calls this function on line 13, it uses the address-of operator to initialize the value of the output parameter `res` with the address of the `result2` variable. So the pointer variable `res` is assigned as pointee the `result2` variable declared in `main()`.
- Line 29 sets the result to the updated value. By dereferencing the `res` parameter, this line ensures that the value of its pointee `result2` is modified.



(a) Result parameter as a reference



(b) Result parameter as a pointer

Figure-4.1: Program-4.2 call stack and heap

Program memory:

- Figure-4.1 shows the contents of the function call stack during a call to each version of the `doubleNum()` function in Program-4.2.
- In the stack frames shown in Figure-4.1a, we see that `result1` is passed to `doubleNum()` as a reference, which means that no separate variable is created for that parameter in `doubleNum()`. As a reference, `res` is simply an alias for the existing variable `result1` declared in `main()`.
- In the stack frames shown in Figure-4.1b, we see that `result2` is passed to `doubleNum()` as a pointer. This means that parameter `res` is a separate variable in the `doubleNum()` stack frame, and its pointee is the existing variable `result2` declared in `main()`.

4.3. Memory allocation

We discuss statically and dynamically allocated memory in C++ programming, as well as the issue of memory leaks.

4.3.1. Concepts

4.3.1.1. What is *memory allocation*?

- *Memory allocation* is the reserving of a specific number of bytes in program memory.
- The number of bytes reserved is based on the data type.
- This allocation may be known at compile time, which is known as static allocation, or it may be performed at runtime, as dynamic allocation.

4.3.1.2. Static memory allocation:

- With static allocation, the number of bytes reserved is known at compile time.
- These bytes are usually stored in the function call stack.
- The use of the word *static* in this context is unrelated to storage class.
- Statically allocated variables include local variables and parameters.
- We statically allocate memory by simply declaring a variable, for example `int x;`

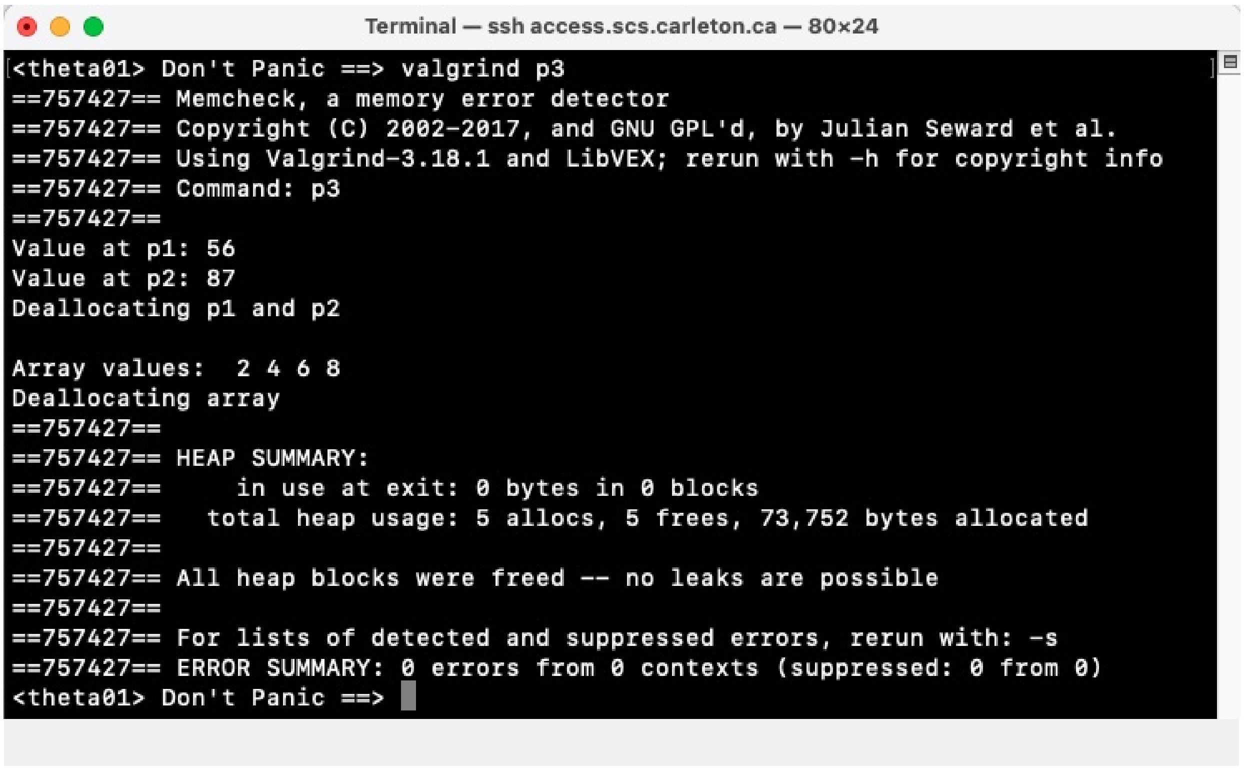
4.3.1.3. **Dynamic memory allocation:**

- With dynamic allocation, the number of bytes reserved is only known at runtime, during the program's execution.
- These bytes are always stored in the heap.
- Memory is dynamically allocated in C++ by using the `new` operator, and it is explicitly deallocated with the `delete` operator.
- Failing to explicitly deallocate dynamically allocated memory will result in a *memory leak*, which is discussed later in this section.

NOTE: All program memory is returned to the OS when the program terminates, so the reader may wonder why bother with explicit deallocation. The reality is that most real-life production-grade software is meant to continue executing for days, or months, or even years *without terminating*. Any memory leak would accumulate over time, and it would eventually cause the program to crash when it runs out of virtual memory. It is considered best practice for programs to clean up after themselves and deallocate memory when it's no longer needed.

4.3.2. Coding example: Dynamic memory allocation

```
#define MAX_ARR   4

int main()
{
  int* p1;
  p1  = new int;
  *p1 = 56;
  cout << "Value at p1: " << *p1 << endl;

  int* p2 = new int(87);
  cout << "Value at p2: " << *p2 << endl;

  cout << "Deallocating p1 and p2" << endl << endl;
  delete p1;
  delete p2;

  int* p3 = new int[MAX_ARR];
  for (int i=0; i<MAX_ARR; i++) {
    p3[i] = (i+1) * 2;
  }

  cout << "Array values:  ";
  for (int i=0; i<MAX_ARR; i++) {
    cout << p3[i] << " ";
  }
  cout << endl;

  cout << "Deallocating array" << endl;
  delete [] p3;

  return 0;
}
```



```
<theta01> Don't Panic ==> valgrind p3
==757427== Memcheck, a memory error detector
==757427== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==757427== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==757427== Command: p3
==757427==
Value at p1: 56
Value at p2: 87
Deallocating p1 and p2

Array values:  2 4 6 8
Deallocating array
==757427==
==757427== HEAP SUMMARY:
==757427==     in use at exit: 0 bytes in 0 blocks
==757427==   total heap usage: 5 allocs, 5 frees, 73,752 bytes allocated
==757427==
==757427== All heap blocks were freed -- no leaks are possible
==757427==
==757427== For lists of detected and suppressed errors, rerun with: -s
==757427== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
<theta01> Don't Panic ==>
```

Program-4.3: Dynamic memory allocation

Program purpose:

- Program-4.3 demonstrates the use of dynamic memory allocation and the corresponding deallocation, with integers and an array.

Lines 5-8:

- These lines show the dynamic allocation of a single integer, its initialization, and printing.
- Line 5 declares an integer pointer variable `p1`, but does not initialize it. This means that the pointer value is garbage at this point.
- Line 6 uses the `new` operator to dynamically allocate an integer (4 bytes) in the heap. This operator returns a pointer to the newly allocated bytes in the heap, and this address is stored in variable `p1`.
- Line 7 uses the dereferencing operator to change the value of the new integer in the heap from garbage to the value `56`.
- Line 8 also uses the dereferencing operator, this time to print the value `56` stored in the heap.

Lines 10-11:

- These lines show the dynamic allocation and initialization of a single integer, and its printing.
- Line 10 uses the `new` operator to dynamically allocate an integer in the heap and store its address in variable `p2`. This line also initializes the new value in the heap to `87`.
- We notice that on line 10, the syntax uses parentheses to initialize the newly allocated integer, in a very similar fashion to a copy constructor. Even though integers are a primitive data type, C++ reuses the same syntax for language consistency.

Lines 14-15:

- These lines show the explicit deallocation of the memory that we have dynamically allocated so far, using the `delete` operator.
- Without this explicit deallocation, the memory would remain reserved, and our program would have a memory leak.

Lines 17-20:

- These lines show the dynamic allocation of an array of integers, its initialization, and printing.
- Line 17 uses the `new` operator to dynamically allocate an array of four integers (16 bytes) as a single contiguous block in the heap. This operator returns a pointer to the first element of the array in the heap, and this address is stored in variable `p3`.
- It's important to note the syntactic differences between lines 10 and 17. Line 10 uses the `new` operator and parentheses to allocate and initialize *a single integer*. Line 17 uses the same operator and square brackets to allocate *an array of integers*.
- Lines 18-20 initialize the array elements to even numbers. Line 19 uses the subscript operator (`[]`) to access each individual element, just like any array.

Line 29:

- This line shows the explicit deallocation of the dynamically allocated array, using the `delete` operator.
- Because we are deallocating an entire array, and not a single object or variable, we must use the empty square brackets with the `delete` operator.
- Failing to use the empty square brackets when deallocating an array may result in unpredictable program behaviour.

NOTE: It is crucial to note that we are responsible for the explicit deallocation of *our dynamically allocated memory only!* Statically allocated memory is stored in the function call stack, and this memory is automatically deallocated when the stack frame containing it is popped off the function call stack. Attempting to explicitly deallocate statically allocated memory will result in a program crash.

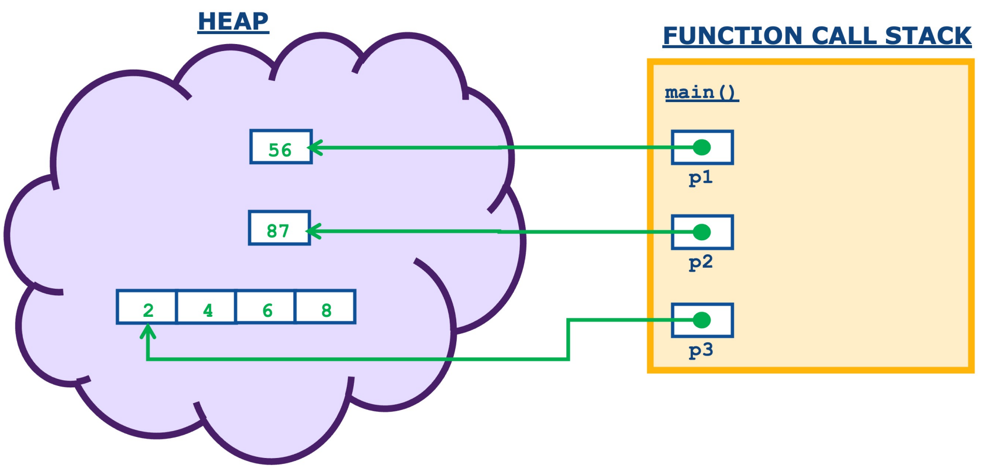


Figure-4.2: Program-4.3 call stack and heap

Program memory:

- Figure-4.2 shows the contents of both the function call stack and the heap during the execution of Program-4.3, before any memory is deallocated.
- In the function call stack, we see that the three pointer variables `p1`, `p2`, and `p3` are stored in the `main()` function's stack frame.
- In the heap, we see the three blocks of dynamically allocated memory. Each one is a pointee of a pointer variable stored in the function call stack.

4.3.3. Memory leaks

4.3.3.1. **What is a *memory leak*:**

- A *memory leak* is a block of memory in the heap that was dynamically allocated by a program, but the program no longer has any pointers to it.
- There are two serious issues with a memory leak:
  - the data inside the memory block is lost because it can no longer be accessed
  - the memory block can never be explicitly deallocated
- In long-lived programs like most production-grade software, the memory leaks accumulate until there is no remaining virtual memory. When this happens, the program crashes.
- Memory leaks are not possible with statically allocated memory, because that type of memory is automatically deallocated when a stack frame is popped off the function call stack.

4.3.3.2. **Possible causes of memory leaks:**

- A memory leak may occur if a pointer into the heap gets *clobbered*, i.e. accidentally overwritten, by the program.
- It may also happen when a pointer variable moves out of scope. For example, if a function stores a pointer into the heap as a local variable, it must either deallocate the block or return the pointer to the calling function. If the only pointer to a dynamically allocated block is a local variable in a stack frame, a memory leak occurs when the stack frame is popped off the function call stack,

4.3.3.3. **Why are memory leaks a problem:**

- A memory leak means that access to the data is permanently lost by the program.
- The program may also run out of heap space:
  - only a finite amount of heap space is allocated to every program
  - once dynamically allocated, a memory block stays reserved until either it is deallocated, or the program terminates
  - for long-running programs, a memory leak accumulates over time until heap space runs out
  - the development of professional software must entail planning for execution durations of months or years

4.3.3.4. **How do we prevent memory leaks:**

- Our programs must always explicitly deallocate dynamically allocated memory as soon as it is no longer required. The `valgrind` utility in Linux can assist in finding memory leaks.
- If a function is tasked with dynamically allocating some memory, the programmer must make sure that pointers to the allocated blocks are passed by reference:
  - passing pointers by reference requires the use of *double pointers*
  - a double pointer is a pointer variable whose value is the address of another pointer variable
- We must make sure that our code never clobbers pointers into the heap.

4.3.3.5. **What is *garbage collection*:**

- *Garbage collection* is an automated mechanism that frees dynamically allocated memory blocks that no longer have any pointers to them.
- In order to support the development of real-time applications and give programmers full control, C and C++ *do not* perform garbage collection, hence the risk of memory leaks.

4.3.4. Dynamic allocation of objects

4.3.4.1. How are objects allocated:

- Static allocation:
  - objects are allocated statically with a simple declaration, as we saw in Program-3.1, on lines 37-38
  - statically allocated objects are stored entirely in the function call stack, in the declaring function's stack frame
- Dynamic allocation:
  - objects are allocated dynamically using the `new` operator
  - dynamically allocated objects are stored entirely in the heap
- Whether an object is allocated statically or dynamically, its constructor is called automatically on allocation.
- Constructor parameters may be specified, otherwise the default constructor executes.

4.3.4.2. How are objects deallocated:

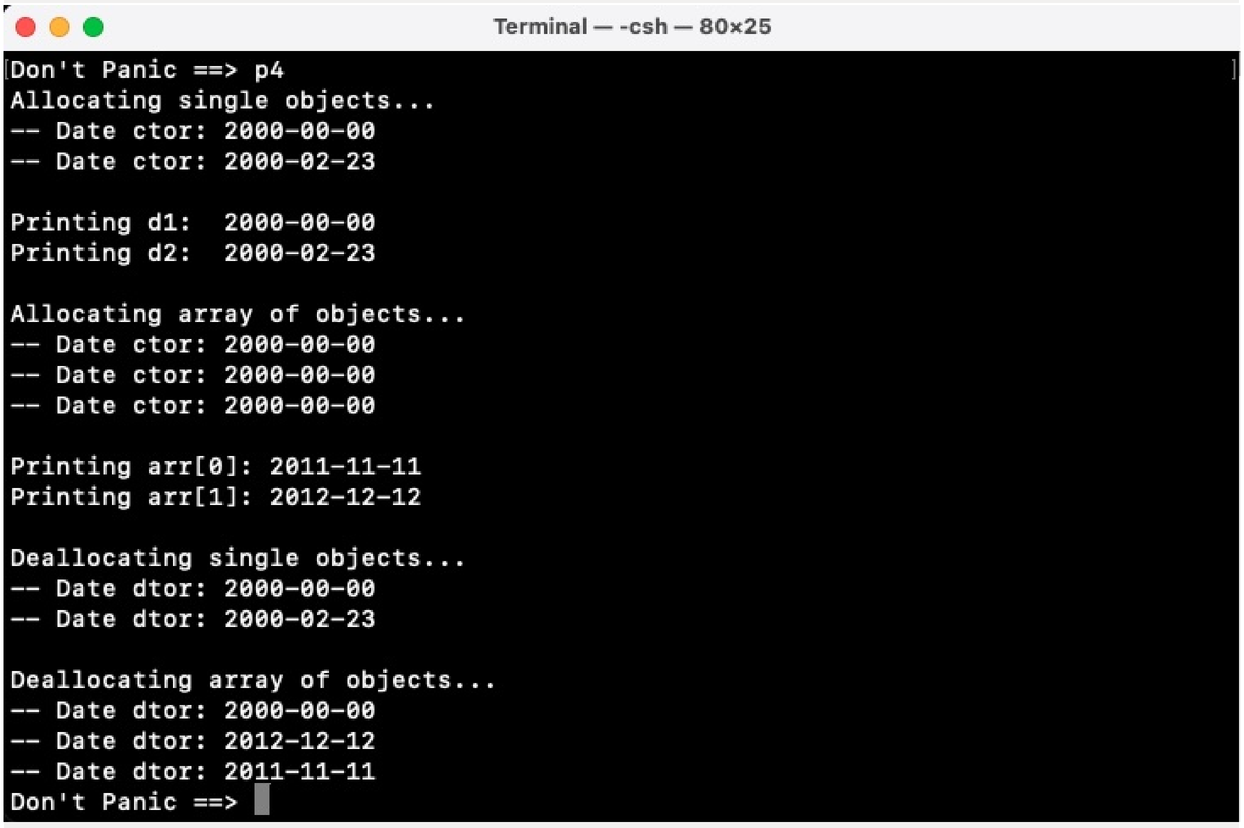
- Statically allocated objects:
  - objects that are allocated statically *do not* require explicit deallocation
  - they are deallocated automatically when the stack frame that contains them is popped off the function call stack
- Dynamically allocated objects:
  - objects that are allocated dynamically must be deallocated using the `delete` operator, otherwise they cause a memory leak
- Whether an object is allocated statically or dynamically, its destructor is called automatically on deallocation.

4.3.4.3. Dynamically allocated containees:

- Classes that have dynamically allocated objects as containees must be implemented carefully in order to avoid memory leaks.
- These container classes should always provide:
  - a copy constructor that performs a deep copy, if needed
  - a destructor that explicitly deallocates the containee objects
  - an overloaded assignment operator (=) that performs a deep copy, if needed
- By default, copy constructors and assignment operators both use member-wise assignment, which only performs a shallow copy.

4.3.5. Coding example: Dynamic allocation of objects

```cpp
int main()
{
  cout << "Allocating single objects..." << endl;
  Date* d1 = new Date;
  Date* d2 = new Date(23, 2);

  cout << endl << "Printing d1:  ";
  d1->print();
  cout << "Printing d2:  ";
  d2->print();

  cout<<endl<<"Allocating array of objects..."<<endl;
  Date* arr = new Date[MAX_ARR_SIZE];
  arr[0].setDate(11, 11, 2011);
  arr[1].setDate(12, 12, 2012);
  cout << endl << "Printing arr[0]: ";
  arr[0].print();
  cout << "Printing arr[1]: ";
  arr[1].print();

  cout<<endl<<"Deallocating single objects..."<<endl;
  delete d1;
  delete d2;

  cout<<endl<<"Deallocating array of objects..."<<endl;
  delete [] arr;
  return 0;
}
```



```
Don't Panic ==> p4
Allocating single objects...
-- Date ctor: 2000-00-00
-- Date ctor: 2000-02-23

Printing d1:  2000-00-00
Printing d2:  2000-02-23

Allocating array of objects...
-- Date ctor: 2000-00-00
-- Date ctor: 2000-00-00
-- Date ctor: 2000-00-00

Printing arr[0]: 2011-11-11
Printing arr[1]: 2012-12-12

Deallocating single objects...
-- Date dtor: 2000-00-00
-- Date dtor: 2000-02-23

Deallocating array of objects...
-- Date dtor: 2000-00-00
-- Date dtor: 2012-12-12
-- Date dtor: 2011-11-11
Don't Panic ==>
```

Program-4.4: Dynamic allocation of objects

Program purpose:

- Program-4.4 demonstrates the dynamic memory allocation of individual `Date` objects and an array of objects, as well as the corresponding deallocation.
- The `Date` class used in this program is identical to Program-3.4, with the addition of a destructor that prints out a debugging message. We assume that the preprocessor constant `MAX_ARR_SIZE` is defined as `3`.

Lines 5-12:

- Line 4 shows the dynamic allocation of a single `Date` object. Since no parameter values are specified, the constructor is called with default values. A pointer to the new object in the heap is stored in local pointer variable `d1`.
- Line 5 shows the dynamic allocation of another single `Date` object and a call to its default constructor with provided parameter values. A pointer to the new object is stored in local pointer variable `d2`.
- Lines 8 and 10 call the printing member function for each new `Date` object. Since `d1` and `d2` are *pointers* to objects, and not actual objects, their members must be accessed using the *arrow operator* (`->`), instead of the dot operator.

Lines 12-19:

- Line 13 shows the dynamic allocation of an array of three `Date` objects and a call to the default constructor for each element in the array. A pointer to the new array in the heap is stored in local pointer variable `arr`.
- Lines 14-15 show calls to the setter member function for the first two elements of the array. Since each element in the array is an actual *object*, and not a pointer to an object, its members must be accessed using the dot operator, instead of the arrow operator.
- Lines 17 and 19 show calls to the printing member function for those same objects.

Lines 21-26:

- Lines 22-23 show the deallocation of both single objects.
- Line 26 deallocates the entire array.

Program execution:

- We see from the program output that the constructor is called when each individual `Date` object is created.
- The constructor is also called three times when the array is allocated, in order to initialize each array element with default values.
- When the individual objects are deallocated on lines 22-23, we see from the output that the destructor is called for each object.
- When the array is deallocated on line 26, the destructor is called for each element of the array, in the reverse order of construction.

Program memory:

- Figure-4.3 shows the contents of both the function call stack and the heap during the execution of Program-4.4, before any memory is deallocated.
- In the function call stack, we see that the three pointer variables `d1`, `d2`, and `arr` are stored in the `main()` function's stack frame.
- In the heap, we see the three blocks of dynamically allocated memory. Each one is a pointee of a pointer variable stored in the function call stack.
- We also note that *there are no `Date` objects in the function call stack in this program*. All the `Date` objects are dynamically allocated in the heap.

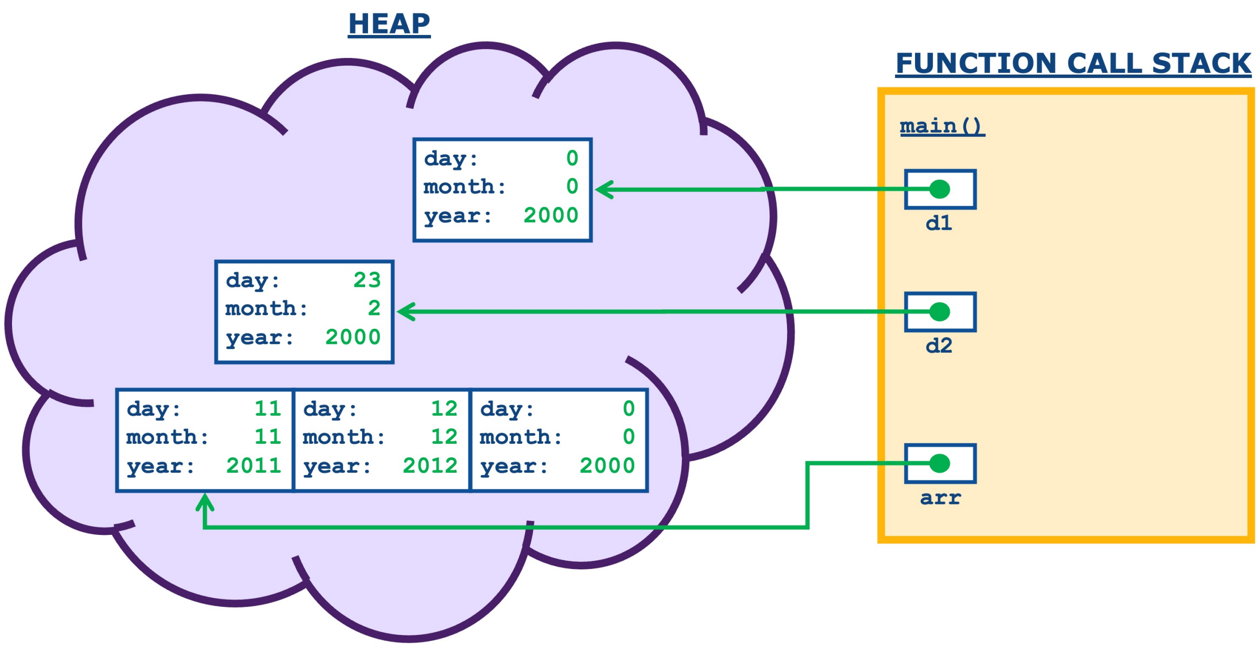


Figure-4.3: Program-4.4 call stack and heap

4.4. Dynamic arrays

We discuss the four different ways in which objects may be stored in arrays in C++ programs. We demonstrate the syntax for the allocation and deallocation of each type of array and its elements.

4.4.1. Types of arrays

4.4.1.1. Types of array allocation:

- Arrays in C++ can be either statically allocated or dynamically allocated.
- With statically allocated arrays, the array itself is stored in the function call stack.
- With dynamically allocated arrays, the array is stored in the heap.

4.4.1.2. Types of array elements:

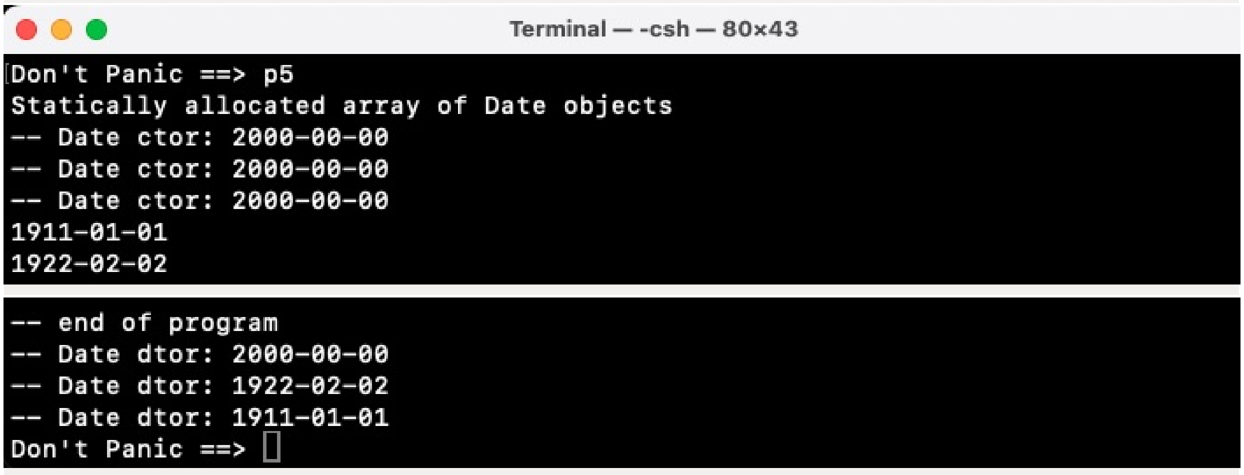
- Arrays can contain either data, as objects or primitive values, or pointers to data.
- If an array stores data, the memory for that data is automatically allocated with the array.
- If an array stores pointers to data, the memory for that data must be allocated separately from the array.

4.4.1.3. Options for storing data in an array:

- There are four ways to store data in an array:
  - in a statically allocated array of objects
  - in a statically allocated array of object pointers
  - in a dynamically allocated array of objects
  - in a dynamically allocated array of object pointers
- Each of the four types of array is stored differently in the function call stack and the heap.
- Each type of array has different allocation and deallocation requirements.

4.4.2. Coding example: Statically allocated array of objects

```
 1 #define MAX_ARR_SIZE  3

 3   cout << "Statically allocated array of Date objects" << endl;
 4   Date arr1[MAX_ARR];

 6   arr1[0].setDate(1,1,1911);
 7   arr1[1].setDate(2,2,1922);
 8   arr1[0].print();
 9   arr1[1].print();

11   cout << endl << "-- end of program" << endl;
12   return 0;
13
```



```
Don't Panic ==> p5
Statically allocated array of Date objects
-- Date ctor: 2000-00-00
-- Date ctor: 2000-00-00
-- Date ctor: 2000-00-00
1911-01-01
1922-02-02

-- end of program
-- Date dtor: 2000-00-00
-- Date dtor: 1922-02-02
-- Date dtor: 1911-01-01
Don't Panic ==> 
```

Program-4.5: Statically allocated array of objects

Partial program purpose:

- Partial Program-4.5 uses the same `Date` class as Program-4.4 to show how a *statically allocated array of objects* is allocated and used.
- Because the entire array and its elements are statically allocated, they are stored in the function call stack. As a result, they do *not* require explicit deallocation.

Line 4:

- This line shows the allocation of the array and its object elements. We see from the program output that this line results in three calls to the `Date` default constructor, once for each element in the array.

Lines 6-7:

- These lines initialize the first two elements of the array with specific values.
- The third element keeps the default values assigned by the constructor.

Lines 8-9:

- These lines print the first two elements of the array.
- Because each element is an object, and not a pointer, the dot operator is used to call the printing member function.

Line 12:

- We see from the program output that all three `Date` elements in the array are deallocated at the end of the program, when the `main()` function's stack frame is popped off the function call stack. When the array is deallocated, the destructor is automatically called for each `Date` object element.

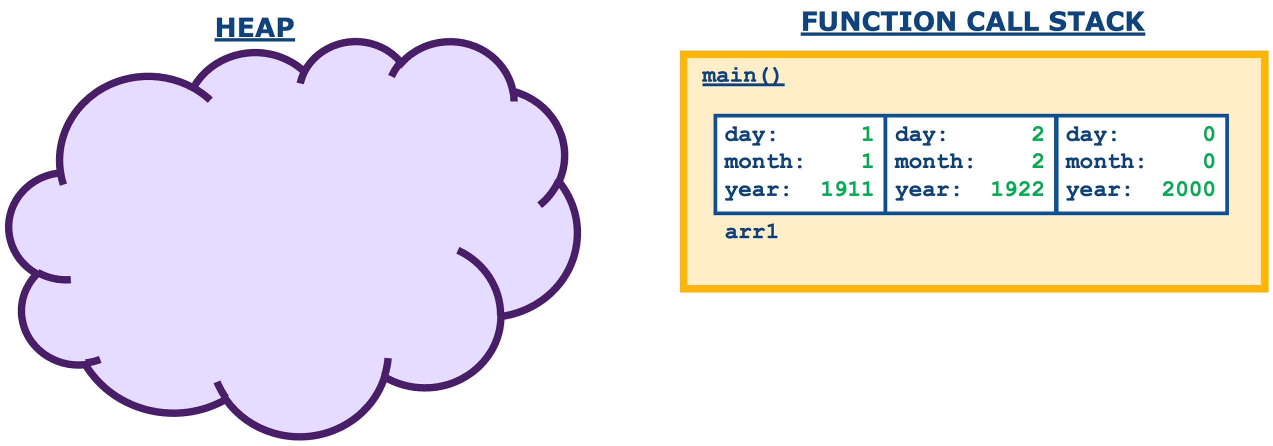


Figure-4.4: Program-4.5 call stack and heap

Program memory:

- Figure-4.4 shows the contents of both the function call stack and the heap during the execution of partial Program-4.5, before any memory is deallocated.
- In the function call stack, we see that the entire array `arr1`, including its `Date` object elements, is stored in `main()`'s stack frame.
- There is nothing in the heap, because no memory is dynamically allocated in this example.

4.4.3. Coding example: Statically allocated array of object pointers

```
cout << "Statically allocated array of Date object pointers"<<endl;
Date* arr2[MAX_ARR];

Date* dt1 = new Date(3,3,1933);
arr2[0] = dt1;
arr2[1] = new Date(4,4,1944);
arr2[0]->print();
arr2[1]->print();

cout << "... deallocating arr2 elements" << endl;
delete arr2[0];
delete arr2[1];

```

```
Statically allocated array of Date object pointers
-- Date ctor: 1933-03-03
-- Date ctor: 1944-04-04
1933-03-03
1944-04-04
... deallocating arr2 elements
-- Date dtor: 1933-03-03
-- Date dtor: 1944-04-04
```

Program-4.6: Statically allocated array of object pointers

Partial program purpose:

- Partial Program-4.6 shows how a *statically allocated array of object pointers* is allocated, used, and deallocated.
- Because the array itself is statically allocated, it is stored in the function call stack. As a result, it does *not* require explicit deallocation.
- But because the array elements point to dynamically allocated `Date` objects, these objects are stored in the heap and must be explicitly deallocated.

Line 2:

- This line shows the static allocation of the array of `Date` pointers.
- We see from the program output that this line does *not* result in any calls to the `Date` default constructor, because no `Date` objects are created on this line. The array elements are *pointers*, which are a primitive data type and not objects.

Lines 4 and 6:

- These lines show the dynamic allocation of two `Date` objects.
- We see from the program output that these lines *do* result in a call to the `Date` constructor.

Lines 5-6:

- These lines initialize the first two elements of the array with pointer values.
- The third element keeps its garbage value, because it has not been initialized.

Lines 7-8:

- These lines print the first two elements of the array.
- Because each element is a pointer, and not an object, the arrow operator is used to call the printing member function.

Lines 11-12:

- These lines deallocate the two dynamically allocated `Date` objects, and we see from the program output that the `Date` destructor is called for each object.
- The array itself is deallocated at the end of the program, when the `main()` function's stack frame is popped off the function call stack.

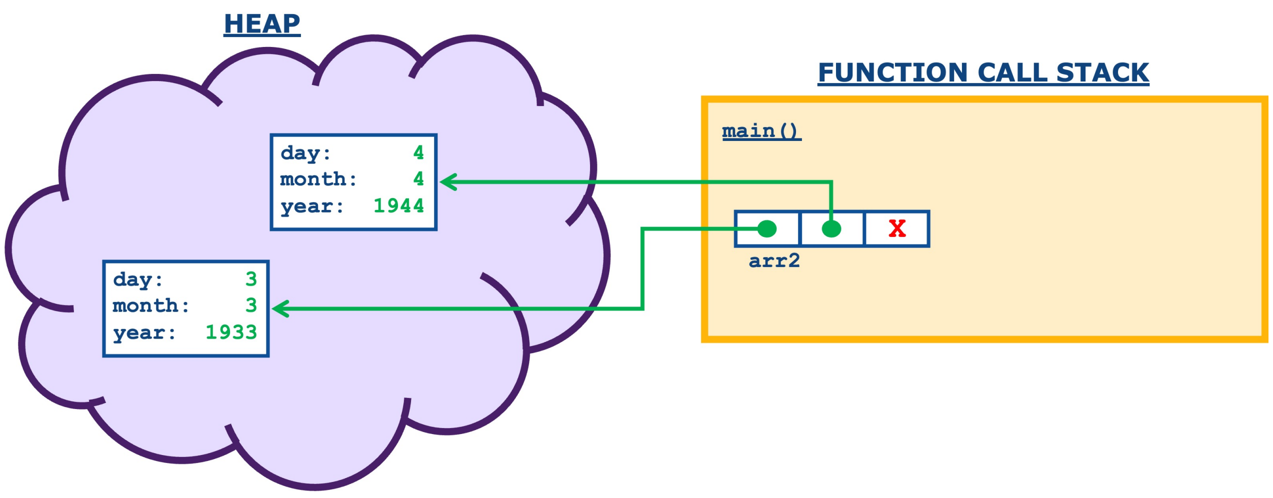


Figure-4.5: Program-4.6 call stack and heap

Program memory:

- Figure-4.5 shows the contents of both the function call stack and the heap during the execution of partial Program-4.6, before any memory is deallocated.
- In the function call stack, we see that the array of pointers `arr2` is stored in `main()`'s stack frame. We also note that *there are no `Date` objects in the call stack in this partial program*.
- In the heap, we see the two dynamically allocated `Date` objects. Each one is a pointee of a pointer element of `arr2`, which is stored in the function call stack.

4.4.4. Coding example: Dynamically allocated array of objects

```
 1  cout << "Dynamically allocated array of Date objects" << endl;
 2  Date* arr3;
 3  arr3 = new Date[MAX_ARR];

 5  arr3[0].setDate(5,5,1955);
 6  arr3[1].setDate(6,6,1966);
 7  arr3[0].print();
 8  arr3[1].print();

10  cout << "... deallocating arr3" << endl;
11  delete [] arr3;
12
```

```
Dynamically allocated array of Date objects
-- Date ctor: 2000-00-00
-- Date ctor: 2000-00-00
-- Date ctor: 2000-00-00
1955-05-05
1966-06-06
... deallocating arr3
-- Date dtor: 2000-00-00
-- Date dtor: 1966-06-06
-- Date dtor: 1955-05-05
```

Program-4.7: Dynamically allocated array of objects

Partial program purpose:

- Partial Program-4.7 shows how a *dynamically allocated array of objects* is allocated, used, and deallocated.
- Because the entire array and its elements are dynamically allocated, they are stored in the heap. As a result, they *do* require explicit deallocation.

Lines 2-3:

- These lines show the declaration and dynamic allocation of the array and its objects.
- We see from the program output that line 3 results in three calls to the `Date` default constructor, once for each element in the array.

Lines 5-6:

- These lines initialize the first two elements of the array with specific values.
- The third element keeps the default values assigned by the constructor.

Lines 7-8:

- These lines print the first two elements of the array.
- Because each element is an object, and not a pointer, the dot operator is used to call the printing member function.

Line 11:

- This line deallocates the dynamically allocated array, including the `Date` objects that are stored directly inside the array.
- We see that the `Date` destructor is automatically called for each object element stored in the array.

Program memory:

- Figure-4.6 shows the contents of both the function call stack and the heap during the execution of partial Program-4.7, before any memory is deallocated.
- In the function call stack, we see that the single pointer `arr3` is stored in `main()`'s stack frame. We also note that there is *no array* and there are *no* `Date` *objects* in the call stack in this partial program.
- In the heap, we see the entire dynamically allocated array, including its `Date` object elements. The array is a pointee of pointer variable `arr3`, which is stored in the function call stack.

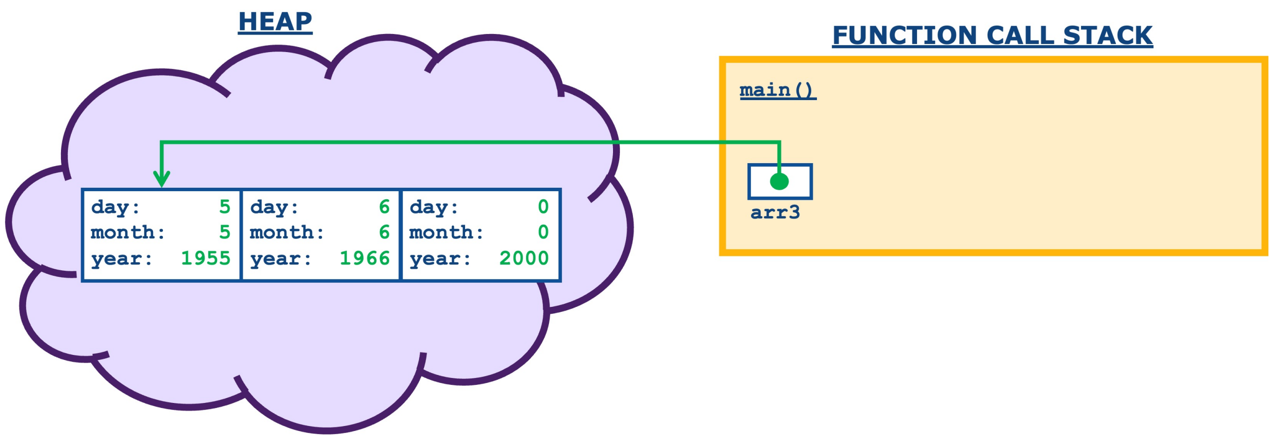
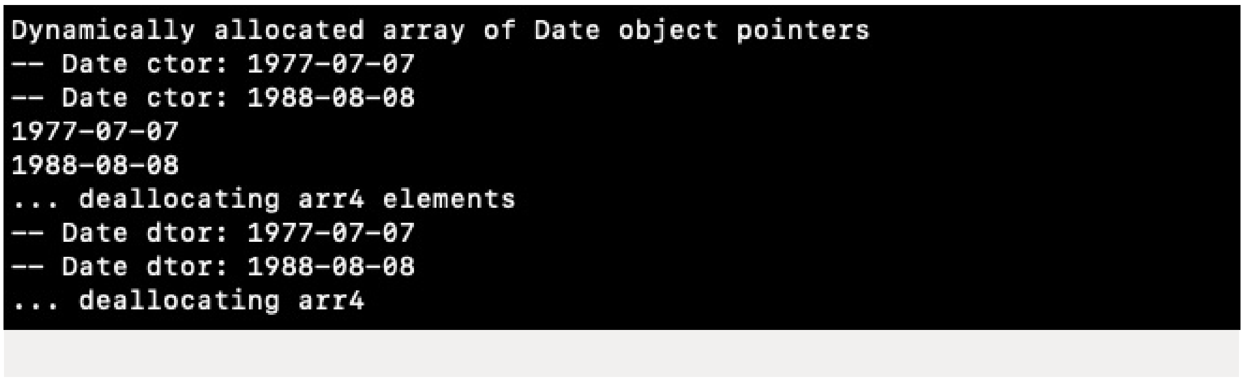


Figure-4.6: Program-4.7 call stack and heap

4.4.5. Coding example: Dynamically allocated array of object pointers

```
cout<<"Dynamically allocated array of Date object pointers"<<endl;
Date** arr4;
arr4 = new Date*[MAX_ARR];

Date* dt2 = new Date(7,7,1977);
arr4[0] = dt2;
arr4[1] = new Date(8,8,1988);
arr4[0]->print();
arr4[1]->print();

cout << "... deallocating arr4 elements" << endl;
delete arr4[0];
delete arr4[1];
cout << "... deallocating arr4" << endl;
delete [] arr4;

```



```
Dynamically allocated array of Date object pointers
-- Date ctor: 1977-07-07
-- Date ctor: 1988-08-08
1977-07-07
1988-08-08
... deallocating arr4 elements
-- Date dtor: 1977-07-07
-- Date dtor: 1988-08-08
... deallocating arr4
```

Program-4.8: Dynamically allocated array of object pointers

Partial program purpose:

- Partial Program-4.8 shows how a *dynamically allocated array of object pointers* is allocated, used, and deallocated.
- Because both the array and its elements are dynamically allocated, they are stored in the heap. As a result, they *do* require explicit deallocation.

Lines 2-3:

- These lines show the declaration and dynamic allocation of the array *only*.
- We see from the program output that this line does *not* result in any calls to the `Date` default constructor, because no `Date` objects are created on these lines. The array elements are *pointers*, which are a primitive data type and not objects.

Lines 5 and 7:

- These lines show the dynamic allocation of two `Date` objects.
- We see from the program output that these lines *do* result in a call to the `Date` constructor.

Lines 6-7:

- These lines initialize the first two elements of the array with pointer values.
- The third element keeps its garbage value, because it has not been initialized.

Lines 8-9:

- These lines print the first two elements of the array.
- Because each element is a pointer, and not an object, the arrow operator is used to call the printing member function.

Lines 11-15:

- Lines 12-13 deallocate the two dynamically allocated `Date` objects, and we see from the program output that the `Date` destructor is called for each object.
- Line 15 deallocates the array itself.

Program memory:

- Figure-4.7 shows the contents of both the function call stack and the heap during the execution of partial Program-4.8, before any memory is deallocated.
- In the function call stack, we see that the single pointer `arr4` is stored in `main()`'s stack frame. We also note that there is *no array* and there are *no `Date` objects* in the call stack in this partial program.

- In the heap, we see the dynamically allocated array of pointers. The array is a pointee of pointer variable `arr4`, which is stored in the function call stack. The heap also contains the two dynamically allocated `Date` objects, each one a pointee of a pointer element of the array.

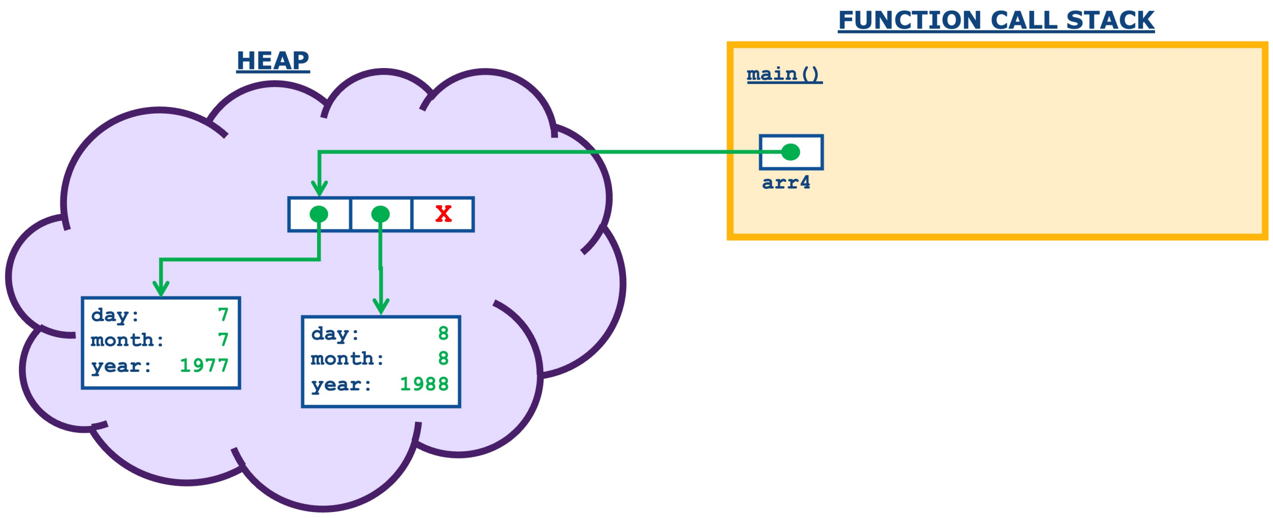


Figure-4.7: Program-4.8 call stack and heap

4.5. Double pointers

Working with double pointers is an essential technique in programming languages where memory management is the programmer's responsibility. In order to achieve a correct design, we must delegate sub-tasks to helper functions, which necessitates the ability to pass any variable by reference. Passing pointers by reference requires the use of double pointers.

4.5.1. Concepts

4.5.1.1. **What is a *double pointer*:**

- A *double pointer* is a pointer variable whose pointee is another pointer variable.
- So the *value* of a double pointer is the address of that other pointer variable.
- For example, `int** p;` declares a pointer variable `p` whose pointee is another pointer variable whose pointee is an integer variable.
- In partial Program-4.8, line 2 declares a dynamically allocated array of `Date` object pointers as `Date** arr4;` which indicates that the double pointer variable `arr4` has a pointee that itself is a pointer. We see from Figure-4.7 that `arr4`'s pointee is the first element of the dynamically allocated array, and this first element is a `Date` object pointer.

  NOTE: Recall from programming with arrays in C that an array variable's pointee is *always* the array's first element. So the *value* of the array variable is the address of that first element.

4.5.1.2. **Uses of double pointers:**

- The most common usage of double pointers is for passing pointers by reference.
- In a correct design with modular functions, it's normal to delegate small tasks to helper functions. If the job of a helper function is to modify a pointer value, then the corresponding pointer variable must be passed by reference, as a double pointer.
- A double pointer may also be used to store an array of pointers, as we saw in partial Program-4.8, or to store a two-dimensional (2D) array.

NOTE: It is almost always a mistake to declare a double pointer as a stand-alone variable, except in the case of an array of pointers or a 2D array. The common usage of a double pointer is to *pass a pointer variable by reference*. The only place where double pointer variables should be declared is as a parameter in a called function.

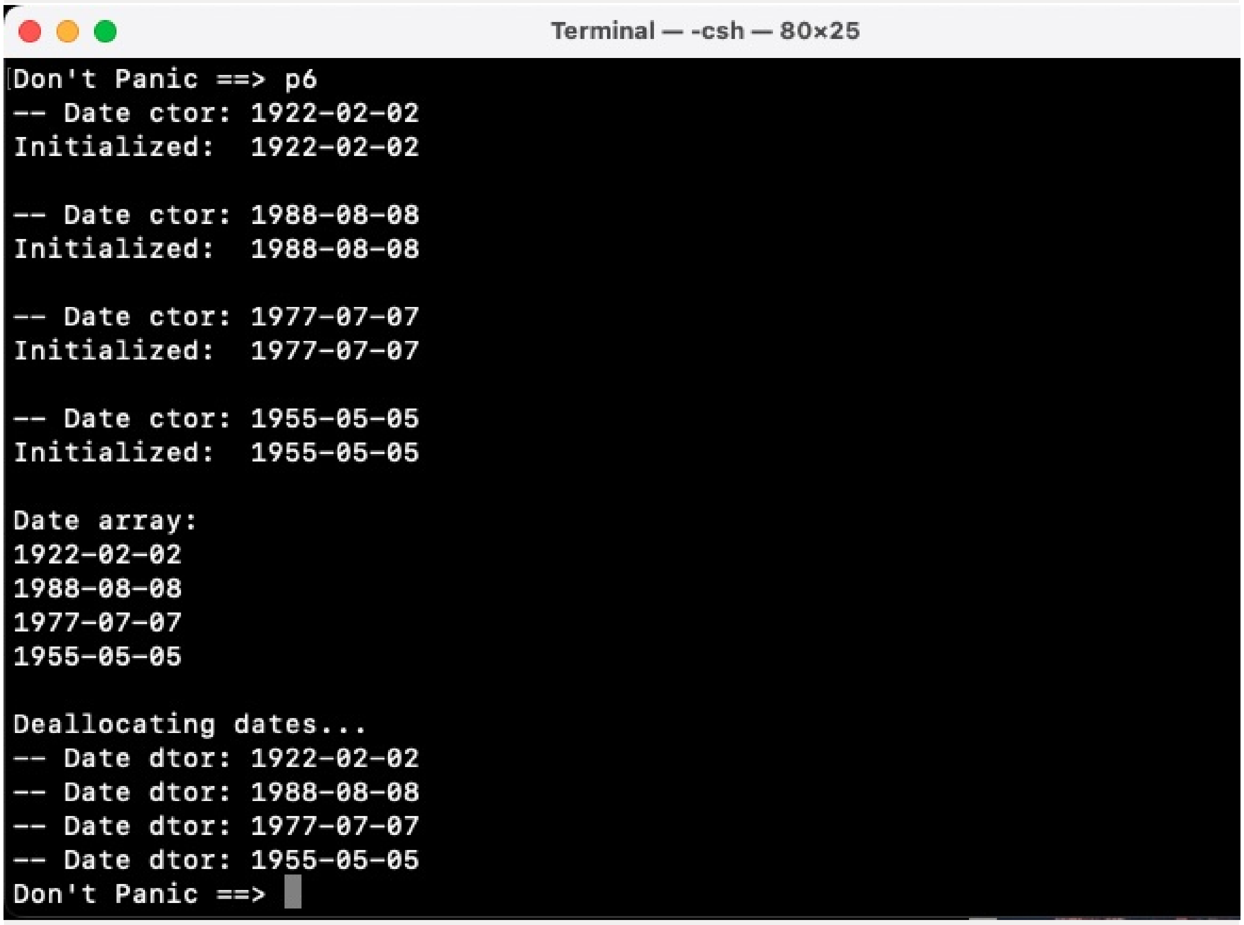
NOTE: For example, if a called function has the prototype `void foo(int** p)`, it is always **incorrect** to declare a double pointer `int** myPtr;` in the calling function and then to call `foo()` using the statement `foo(myPtr)`. All this does is copy the garbage inside `myPtr` into the called function `foo()`, which will modify its *local copy* of parameter `p` to a valid value. When `foo()` returns and `p` is popped off the function call stack, the `myPtr` variable in the calling function will still contain the same garbage it originally had.

Passing a double pointer by value is NOT the same as passing a single pointer by reference, even though the called function prototype may look the same.

4.5.2. Coding example: Double pointers

```
#define MAX_ARR  5
void initDate(int, int, int, Date**);

int main()
{
  Date *someDates[MAX_ARR];
  Date *tmpDate;

  initDate(1922, 2, 2, &tmpDate);
  someDates[0] = tmpDate;
  cout << "Initialized:  ";
  tmpDate->print();  cout << endl;

  initDate(1988, 8, 8, &tmpDate);
  someDates[1] = tmpDate;
  cout << "Initialized:  ";
  tmpDate->print();  cout << endl;

  initDate(1977, 7, 7, &someDates[2]);
  cout << "Initialized:  ";
  someDates[2]->print();  cout << endl;

  initDate(1955, 5, 5, someDates+3);
  cout << "Initialized:  ";
  someDates[3]->print();  cout << endl;

  cout << "Date array:" << endl;
  for (int i=0; i<4; ++i) {
    someDates[i]->print();
  }
  cout << endl;

  cout << "Deallocating dates... " << endl;
  for (int i=0; i<4; ++i) {
    delete someDates[i];
  }

  return 0;
}

```

```
41 void initDate(int y, int m, int d, Date** dt)
42 {
43   *dt = new Date(d, m, y);
44 }
```



```
Don't Panic ==> p6
-- Date ctor: 1922-02-02
Initialized:  1922-02-02

-- Date ctor: 1988-08-08
Initialized:  1988-08-08

-- Date ctor: 1977-07-07
Initialized:  1977-07-07

-- Date ctor: 1955-05-05
Initialized:  1955-05-05

Date array:
1922-02-02
1988-08-08
1977-07-07
1955-05-05

Deallocating dates...
-- Date dtor: 1922-02-02
-- Date dtor: 1988-08-08
-- Date dtor: 1977-07-07
-- Date dtor: 1955-05-05
Don't Panic ==> 
```

Program-4.9: Double pointers

Program purpose:

- Program-4.9 demonstrates the use of double pointers to pass a pointer by reference. The `initDate()` helper function is called from `main()` multiple times to allocate and initialize a `Date` object, using a pointer to the allocated object as an output parameter.
- The `Date` class used in this program is identical to Program-4.4.

Lines 2 and 41:

- These lines show the prototype for function `initDate()`.
- The job of the `initDate()` function is to dynamically allocate memory for a new `Date` object, initialize its data members to the values provided in the input parameters, and return a pointer to the new object using an output parameter.
- We see that this function takes the following parameters:
  - three input parameters for the new date's year, month, and day
  - an output parameter to return to the calling function a pointer to a newly allocated and initialized `Date` object
- The `initDate()` function expects the calling function to declare a `Date` pointer variable that is passed by reference using a double pointer parameter.

Lines 6-7:

- Line 6 declares a statically allocated array of `Date` object pointers, as we did in partial Program-4.6. With this type of array, the array of pointers is located in the function call stack, while its object elements are in the heap.
- Line 7 declares a temporary `Date` pointer that to be used as output parameter in the calls to the `initDate()` function.

Lines 9-12:

- Line 9 calls `initDate()` to allocate and initialize a new `Date` object, using the locally declared `tmpDate` pointer variable as the output parameter.
- After line 9, the value inside `tmpDate` is the address of the new block of memory in the heap that contains the new `Date` object.
- Line 10 initializes the first element of the `someDates` array to that same address in the heap. This line is a simple assignment of one pointer value to another, as a primitive data type.
- After line 10, both `tmpDate` and the first element of `someDates` point to the newly created `Date` object in the heap.
- Figure-4.8a shows the contents of the function call stack and heap after line 10.

Lines 14-17:

- Line 14 calls `initDate()` again to allocate and initialize another new `Date` object, using the locally declared `tmpDate` pointer variable as the output parameter.
- Figure-4.8b shows the contents of the function call stack and heap during the call to `initDate()` from line 14.
- After line 14, the value inside `tmpDate` is the address of the new block of memory in the heap that contains the new `Date` object.
- Line 15 initializes the second element of the `someDates` array to that same address in the heap.
- After line 15, both `tmpDate` and the second element of `someDates` point to the newly created `Date` object in the heap.
- Figure-4.8c shows the contents of the function call stack and heap after line 15.

Lines 19-21:

- Line 19 calls `initDate()` to allocate and initialize another new `Date` object. This time, the third element of the `someDates` array is used directly as the output parameter.
- Because the output parameter to `initDate()` must be the address of a pointer variable, the function is called with the address of the array's third element, as `&someDates[2]`.

Lines 23-25:

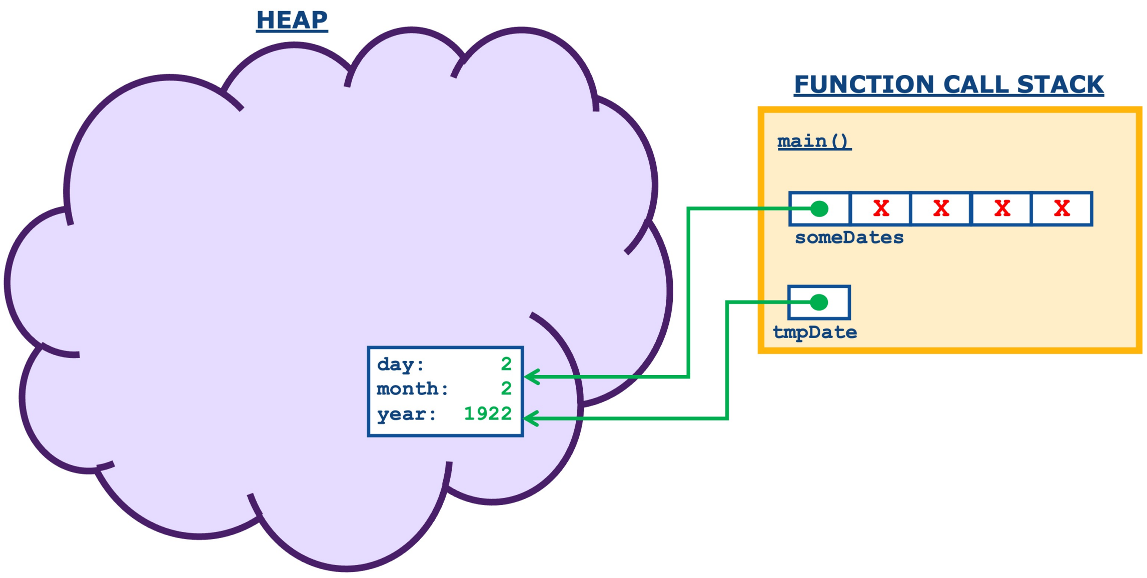
- Line 23 calls `initDate()` to allocate and initialize another new `Date` object. The fourth element of the `someDates` array is used as the output parameter.
- The address of the array's fourth element is used as output parameter. As we know from pointer arithmetic principles, `someDates+3` is equivalent to `&someDates[3]`. The reader is encouraged to review the pointer arithmetic concepts that were covered in COMP 2401.
- Figure-4.8d shows the contents of the function call stack and heap after line 23, where the `someDates` array has been populated with pointers to the four `Date` objects.

Lines 27-36:

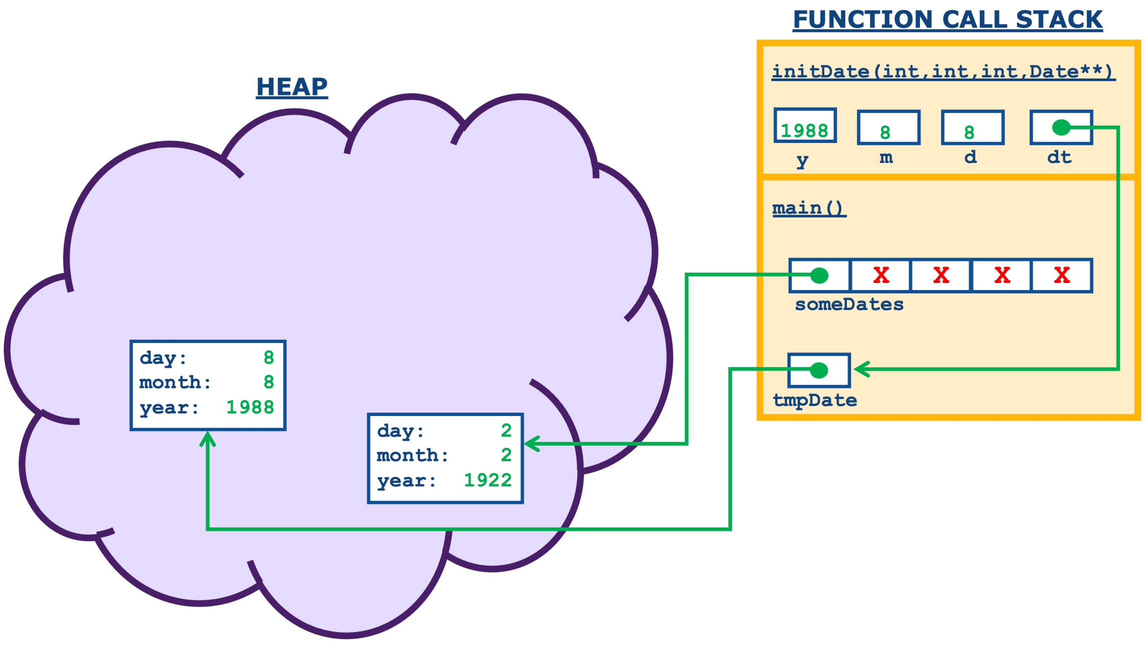
- Lines 28-30 print out every object element in the `someDates` array.
- Lines 34-36 loop over the `someDates` array to deallocate all four `Date` objects. The array itself is in the function call stack and doesn't need explicit deallocation.

Lines 41-44:

- These lines show the implementation of the `initDate()` function.
- Line 43 allocates and initializes a new `Date` object. It stores the pointer to the new object inside a pointer variable or an array element declared in `main()` and used as the output parameter.

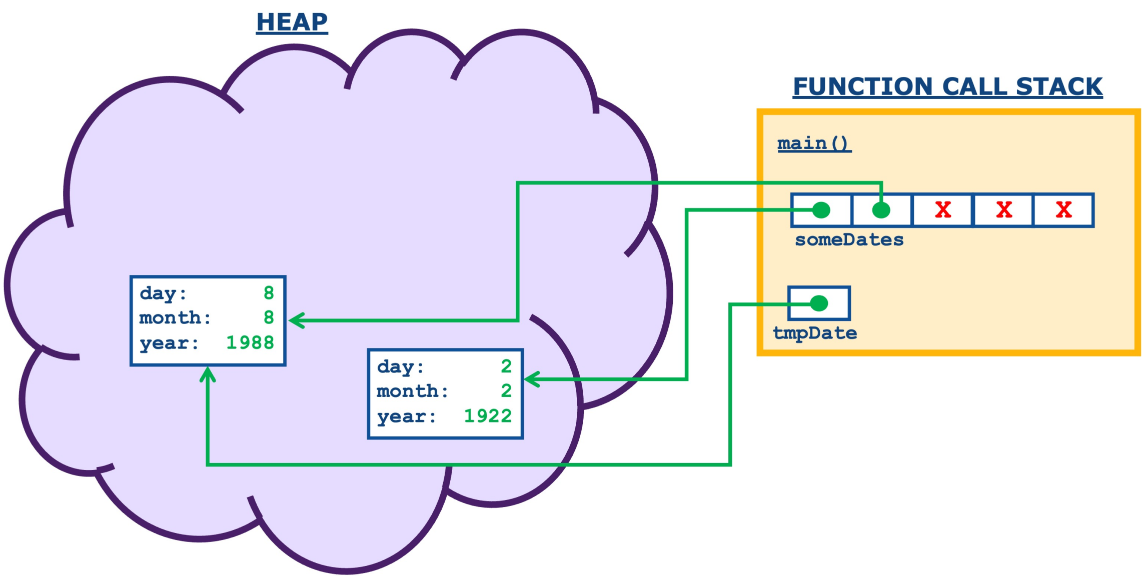


(a) After line 10

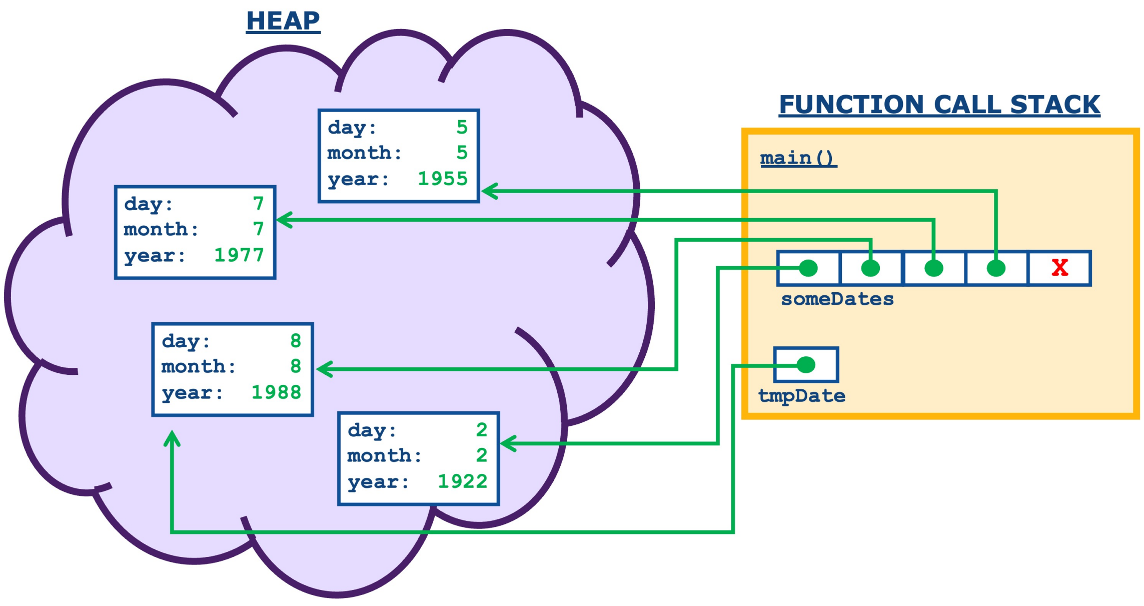


(b) After line 43, with `initDate()` called on line 14

Figure-4.8: Program-4.9 call stack and heap



(c) After line 15



(d) After line 23

Figure-4.8: Program-4.9 call stack and heap (cont.)

Program memory:

- Figure-4.8 shows the contents of both the function call stack and the heap during the execution of Program-4.9, before any memory is deallocated.
- Figure-4.8a represents the program memory after line 10, where one new `Date` object has been allocated and initialized:
  - the function call stack contains:
    - ➢ a statically allocated array of pointers called `someDates`, and
    - ➢ a single `Date` pointer called `tmpDate`, both stored in `main()`'s stack frame
  - the heap contains only one `Date` object at this point
- Figure-4.8b shows the program memory during the call to `initDate()` from line 14:
  - the function call stack shows the stack frames for both `main()` and `initDate()`
  - it's important to note that the double pointer parameter `dt` inside `initDate()`'s stack frame points to the `Date` pointer `tmpDate` declared in `main()`'s stack frame; this is how double pointers must be used to pass pointers by reference
  - the heap contains the two `Date` objects that have been allocated at this point
- Figure-4.8c illustrates the program memory after line 15, where the second new `Date` object has been allocated, initialized, and its address stored as the second element of the `someDates` array. The heap contains the two `Date` objects that have been allocated at this point.
- Figure-4.8d shows the program memory after line 23, where all four `Date` objects have been allocated, initialized, and their addresses stored as the first through fourth elements of the `someDates` array. The heap contains the four `Date` objects that are created in this program.

Part II

Basics of Object-Oriented Design

Chapter 5

Design Principles

We introduce some basic concepts of object-oriented design, including data abstraction, encapsulation, and the principle of least privilege.

5.1. Concepts

5.1.1. **Why object-oriented (OO) design:**

- Any software system worth building must be built *correctly*. This involves thinking through how the program functionality should be distributed over what objects, *before* we start coding.
- For every program we write, there are typically two or three correct ways to design it and hundreds of incorrect ways.
- Simply writing code that works is not sufficient. Professionally developed code must also follow the principles of good software engineering, and a significant portion of correct development must be planned at the design stage.

5.1.2. **For a good OO design, we need to think about:**

- What objects do we need, what data should they store, what behaviour should they have?
- Can we reuse classes from another source?
- What do our classes have in common with each other? Do they contain generalized or specialized data from a common class or from each other?
- What information should be hidden inside each class? What information should be available to other classes?

5.1.3. **Characteristics of good OO designs:**

- We design *single responsibility* classes:
  - our objects should have **one** purpose
- We follow the principles of *data abstraction*:
  - we separate **what** an object does from **how** it does it
- We incorporate *encapsulation* into our design:
  - we protect our runtime objects from bad code (our own or other developers')
- We follow the *principle of least privilege*:
  - we allow as little access to our runtime objects as possible

5.2. Data abstraction and encapsulation

5.2.1. Goal of *data abstraction*:

- We separate **what** an object does from **how** it does it.
- For example:
  - most people drive a car without understanding the mechanical principles behind how cars are built
  - a driver's conceptual idea of a car is much simpler than the engineering principles involved in car design and construction; this conceptual idea is called an *abstraction*
  - the same is true of software; our class user doesn't have to understand *how* our classes work, they simply need to know *what* they do
- We separate the class interface from the *implementation details*:
  - the class interface is the set of public properties and behaviours of our objects, and it is shared with our class user
  - the implementation details are private to the class developer and are *not* shared with the class user

5.2.2. *Encapsulation* is another important design concept:

- We hide the implementation details inside our classes. This includes the underlying data structures, algorithmic details, and so on.
- Our class user should never design their code based on our implementation details, only based on our class interface.
- We discuss some key tools for promoting encapsulation in chapter 8, including composition (aggregation), inheritance, and the principle of least privilege.

5.2.3. Why are data abstraction and encapsulation important:

- The biggest threat to timely software development is **change**:
  - clients change their mind about what they want
  - designers misunderstand the requirements, so the design is incorrect
  - clients want more features, which may have a significant impact on the design
- Change is always disruptive, so its consequences must be *minimized*.
- What developers can do to minimize the impact of changes:
  - we must design classes that can be modified with minimal impact on other classes
  - we can make sure our class users don't rely on our class implementation details

5.2.4. Approaches for good data abstraction:

- We must design our class interfaces to be simple and intuitive, easy to use, and not require knowledge of implementation details.
- We must group together common data and functionality.
- We must grant the least amount of access to other classes. Our data members should always be implemented with private or protected access specifiers only.
- We must maximize the reuse of existing code by finding the similarities between our classes and reusing the code that implements these similarities.
- We must design code that can be reused, within the same program and even across other applications and platforms.

5.2.5. Good data abstraction means separating object functionality:

- *Separation of concerns* is a design technique where each part of a software system focuses on its own distinct responsibilities. One example is the design of our objects into categories, as discussed in chapter 6.
- *Control objects* are responsible for the program control flow.
- *Boundary objects* conduct all communications with the end-user and with other systems or applications. These are sometimes called user-interface (UI) or view objects.
- *Entity objects* represent the real-world domain concepts in the program.
- *Collection objects* contain data structures and their related operations to store program data.

5.3. Principle of least privilege

5.3.1. What is the *principle of least privilege*:

- The *principle of least privilege* is an important OO design technique.
- It requires that:
  - we give the rest of the program minimal access to our objects, and *only as needed*
  - we never grant more access than is strictly required

5.3.2. Characteristics of the principle of least privilege:

- The principle applies to:
  - all class members
  - variables, parameters, objects
- In practical terms, it means:
  - all data members should have private or protected access specifiers only
  - we never provide getter and setter member functions for our classes, unless there exists no alternative design that can avoid them
  - helper member functions that are only used inside the class should have private or protected access specifiers
  - only the most minimal set of public member functions should be available to class users
  - the public member functions should provide a simple class interface that reveals no implementation details about the class

Chapter 6

Object Design Categories

We discuss how we can promote good data abstraction by designing classes that belong to object design categories, as introduced by Jacobson et al. [6]

6.1. Concepts

6.1.1. **What are *object design categories*:**

- *Object design categories* are an OO design approach for identifying and separating objects by area of responsibility.
- Each category of objects has specific tasks for which it is responsible.

6.1.2. **Why should we use object design categories:**

- Object design categories assist in distributing program functionality over objects.
- They promote data abstraction and encapsulation.
- They help with designing classes that are single-purpose and reusable.

6.1.3. **What are object design categories not:**

- They are **not** the same as the Model-View-Controller (MVC) architectural pattern.
- They are *structured* like MVC, in the sense that they are comprised of the same kinds of objects.
- They do not *behave* like MVC, in that they do not use the Observer design pattern.

6.2. Types of object categories

6.2.1. **What are the different object design categories:**

- Entity objects;
- Control objects;
- Boundary objects, also sometimes called view or user interface (UI) objects; and
- Collection objects.

6.2.2. **Entity objects:**

- An ***entity*** object represents real-world information maintained by the program.
- They often represent *persistent* information. Persistent objects survive between program executions. They are saved to persistent storage, such as a hard disk or solid-state drive.

- Examples of entity objects: `Library`, `Book`, `Patron`

6.2.3. **Control objects:**

- A control object is in charge of the program *control flow*, i.e. the sequence of operations.
- They manage how classes interact with each other.
- In a typical OO program, the `main()` function has two lines of code: it creates a control object, and it calls a member function of that object to launch the rest of the control flow.
- The control object then takes charge of the program control flow.
- Large programs may have multiple control objects, each with a different set of responsibilities.

6.2.4. **Boundary objects:**

- A boundary object is responsible for interactions with end-users and other systems.
- In this textbook, we focus only on end-user interactions, specifically user I/O.
- Ideally, no other classes in the program should communicate with the end-user.
- The use of boundary objects simplifies switching from one UI library to another.

6.2.5. **Why separate objects into design categories:**

- It's easier to make changes. Modifiability and extensibility are very important software engineering (SE) qualities.
- For example: entity objects can be reused between programs
  - if we model the `Book` class correctly, it should be reusable between different applications
- For example: replacing a UI simply means implementing new boundary objects
  - the entity objects should not change
  - the control objects should only require very simple changes

6.3. Collection classes

6.3.1. **What is a *collection*:**

- A *collection* is a generic term for a data structure that stores multiple instances of the same data type.
- There are two options for storing data in a collection: a primitive collection, or a collection class

6.3.2. **What is a *primitive collection*:**

- A *primitive collection* is a type of collection that is built directly into a programming language.
- These are very basic and have no special features or operators.
- An array is an example of a primitive collection that exists in most programming languages. We use the term ***primitive array*** in this textbook to denote this type of built-in collection.

6.3.3. **What is a *collection class*:**

- A ***collection class*** is a class whose only purpose is to store a collection of data and provide operations on it.
- In C++, it's sometimes also called a *container*.

- A collection class must use an internal data structure to store the data:
  - that internal data structure is called the *underlying container* or collection
  - it may be a primitive collection or an instance of another collection class
- There are libraries in C++ that provide collection classes, but these are not always efficient. They are discussed in chapter 15.

6.3.4. **Why use a collection class:**

- Why not simply use arrays everywhere? Let's think about the principle of least privilege.
- A primitive array does **not** restrict operations on itself. Any part of the program can add to, delete from, or access the elements of a primitive array.
- This is **bad** software engineering.

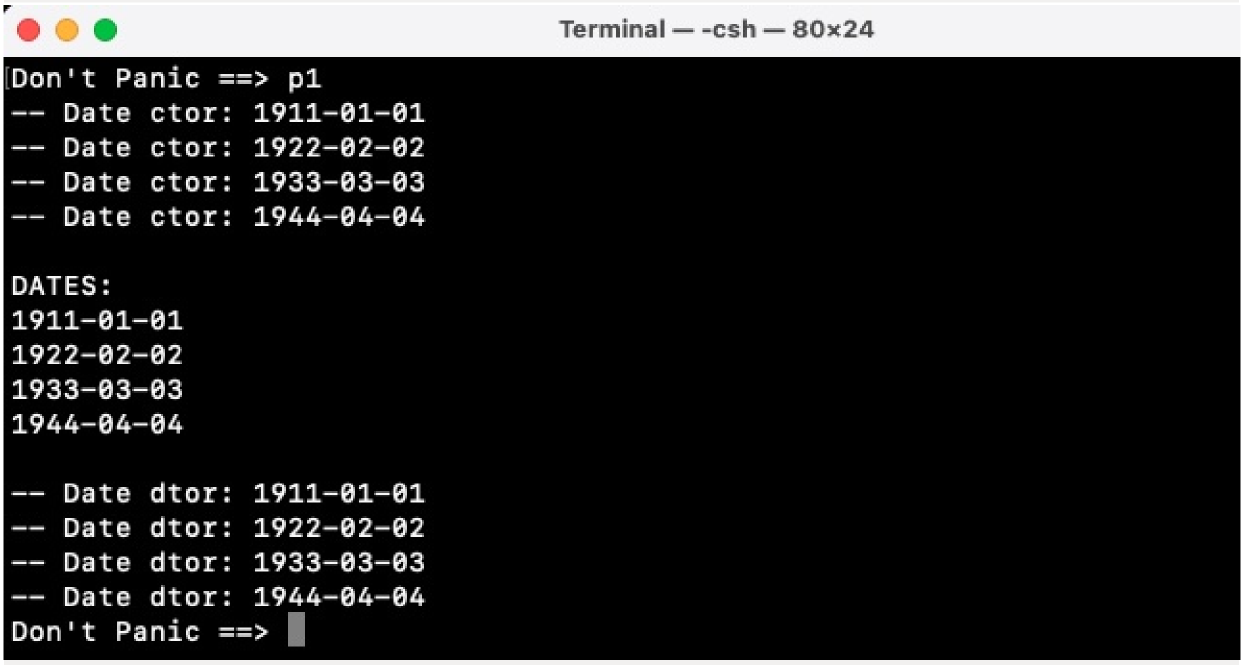
6.3.5. **Advantages of using a collection class:**

- A collection class effectively hides the data from the rest of the program.
- The collection class developer has full control over *how* the data is accessed and modified by choosing what member functions to make available.
- This is **good** software engineering.

6.4. Coding example: Collection classes

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Array.h            *
 * * * * * * * * * * * * * * * * */
#define MAX_ARR_SIZE 64

class Array
{
  public:
    Array();
    ~Array();
    void add(Date*);
    void print();
  private:
    Date* elements[MAX_ARR_SIZE];
    int   size;
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Array.cc           *
 * * * * * * * * * * * * * * * * */
Array::Array()
{
  size = 0;
}

Array::~Array()
{
  for (int i=0; i<size; ++i) {
    delete elements[i];
  }
}
```

```
void Array::add(Date* d)
{
  if (size >= MAX_ARR_SIZE)
    return;

  elements[size++] = d;
}

void Array::print()
{
  cout << endl << "DATES:" << endl;
  for (int i=0; i<size; ++i) {
    elements[i]->print();
  }
  cout<<endl;
}

/* * * * * * * * * * * * * * * * * *
 * Filename:  main.cc              *
 * * * * * * * * * * * * * * * * * */
int main()
{
  Array arr;
  Date* d;

  d = new Date(1,1,1911);
  arr.add(d);
  d = new Date(2,2,1922);
  arr.add(d);
  arr.add(new Date(3,3,1933));
  arr.add(new Date(4,4,1944));
  arr.print();

  return 0;
}
```



```
Don't Panic ==> p1
-- Date ctor: 1911-01-01
-- Date ctor: 1922-02-02
-- Date ctor: 1933-03-03
-- Date ctor: 1944-04-04

DATES:
1911-01-01
1922-02-02
1933-03-03
1944-04-04

-- Date dtor: 1911-01-01
-- Date dtor: 1922-02-02
-- Date dtor: 1933-03-03
-- Date dtor: 1944-04-04
Don't Panic ==>
```

Program-6.1: Collection classes

Program purpose:

- Program-6.1 demonstrates the implementation and usage of a collection class called `Array`.
- The `Date` class used in this program is identical to Program-4.4.

Lines 6-16:

- These lines contain the class definition for the `Array` collection class.
- Line 14 shows that the collection class uses a primitive, statically allocated array as the underlying container to store `Date` object pointers.
- Line 15 declares a `size` data member to track of the current number of elements in the array.
- Lines 9-12 show that the collection class provides a default constructor, a destructor, and member functions for adding a new `Date` pointer to the array and for printing the entire array.

Lines 21-47:

- These lines show the member function implementations for the `Array` class.
- Lines 21-24 implement a default constructor that initializes only the `size` data member to indicate that a new array contains no elements.
- The `elements` primitive array of pointers does *not* require initialization, because correctly written code should never access array elements beyond the current `size`.
- Lines 26-31 show the destructor deallocating the array's `Date` objects stored in the heap.
- It is common practice for collection class destructors to clean up their elements. The collection class is often the only place that holds pointers to this memory.
- Lines 32-38 show the `add()` member function that adds a new element to the end of the array.
- Lines 40-47 implement the `print()` member function that prints to the screen every element of the array.
- It is important to note that the `print()` member function does **not** contain any knowledge of how to print a `Date` object. The only class in the program that knows how to print a `Date` object is the `Date` class itself, specifically its `print()` member function.
- Ensuring that the `Array` class's job is to know about `Array` objects *only* is a fundamental example of correct data abstraction and encapsulation.

Lines 52-66:

- These lines show the implementation of the `main()` function.
- To maintain correct encapsulation, the `main()` function knows no details about any class. The `Array` class provides a simple class interface for `main()` to create and manipulate an `Array` object by calling its member functions.
- Line 54 declares a new `Array` object, and its constructor initializes it.
- Lines 57, 59, 61, and 62 each dynamically allocate a new `Date` object.
- Lines 58, 60, 61, and 62 each add a `Date` object pointer to the array.
- Line 63 prints out the contents of the array.

Chapter 7

UML Class Diagrams

We introduce the concepts of Unified Modelling Language (UML) class diagrams, which are used to represent an OO software design graphically, as introduced by Booch et al. [7] We show how to represent classes and their relationships in a UML class diagram.

7.1. Concepts

7.1.1. What is the *Unified Modelling Language (UML)*:

- The *Unified Modelling Language (UML)* is a family of notations that represent OO models.
- A *model* is a virtual representation of something, in our case software.
- UML is an important tool for documenting OO designs. As a modelling *language*, its goal is to facilitate communication between software developers.
- UML is **programming language independent**. In principle, a design that's based on a UML diagram can be implemented in any OO programming language.
- There are many types of UML design diagrams, but we focus on UML *class diagrams*.

7.1.2. UML class diagrams are used to represent:

- Classes, including:
  - *class attributes*: this is the general term for data members or instance variables
  - *class operations*: this is the general term for member functions or methods
- Class relationships, including:
  - inheritance
  - composition/aggregation
  - dependency (the "uses" relationship)
- In this textbook, we focus on the inheritance and composition/aggregation class relationships.

7.2. Representing classes in UML

7.2.1. Showing classes:

- Each class consists of three vertically stacked boxes:
  - the top box contains the class name
  - the middle box shows the class attributes (data members, instance variables)
  - the bottom box shows the class operations (member functions, methods)

- If a class is abstract, its name must be *italicized*.
- We **never** show collection classes in a UML class diagram. Instead, a composition relationship must be shown between the class that contains the collection and the class of the collection elements. For example, if a `Library` class contains a `BookArray` object that contains `Book` objects or pointers, we *do not* shown the `BookArray` class at all! Instead, we use a composition class relationship to show that a `Library` contains multiple `Book`s.
- The name of every attribute and operation must be preceded by its access specifier:
  - private is represented with a dash (−)
  - protected with a hash symbol (#)
  - public with a plus symbol (+)

7.2.2. **Showing attributes:**

- Each attribute is shown with its access specifier and name, followed by a colon (`:`) and the attribute's data type.
- We **never** show other objects among a class's attributes. If a class has an attribute that's an object of another class, this must be shown as a relationship between the two classes instead.
- UML **does not** use the C++ variable declaration syntax.

7.2.3. **Showing operations:**

- Each operation is shown with its access specifier, name, and list of parameters, followed by a colon and the operation's return type.
- Each parameter in the list must indicate the parameter direction (in, out, or inout) and name, followed by a colon and the parameter's data type.
- UML **does not** use the C++ function prototype syntax.

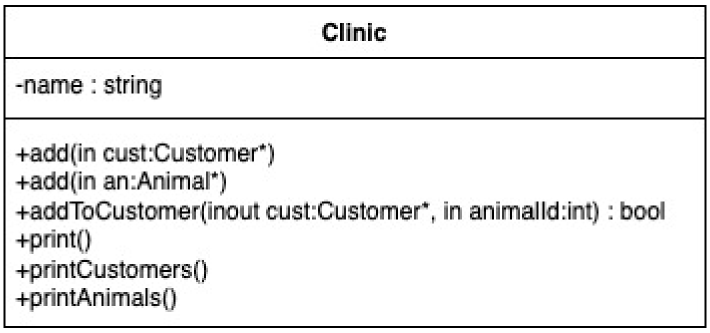


Figure-7.1: UML class diagram with a single class

7.2.4. **UML class example:**

- Figure-7.1 shows an example of how a single class is represented in a UML class diagram.
- The top box specifies the class name `Clinic`.
- The middle box indicates that the `Clinic` class has one attribute that isn't an instance of another class. Remember, data attributes that are objects are *not* shown as attributes, but using class relationships instead. We see that the `name` attribute is a string, and it's private to the `Clinic` class.
- The bottom box shows the six operations for the `Clinic` class, all of which are public. Each operation lists its parameters, and it specifies the direction, name, and data type of each one. The `Clinic` class and its operations are further explained in a later example.

7.3. Representing class relationships in UML

7.3.1. **Types of class relationships:**

- A *class relationship* is a relationship between two classes in an OO design.
- There are several types of relationships between classes:
  - inheritance, which is more formally called generalization / specialization
  - association, which includes composition and aggregation
  - dependency, which is sometimes called a "uses" relationship
- The differences between composition and aggregation are subtle and outside the scope of this textbook. We use the term *composition* to denote both.

7.3.2. **Showing class relationships:**

- In our class diagrams, we show *inheritance* and *composition* relationships only.
- A dependency relationship is represented in UML with a dashed line between two classes, where one class uses the services of the other. For example, one class may call the operations of the other class, or use temporary instances of it as parameter.
- We do not discuss dependency relationships in this textbook, as they are a much weaker type of relationship between classes than inheritance or composition. We leave it to upper year software engineering courses to delve into the degrees of dependency in class relationships.

7.3.3. **What is *inheritance*:**

- *Inheritance* is an *is-a* relationship between two classes, where one class is a specialization of another, more general class. We discuss inheritance further in chapter 9.
- The *superclass* is the generalized class, and it's sometimes informally called the "parent" class.
- The *subclass* is the specialized class, sometimes called the "child" class.
- Inheritance is shown in UML with a line between the two classes, and a clear (not filled) triangle at the superclass end of the line.

7.3.4. **What is *composition*:**

- *Composition* is a *has-a* relationship between two classes, where each instance of one class contains one or more instances of the other class.
- The *container* is the class that contains one or more instances of the other class.
- The *containee* is the class that is contained within the other one.
- Composition is shown in UML with a line between two classes, based on directionality and multiplicity as described below.

7.3.5. **What is *directionality*:**

- *Directionality* in a composition relationship indicates which class is the container and which one is the containee.
- Composition relationships may be:
  - unidirectional (one-way), or
  - bidirectional (two-way)
- In a unidirectional relationship:
  - the container has one or more instances of the containee, and
  - the containee does **not** know about the container; it doesn't even know that it is a containee

- A unidirectional relationship is shown with an open-headed arrow *from container to containee*.
- In a bidirectional relationship:
  - the container has one or more instances of the containee, and
  - the containee has one or more instances of the container
- A bidirectional relationship is shown with a simple line *with no arrows*.
- Directionality is a property of *composition relationships only*. Inheritance relationships **do not** have directionality.

7.3.6. What is ***multiplicity***:

- *Multiplicity* in a composition relationship indicates how many instances of the containee are stored in the container.
- Values may be a single number or an asterisk (`*`) to represent "many", or it may be a range.
- A range of values is represented with a lower bound, followed by two dots, followed by an upper bound.
- If the container has a collection of containee objects, we *do not* show a specific upper bound. We use the asterisk to denote "many" instead.
- Multiplicity is shown at the containee end of the relationship.
- Multiplicity is a property of *composition relationships only*. Inheritance relationships **do not** have multiplicity.

7.4. Example

Figure-7.2 shows an example of a design that is separated into control, boundary, and entity classes, with both inheritance and composition relationships. The design is for a program that manages the data for a veterinary clinic, including customer and animal information.

The `Control` class:

- This class manages the data and control flow for the entire program.
- It contains one instance of the clinic that it manages and one instance of the boundary object (the `View` class) that is responsible for most communication with the end-user.
- We see that the multiplicity of the composition relationships indicates that the control object contains a single instance each of the clinic and and view objects.
- The composition relationships are also both shown as unidirectional. The control object knows about the clinic and the view, but they have no knowledge of or access to the control object.
- The public `launch()` operation manages the program's overall control flow.
- The private `initCustomers()` and `initAnimals()` operations are helpers that initialize the program data.

The `View` class:

- This class manages most communications with the end-user.
- It contains operations that print out the main menu to the end-user and perform basic end-user I/O.

The `Clinic` class:

- This is the primary entity class in the program.
- It has an attribute to store the clinic name.
- From the composition relationships and their multiplicity, we see that the clinic also contains two collections: a customer collection and an animal collection.
- The customer collection stores data on the human clients of the clinic, and the animal collection contains information on all the customers' pets.
- We note that the exact types of collections are *not* specified. UML class diagrams are created at the software design stage, but choices of collection types are implementation decisions.
- The diagram also does not specify whether the collections store objects or object pointers, since that is also an implementation decision. However, it's always a better design choice to use pointers, so we can assume the use of pointers as collection elements.
- The two `add()` operations allow a class user to add a new element to the clinic's customers and animals collections.
- The `addToCustomer()` operation establishes a new adoption relationship between a customer and an animal. It adds the animal with the given id to the given customer's collection of adoptees, and it sets the animal's parent to that same customer.
- The `print()` operations print out the corresponding data to the screen. Alternatively, we could design the `View` class to print out the clinic data, but instead we choose to encapsulate it entirely inside the `Clinic` class.

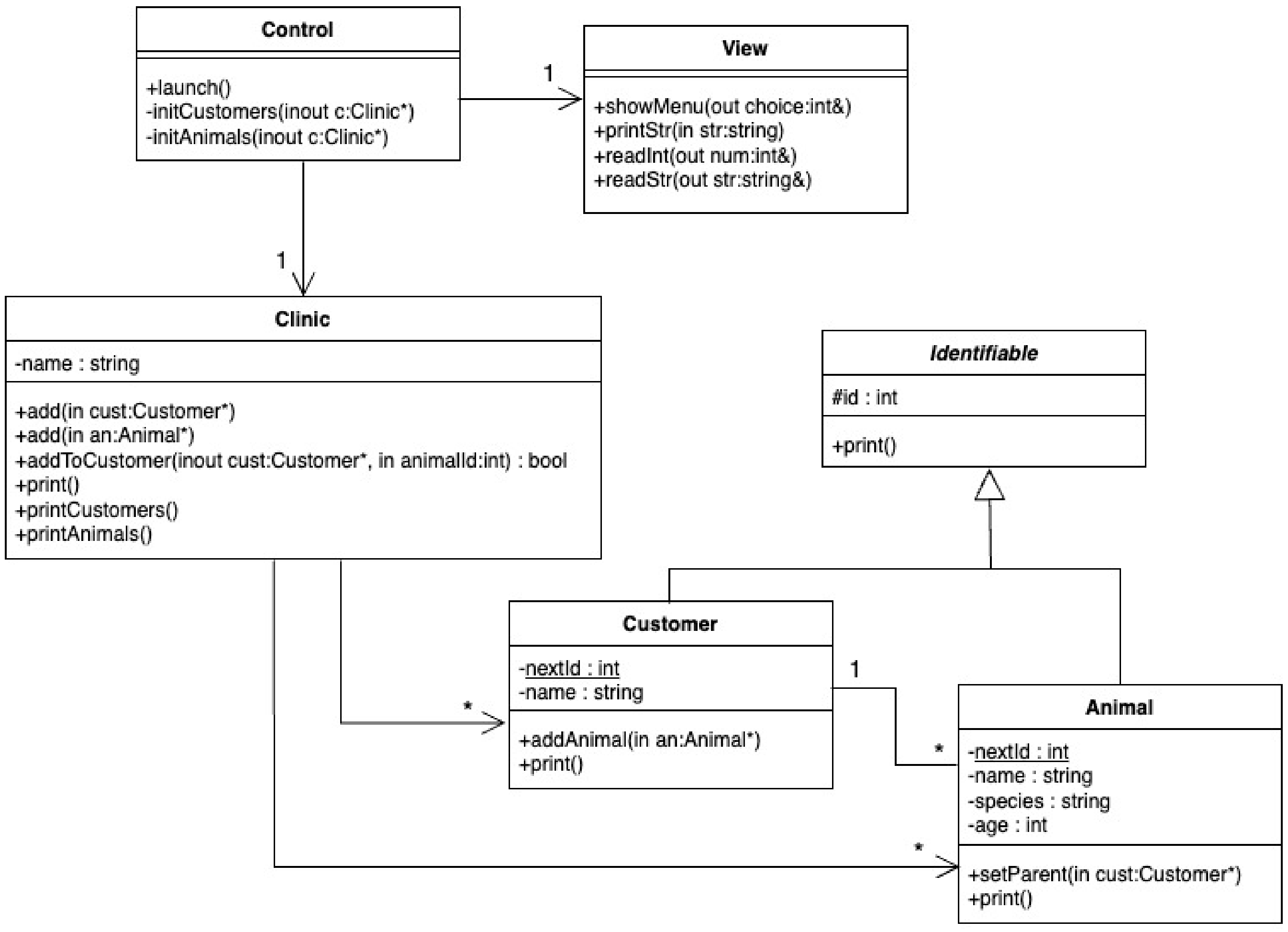


Figure-7.2: UML class diagram with class relationships

The `Identifiable` class:

- This class is used as a superclass for all objects that have a unique identifier.
- It contains a unique id attribute.
- It's the role of the subclass to track the id to be assigned to the next subclass object created. However, this superclass assigns the id and ensures their sequential incrementation.

The `Customer` class:

- This class represents a single human client of the clinic.
- It is a subclass of the `Identifiable` class, as we can see from the clear triangle.
- Each customer has three attributes: the next id to be assigned to a new customer object; the customer's name; and a collection of the customer's pets.
- The underlining of the next id attribute indicates that it is a static attribute that belongs to the class as a whole, instead of a unique value belonging to each instance. We discuss static class members in chapter 8.
- The `Customer` and `Animal` classes have a bidirectional composition relationship, which means that each serves as both a container and a containee to the other.
- We see that each customer's animal collection is represented with multiplicity of "many".

The `Animal` class:

- This class represents a single animal in the clinic, and it is a subclass of the `Identifiable` class.
- Each animal has five attributes: the next id to be assigned to a new animal object, as a static class member; the animal's name, species and age; and a reference to the customer object to which the animal belongs, as a pointer to avoid object duplication.
- We see that each animal has one parent, based on the multiplicity of 1.

Important conventions:

- We never show collection classes in UML. They are implied using a composition relationship between the container and containee classes, using a multiplicity of "many".
- Abstract class names are shown in *italics*.
- Inheritance relationships *do not* have directionality or multiplicity. Only composition relationships do.
- In C++, *friendship* is not shown in a UML class diagram. Friendship is a concept unique to C++, and not to OO programming in general, and it is discussed in chapter 8.
- We usually do not show constructors, destructors, or simple getter/setter operations in a UML class diagram.

Part III

Essential Object-Oriented Techniques

Chapter 8

Encapsulation

Most OO programming languages provide standard design and implementation techniques that help promote the encapsulation of data and behaviour into objects. In this chapter, we discuss how C++ implements these essential techniques, including composition class relationships, the use of constants, friendship, static class members, and namespaces. We also delve into a detailed design and implementation example of linked lists that synthesizes several of the concepts discussed in this textbook so far.

8.1. Composition

Composition relationships allow for the design of classes that work together in a container-containee capacity. The design goal is for both classes to maintain their independence from each other by encapsulating data and behaviour in the class where they belong.

We show how composition relationships are implemented between classes in C++, as well as the special syntax used to prevent the unnecessary creation of temporary objects. We discuss how the construction and destruction of objects work within the container-containee class relationship.

8.1.1. Concepts

8.1.1.1. **What is a *composition* relationship:**

- A *composition* relationship between two classes means that one class declares a data member that's an object or a pointer to an object of another class.
- Composition is also sometimes informally called a **has-a** relationship between two classes.
- The ***container*** class in a composition relationship is the one that contains one or more instances of another class.
- The ***containee*** class is is the one contained within an instance of the container class. The containee may be declared as an actual object or a pointer to an object.
- For example, if the `Student` class declares a data member that's an instance of the `Address` class, then the two classes have a composition relationship. In this case, `Student` is the container class, and `Address` is the containee.

8.1.1.2. **Container and containee object construction:**

- When an object is allocated, either statically or dynamically, its constructor is always called.
- If the new object is a container, then a call to its constructor automatically triggers a call to the containee constructor.

- If a container object has several containees, the containee constructors are called in the order in which they are declared in the container class definition.
- It is convention for the container constructor to take all the parameter values required to initialize both the container and the containee data members.
- To uphold correct encapsulation, the container constructor should *only* initialize its own data members. The containee data members must be initialized with a call to the containee constructor that uses the values provided as parameters to the container constructor.
- The **correct** approach for a container constructor to pass parameters to a containee constructor is to use *member initializer syntax*. All other approaches result in the creation of temporary containee objects, which is a waste of computational resources.

NOTE: There are **many** cases in C++ where the default syntax will result in the automatic creation of duplicate or temporary objects. These are unnecessary and *should be avoided*. It's important to understand what syntax triggers the creation of these extra objects, so that it can be avoided.

8.1.1.3. **What is *member initializer syntax*:**

- We use ***member initializer syntax*** to have a container object constructor call a containee constructor with provided parameter values, during the construction of the container object.
- Without this syntax, the following situations arise:
  - the containee's default constructor is called automatically to initialize the containee data members with default values, instead of the values provided as parameters to the container constructor
  - in order for the container constructor to do its job of initializing the containee data members, the containee class must provide setter functions for all its data members
- Both situations are examples of **bad** software engineering. Using member initializer syntax avoids both issues entirely.
- Member initializer syntax appears with the container constructor implementation, between the prototype and the opening brace of its implementation. It begins with a colon (`:`), followed by the containee data member name, as declared in the container class definition, followed by the list of containee constructor parameter values within a pair of parentheses.
- Member initializer syntax can also be used for the initialization of non-object data members. In this case, the syntax is also located between the constructor prototype and its implementation. It begins with a colon (`:`), followed by the data member name, followed by the value to be assigned to the data member, within a pair of parentheses.
- If a constructor initializes multiple data members and/or containee objects using member initializer syntax, the initializations must be separated with a comma (`,`) which is the *sequencing* operator.

8.1.1.4. **Order of execution with composition relationships:**

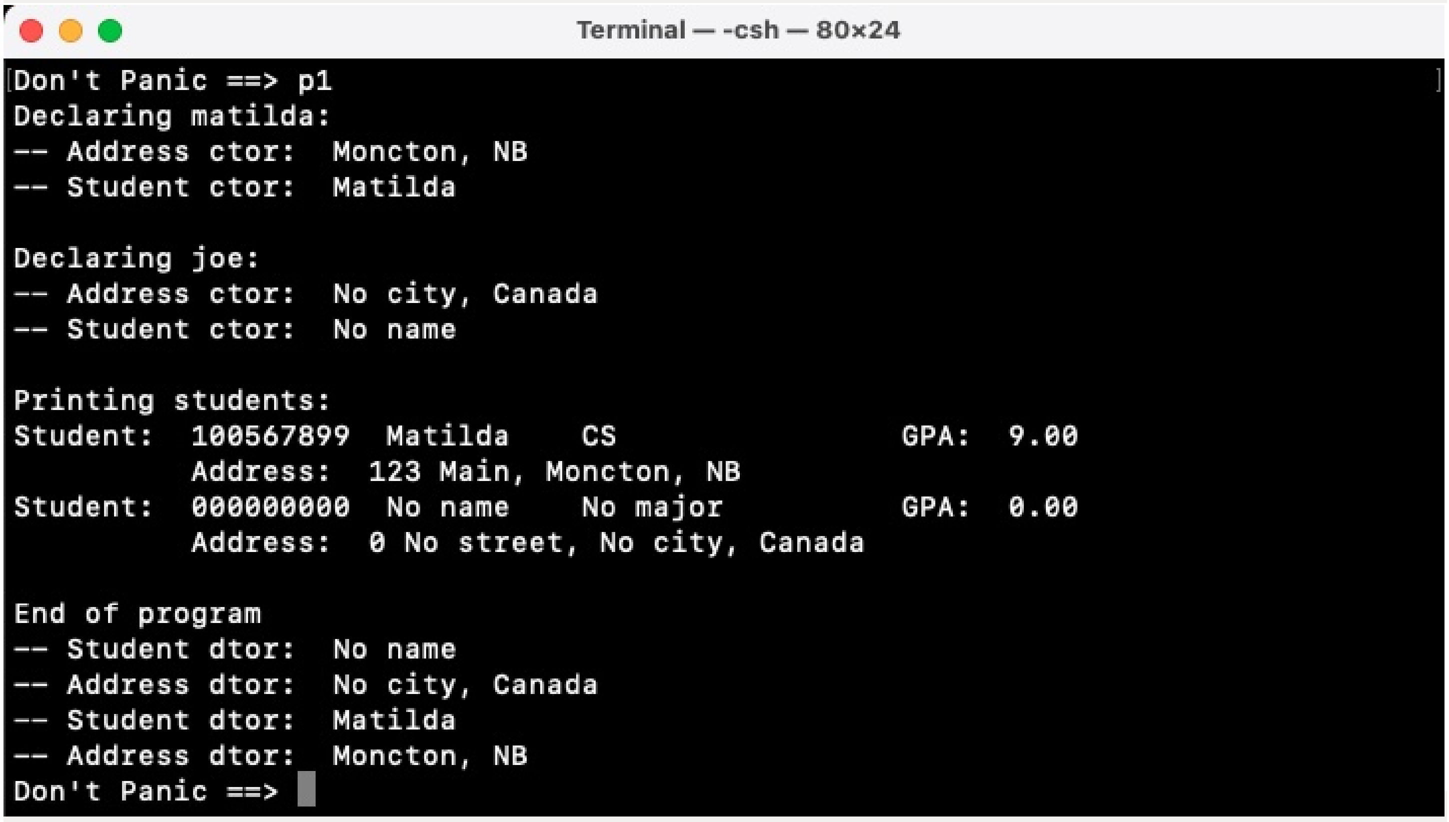
- Constructors:
  - objects are built from the *inside out*
  - the containee object(s) are constructed first, in order of declaration in the class definition
  - the container object constructor executes last
- Destructors:
  - objects are destroyed from the *outside in*
  - the container destructor executes first, then the containee destructor(s)
  - destructors are usually invoked in the reverse order of constructors

8.1.2. Coding example: Member initializer syntax

```
/* * * * * * * * * * * * * * * * * *
 * Filename:  Address.h           *
 * * * * * * * * * * * * * * * * */
class Address
{
  public:
    Address(int=0, string="No street", string="No city", string="Canada");
    ~Address();
    void print();

  private:
    int    number;
    string street;
    string city;
    string province;
};

/* * * * * * * * * * * * * * * * * *
 * Filename:  Address.cc          *
 * * * * * * * * * * * * * * * * */
Address::Address(int n, string s, string c, string p)
{
  number   = n;
  street   = s;
  city     = c;
  province = p;

  cout<<"-- Address ctor:  "<< city << ", " << province << endl;
}

Address::~Address()
{
  cout<<"-- Address dtor:  "<< city << ", " << province << endl;
}

void Address::print()
{
  cout<<"          Address:  "<<number<<" "<<street<<", "<<city<<", "<<province;
}

```

```cpp
/* * * * * * * * * * * * * * * * *
 * Filename:  Student.h          *
 * * * * * * * * * * * * * * * * */
class Student
{
  public:
    Student(string="000000000", string="No name", string="No major",
            float=0.0f, int=0, string="No street", string="No city",
            string="Canada");
    ~Student();
    void print();

  private:
    string  number;
    string  name;
    string  majorPgm;
    float   gpa;
    Address homeAddr;
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Student.cc         *
 * * * * * * * * * * * * * * * * */
/*  Version 1:  Uses member initializer syntax for containee ctor  */
Student::Student(string s1, string s2, string s3, float g,
                 int n, string s, string c, string p)
    : homeAddr(n,s,c,p)
{
  cout<<"-- Student ctor:  "<< s2 <<endl;
  number   = s1;
  name     = s2;
  majorPgm = s3;
  gpa      = g;
}
/*  Version 2:  Uses member initializer syntax for everything  */
/*
Student::Student(string s1, string s2, string s3, float g,
                 int n, string s, string c, string p)
    : number(s1), name(s2), majorPgm(s3), gpa(g), homeAddr(n,s,c,p)
{
  cout<<"-- Student ctor:  "<< s2 <<endl;
}
*/

Student::~Student()
{
  cout<<"-- Student dtor:  "<<name <<endl;
}
void Student::print()
{
  cout << "Student:  " << number << "  " << left << setw(10) << name << " "
       << setw(15) << majorPgm << "   GPA: "
       << fixed << setprecision(2) << setw(5) << right << gpa << endl;
  homeAddr.print();
  cout << endl;
}
```

```
/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc            *
 * * * * * * * * * * * * * * * * */
int main()
{
  cout << "Declaring matilda:" << endl;
  Student matilda("100567899", "Matilda", "CS", 9.0f, 123, "Main",
                  "Moncton", "NB");
  cout << endl;
  cout << "Declaring joe:" << endl;
  Student joe;
  cout << endl;

  cout << "Printing students:" << endl;
  matilda.print();
  joe.print();

  cout<< endl << "End of program" << endl;
  return 0;
}
```


Terminal — -csh — 80×24

```
Don't Panic ==> p1
Declaring matilda:
-- Address ctor:  Moncton, NB
-- Student ctor:  Matilda

Declaring joe:
-- Address ctor:  No city, Canada
-- Student ctor:  No name

Printing students:
Student:  100567899  Matilda    CS                 GPA:  9.00
          Address:  123 Main, Moncton, NB
Student:  000000000  No name    No major           GPA:  0.00
          Address:  0 No street, No city, Canada

End of program
-- Student dtor:  No name
-- Address dtor:  No city, Canada
-- Student dtor:  Matilda
-- Address dtor:  Moncton, NB
Don't Panic ==> 
```

Program-8.1: Member initializer syntax

Program purpose:

- Program-8.1 demonstrates the *correct* construction of container/containee objects, by using member initializer syntax.
- In this example, the `Student` class is the container, and the `Address` class is the containee. So every `Student` object created has its own `Address` object contained inside it.

Lines 4-16:

- These lines show the `Address` class definition.
- An address has four data members, as shown on lines 12-15: the house number, street name, city, and province.
- We see that the class definition contains no setter functions for the data members. If this solution didn't use member initializer syntax, we would be forced to provide setters, which would violate the principle of least privilege.

Lines 21-39:

- These lines show the implementations of the `Address` member functions.
- Lines 21-29 contain the default constructor, which initializes every data member from the provided parameters.

Lines 44-59:

- These lines show the `Student` class definition.
- A student has five data members, as shown on lines 54-58: the student number, their name, major, gpa, and home address as an `Address` object.
- Line 58 establishes the composition relationship between the two classes, by declaring an `Address` object as a data member of the `Student` class.
- We see on lines 47-49 that the `Student` constructor takes all the parameters required to initialize **both** the new `Student` object and its containee `Address` object.

Lines 65-74:

- These lines show the correct implementation of the `Student` default constructor.
- The constructor is responsible for initializing the entire `Student` object, including calling the constructor for its containee `Address` object using the correct parameters.
- **Issue:** In a composition relationship, the containee object constructor is called **before** the first line of the container object constructor. So by the time we reach line 69, the containee `Address` object has *already been constructed* using default values.
- Without member initializer syntax, we would be forced to implement and call setter member functions to initialize the `Address` data members *a second time* from inside the `Student` constructor. This would be a violation of the principle of least privilege.
- However, this solution **does** use member initializer syntax on line 67. This line explicitly calls and passes parameter values to the `Address` object constructor **before** the first line of the container `Student` object constructor on line 69.
- The syntax for member initializer syntax must appear after the constructor prototype and before the constructor body. It begins with a colon (`:`), followed by the containee data member name `homeAddr` (as declared in the `Student` class definition on line 58), followed by the `Address` constructor parameter values within a pair of parentheses.
- In this version of the `Student` constructor, the other data members are initialized in the body of the constructor, as usual.

Lines 77-82:

- These lines show an alternative, commented-out version of the `Student` default constructor.
- In this version, *all the `Student` data members* are initialized using member initializer syntax on line 79. First, all four non-object data members are initialized, and then the `Address` constructor is called with the correct parameter values.
- The member initializer syntax must appear after the constructor prototype and before the constructor body. The syntax begins with a colon (`:`), followed by the data member name, followed by the value to be assigned to the data member, within a pair of parentheses.

- With this syntax, it is expected that the body of the constructor would be empty, if we did not have a print statement for debugging purposes. This is a normal occurrence in C++, even if it may look somewhat unusual.

Lines 89-96:

- These lines show the implementation of the `Student` printing function.
- This member function must print out all the data members of the `Student` object to the screen, and this must include the data members of the containee `Address` object. However, it would be **bad** encapsulation for the `Student` class to have knowledge about how to print an `Address` object.
- The correct solution is seen on line 94, where the `Student` printing function calls the `Address` class's printing function to output the `Address` object's data members.

Lines 100-116:

- These lines show the implementation of the `main()` function.
- We see from lines 103-104 and the program output that the declaration of the `matilda` object with parameter values for all `Student` and `Address` data members results in both the `Student` and `Address` constructors being called.
- We also see from the output that objects are constructed *from the inside out*, with the containee object constructed first, then the container object.
- The program output also shows that the `Address` object is correctly initialized with parameter values immediately on creation. It is **not** initialized with default values, as it would be without member initializer syntax.

8.2. Constants

The use of constant objects, constant member functions, and constant data members in C++ protects the integrity of our runtime objects and ensure they are not misused. We discuss how these mechanisms are implemented in C++.

8.2.1. Concepts

8.2.1.1. **What are *constants* in C++:**

- Constants are used to *protect* our runtime objects from bad code, either our own or code written by other programmers for the same software system.
- Constants help to promote encapsulation and enforce the principle of least privilege by minimizing access to our objects from the rest of the program.
- In C, we declare constant variables and parameters with the `const` keyword.
- C++ provides three additional ways to use constants:
  - constant objects
  - constant member functions
  - constant data members

8.2.1.2. **Properties of constant variables:**

- In C/C++, once a constant variable is declared, its value can *never* be modified.
- A constant variable can never be used as an *lvalue*.

- In programming, an *lvalue* can be used on the left-hand side of an assignment operation:
  - a valid lvalue must represent an area of storage, for example a variable
  - an expression or a literal cannot be used as an lvalue
- An *rvalue* can be used on the right-hand side of an assignment operation:
  - a valid rvalue can represent an area of storage, or an expression, or a literal
  - all lvalues can be used as rvalues, but not all rvalues can be used as lvalues
- A constant variable or object must be assigned a value *on initialization*.

8.2.2. Constant objects

8.2.2.1. **What is a *constant object*:**

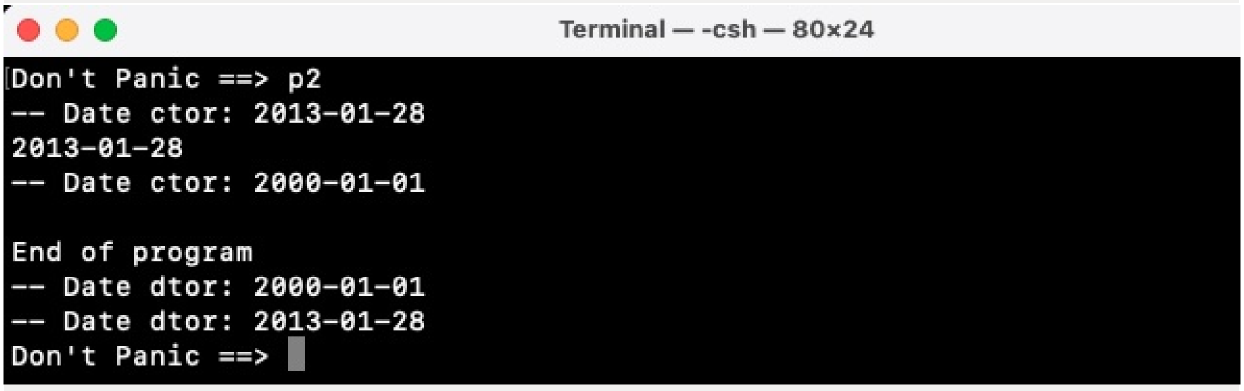
- A *constant object* is an object that is declared as a constant variable.
- Once initialized, the object's data members can *never be modified*.
- No part of the program is allowed to modify the members of a constant object, not even the class's own member functions.
- Only *constant member functions* can be called on a constant object, but they cannot modify its members. Constant member functions are discussed in the next section.

8.2.2.2. **Applying the principle of least privilege:**

- We need to think about what objects are required in a program.
- If an object does not need to be modified after initialization, we make it a constant object.
- This guarantees the integrity of the object and protects it from bad code.

8.2.3. Coding example: Constant objects

```cpp
int main()
{
  Date d1(28, 1, 2013);
  d1.print();

  const Date newYear(1, 1);
//  newYear.setDate(2, 2, 2013);
//  newYear.print();

  cout<< endl << "End of program" << endl;
  return 0;
}
```



```
Don't Panic ==> p2
-- Date ctor: 2013-01-28
2013-01-28
-- Date ctor: 2000-01-01

End of program
-- Date dtor: 2000-01-01
-- Date dtor: 2013-01-28
Don't Panic ==>
```

Program-8.2: Constant objects

Program purpose:

- Program-8.2 demonstrates the declaration and usage of constant and non-constant objects.
- The `Date` class is identical to Program-3.4, with the addition of a destructor that prints out a debugging message.

Lines 3-4:

- These lines show the declaration and printing of a non-constant `Date` object called `d1`.

Lines 6-8:

- Line 6 shows the declaration of a *constant object* called `newYear`. The `Date` constructor is called with parameter values to initialize the object's data members.
- The constructor is the only non-constant member function that is allowed to execute on a constant object. Once the constructor has finished executing, *no other member functions* can be called on a constant object, unless they are *constant member functions*.
- Lines 7-8 are commented out because they cause a compilation error. The member functions called are not allowed on a constant object.

8.2.4. Constant member functions

8.2.4.1. **What is a *constant member function*:**

- A *constant member function* is a member function that is allowed to access the members of a constant object, on a read-only basis.
- Declaring a member function as constant is a guarantee that this function does **not** modify any part of the object.

8.2.4.2. **Characteristics of constant member functions:**

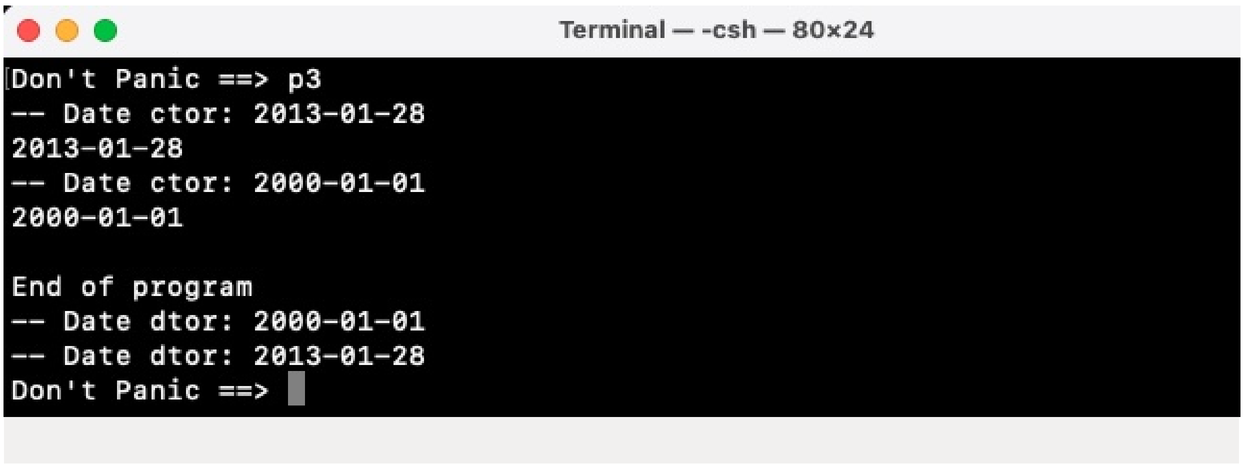
- Constant member functions are the *only* functions that can be called on a constant object, except for the constructor and destructor.
- They are not allowed to modify the value of any data member.
- They are not allowed to call a non-constant member function.
- Constructors and destructors cannot be constant, but they can be called on constant objects.

8.2.4.3. **Applying the principle of least privilege:**

- We need to think about what member functions are required for a class.
- If a member function doesn't need to modify the object, it should be made constant. This ensures that it can be called on constant objects.
- Declaring a member function as constant makes our class more usable by other programmers, if they need to declare constant objects in their code.

8.2.5. Coding example: Constant member functions

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Date.h             *
 * * * * * * * * * * * * * * * * */
class Date
{
  public:
    Date(int=0, int=0, int=2000);
    ~Date();
    void print() const;

  private:
    int  day;
    int  month;
    int  year;
    void setDate(int, int, int);
    int  lastDayInMonth(int, int);
    bool leapYear(int);
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Date.cc            *
 * * * * * * * * * * * * * * * * */
void Date::print() const
{
  cout << setfill('0') << setw(4) << year << "-"
       << setfill('0') << setw(2) << month << "-"
       << setfill('0') << setw(2) << day << endl;
}
/* The rest of the member functions are not shown */

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc            *
 * * * * * * * * * * * * * * * * */
int main()
{
  Date d1(28, 1, 2013);
  d1.print();

  const Date newYear(1, 1);
  newYear.print();

  cout<< endl << "End of program" << endl;
  return 0;
}
```



```
Don't Panic ==> p3
-- Date ctor: 2013-01-28
2013-01-28
-- Date ctor: 2000-01-01
2000-01-01

End of program
-- Date dtor: 2000-01-01
-- Date dtor: 2013-01-28
Don't Panic ==> 
```

Program-8.3: Constant member functions

Program purpose:

- Program-8.3 demonstrates the use of constant member functions.

Lines 4-18:

- These lines show the `Date` class definition.
- Line 9 shows the function prototype for `print()`, which is declared as a constant member function. The `const` keyword must appear at the end of the member function prototype.
- Once a member function has been declared as constant, the `const` qualifier becomes part of the function prototype. It must appear everywhere that the prototype is used.

Lines 23-28:

- These lines show the implementation of the `print()` member function.
- On line 23, we see the `const` qualifier appearing in the function prototype, just as in the class definition on line 9.

Lines 34-44:

- These lines show the implementation of the `main()` function.
- Lines 36-37 show the declaration and printing of a non-constant `Date` object called `d1`.
- Line 39 shows the declaration of a constant object called `newYear`. The `Date` constructor is called with parameter values to initialize the object's data members.
- On line 40, the `print()` member function is called on the `newYear` constant object. This is allowed because it's declared as a constant member function. The program output shows that the constant object's data is printed correctly.

8.2.6. Constant data members

8.2.6.1. **What is a *constant data member*:**

- A *constant data member* is a data member that can *never* be modified.
- A constant data member cannot be modified by any of the class member functions, *not even the constructor*.

8.2.6.2. **Characteristics of constant data members:**

- Because a constant data member can never be modified, even inside the constructor, it must be initialized *before* the constructor body.
- As a result, its initialization **must** use member initializer syntax.

8.2.6.3. **Constant data members and member initializer syntax:**

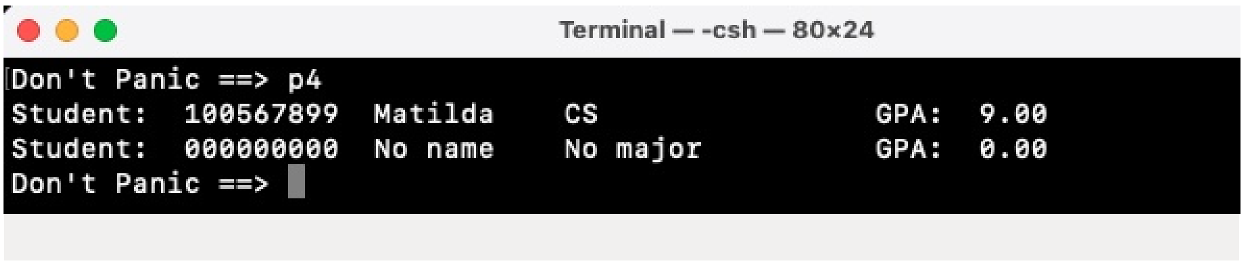
- Member initializer syntax *can* be used to initialize non-constant data members.
- But it *must* be used to initialize constant data members, because it executes *before* the body of the constructor.

8.2.6.4. **Applying the principle of least privilege:**

- We need to think about what data members are required for a class.
- If a data member never needs to be modified after initialization, we make it constant.
- This protects the integrity of the data member, even from our own member functions.

8.2.7. Coding example: Constant data members

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Student.h          *
 * * * * * * * * * * * * * * * * */
class Student
{
  public:
    Student(string="000000000", string="No name", string="No major",
            float=0.0f);
    void print() const;
  private:
    const string number;
    string name;
    string majorPgm;
    float  gpa;
};
/* * * * * * * * * * * * * * * * *
 * Filename:  Student.cc         *
 * * * * * * * * * * * * * * * * */
Student::Student(string s1, string s2, string s3, float g)
  : number(s1), name(s2), majorPgm(s3), gpa(g)
{
}

void Student::print() const
{
  cout << "Student:  " << number << "  " << left << setw(10) << name
       << " " << setw(15) << majorPgm << "   GPA: "
       << fixed << setprecision(2) << setw(5) << right << gpa << endl;
}
/* The rest of the member functions are not shown */

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc            *
 * * * * * * * * * * * * * * * * */
int main()
{
  Student matilda("100567899", "Matilda", "CS", 9.0f);
  Student joe;
  matilda.print();
  joe.print();
  return 0;
}
```



```
Don't Panic ==> p4
Student:  100567899  Matilda    CS                GPA:  9.00
Student:  000000000  No name    No major          GPA:  0.00
Don't Panic ==>
```

Program-8.4: Constant data members

Program purpose:

- Program-8.4 demonstrates the use of constant and non-constant data members, as well as the use of member initializer syntax.

Lines 4-15:

- These lines show the `Student` class definition.
- Line 11 declares the student number as a constant data member.

Lines 19-22:

- These lines show the implementation of the `Student` default constructor.
- Line 20 demonstrates the use of member initializer syntax to initialize all the data members from the given parameter values.
- While member initializer syntax is mandatory for initializing the `number` data member, which is constant, it is optional for initializing the other data members.
- If any member function, including the constructor, tried to assign a value to the `number` data member using the assignment operator, it would result in a compilation error.
- We see on lines 21-22 that the constructor body is empty, since all the data members are initialized using member initializer syntax.

8.3. Friendship

Friendship is a unique C++ feature that has value in very specific situations, but otherwise strains the principle of least privilege to near its breaking point. It's worth noting that friendship is a design choice that should **never** be made without a complete understanding of its ramifications.

In the COMP 2404 course supported by this textbook, the *only* acceptable use of friendship is discussed in Chapter 12, where it's necessary for overloading some types of operators.

8.3.1. Concepts

8.3.1.1. **What is *friendship* in C++:**

- *Friendship* in C++ is a special relationship between either:
  - two classes, or
  - a class and a global function
- A class can grant friendship to:
  - a global function (not a member function), and/or
  - another class
- A ***friend function*** is a global function that is granted friendship by a class.
- A *friend class* is a class that is granted friendship by another class.

8.3.1.2. **Characteristics of friendship:**

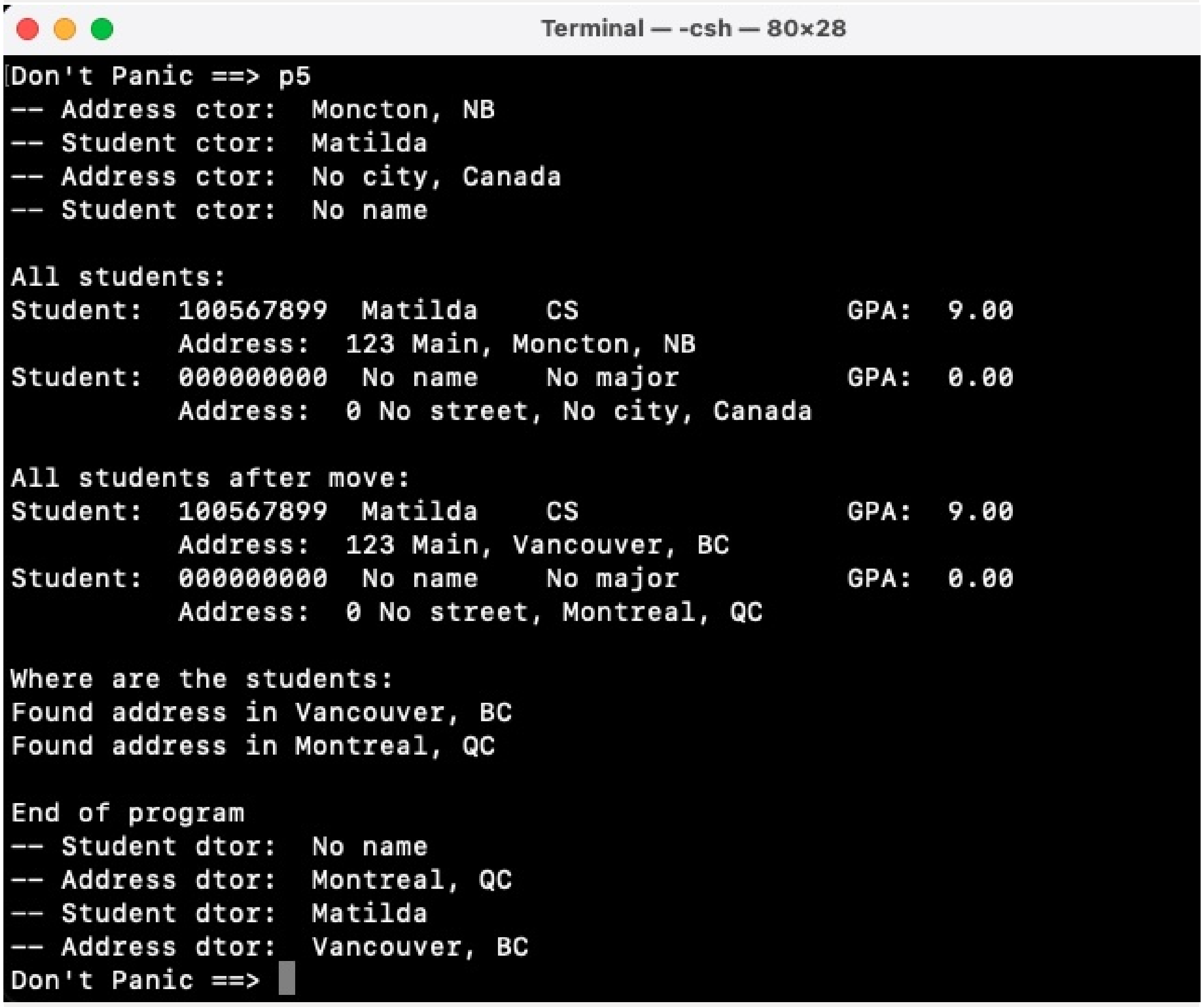
- If a class grants friendship, it gives away complete access to its class members, even the private and protected ones:
  - for example, if class `A` grants friendship to class `B`, then all instances of `B` can directly access any member inside an instance of `A`
- Friendship can only be granted (i.e. given), and it cannot be taken:
  - for example, class `A` can grant friendship to class `B`, but `B` cannot take friendship from class `A` without `A` explicitly granting it
- Friendship is neither symmetric nor transitive:
  - for example, if class `A` grants friendship to class `B`, it does *not* mean that `B` automatically grants friendship to `A`
  - if `A` grants friendship to `B`, and `B` grants friendship to `C`, it does *not* mean that `A` automatically grants friendship to `C`
- Friendship is *not* inherited from a superclass to a subclass.

8.3.1.3. **Friendship is bad software engineering:**

- Friendship violates both encapsulation and the principle of least privilege by giving away access to all class members, even private and protected ones.
- Outside of **very** specific situations, for example in the implementation of some overloaded operators, friendship should **never** be used in our programs.

8.3.2. Coding example: Friendship

```
/* * * * * * * * * * * * * * * * * *
 * Filename:  Address.h            *
 * * * * * * * * * * * * * * * * */
class Student;

class Address
{
  friend class Student;
  friend void where(Student&);

  public:
    Address(int=0, string="No street", string="No city",
            string="Canada");
    ~Address();
    void print() const;

  private:
    int    number;
    string street;
    string city;
    string province;
};
/*  The Address.cc file is not shown.  */

/* * * * * * * * * * * * * * * * *
 * Filename:  Student.h          *
 * * * * * * * * * * * * * * * * */
#include "Address.h"

class Student
{
  friend void where(Student&);

  public:
    Student(string="000000000", string="No name", string="No major",
            float=0.0f, int=0, string="No street", string="No city",
            string="Canada");
    ~Student();
    void moveTo(string,string);
    void print() const;

  private:
    const string number;
    string  name;
    string  majorPgm;
    float   gpa;
    Address homeAddr;
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Student.cc         *
 * * * * * * * * * * * * * * * * */
void Student::moveTo(string c, string p)
{
  homeAddr.city     = c;
  homeAddr.province = p;
}
/*  The rest of the file is not shown.  */

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc            *
 * * * * * * * * * * * * * * * * */
int main()
{
  Student matilda("100567899", "Matilda", "CS", 9.0f, 123, "Main",
                  "Moncton", "NB");
  Student joe;

  cout<<endl<<"All students:"<<endl;
  matilda.print();
  joe.print();

  matilda.moveTo("Vancouver", "BC");
  joe.moveTo("Montreal", "QC");

  cout<<endl<<"All students after move:"<<endl;
  matilda.print();
  joe.print();

  cout<<endl<<"Where are the students:"<<endl;
  where(matilda);
  where(joe);

  cout<< endl << "End of program" << endl;
  return 0;
}

void where(Student& s)
{
  cout << "Found address in " << s.homeAddr.city
       << ", " << s.homeAddr.province << endl;
}
```



```
Don't Panic ==> p5
-- Address ctor:  Moncton, NB
-- Student ctor:  Matilda
-- Address ctor:  No city, Canada
-- Student ctor:  No name

All students:
Student:  100567899  Matilda    CS                GPA:  9.00
          Address:  123 Main, Moncton, NB
Student:  000000000  No name    No major          GPA:  0.00
          Address:  0 No street, No city, Canada

All students after move:
Student:  100567899  Matilda    CS                GPA:  9.00
          Address:  123 Main, Vancouver, BC
Student:  000000000  No name    No major          GPA:  0.00
          Address:  0 No street, Montreal, QC

Where are the students:
Found address in Vancouver, BC
Found address in Montreal, QC

End of program
-- Student dtor:  No name
-- Address dtor:  Montreal, QC
-- Student dtor:  Matilda
-- Address dtor:  Vancouver, BC
Don't Panic ==> 
```

Program-8.5: Friendship

Program purpose:

- Program-8.5 modifies the `Address` and `Student` classes from Program-8.1 to demonstrate the use of friendship between two classes and between a class and a global function.

Line 4:

- This line is a *forward reference*, a technique sometimes necessary when packaging software.
- This program has a `Student` class that contains an instance of the `Address` class. As a result, the `Student.h` file must use the `include` preprocessor command to access the `Address` class definition found inside the `Address.h` file.

- However, because of the friendship relationship in this example, the `Address` class definition in the `Address.h` file also needs to know about the `Student` class definition found in the `Student.h` file.
- C++ prohibits circular inclusions like two files including each other.
- The solution is for the `Student.h` file to continue including the `Address.h` file, as we see on line 28. But instead of having the reverse inclusion, we use a forward reference on line 4 to inform the `Address` class that the `Student` class exists, without having to include the `Student.h` file.

Lines 6-22:

- These lines show the `Address` class definition.
- Line 8 grants friendship from the `Address` class to the `Student` class. This gives the `Student` member functions full access to the private data members of their containee `Address` object.
- Line 9 grants friendship from the `Address` class to the `where()` global function. This gives the function complete access to the private data members of an `Address` object.
- The member function implementations for the `Address` class are not shown, because they are identical to Program-8.1.

Lines 30-48:

- These lines show the `Student` class definition.
- Line 32 grants friendship from the `Student` class to the `where()` global function. This gives the function full access to the private data members of a `Student` object.
- Line 39 declares a new `moveTo()` member function that updates a student address's city and province to new values.

Lines 53-57:

- These lines show the implementation of the `moveTo()` member function.
- Lines 55-56 update the city and province data members of the containee `Address` object to the given parameters. For this to work, it is necessary for the `Address` class to grant friendship to the `Student` class, as we see on line 8.

Lines 88-92:

- These lines show the implementation of the `where()` global function.
- This function prints out the city and province where the given student resides.
- In order for this global function to print out the information, it must have access to the `Student` object's private `homeAddr` data member, as well as the containee `Address` object's `city` and `province` private data members.
- To get access to the needed data, it is necessary for both the `Student` and `Address` classes to grant friendship to the `where()` global function, as we see on lines 9 and 32.

Lines 63-86:

- These lines show the implementation of the `main()` function.
- The function declares and initializes a `Student` object called `matilda` on lines 65-66.
- Another `Student` object called `joe` is declared on line 67, using default values.
- Lines 73-74 call the `Student` class's `moveTo()` member function to update the city and province of both `Student` objects.
- Lines 81-82 call the `where()` global function to print out the city and province of both `Student` objects.
- We see from the program output that the changes on lines 73-74 are performed, and lines 81-82 print out the correct data.

8.4. Static class members

Most OO programming languages support static class members. When a class has a property that is specific to the class itself, regardless of any instances of the class, then that property can be represented as a static data member. If a class provides a service to other classes, independently of any instances of that class, then that service can be implemented as a static member function.

8.4.1. Concepts

8.4.1.1. **What is a *static member* of a class:**

- A *static member* is a data member or member function that is a property or behaviour of the class as a whole.
- A static member does *not* have different values or behaviours for each instance of the class.

8.4.1.2. **Characteristics of static members:**

- Only *one instance* of a static member exists for the entire class.
- Static members exist and can be used in the program, even if *no objects of the class are ever created*.
- Static members can be accessed either:
  - using the class name with the scope resolution operator, or
  - using any object of the class
- Be careful of the `static` keyword in C/C++! When used at file scope, the corresponding identifier becomes invisible outside that file.

8.4.1.3. **What is a *static data member*:**

- A *static data member* is a data member that's a property of the entire class as a whole.
- It contains a single value that is *shared* by all instances of that class.
- Only one value exists at a time for each static data member, and that value belongs to the entire class and all its objects.
- Static data members must be initialized at file scope. By convention, this initialization is placed in the source file.

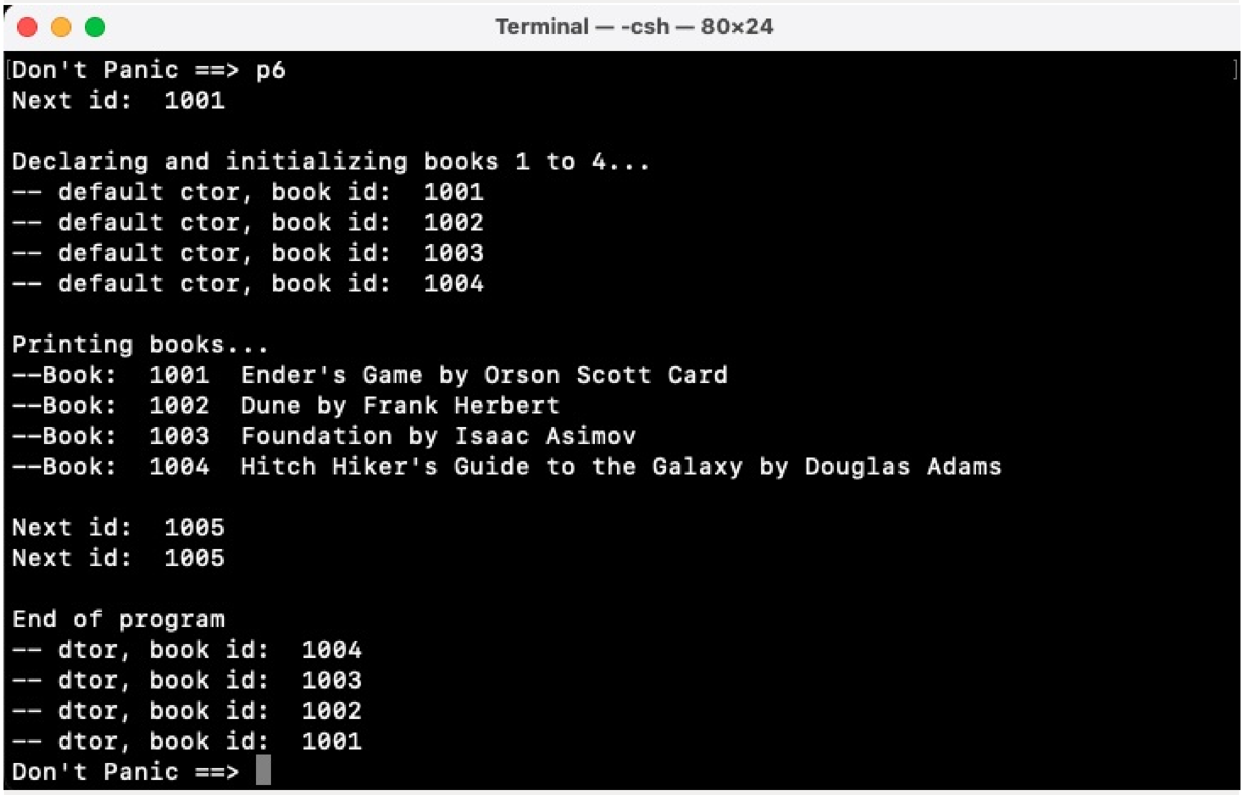
8.4.1.4. **What is a *static member function*:**

- A *static member function* is a member function that performs a service of the class as a whole.
- It must behave correctly, even if no objects of the class are ever created.
- A static member function may *only* access static data members. It cannot use any non-static data members or call any non-static member functions.
- It must be specified as static in the class definition only, and not in the source file, otherwise it cannot be visible outside the file.

8.4.2. Coding example: Static class members

```
/* * * * * * * * * * * * * * * * *
 * Filename:   Book.h              *
 * * * * * * * * * * * * * * * * */
class Book
{
  public:
    Book(string="Unknown", string="Unknown");
    Book(Book&);
    ~Book();
    static int getNextId();
    void print() const;

  private:
    static int nextId;
    int id;
    string title;
    string author;
};

/* * * * * * * * * * * * * * * * *
 * Filename:   Book.cc             *
 * * * * * * * * * * * * * * * * */
int Book::nextId = 1001;

Book::Book(string t, string a)
  : id(nextId++), title(t), author(a)
{
  cout<<"-- default ctor, book id:  "<< id <<endl;
}

Book::Book(Book& orig)
{
  id     = orig.id;
  title  = orig.title;
  author = orig.author;
  cout<<"-- copy ctor, book id:  "<< id <<endl;
}

Book::~Book()
{
  cout<<"-- dtor, book id:  "<< id <<endl;
}

int Book::getNextId()
{
  return nextId;
}

void Book::print() const
{
  cout<<"--Book:  "<< id << "  " << title <<" by "<<author<<endl;
}
```

```
/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc            *
 * * * * * * * * * * * * * * * * */
int main()
{
  cout<<"Next id:  "<< Book::getNextId() <<endl;

  cout<<endl<<"Declaring and initializing books 1 to 4..."<<endl;
  Book b1("Ender's Game", "Orson Scott Card");
  Book b2("Dune", "Frank Herbert");
  Book b3("Foundation", "Isaac Asimov");
  Book b4("Hitch Hiker's Guide to the Galaxy", "Douglas Adams");

  cout<<endl<<"Printing books..."<<endl;
  b1.print();
  b2.print();
  b3.print();
  b4.print();
  cout<<endl;

  cout<<"Next id:  "<< Book::getNextId() <<endl;
  cout<<"Next id:  "<< b4.getNextId() <<endl;

  cout<< endl << "End of program" << endl;
  return 0;
}
```



Terminal — -csh — 80×24

```
Don't Panic ==> p6
Next id:  1001

Declaring and initializing books 1 to 4...
-- default ctor, book id:  1001
-- default ctor, book id:  1002
-- default ctor, book id:  1003
-- default ctor, book id:  1004

Printing books...
--Book:  1001  Ender's Game by Orson Scott Card
--Book:  1002  Dune by Frank Herbert
--Book:  1003  Foundation by Isaac Asimov
--Book:  1004  Hitch Hiker's Guide to the Galaxy by Douglas Adams

Next id:  1005
Next id:  1005

End of program
-- dtor, book id:  1004
-- dtor, book id:  1003
-- dtor, book id:  1002
-- dtor, book id:  1001
Don't Panic ==> 
```

Program-8.6: Static class members

Program purpose:

- Program-8.6 demonstrates the use of static data members and static member functions.
- The program uses and modifies the `Book` class from Program-3.6.

Lines 4-18:

- These lines show the `Book` class definition.
- Line 14 declares an integer called `nextId` as a static data member.
- As a static data member, `nextId` stores a single value for the entire `Book` class, and that value is shared by all the objects of the class.
- In this example, the `nextId` data member contains the unique id that our code automatically assigns to the next `Book` object that is created.
- Line 15 declares the unique id for each `Book` object. Because it's a non-static data member, each `Book` object stores its own separate value for this data member.
- Line 10 declares a static member function called `getNextId()`.
- As a static member function, `getNextId()` is a service of the entire class, and it can be called even if no `Book` objects are ever created.

Line 23:

- This line initializes the `nextId` static data member with an initial value of `1001`.
- The initialization is performed at file scope, and it must use the class name and scope resolution operator, as shown on line 23.

Lines 25-29:

- These lines show the implementation of the default constructor for the `Book` class.
- Line 26 uses member initializer syntax to initialize all the data members of a new `Book` object.
- Line 26 also shows how the new `Book` object's unique `id` is initialized automatically using the static data member `nextId`.
- Line 26 uses the *postfix* version of the increment operator, which is discussed in Program-2.1. The postfix operator increments its operand, but it returns the *original value*, i.e. the value before the operand was incremented.
- On line 26, this original value is assigned to the `id` data member of the new `Book` object, and the incremented `nextId` is ready to be assigned the next time a new `Book` object is created.

Lines 44-47:

- These lines show the implementation of the `getNextId()` static member function.
- As a static member function, it can *only* access static data members, specifically `nextId`.

Lines 56-78:

- These lines show the implementation of the `main()` function.
- Line 58 shows a call to the `Book` class's `getNextId()` static member function, using the class name and scope resolution operator. Even though no `Book` objects exist yet on this line, the `nextId` static data member has already been initialized on line 23.
- Lines 60-71 create four `Book` objects and print them out. We see from the program output that the constructor assigns each `Book` object a sequentially increasing unique book id.
- Lines 73-74 both print out the current value of `nextId`. Line 73 calls `getNextId()` using the class name and scope resolution operator, and line 74 uses one of the `Book` objects.
- Because no additional `Book` objects are created on lines 73-74, they both print the same value, as we see in the program output.

8.5. Namespaces

We discuss namespaces and their usage in C++.

8.5.1. Concepts

8.5.1.1. **What is a *namespace*:**

- In C++, a *namespace* defines a self-contained scope.
- It groups together a set of identifiers (for example: variables, functions, data types) under a unique name.
- A namespace allows the programmer to encapsulate inside a scope a group of variables and functions that belong together functionally, but that *do not* represent a class.

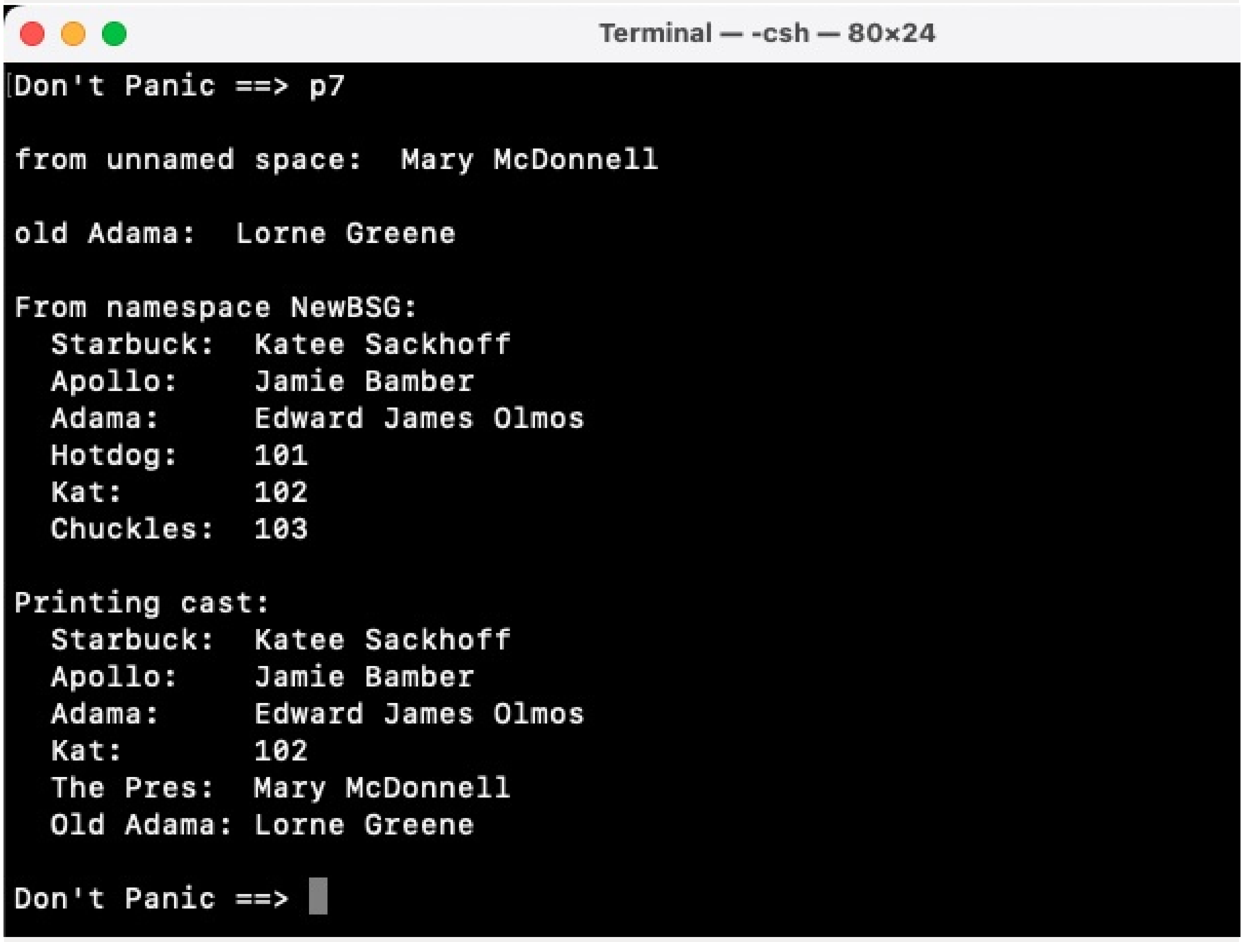
8.5.1.2. **Characteristics of namespaces:**

- A namespace is **not** a class or a data type.
- No instances of a namespace can be created, and it occupies no memory.
- There are two ways to use the members of a namespace inside our code:
  - the namespace can be *scoped in* with the `using` keyword, which brings all the members of the namespace into the current scope, or
  - every usage of a namespace member can be preceded by the namespace identifier and the scope resolution operator
- For example:
  - we use the `std` namespace for standard I/O in most of our coding examples
  - to access the I/O objects and functions, our examples bring into scope the entire `std` namespace, with the statement: `using namespace std;` in our programs
  - another way to access each identifier inside the `std` namespace is to use the scope resolution operator instead, for example: `std::cout << "Hello world!" << std::endl;`
- A namespace may be *unnamed*. An unnamed namespace is automatically scoped in.

8.5.2. Coding example: Namespaces

```
string adama = "Lorne Greene";

namespace NewBSG
{
  string starbuck = "Katee Sackhoff";
  string apollo   = "Jamie Bamber";
  string adama    = "Edward James Olmos";

  void printCast();

  namespace Nuggets
  {
    enum NuggetNames { HOTDOG=101, KAT, CHUCKLES };
  }
}

namespace
{
  string thePres = "Mary McDonnell";
}
```

```
int main()
{
  cout << endl << "from unnamed space:  " << thePres << endl << endl;
  cout << "old Adama:  "<< adama << endl << endl;

  cout << "From namespace NewBSG:"<< endl;
  cout << "  Starbuck:  "<< NewBSG::starbuck << endl;
  cout << "  Apollo:    "<< NewBSG::apollo   << endl;
  cout << "  Adama:     "<< NewBSG::adama    << endl;
  cout << "  Hotdog:    "<< NewBSG::Nuggets::HOTDOG   << endl;
  cout << "  Kat:       "<< NewBSG::Nuggets::KAT      << endl;
  cout << "  Chuckles:  "<< NewBSG::Nuggets::CHUCKLES << endl;

  NewBSG::printCast();
  return 0;
}

void NewBSG::printCast()
{
  cout << endl << "Printing cast:" << endl;
  cout << "  Starbuck:   " << starbuck << endl;
  cout << "  Apollo:     " << apollo   << endl;
  cout << "  Adama:      " << adama    << endl;
  cout << "  Kat:        " << Nuggets::KAT << endl;
  cout << "  The Pres:   " << thePres  << endl;
  cout << "  Old Adama: " << ::adama  << endl << endl;
}
```



```
Don't Panic ==> p7

from unnamed space:  Mary McDonnell

old Adama:  Lorne Greene

From namespace NewBSG:
  Starbuck:  Katee Sackhoff
  Apollo:    Jamie Bamber
  Adama:     Edward James Olmos
  Hotdog:    101
  Kat:       102
  Chuckles:  103

Printing cast:
  Starbuck:  Katee Sackhoff
  Apollo:    Jamie Bamber
  Adama:     Edward James Olmos
  Kat:       102
  The Pres:  Mary McDonnell
  Old Adama: Lorne Greene

Don't Panic ==> 
```

Program-8.7: Namespaces

Program purpose:

- Program-8.7 demonstrates the use of namespaces to encapsulate identifiers, including variables, functions, and data types, into a named or unnamed scope.

Line 1:

- This line declares a global variable called `adama`, initialized with the name of actor Lorne Greene.

Lines 3-15:

- These lines declare a namespace called `NewBSG`.
- The namespace contains:
  - three string variables (`starbuck`, `apollo`, and `adama`) declared on lines 5-7;
  - a function prototype (`printCast()`) on line 9; and
  - a nested namespace called `Nuggets` on lines 11-14
- The nested namespace `Nuggets` contains the definition of an enumerated data type called `NuggetNames` on line 13.
- The `NewBSG` namespace defines a variable called `adama` on line 7, initialized with the name of actor Edward James Olmos, even though another variable of the same name but a different value has already been declared as a global variable on line 1.
- In C++, a variable can be declared in an inner scope with the same name as a variable in an outer scope. This is a valid technique, but the inner scope variable effectively "hides" the outer scope variable of the same name.
- Because the namespace contains a function prototype, but not the corresponding function implementation, this implementation must be provided elsewhere in the source file, in this case on lines 38-47.

Lines 17-20:

- These lines declare a unnamed namespace, which contains a single string variable called `thePres`.
- This namespace is scoped in automatically, which means that the program can use the `thePres` variable directly.

Lines 38-47:

- These lines contain the implementation of the `printCast()` function that belongs to the `NewBSG` namespace.
- We see on line 38 that, in the function prototype, the function name must include the namespace name, followed by the scope resolution operator. This is unnecessary on line 9, because that line is inside the namespace definition.
- Lines 41-43 print out the variables from the `NewBSG` namespace. In particular, line 43 prints out the value of the inner scope `adama` variable declared in the namespace.
- Line 44 prints out a value from the enumerated data type declared in the nested namespace `Nuggets`. We see how the scope resolution operator is used to access this value.
- Line 45 demonstrates that the `thePres` variable declared in the unnamed namespace can be accessed directly.
- Line 46 shows how a global variable can be accessed, even if another variable with the same name has been declared in an inner scope. The scope resolution operator, used as a unary operator, allows a function in the `NewBSG` namespace to access the `adama` global variable, even though a variable of the same name exists inside the namespace.

Lines 21-36:

- These lines show the implementation of the `main()` function.
- Line 23 shows how the function accesses the contents of the unnamed namespace directly.
- Line 24 prints out the `adama` global variable.
- Lines 27-32 print out information from the `NewBSG` namespace and from its nested namespace, using the scope resolution operator to access each namespace member.
- Line 34 calls the `NewBSG` namespace's `printCast()` function, again using the scope resolution operator.

8.6. Encapsulation example: Linked lists

In sections 8.6 and 8.7, we develop a design and implementation example that supports several of the concepts covered so far in this textbook: pointers, dynamic memory allocation, collection classes, data abstraction, encapsulation, and the principle of least privilege.

Section 8.6 introduces the concepts behind the linked list data structure, how its design can be optimized to promote encapsulation, and how the common operations performed on linked lists can be designed to uphold the principle of least privilege. Section 8.7 illustrates these concepts with a coding example.

8.6.1. Concepts

8.6.1.1. **Use of collections:**

- When a data processing application requires the storage of data into program memory, *how* the data is stored has important implications for computational and memory efficiency.
- We want to use the least amount of memory, while maintaining the fastest access to the data. The trade-off between space (memory) and time (computational efficiency) is the programmer's eternal quandary.
- The correct choice of collection type always depends on the nature of the application. There is no "right choice" that is always correct, and every type of collection has its advantages and disadvantages.
- The major types of collections are:
  - sequential (for example: arrays, linked lists, deques)
  - associative (for example: sets, maps, trees)
  - higher-order (for example: stacks, queues, priority queues)
- In the examples of this textbook, we focus mainly on arrays and linked lists.

8.6.1.2. **Terminology:**

- In computing, the *back* of a collection means its end, where the last element is located.
- The *front* of a collection is its beginning, where the first element is.

8.6.1.3. **Using arrays:**

- Using arrays to store our program data has many advantages:
  - array elements are guaranteed to be stored *contiguously*
  - contiguous storage means that the elements are stored all together in a single unbroken block of memory, without gaps
  - this contiguous storage results in faster sequential access to the elements

- But arrays also incur disadvantages:
  - a statically allocated array cannot be resized; it can neither grow nor shrink to accommodate additional or fewer elements
  - resizing a dynamically allocated array is often inefficient, because it requires that elements be copied from one memory location to another
  - adding or removing an element from anywhere in the array except at the back is inefficient

8.6.1.4. **Using linked lists:**

- Using linked lists to store program data has the following advantages:
  - there is no need to ever resize the list
  - we only allocate as much memory as we need, no more and no less
  - elements can be efficiently added, removed, or shifted anywhere in the list
- And the following disadvantage:
  - the elements are *not* stored contiguously in memory, so sequential access is slower

8.6.1.5. **Rationale for implementing linked lists in C++:**

- The C++ standard template library (STL) does provide a linked list container called `list`.
- As we discuss in a later chapter, there are issues with the STL `list` container that render its usage problematic for some applications. As a result, it's a far-from-ideal tool for beginner C++ programmers.
- In this section, we focus on our own implementation of a linked list, as an exercise for deeper learning of memory management, and OO data abstraction and encapsulation concepts.

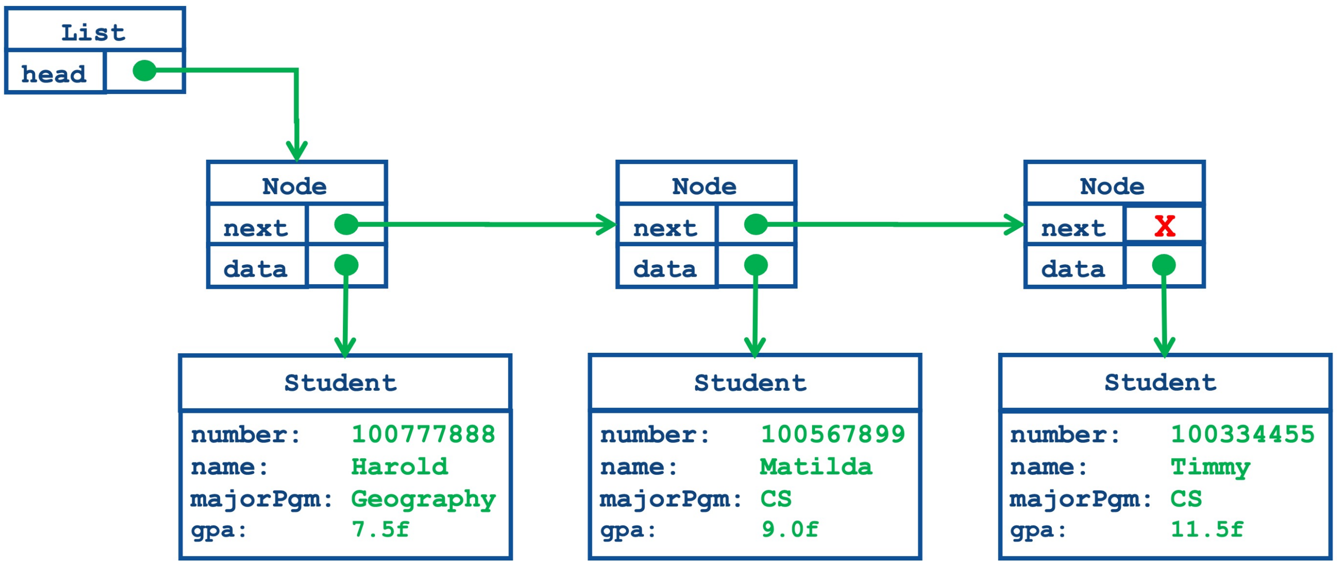


Figure-8.1: Example of linked list classes

8.6.1.6. **Classes for implementing a linked list:**

- Figure-8.1 shows the major classes involved in our implementation of a linked list.
- A linked list, represented by the `List` class, is composed of the following:
  - a set of `Node` objects;
  - a pointer to the first node in the list, called the `head`;
  - optionally, a pointer to the last node in the list, called the `tail`; and
  - a set of data objects

- The `Node` class represents a single node, and it contains:
  - a pointer to the corresponding `data` element;
  - a `next` pointer to the following node in the list; and
  - optionally, if the list is *doubly linked*, a `prev` pointer to the preceding node in the list
- The data class represents a single element in the linked list. In Figure-8.1, our example shows that data are `Student` objects.

8.6.1.7. **Characteristics of our linked list implementation:**

- **Do not** confuse the name of our `List` class with the STL `list` container. Unix-based programming is *always* case-sensitive, so the two class names are entirely different.
- At all times, there is *exactly one node* for each data element in the linked list. If an element is added to the list, a new node must be created. If an element is removed, its node must be deleted.
- Nodes can **never** be reused by different data elements, not even if an element is removed and then added again. They can never be created in advance, before an element is added.
- The `next` pointer of the last node is set to a null value, as is the `prev` pointer of the first node in a doubly linked list.

8.6.1.8. **Types of linked lists:**

- Some linked lists are *singly linked*:
  - each node stores knowledge of the next node in the list
  - the node does not keep track of the previous node
  - a singly linked list can only be traversed in the forward direction, starting at the head until the end of the list
- Some linked lists are *doubly linked*:
  - each node stores knowledge of the next node in the list
  - the node also keeps track of the previous node
  - a doubly linked list can be traversed either in the forward direction, starting at the head until the end of the list, or in the backward direction, starting at the end until the head
- All lists maintain a pointer to the first node, called the head.
- Some lists maintain a pointer to the last node, called the tail, and some do not.
- In all linked lists, the last node's next node pointer is set to a null value.
- In doubly linked lists, the first node's previous node pointer is set to null.

8.6.1.9. **Alternative linked list implementations:**

- Some linked list implementations use extra nodes, called *dummy nodes*:
  - a *dummy node* is a placeholder node that has no data element associated with it
  - some implementations use one dummy node to represent the list head, and one for the tail
  - we do **not** use dummy nodes in these examples; all the nodes we implement correspond to a data element
- Some linked list implementations are *circular*:
  - in a *circular linked list*, the last node's next node pointee is the first node
  - in a doubly linked circular list, the first node's previous node pointee is the last node
  - by default, linked lists are **not** circular

8.6.1.10. **Why are so many classes used in linked list implementation:**

- We separate our classes by functionality, in order to maintain encapsulation and correct data abstraction.
- We keep the data-related and the list-related knowledge separate:
  - it's good encapsulation to compartmentalize what knowledge each object contains
  - for example, a data object should never know that it's an element in a collection
- This separation of functionality facilitates code reuse. For example, without nodes, each data object would be hard-coded to point to a fixed next element. As a result, the same data object could not be stored as an element in multiple lists, which would be a limiting design.
- The separation of data-related and list-related knowledge is **good** software engineering.

8.6.2. Linked list operations

8.6.2.1. **Inserting a list element:**

- A new element can be inserted anywhere in a linked list, simply by shifting next node pointer values between different nodes.
- When inserting a new element, our implementation must handle the following four cases:
  - the new element is added to a currently empty list
  - the element is inserted at the beginning of the list
  - the element is inserted in the middle of the list
  - the element is inserted at the end of the list

8.6.2.2. **Removing a list element:**

- An element can be removed from anywhere in a linked list, simply by shifting pointer values between different nodes.
- When removing an element, our implementation must handle the following six cases:
  - the list is empty
  - the element removed is the only element currently in the list
  - the element is removed from the beginning of the list
  - the element is removed from the middle of the list
  - the element is removed from the end of the list
  - the element to be removed is not found in the list

8.6.2.3. **List cleanup:**

- A correct implementation of linked lists must use dynamically allocated memory for the nodes, and most likely for the data as well.
- Dynamically allocated memory must always be explicitly deallocated when no longer needed. In C++, destructors are naturally suited to perform the deallocation tasks.
- But what exactly must be deallocated? The nodes only, or both nodes and data? This depends on the nature of the application.
- When to deallocate linked list nodes:
  - when an element is removed from the list, the corresponding node must be deallocated
  - all nodes must be deallocated when the list itself is deallocated

- When to deallocate linked list data:
  - data must be deallocated only **once** in the program
  - we must ensure that data is deallocated *only if it is **no longer required** in the program*
  - if the data is not deallocated when the list itself is deallocated, we must ensure that we retain a pointer to the data, so it can be deallocated later in the program

8.7. Coding example: Linked lists

We combine together the concepts of section 8.6 and demonstrate their implementation in a coding example.

8.7.1. Class definitions

```
 1 /* * * * * * * * * * * * * * * * *
 2  * Filename:  List.h             *
 3  * * * * * * * * * * * * * * * * */
 4 class List
 5 {
 6   class Node
 7   {
 8     public:
 9       Student* data;
10       Node*    next;
11   };

13   public:
14     List();
15     ~List();
16     void add(Student*);
17     void del(const string&, Student**);
18     void print() const;

20   private:
21     Node* head;
22 };
```

Program-8.8: Linked list class definitions

Partial program purpose:

- Partial Program-8.8 shows the class definitions used for a linked list that stores `Student` object pointers as data.
- The `Student` class is identical to Program-8.4, with the addition of a getter for the `name` data member.

Lines 6-11:

- These lines show the `Node` class definition, which is defined *inside* the `List` class definition. We call this a *nested class*.
- The use of nested classes is reserved for cases where there is a strong dependency between two classes, as there is between `List` and `Node`, and we want to prohibit access to the nested class from outside the outer class.
- Remember that the default access specifier inside a class is *private*. Lines 6-11 are not under the public heading on line 13, so the `Node` class definition is private within the `List` class. As a result, no other class can access it or even know that it exists, which is good encapsulation.

- With the `Node` class definition private to the `List` class, we make its data members public on line 8, so that the `List` class can access the `Node` data members directly. This is the *only* acceptable use of public data members.
- Line 9 specifies that each list element is stored as a `Student` object pointer.
- Line 10 declares a pointer to the next node in the list.
- These lines also indicate that the list is singly linked. If it was doubly linked, there would be an additional data member for a pointer to the previous node in the list.

Line 21:

- This line declares the only `List` data member as the head of the linked list, which is a pointer to the list's first node.
- This line also reveals that the linked list does *not* maintain a tail.

Lines 14-18:

- Line 14 declares a default constructor and line 15 a destructor for the `List` class, respectively.
- Line 18 declares a printing function that prints out the contents of the linked list to the screen.
- Line 16 line declares the `add()` member function that adds a new student as an element of the linked list.
- The `add()` function takes a `Student` pointer as parameter, and **not** a `Node`. It would be very bad data abstraction and encapsulation for a class or function outside the `List` class to have any knowledge of nodes.
- In this example, the `add()` member function inserts a new element into the list so that the students are stored in *ascending* (increasing) alphabetical order by name.
- Line 17 declares the `del()` member function that removes from the linked list the student with the name provided as the first parameter. The function then returns the removed student to the calling function, using the second parameter.
- The second parameter to `del()` is a double pointer, because we need to return a single `Student` pointer by reference. We do **not** use the return value to return the `Student` pointer. Instead, we use an output parameter to practice with both output parameters and double pointers, which are useful when a function needs to return multiple values.

8.7.2. Initializing the list

```
1 /* * * * * * * * * * * * * * * * *
2  * Filename:  List.cc (partial)   *
3  * * * * * * * * * * * * * * * * */
4 List::List() : head(nullptr) { }
```

Program-8.9: Linked list initialization

Partial program purpose:

- Partial Program-8.9 shows the implementation of the `List` constructor that initializes a linked list as defined in partial Program-8.8.

Line 4:

- The job of any constructor is to initialize all the data members of a new object.
- The `List` class has only one data member, the head of the list. Since every new list starts out empty, we initialize the `head` data member to a null pointer.
- Since we use member initializer syntax to initialize the list head, the constructor body is empty. The empty braces are still mandatory in order for the compiler to recognize this line as a function implementation.

8.7.3. Inserting a list element

```
 1 /* * * * * * * * * * * * * * * * *
 2  * Filename:  List.cc (partial)   *
 3  * * * * * * * * * * * * * * * * */
 4 void List::add(Student* newStu)
 5 {
 6   Node* newNode = new Node;
 7   newNode->data = newStu;
 8   newNode->next = nullptr;

10   Node *currNode, *prevNode;
11   currNode = head;
12   prevNode = nullptr;

14   while (currNode != nullptr) {
15     if (newNode->data->getName() < currNode->data->getName())
16       break;
17     prevNode = currNode;
18     currNode = currNode->next;
19   }

21   if (prevNode == nullptr) {
22     head = newNode;
23   }
24   else {
25     prevNode->next = newNode;
26   }
27   newNode->next = currNode;
28 }
```

Program-8.10: Linked list insertion

Partial program purpose:

- Partial Program-8.10 shows the implementation of the `add()` member function of the `List` class defined in partial Program-8.8.
- This function inserts a new student into the linked list in its correct position, so that the list remains in ascending alphabetical order by student name at all times.

Lines 6-8:

- Every data element in the list needs a corresponding node. Since we are adding a new element, these lines create and initialize a new node for it.
- Line 6 dynamically allocates a new node for the new student element.
- Line 7 sets the new node's `data` data member to the new student element, which is passed in as a parameter to the `add()` member function.
- At this point, we don't yet know where in the list the new element belongs. So for now, we set the new node's `next` data member to a null value on line 8.

Lines 10-12:

- Next, we must find the new element's *insertion point*, where the new student belongs so that the list remains in the correct order.
- The end goal in finding the insertion point is to correctly set two `Node` pointer variables: one pointer to the current list node that should immediately precede the new element, and one pointer to the node that should immediately follow the new one.

- Line 10 declares these two node pointers. The `currNode` pointer is used as a looping variable to traverse the list, and it ends up set to the list node that should immediately follow the new element to be added. The `prevNode` pointer ends up set to the list node that should immediately precede the new element.
- Lines 11-12 set both `currNode` and `prevNode` to their initial values before the loop that traverses the list looking for the insertion point. As the looping variable, `currNode` starts at the list head. As the node preceding `currNode`, `prevNode` is initialized to a null value.

Lines 14-19:

- These lines show the loop that traverses the linked list and looks at each student element to find the insertion point for the new student.
- Line 14 is the `while` loop header that terminates the loop if `currNode` has reached the null pointer value at the end of the linked list.
- The loop in lines 14-19 exits under one of two conditions: we reach the end of the list and exit at the loop header on line 14, or we find the insertion point and break out of the loop on line 16. If we reach the end of the list, it means that the new student must be added after the last element.
- Line 15 determines if the loop's current iteration has found the insertion point. If the new student's name is alphabetically less than the current student element we are looking at, then we have found the insertion point and break out of the loop on line 16.
- If the new student's name is alphabetically greater than the current element, then we haven't yet found the insertion point, and we must move on to examine the next element in the list.
- Before we advance `currNode` forward in the list, we must ensure that the `prevNode` pointer is always set to the node that precedes `currNode`. This is done on line 17.
- Line 18 advances `currNode` to the next node in the list, so that the next iteration of the loop can examine the next student element.

Lines 21-26:

- These lines connect the new student element's node to the preceding node in the list.
- If the new student is added to the beginning of the list as the first element, then the list head must be updated. We detect this case by checking if `prevNode` has a null value on line 21. If it does, then the list head is updated on line 22.
- If the new student is added to the middle or end of the list, then its preceding node is stored in `prevNode`. We set that node's `next` data member to the new element's node on line 25.

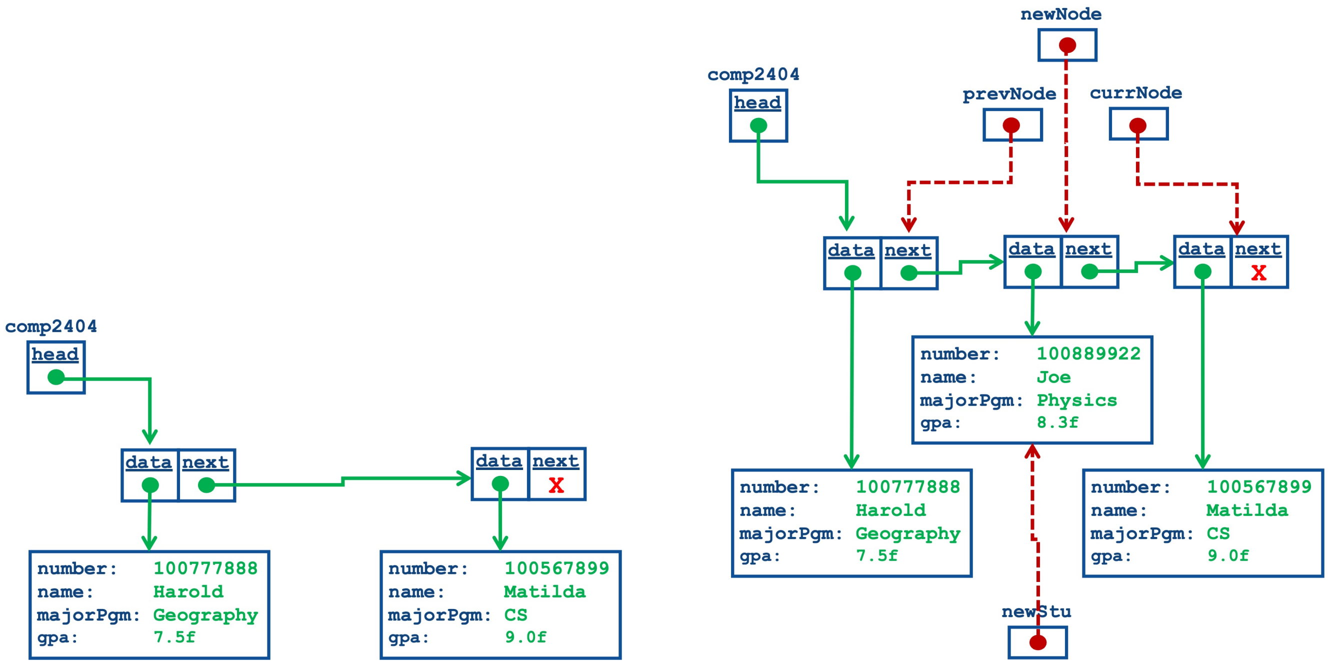
Line 27:

- This line connects the new student element's node to the following node in the list, as stored in the `currNode` pointer.
- In a singly linked list without a tail, there is no special case for linking the new node to the following one. We set the new element node's `next` data member to the following node on this line. If the new student is added to the end of the list, the following node is a null pointer.

8.7.3.1. **Insertion exercise #1: adding to the middle of the list**

- Figure-8.2 demonstrates the insertion of a new student *in the middle* of a linked list, using the `List` class's `add()` member function shown in partial Program-8.10.
- Figure-8.2a shows the linked list before the new student is added. The existing list contains two students named Harold and Matilda. This exercise shows the insertion of new student Joe. The list head is stored in a node pointer called `comp2404` in the program's `main()` function. In Figure-8.2a, the head points the node corresponding to student Harold, and that node points to the next node corresponding to student Matilda. Matilda's node is the last node in the list, so its next pointer value is null, as represented by a red X.

- Figure-8.2b shows the linked list after the insertion of student Joe, as well as the values of the temporary pointers inside the `add()` member function. The new student Joe is passed in to the function as parameter `newStu`, and a new node is created for Joe using the `newNode` pointer.
- In Figure-8.2b, we see that the insertion point for new student Joe is found between students Harold and Matilda. So when the control flow breaks out of the loop on line 16 of partial Program-8.10, the preceding node pointer `prevNode`'s pointee is Harold's node, and the following node pointer `currNode`'s pointee is Matilda's node.
- Lines 25 and 27 of partial Program-8.10 reset the `next` pointers inside both Harold and Joe's nodes respectively, so that Joe's node is inserted into the linked list at the correct position.



(a) Before insertion (b) After insertion

Figure-8.2: Linked list exercise: adding to the middle of the list

8.7.3.2. **Insertion exercise #2: adding to the end of the list**

- Figure-8.3 demonstrates the insertion of a new student *at the end* of a linked list, using the `List` class's `add()` member function shown in partial Program-8.10.
- Figure-8.2b shows the linked list before the new student is added. The existing list contains three students named Harold, Joe, and Matilda. This exercise shows the insertion of new student Timmy.
- Figure-8.3 shows the linked list after the insertion of student Timmy, as well as the values of the temporary pointers inside the `add()` member function. In the same figure, we also see that the insertion point for new student Timmy is found after student Matilda. So when the control flow breaks out of the loop on line 14 of partial Program-8.10, the preceding node pointer `prevNode`'s pointee is Matilda's node, and the following node pointer `currNode` is null.
- Lines 25 resets the `next` pointer inside Matilda's node, so that Timmy's node is inserted into the linked list at the end.
- Line 27 resets the `next` pointer of Timmy's node to a null value. Even though this action is unnecessary, since a new node's `next` pointer is always set to null on line 8, we do *not* add a new special case to the code. We could place an `if`-statement on line 27 to not perform this action when adding to the end of the list, but that would complicate the code and impede its readability.

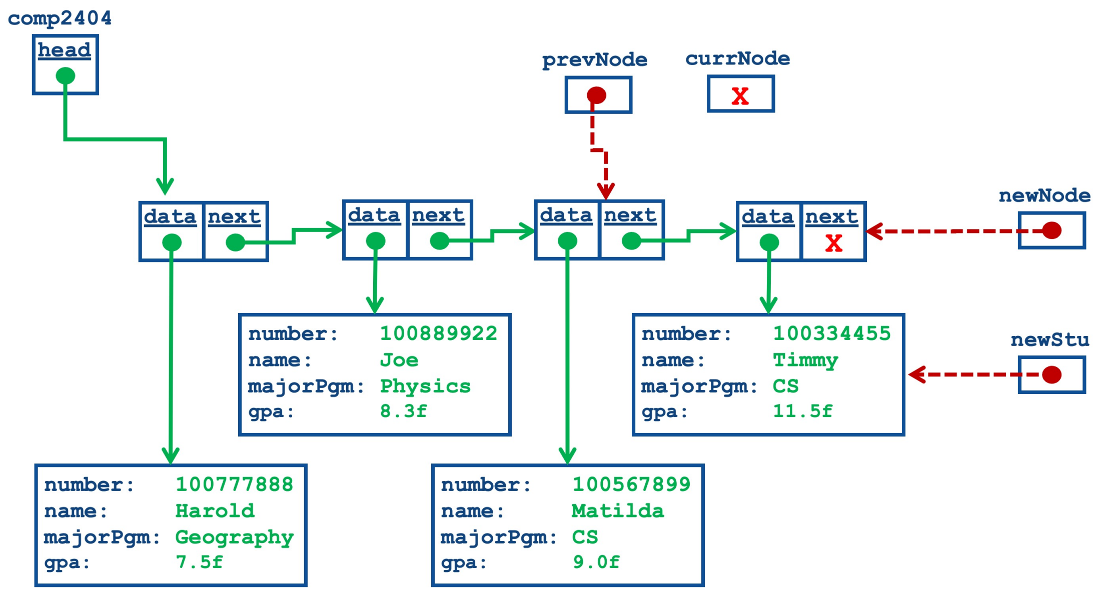


Figure-8.3: Linked list exercise: adding to the end of the list

8.7.3.3. **Insertion exercise #3: adding to the beginning of the list**

- Figure-8.4 demonstrates the insertion of a new student *at the beginning* of a linked list, using the `List` class's `add()` member function shown in partial Program-8.10.
- Figure-8.3 shows the linked list before the new student is added. The existing list contains four students named Harold, Joe, Matilda, and Timmy. This exercise shows the insertion of new student Amy.
- Figure-8.4 shows the linked list after the insertion of student Amy, as well as the values of the temporary pointers inside the `add()` member function. In the same figure, we also see that the insertion point for new student Amy is found before student Harold. So when the control flow breaks out of the loop on line 16 of partial Program-8.10, the preceding node pointer `prevNode` has a null value, and the following node pointer `currNode`'s pointee is Harold's node.
- Line 21 detects that we are inserting at the beginning of the list, so the list head must be updated.
- Lines 22 resets the list head from Harold's node, as shown in Figure-8.3, to Amy's node, as shown in Figure-8.4, so that Amy's node is inserted into the linked list at the beginning.
- Line 27 resets the `next` pointer of Amy's node to Harold's node.

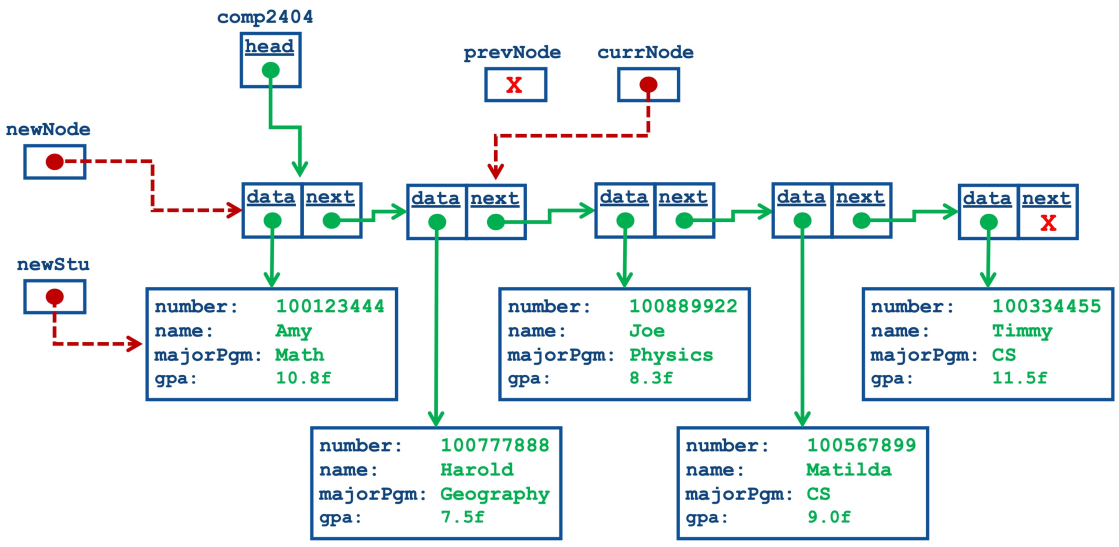


Figure-8.4: Linked list exercise: adding to the beginning of the list

8.7.4. Deleting a list element

```
 1 /* * * * * * * * * * * * * * * * *
 2  * Filename:  List.cc (partial)   *
 3  * * * * * * * * * * * * * * * * */
 4 void List::del(const string& name, Student** goner)
 5 {
 6   Node *currNode, *prevNode;
 7   prevNode = nullptr;
 8   currNode = head;

10   while (currNode != nullptr) {
11     if (currNode->data->getName() == name)
12       break;
13     prevNode = currNode;
14     currNode = currNode->next;
15   }

17   if (currNode == nullptr) {
18     *goner = nullptr;
19     return;
20   }

22   if (prevNode == nullptr) {
23     head = currNode->next;
24   }
25   else {
26     prevNode->next = currNode->next;
27   }
28   *goner = currNode->data;
29   delete currNode;
30 }
```

Program-8.11: Linked list deletion

Partial program purpose:

- Partial Program-8.11 shows the implementation of the `del()` member function of the `List` class defined in partial Program-8.8.
- This function removes from the linked list a student with the name provided in the `name` parameter, so that the list remains fully linked with no gaps between the nodes. The `goner` parameter returns to the calling function a pointer to the removed `Student` object.

Lines 6-8:

- First, we must traverse the linked list and find the target student to be deleted and its corresponding node.
- The end goal in finding the target student's node is to correctly set two `Node` pointer variables: one pointer to the target node, and one pointer to the node that immediately precedes it.
- Line 6 declares these two node pointers. The `currNode` pointer is used as a looping variable to traverse the list, and it ends up set to the list node that corresponds to the target student element to be deleted. The `prevNode` pointer ends up set to the list node that immediately precedes the target node.
- Lines 7-8 set both `currNode` and `prevNode` to their initial values before the loop that traverses the list looking for the target student. As the looping variable, `currNode` starts at the list head. As the node preceding `currNode`, `prevNode` is initialized to a null value.

Lines 10-15:

- These lines show the loop that traverses the linked list and looks at each student element to find the target student to be deleted.
- Line 10 is the `while` loop header that terminates the loop if `currNode` has reached the null pointer value at the end of the linked list.
- The loop in lines 10-15 exits under one of two conditions: we reach the end of the list and exit at the loop header on line 10, or we find the target element to be deleted and break out of the loop on line 12. If we reach the end of the list, it means that we did not find any student with a name matching the parameter.
- Line 11 determines if the loop's current iteration has found the target student. If the name of the current student element matches the `name` parameter, then we have found the target student and break out of the loop on line 12.
- If the current element's name doesn't match the parameter, then we haven't yet found the target student, and we must move on to examine the next element in the list.
- Before we advance `currNode` forward in the list, we must ensure that the `prevNode` pointer is always set to the node that precedes `currNode`. This is done on line 13.
- Line 14 advances `currNode` to the next node in the list, so that the next iteration of the loop can examine that next student element.
- When we exit the loop, the `currNode` pointer is set to the target student's node, if it has been found, and `prevNode` is set to its preceding node. If the target student has not been found, then the end of the list was reached, and `currNode` has a null pointer value.

Lines 17-20:

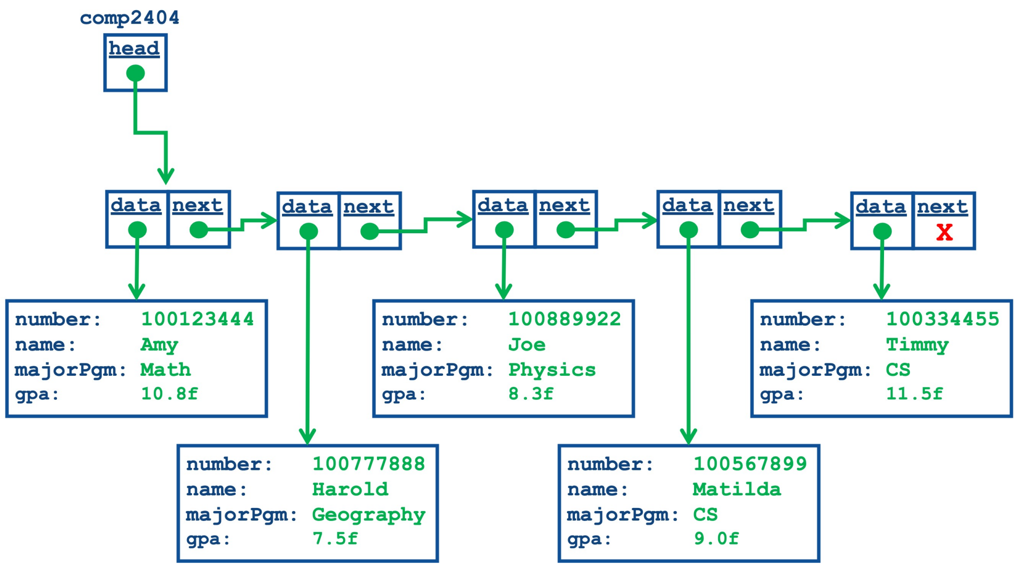
- These lines deal with the case where we have traversed the entire linked list and the target student has not been found. In other words, there is no student element in the list whose name matches the `name` parameter. In this case, there are no changes to be made to the list.
- Line 18 initializes the target student pointer in the `goner` parameter to a null value, and line 19 returns to the calling function.

Lines 22-27:

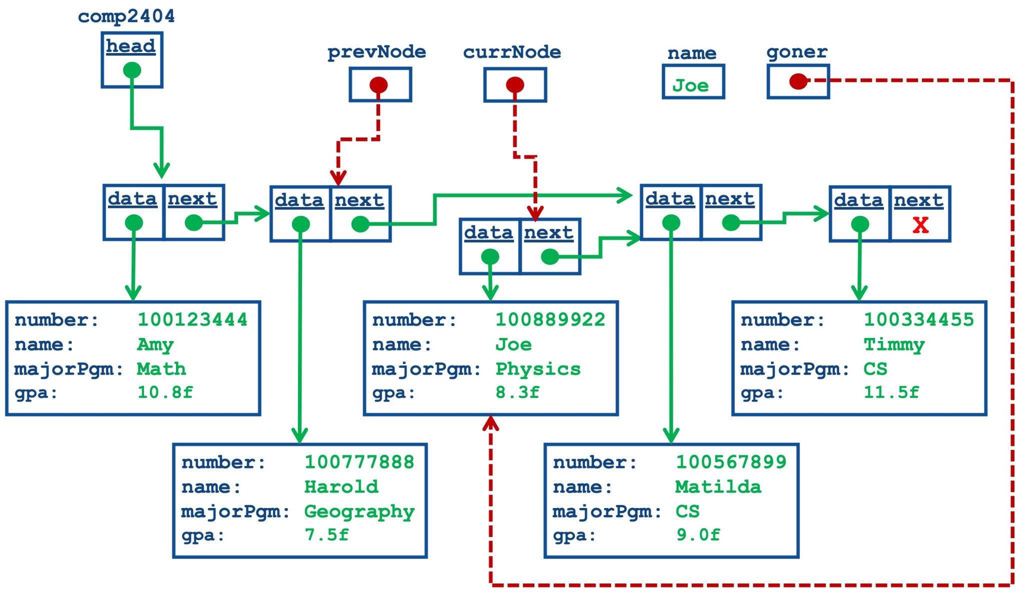
- These lines connect the target student's preceding node to the target student's following node, effectively removing the target student from the list.
- If the target student is located at the very beginning of the list, then the list head must be updated to the target's next node in the list, found in `currNode->next`. We detect this case by checking if `prevNode` has a null value on line 22. If it does, then the list head is updated on line 23.
- If the target student is located in the middle or end of the list, then its preceding node `prevNode` is linked to the target's following node, found in `currNode->next` on line 26.

Lines 28-29:

- These lines perform the final housekeeping duties in this function.
- Line 28 initializes the target student pointer in the `goner` parameter to the `Student` pointer stored in the target node `currNode`.
- Line 29 deallocates the dynamically allocated memory associated with the target student's node. Because the only job of a node is to link a data element into the linked list, if the element is removed from the list, then the node is no longer needed. We **do not** deallocate the data, since it is returned in the `goner` parameter for the calling function to use.



(a) Before deletion



(b) After deletion

Figure-8.5: Linked list exercise: removing from the middle or end of the list

8.7.4.1. **Deletion exercise #1: removing from the middle or end of the list**

- Figure-8.5 demonstrates the deletion of a student *from the middle or end* of a linked list, using the `List` class's `del()` member function shown in partial Program-8.11.
- Figure-8.5a shows the linked list before the student is removed. The existing list contains five students named Amy, Harold, Joe, Matilda, and Timmy. This exercise shows the deletion of student Joe.
- Figure-8.5b shows the linked list after the deletion of student Joe, as well as the values of the parameters and temporary pointers inside the `del()` member function. The student name `"Joe"` is passed in to the function as the `name` parameter.
- In Figure-8.5b, we see that student Joe was found. So when the control flow breaks out of the loop on line 12 of partial Program-8.11, the preceding node pointer `prevNode`'s pointee is Harold's node, and the target node pointer `currNode`'s pointee is Joe's node.
- Line 26 resets the `next` pointer inside Harold's node to the node following Joe, which is Matilda's node. That way, Joe's node is removed from the linked list.

- This logic also works for removing the last element of the list, for example student Timmy. In that case, the loop ends with `prevNode` pointing to Matilda's node and `currNode` pointing to Timmy's node. Setting the `next` pointer inside Matilda's node to the node following Timmy, which is the null value, successfully removes Timmy from the list.
- Line 28 sets the `goner` output parameter's pointee to the `Student` object corresponding to the target student Joe.
- Line 29 deallocates Joe's node, since its element is no longer in the list.

8.7.4.2. **Deletion exercise #2: removing from the beginning of the list**

- Figure-8.6 demonstrates the deletion of a student *from the beginning* of a linked list, using the `List` class's `del()` member function shown in partial Program-8.11.
- Figure-8.5b shows the list before the student is removed. It contains four students named Amy, Harold, Matilda, and Timmy. This exercise shows the deletion of student Amy.
- Figure-8.6 shows the linked list after the deletion of student Amy, as well as the values of the parameters and temporary pointers inside the `del()` member function. The student name `"Amy"` is passed in to the function as the `name` parameter.
- In Figure-8.6, we see that student Amy was found. So when the control flow breaks out of the loop on line 12 of Program-8.11, the preceding node pointer `prevNode` has a null value, and the target node pointer `currNode`'s pointee is Amy's node.
- Line 22 detects that we are removing the first element, so the list head must be updated.
- Line 23 resets the list head from Amy's node to the following node, which is Harold's node. That way, Amy's node is removed from the linked list.
- Line 28 sets the `goner` output parameter's pointee to the `Student` object corresponding to the deleted student Amy.
- Line 29 deallocates Amy's node, since its element is no longer in the list.

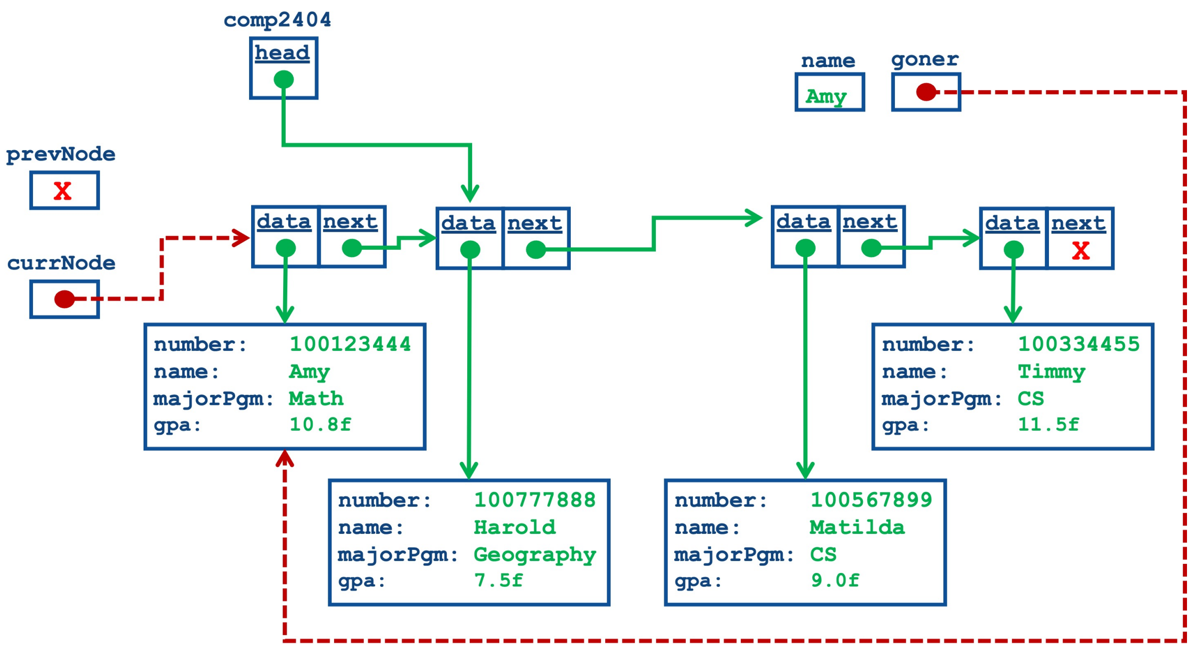


Figure-8.6: Linked list exercise: removing from the beginning of the list

8.7.5. Printing the list

```
 1 /* * * * * * * * * * * * * * * * *
 2  * Filename:  List.cc (partial)   *
 3  * * * * * * * * * * * * * * * * */
 4 void List::print() const
 5 {
 6   Node* currNode;

 8   cout << "STUDENTS:" << endl;
 9   for (Node* currNode=head;
10        currNode != nullptr;
11        currNode = currNode->next) {
12     currNode->data->print();
13   }
14 }
```

Program-8.12: Linked list printing

Partial program purpose:

- Partial Program-8.12 shows the implementation of the `print()` member function of the `List` class defined in partial Program-8.8.
- This function traverses the linked list and prints the student data of every list element.

Line 6:

- To traverse the linked list, we need a `Node` pointer as a looping variable.
- Line 6 declares the `currNode` pointer for this purpose.

Lines 9-13:

- These lines show the loop that traverses the linked list and prints each student element.
- Lines 9-11 define the loop header of a `for`-loop that iterates through the list. We could have used a `while` loop very similar to both partial Program-8.10 and partial Program-8.11, but here we use a `for`-loop to demonstrate an alternative implementation.
- The first part of a `for`-loop header initializes the looping variable. Here, line 9 shows that the `currNode` looping variable is initialized to the list head.
- The second part of a `for`-loop header is the iteration condition. As long as this condition is true, the loop keeps executing for one more iteration. Line 10 shows that the loop iterates until the end of the list is reached when `currNode` is set to a null value.
- The third part of a `for`-loop header is the looping variable's advancing statement, which is often an increment or decrement operation. Line 11 shows that we advance to the next element by setting the `currNode` looping variable to the next element's node in the list.
- Line 12 calls the `Student` class's `print()` member function for the current element, which is the `Student` object corresponding to the current node.
- This loop does **not** try to print the student information itself, since that would be very bad encapsulation. The `List` class should have no knowledge of `Student` information. The only part of the program that should know how to print student data is the `Student` class.

8.7.6. Cleaning up the list

```
 1 /* * * * * * * * * * * * * * * * *
 2  * Filename:  List.cc (partial)   *
 3  * * * * * * * * * * * * * * * * */
 4 List::~List()
 5 {
 6   Node *currNode, *nextNode;
 7   currNode = head;

 9   while (currNode != nullptr) {
10     nextNode = currNode->next;
11     delete currNode->data;
12     delete currNode;
13     currNode = nextNode;
14   }
15 }
```

Program-8.13: Linked list cleanup

Partial program purpose:

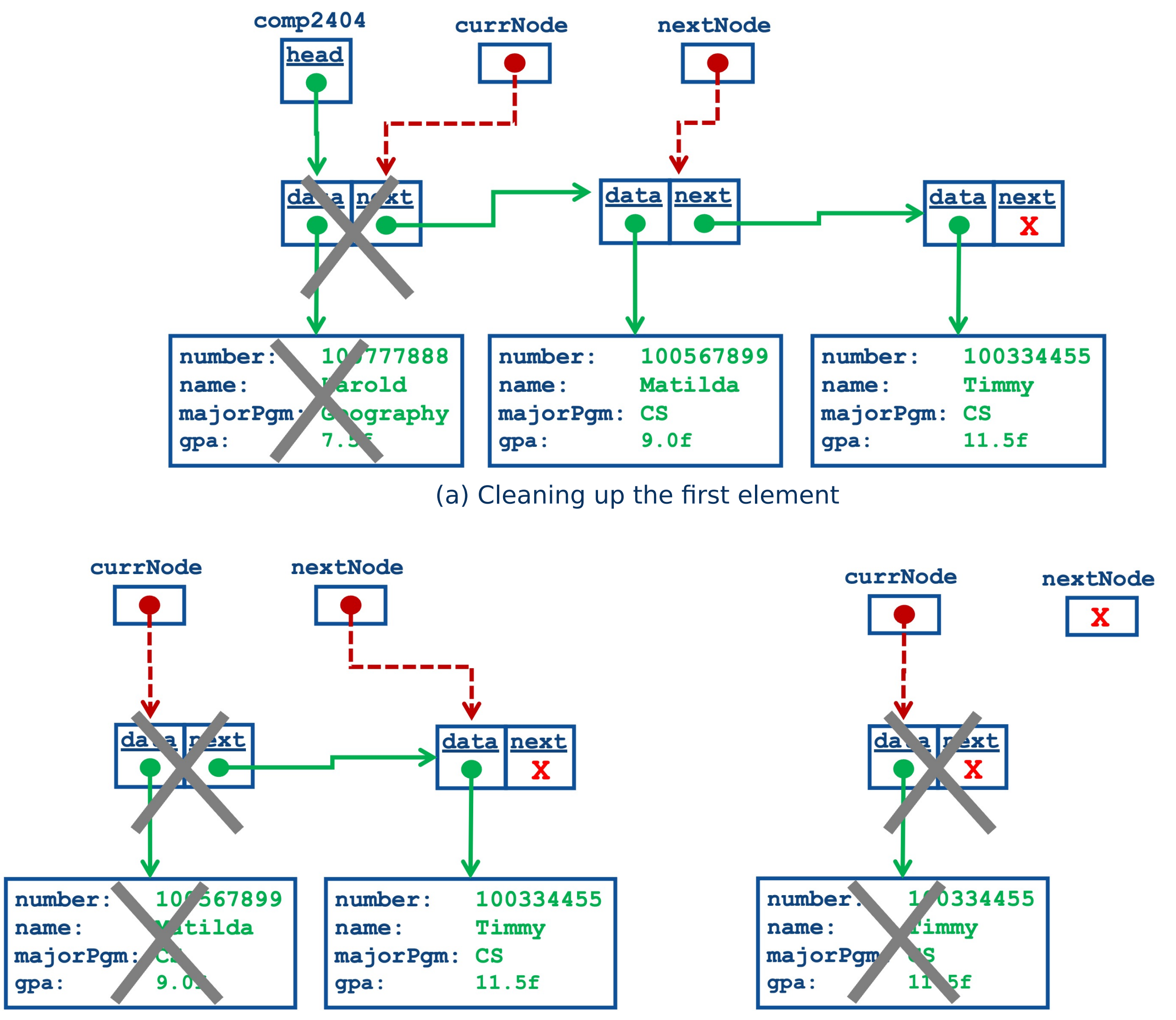
- Partial Program-8.13 shows the implementation of the `List` class destructor.
- The destructor is called automatically when a statically allocated `List` object moves out of scope, or when a dynamically allocated `List` object is deallocated.

Lines 6-7:

- The goal of the destructor is to traverse the linked list and deallocate every remaining data element its corresponding node.
- Line 6 declares two node pointers. The `currNode` pointer is used as the looping variable to traverse the list, and the `nextNode` pointer stores the following node as its pointee.
- Unlike the `add()` and `del()` member functions, the destructor does *not* store a preceding node. Instead, it keeps track of a next node because the loop on lines 9-14 is a *destructive* loop, which means that it alters the list irreparably. We must save the link to the following node into a separate variable before we deallocate the current node. Otherwise, when we deallocate the current node, we lose the rest of the list because the memory where the next node pointer is stored is no longer accessible.
- Line 7 ensures that the list traversal begins at the head.

Lines 9-14:

- These lines show the loop that traverses the linked list and deallocates its nodes and data.
- Line 9 declares the header of a `while` loop that terminates if `currNode` has reached the null pointer value at the end of the linked list.
- Before we deallocate the current element, we save the following node pointer into a separate `nextNode` variable on line 10.
- Line 11 deallocates the current `Student` data element. This must be done before deallocating the node, otherwise we lose access to the data element.
- Line 12 deallocates the current element's node.
- Line 13 advances `currNode` to the next node in list, as saved in the `nextNode` variable, so that the next iteration of the loop can deallocate the next element.



(a) Cleaning up the first element

(b) Cleaning up the second element

(c) Cleaning up the third element

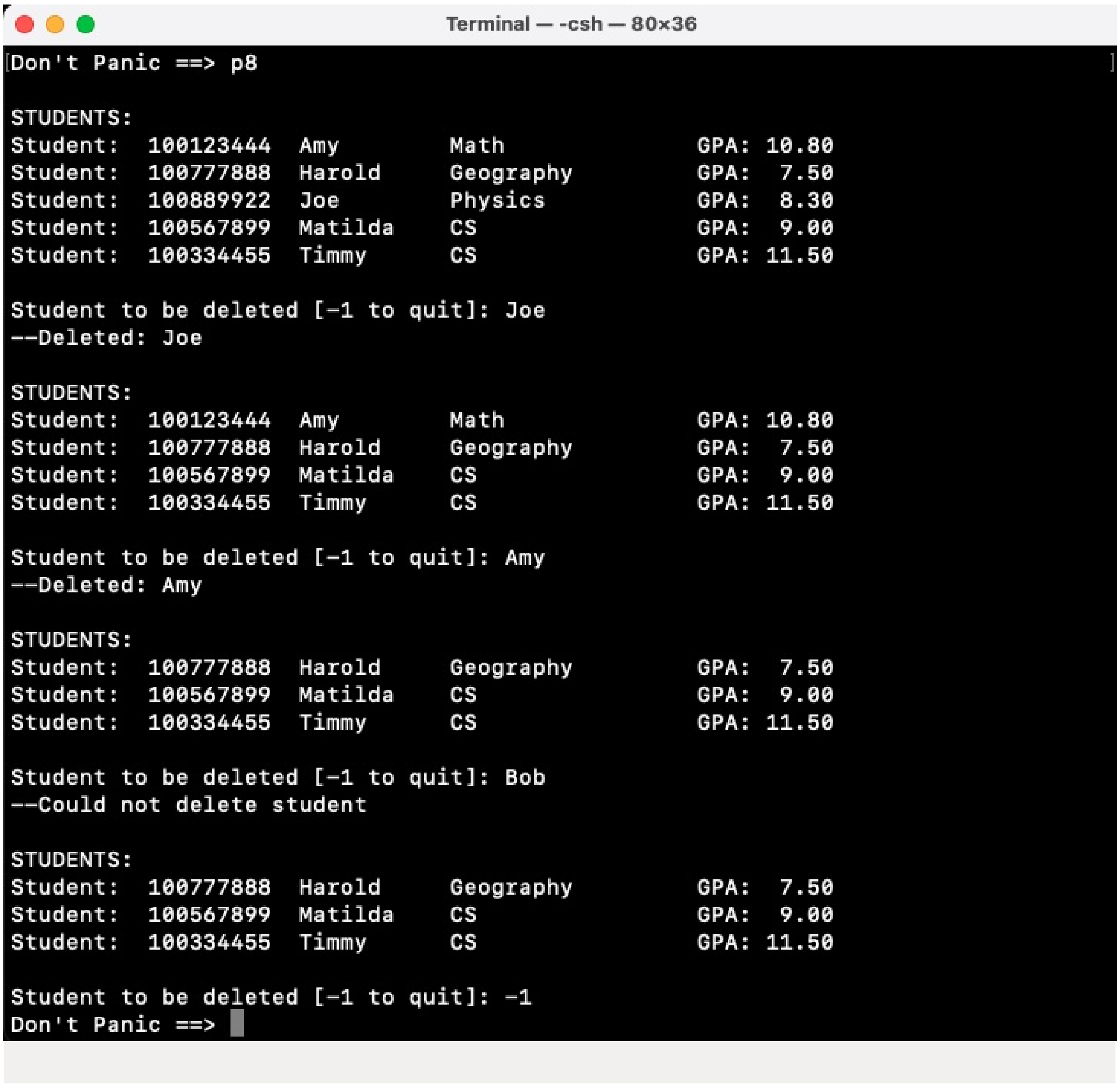
Figure-8.7: Linked list exercise: cleaning up the nodes and data

8.7.6.1. **Clean up exercise: cleaning up the nodes and data**

- Figure-8.7 demonstrates the cleaning up of a linked list by deallocating the memory occupied by its data and nodes, using the `List` class's destructor shown in partial Program-8.13. In the figure, a large X over an object represents the deallocation of that object.
- All the memory in the linked list exercise is dynamically allocated, except for the `List` object called `comp2404` that contains the list head. This variable is local to the `main()` function, as we see in the next part of the exercise.
- Figure-8.7a shows the linked list at the end of the loop's first iteration, as well as the values of the temporary pointers inside the destructor. We see that the current node's pointee is Harold's node, which is the first element deallocated. The following node is saved into `nextNode` as a pointer to Matilda's node.
- In Figure-8.7b, we see the second iteration of the loop. The current node's pointee is Matilda's node, which is deallocated. The following node is set as a pointer to Timmy's node.
- Figure-8.7c shows the third and final iteration of the loop. The current node's pointee is Timmy's node, which is deallocated. The following node is set to a null value, since we are at the last element in the list.
- After deallocating Timmy's data and node, the loop terminates.

8.7.7. Main control flow

```cpp
/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc             *
 * * * * * * * * * * * * * * * * */
int main()
{
  Student *matilda = new Student("100567899", "Matilda", "CS", 9.0f);
  Student *harold  = new Student("100777888", "Harold", "Geography",
                                 7.5f);
  Student *joe     = new Student("100889922", "Joe", "Physics", 8.3f);
  Student *timmy   = new Student("100334455", "Timmy", "CS", 11.5f);
  Student *amy     = new Student("100123444", "Amy", "Math", 10.8f);

  List comp2404;
  comp2404.add(matilda);
  comp2404.add(harold);
  comp2404.add(joe);
  comp2404.add(timmy);
  comp2404.add(amy);
  comp2404.print();

  Student* someStu;
  string   stuName = "";

  while (1) {
    cout << "Student to be deleted [-1 to quit]: ";
    cin  >> stuName;
    if (stuName == "-1")
      break;

    comp2404.del(stuName, &someStu);
    if (someStu == NULL) {
      cout << "--Could not delete student" << endl;
    }
    else {
      cout << "--Deleted: " << someStu->getName() << endl;
      delete someStu;
    }

    comp2404.print();
  }

  return 0;
}
```



```
Don't Panic ==> p8

STUDENTS:
Student:  100123444  Amy        Math             GPA: 10.80
Student:  100777888  Harold     Geography        GPA:  7.50
Student:  100889922  Joe        Physics          GPA:  8.30
Student:  100567899  Matilda    CS               GPA:  9.00
Student:  100334455  Timmy      CS               GPA: 11.50

Student to be deleted [-1 to quit]: Joe
--Deleted: Joe

STUDENTS:
Student:  100123444  Amy        Math             GPA: 10.80
Student:  100777888  Harold     Geography        GPA:  7.50
Student:  100567899  Matilda    CS               GPA:  9.00
Student:  100334455  Timmy      CS               GPA: 11.50

Student to be deleted [-1 to quit]: Amy
--Deleted: Amy

STUDENTS:
Student:  100777888  Harold     Geography        GPA:  7.50
Student:  100567899  Matilda    CS               GPA:  9.00
Student:  100334455  Timmy      CS               GPA: 11.50

Student to be deleted [-1 to quit]: Bob
--Could not delete student

STUDENTS:
Student:  100777888  Harold     Geography        GPA:  7.50
Student:  100567899  Matilda    CS               GPA:  9.00
Student:  100334455  Timmy      CS               GPA: 11.50

Student to be deleted [-1 to quit]: -1
Don't Panic ==> 
```

Program-8.14: Linked list exercise: `main()` function

Partial program purpose:

- Program-8.14 demonstrates the `main()` function's use of the `List` class previously defined in partial Program-8.8.
- The program declares a local `List` object and adds some `Student` objects to it. The end-user is prompted repeatedly for the name of a student to remove from the list, which is printed out after every deletion.

Lines 6-19:

- These lines allocate and initialize five students, as well as a linked list. They add the students to the list and print it out to the screen.
- Lines 6-11 dynamically allocate and initialize five `Student` objects.
- Line 13 declares a `List` object, as defined partial Program-8.8. This calls the `List` default constructor shown in partial Program-8.9.
- Lines 14-18 add each of the five students to the linked list by calling the `List` class's `add()` member function shown in partial Program-8.10.
- Line 19 calls the `print()` member function shown in partial Program-8.12.

Lines 24-40:

- These lines show the `while` loop that repeatedly deletes a user-selected student from the linked list and prints the list.
- On lines 25-28, the end-user is prompted to enter a student name, and we break out of the loop if the sentinel value `"-1"` is entered.
- Line 30 removes from the linked list the student with the user-entered name by calling the `List` class's `del()` member function shown in partial Program-8.11.
- Lines 31-33 deal with the case where the user-entered student name does not appear as an element in the linked list. This is detected by checking the `del()` member function's output parameter, returned into the `someStu` local variable, for a null value.
- Lines 34-37 deal with the case where the user-entered student name was found in the list. The student's name is printed out on line 35, using the returned `Student` pointer. The corresponding `Student` object is deallocated on line 36, as `someStu` is the only remaining pointer to that object.
- Line 39 prints out the contents of the linked list at the end of each iteration.
- Line 42 terminates the `main()` function. As a result, the statically allocated linked list `comp2404` is automatically deallocated, which executes the `List` object's destructor and deallocates all the remaining nodes and student data, as shown in partial Program-8.13.

NOTE: In programming, a ***sentinel value*** is a value that has a special meaning. It's often used as a breaking condition in a loop to indicate the end of the input.

Chapter 9

Inheritance

Inheritance is a key OO design technique that allows programmers to organize objects into a generalization and specialization hierarchy. Informally known as an "is-a" relationship, inheritance between two classes establishes one class as a specialized type of a more generalized class. Inheritance is a key technique for abstraction, as presented by Liskov [8].

In this chapter, we discuss the basics of inheritance and their implementation in C++, including the construction and destruction of objects. We also introduce the different types of inheritance and how C++ enables multiple inheritance.

9.1. Principles

We introduce the principles of inheritance and their terminology in C++, and we discuss how access specifiers behave with classes in an inheritance relationship.

9.1.1. Terminology

9.1.1.1. **What is *inheritance*:**

- In OO design, ***inheritance*** is a relationship between two classes where one is a specialized sub-type of a more generalized one.
- It is often conceptualized as an **is-a** relationship between two classes.
- For example, an *undergraduate student* and a *graduate student* are both a specialized kind of *student*. Both sub-types of students have many attributes in common that can be captured in the more generalized *student* class, for example their name, student number, and GPA.
- In less formal terms, the more generalized class is sometimes called the *parent* class or the *superclass*, and the more specialized class is called the *child* class or the *subclass*.
- Inheritance is an important technique for the abstraction and encapsulation of object data and behaviour.

9.1.1.2. **C++ terminology:**

- In C++, the more generalized class in an inheritance relationship is called the ***base class***, and the more specialized class is called the ***derived class***.
- In the example above, the `Student` class is the base class, and both `UndergradStudent` and `GradStudent` are the derived classes. We say that `UndergradStudent` and `GradStudent` are *derived from* the `Student` class.
- In UML, the inheritance relationship is shown with a clear (not filled) triangle at the superclass end of the line between the two classes, as discussed in section 7.3.

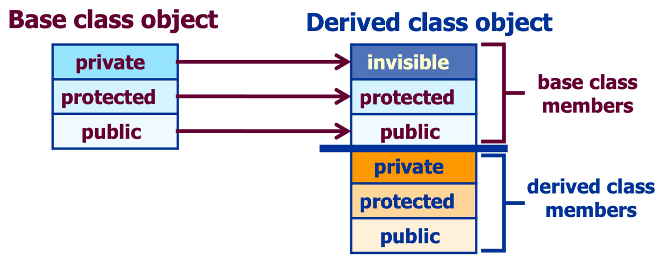
9.1.2. Member access

9.1.2.1. **Inherited members:**

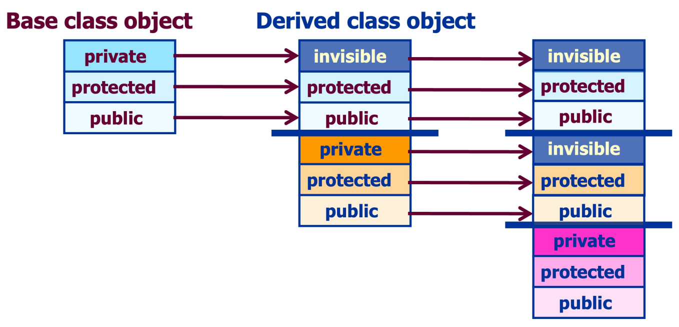
- All class members defined in a base class are automatically inherited by its derived class(es).
- Inherited members class become part of the derived class simply by declaring the inheritance relationship. These members should **never** be redefined in the derived class, either in the code or in a UML class diagram.
- Inherited members include *all class members*, including data members and member functions, whether they have public, protected, or private access specifiers.
- Composition relationships are inherited, since every data member of a base class becomes part of a derived class object, including the base class's containee objects.
- Friendship is *not* inherited.

9.1.2.2. **Accessing inherited members:**

- While all base class members are inherited by the derived class, they are **not** all directly accessible from the derived class.
- *Only public and protected base class members* can be accessed directly in the derived class. Those members can be accessed as if they were members of the derived class.
- Private base class members are *not* directly accessible in the derived class.
- If a base class declares some of its members as private, it means that those members are *private to the base class*.
- Inherited private base class members are still part of each derived class object! But they are invisible and inaccessible directly in the derived class.
- Private base class members can be accessed by the derived class using the base class's public or protected member functions, or the base class's friend classes and friend functions.



(a) One level of inheritance



(b) Two levels of inheritance

Figure-9.1: Class member inheritance

9.1.2.3. **Example:**

- Figure-9.1 shows the behaviour of access specifiers in derived objects.
- In Figure-9.1a, we see that private members in the base class become invisible in a derived class object, but public and protected members keep their original access.
- In Figure-9.1b, we see the same effect with two levels of inheritance. Members that are invisible or private in the base class are invisible in the derived class object. Public and protected members maintain the same access no matter the level of inheritance.

9.1.3. Coding example: Simple inheritance

The program in this example implements the UML class diagram shown in Figure-9.2.

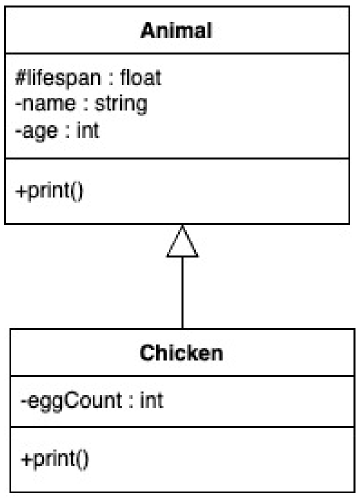


Figure-9.2: UML class diagram for Program-9.1

The `Animal` class:

- We see from Figure-9.2 that the `Animal` class contains three data members: a protected data member for the animal's expected lifespan in years, a private data member for the animal's name, and a private data member for the animal's current age in years.
- The `Animal` class has a public member function that prints out the animal's data.
- It also has other member functions that are not shown in the UML class diagram, including a default constructor, a destructor, and a getter for the name data member.

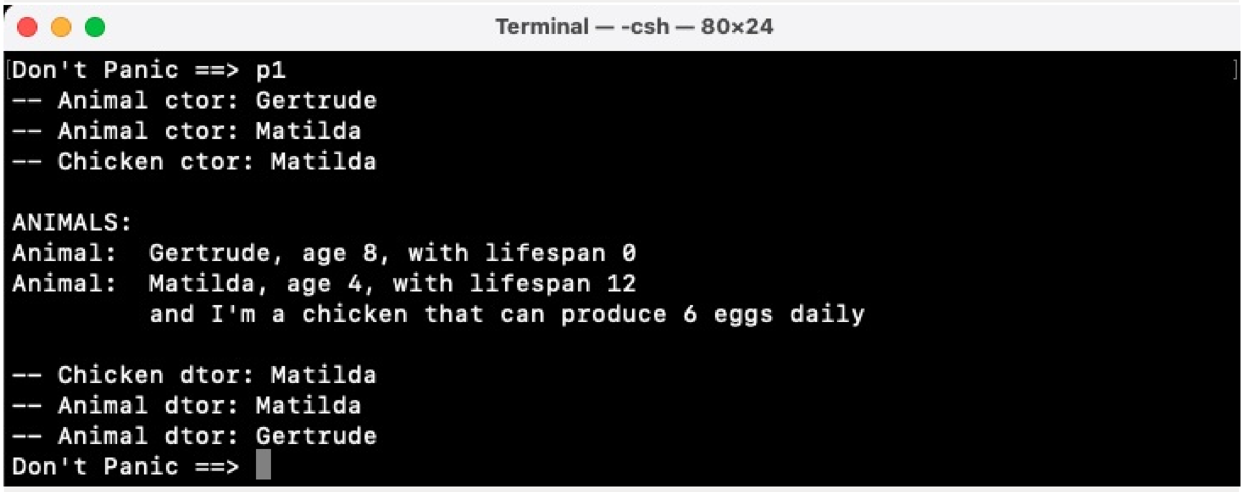
The `Chicken` class:

- Figure-9.2 shows that the `Chicken` class is derived from the `Animal` class.
- The `Chicken` class contains four data members: the three data members inherited from the `Animal` class, and a private data member that stores the number of eggs that the chicken is expected to produce on a daily basis.
- The `Chicken` class has a public member function that prints out the chicken's data, as well as a default constructor and a destructor.

```cpp
/* * * * * * * * * * * * * * * * *
 * Filename:  Animal.h            *
 * * * * * * * * * * * * * * * * */
class Animal
{
  public:
    Animal(string="Fluffy", int=0, float=0);
    ~Animal();
    string getName() const;
    void   print()   const;
  protected:
    float  lifespan;
  private:
    string name;
    int    age;
};

```

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Animal.cc          *
 * * * * * * * * * * * * * * * * */
Animal::Animal(string n, int a, float lf)
    : name(n), age(a), lifespan(lf)
{
  cout << "-- Animal ctor: " << name << endl;
}

Animal::~Animal()
{
  cout << "-- Animal dtor: " << name << endl;
}

string Animal::getName() const { return name; }

void Animal::print() const
{
  cout << "Animal:  " << name << ", age " << age
       << ", with lifespan " << lifespan << endl;
}

/* * * * * * * * * * * * * * * * *
 * Filename:  Chicken.h          *
 * * * * * * * * * * * * * * * * */
class Chicken : public Animal
{
  public:
    Chicken(string="Little Red Hen", int=0, float=0, int=0);
    ~Chicken();
    void   print() const;
  private:
    int eggCount;
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Chicken.cc         *
 * * * * * * * * * * * * * * * * */
Chicken::Chicken(string n, int a, float lf, int ec)
    : Animal(n,a,lf), eggCount(ec)
{
  cout << "-- Chicken ctor: " << getName() << endl;
}

Chicken::~Chicken()
{
  cout << "-- Chicken dtor: " << getName() << endl;
}

void Chicken::print() const
{
  Animal::print();
  cout << "         and I'm a chicken that can produce "
       << eggCount << " eggs daily" << endl;
}

```

```
74 /* * * * * * * * * * * * * * * * *
75  * Filename:  main.cc            *
76  * * * * * * * * * * * * * * * * */
77 int main()
78 {
79   Animal  gertrude("Gertrude", 8);
80   Chicken matilda("Matilda", 4, 12, 6);

82   cout << endl << "ANIMALS:" << endl;
83   gertrude.print();
84   matilda.print();

86   return 0;
87 }
```



```
Don't Panic ==> p1
-- Animal ctor: Gertrude
-- Animal ctor: Matilda
-- Chicken ctor: Matilda

ANIMALS:
Animal:  Gertrude, age 8, with lifespan 0
Animal:  Matilda, age 4, with lifespan 12
         and I'm a chicken that can produce 6 eggs daily

-- Chicken dtor: Matilda
-- Animal dtor: Matilda
-- Animal dtor: Gertrude
Don't Panic ==> 
```

Program-9.1: Simple inheritance

Program purpose:

- Program-9.1 demonstrates the use of inheritance in C++ with two classes, based on the design shown in Figure-9.2.

Lines 4-16:

- These lines show the `Animal` class definition.
- Line 12 declares the `lifespan` data member in the protected area defined on line 11, which means that derived class objects can access the data member directly.
- Lines 14-15 declare the two data members `name` and `age` in the private area defined on line 13, which means that derived class objects *cannot* access them directly.
- We see on line 7 that the default constructor takes three parameters, one for each data member declared in this class.

Lines 43-51:

- These lines show the `Chicken` class definition.
- Line 43 defines the `Chicken` class as derived from the `Animal` class. The syntax requires that the derived class name is followed by a colon, then the type of inheritance (public), then the name of the base class. Types of inheritance are discussed later in this chapter.
- Line 50 declares the `eggCount` data member as private. We do **not** redefine the three data members inherited from `Animal`, but they are still present in every `Chicken` object created.
- We see on line 46 that the default constructor takes four parameters, one for each data member declared in the `Animal` class and in the `Chicken` class.

- In section 8.1.1, we noted that correct encapsulation requires that a container object constructor only initializes its own data members, and not the containee data members, which must be initialized by the containee object constructor.
- The same encapsulation rules apply to objects related by inheritance. A derived class constructor must only initialize its own data members, and not the base class members. The base class data members must be initialized by the base class constructor.

Lines 56-60:

- These lines show the implementation of the `Chicken` class's default constructor.
- We see on line 56 that the `Chicken` constructor takes in all the parameter values necessary to initialize a new `Chicken` object, including both its `Animal` and its `Chicken` data members.
- However, it is **not** the responsibility of the `Chicken` constructor to initialize the `Animal` data members. It must call the `Animal` constructor to do this.
- We see how the `Chicken` constructor calls the `Animal` constructor on line 57, using *base class initializer syntax*.
- Remember member initializer syntax from section in 8.1.1? We used member initializer syntax to avoid the default initialization of containee objects when creating a container object. Similarly, base class initializer syntax avoids the default initialization of base class data members by calling the base class constructor with the correct parameter values.
- Line 57 does two things:
  - it uses base class initializer syntax to call the base class `Animal` constructor with the first three parameter values, in order to correctly initialize the `Animal` data members, and
  - it uses member initializer syntax to initialize the `Chicken` data member using the remaining parameter value
- Line 59 prints out a debugging statement with the new `Chicken` object's name. But because the `name` data member is private to the `Animal` base class, the `Chicken` class cannot access it directly. A public member function in the `Animal` class is called to get the chicken's name.

Lines 67-72:

- These lines show the implementation of the `Chicken` class's printing member function. This function is responsible for printing out *all* the `Chicken` object's data members, including the `Animal` ones. However, correct encapsulation requires that only the `Animal` class knows how to print out `Animal` data members.
- The solution is simply for the `Chicken` printing function to output only the `Chicken` data member, and to call the `Animal` class's printing function for the printing of its data members.
- We see on line 69 how the `Animal` printing function is called. This line *cannot* simply call `print()`, since that would create an endless recursive call to the `Chicken` printing function. Instead, we must use the `Animal` class name followed by the scope resolution operator to specify that it's the `Animal` printing function that is called on this line.

Lines 77-87:

- These lines show the implementation of the `main()` function.
- Line 79 shows the creation of an `Animal` object. The `Animal` constructor is called with only two parameters: the new animal's name and its age. The constructor uses the default value indicated on line 7 to initialize the lifespan data member.
- Line 80 declares a `Chicken` object. In this case, all four parameter values are provided, including the egg count.
- Lines 83-84 print out the contents of both objects, and we see the correct values in the program output.

NOTE: The reader should note that all the animals in these coding examples are highly fictionalized. This author is aware that hens do not produce multiple eggs per day, but alas creative license must be suffered.

9.2. Overloading member functions

We discuss the concepts of overloading and overriding member functions in C++ and their consequences for program behaviour.

9.2.1. Concepts

9.2.1.1. **Function prototypes and signatures:**

- A function ***prototype*** is the combination of a function name, the set of its parameter data types, and its return type.
- A function ***signature*** is the combination of a function name and the set of its parameter data types only, without the return type.
- This distinction is important in all discussions on overloading in C++. The topic is expanded upon in chapter 12.

9.2.1.2. ***Overriding* a class member function:**

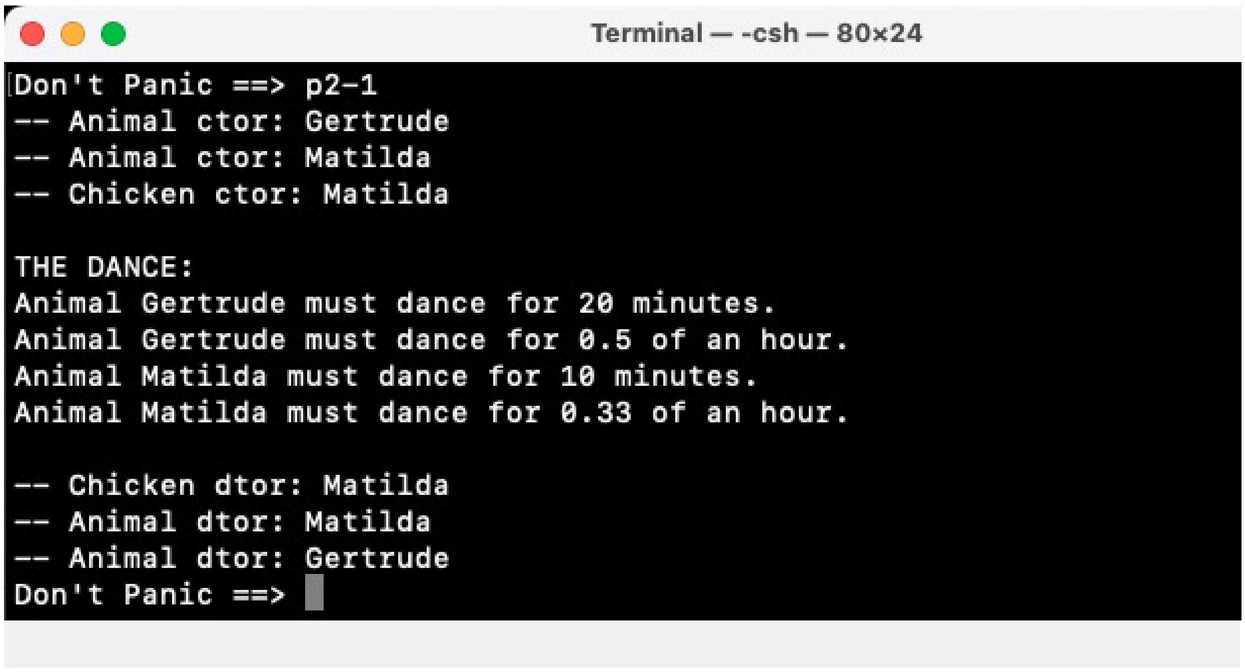
- *Overriding* redefines a function inherited from a base class with another function with the same name, and *with the exact same signature*, in a derived class.
- This feature is one of the fundamental requirements for *runtime polymorphism*, which is a highly powerful technique in OO design and the topic of chapter 10.
- An overridden base class function can still be accessed in the derived class, but it must be called using the base class name and the scope resolution operator, as we did on line 69 of Program-9.1.

9.2.1.3. ***Overloading* a class member function:**

- *Overloading* redefines a function, either declared in the same class or inherited from a base class, with another function of the same name, but *with a different signature*.
- When a derived class overloads an inherited member function, it effectively **hides** the base class function from the derived class objects, even though the base class function is inherited.
- Just as with an overridden function, an overloaded inherited one can still be called using the scope resolution operator.
- Overloading is an important feature in many OO languages, and we explore this topic further in chapter 12.

9.2.2. Coding example: Using inherited member functions

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Animal.h           *
 * * * * * * * * * * * * * * * * */
class Animal
{
  public:
    Animal(string="Fluffy", int=0, float=0);
    ~Animal();
    string getName()      const;
    void   print()        const;
    void   dance(int)     const;
    void   dance(double)  const;
  protected:
    float  lifespan;
  private:
    string name;
    int    age;
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Animal.cc (partial)*
 * * * * * * * * * * * * * * * * */
void   Animal::dance(int mins) const
{
  cout << "Animal " << name << " must dance for "
       << mins << " minutes." << endl;
}

void   Animal::dance(double frac) const
{
  cout << "Animal " << name << " must dance for "
       << frac << " of an hour." << endl;
}

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc            *
 * * * * * * * * * * * * * * * * */
int main()
{
  Animal  gertrude("Gertrude", 8);
  Chicken matilda("Matilda", 4, 12, 6);

  cout << endl << "THE DANCE:" << endl;
  gertrude.dance(20);
  gertrude.dance(0.5);
  matilda.dance(10);
  matilda.dance(0.33);
  cout << endl;

  return 0;
}
```



```
Don't Panic ==> p2-1
-- Animal ctor: Gertrude
-- Animal ctor: Matilda
-- Chicken ctor: Matilda

THE DANCE:
Animal Gertrude must dance for 20 minutes.
Animal Gertrude must dance for 0.5 of an hour.
Animal Matilda must dance for 10 minutes.
Animal Matilda must dance for 0.33 of an hour.

-- Chicken dtor: Matilda
-- Animal dtor: Matilda
-- Animal dtor: Gertrude
Don't Panic ==> 
```

Program-9.2: Using inherited member functions

Program purpose:

- Program-9.2 demonstrates the use of inherited functions *without* overloading them in the derived class.
- The program uses the same `Chicken` class defined in Program-9.1.

Lines 11-12:

- These lines show the declaration of two member functions called `dance()` in the `Animal` class definition.
- The two `dance()` member functions are overloaded in the `Animal` class, but *not* overloaded in the `Chicken` class. One `dance()` member function takes an integer as parameter, and the other takes a double.

Lines 23-33:

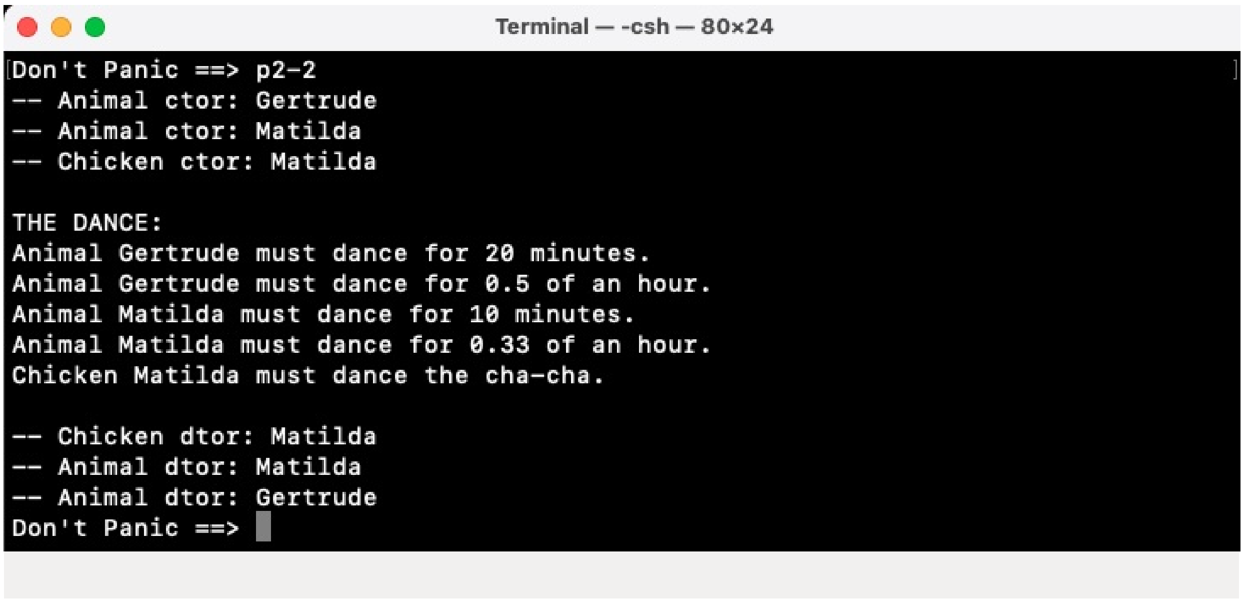
- These lines show the implementation of the two `dance()` member functions.
- Each member function prints out a different message.

Lines 38-51:

- These lines show the implementation of the `main()` function.
- Lines 40-41 create an `Animal` object and a `Chicken` object.
- Lines 44-47 show the two `dance()` member functions called on each object.
- The program output shows that the `Animal` class's two `dance()` member functions are successfully called on both the `Animal` object and the `Chicken` object.

9.2.3. Coding example: Using overloaded inherited member functions

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Chicken.h          *
 * * * * * * * * * * * * * * * * */
class Chicken : public Animal
{
  public:
    Chicken(string="Little Red Hen", int=0, float=0, int=0);
    ~Chicken();
    void print() const;
    void dance(string) const;
  private:
    int eggCount;
};

/* * * * * * * * * * * * * * * * * *
 * Filename:  Chicken.cc (partial) *
 * * * * * * * * * * * * * * * * * */
void Chicken::dance(string d) const
{
  cout << "Chicken " << getName() << " must dance the "
       << d << "." << endl;
}

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc            *
 * * * * * * * * * * * * * * * * */
int main()
{
  Animal  gertrude("Gertrude", 8);
  Chicken matilda("Matilda", 4, 12, 6);

  cout << endl << "THE DANCE:" << endl;
//  gertrude.dance("tango");
  gertrude.dance(20);
  gertrude.dance(0.5);

//  matilda.dance(10);
//  matilda.dance(0.33);
  matilda.Animal::dance(10);
  matilda.Animal::dance(0.33);
  matilda.dance("cha-cha");
  cout << endl;

  return 0;
}
```



```
Don't Panic ==> p2-2
-- Animal ctor: Gertrude
-- Animal ctor: Matilda
-- Chicken ctor: Matilda

THE DANCE:
Animal Gertrude must dance for 20 minutes.
Animal Gertrude must dance for 0.5 of an hour.
Animal Matilda must dance for 10 minutes.
Animal Matilda must dance for 0.33 of an hour.
Chicken Matilda must dance the cha-cha.

-- Chicken dtor: Matilda
-- Animal dtor: Matilda
-- Animal dtor: Gertrude
Don't Panic ==>
```

Program-9.3: Using overloaded inherited member functions

Program purpose:

- Program-9.3 demonstrates the use of both inherited functions and overloaded inherited functions in the derived class.
- The program uses the same `Animal` class defined in Program-9.2.

Line 10:

- This line shows the declaration of a `dance()` member function in the `Chicken` class definition.
- The `Animal` base class still contains two `dance()` member functions with different signatures, so the `dance()` member function declared on line 10 overloads the other two.
- The `Chicken` class now has three overloaded `dance()` member functions: two are inherited from `Animal` and take an integer and a double as parameter, respectively; and one is overloaded in `Chicken` and takes a string as parameter.

Lines 18-22:

- These lines show the implementation of the `Chicken` class's `dance()` member function that takes a string as parameter.

Lines 27-45:

- These lines show the implementation of the `main()` function.
- Lines 29-30 create an `Animal` object and a `Chicken` object.
- Line 33 is commented out because it doesn't compile. It attempts to call the `dance()` member function that takes a string as parameter on an `Animal` object. Since that member function is defined in the `Chicken` class only, it cannot be called on an `Animal` object.
- Lines 34-35 show the two `dance()` member functions from the `Animal` class called on an `Animal` object, which executes successfully.
- Lines 37-38 are also commented out because they don't compile. Although the `Animal` class's `dance()` member functions are inherited in `Chicken`, *they cannot be called on a `Chicken` object* using the syntax shown on lines 37-38. That's because the `Chicken` class has overloaded the `dance()` member functions with its own version.
- Lines 39-40 show the correct syntax necessary to call the base class's `dance()` member functions, using the class name and the scope resolution operator.
- Line 41 shows the `Chicken` class's `dance()` function called on a `Chicken` object, which uses regular syntax.

9.3. Constructing and destroying objects

We discuss the construction and destruction of objects with an inheritance relationship.

9.3.1. Concepts

9.3.1.1. What is *base class initializer syntax*:

- We use ***base class initializer syntax*** to provide a base class constructor with parameter values during the construction of a derived class object. It is closely related to member initializer syntax.
- When a derived class object is allocated and its constructor is called, the base class constructor also executes automatically. The call to the base class constructor can be implicit, which uses the constructor's default arguments, or explicit if *base class initializer syntax* is used.
- We use base class initializer syntax to:
  - provide parameter values to the base class constructor
  - avoid an automatic call to the base class constructor using default values
  - avoid having to provide setter member functions in the base class
- Correct encapsulation requires that each constructor initializes its own members **only**, and never the members of another class, not even a base class.
- For example, allocating a `Chicken` object automatically calls the `Chicken` constructor, which automatically calls the base class `Animal` constructor. The derived class `Chicken` constructor is responsible for initializing the `Chicken` portion of the object, and the base class constructor initializes the `Animal` part.

9.3.1.2. Order of construction of objects with inheritance:

- Objects are built "top-down". The base class portion of an object is constructed first, followed by the derived class part.
- For example, in Program-9.3, we know that the `Chicken` class is derived from `Animal`. So the construction of the `Chicken` object `matilda` generates a call to both constructors, as we see in the program output. The `Animal` portion of the `Chicken` object `matilda` is initialized first, then its `Chicken` portion.

9.3.1.3. Order of destruction of objects with inheritance:

- Objects are destroyed "bottom-up". The derived class portion of an object is destroyed first, followed by the base class part.
- For example, in Program-9.3, the destruction of the `Chicken` object `matilda` generates a call to both destructors, as we see in the program output. The `Chicken` portion of `matilda` is destroyed first, then its `Animal` portion.

9.3.1.4. Order of construction of objects with both composition and inheritance:

- Objects are built top-down and inside-out.
- The base class portion of an object is constructed first, with the base class containee objects built before the base class container.
- The derived class portion of an object is constructed last, with the derived class containee objects built before the derived class container.

9.3.1.5. **Order of destruction of objects with both composition and inheritance:**

- Objects are destroyed bottom-up and outside-in.
- The derived class portion of an object is destroyed first, with the derived class container object destroyed before the derived class containees.
- The base class portion of an object is destroyed last, with the base class container object destroyed before the base class containees.
- Destructors are usually invoked in the reverse order of constructors.

9.3.2. Coding example: Inheritance and composition

The program in this example implements the UML class diagram shown in Figure-9.3.

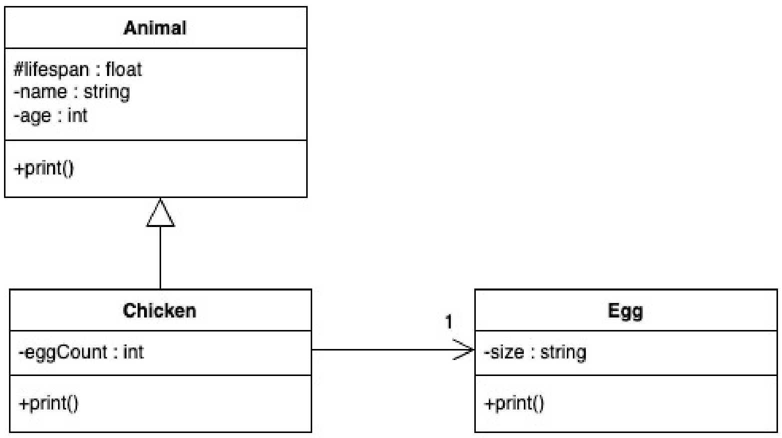


Figure-9.3: UML class diagram for Program-9.4

The `Animal` class:

- The `Animal` class in Figure-9.3 is identical to Figure-9.2.

The `Chicken` class:

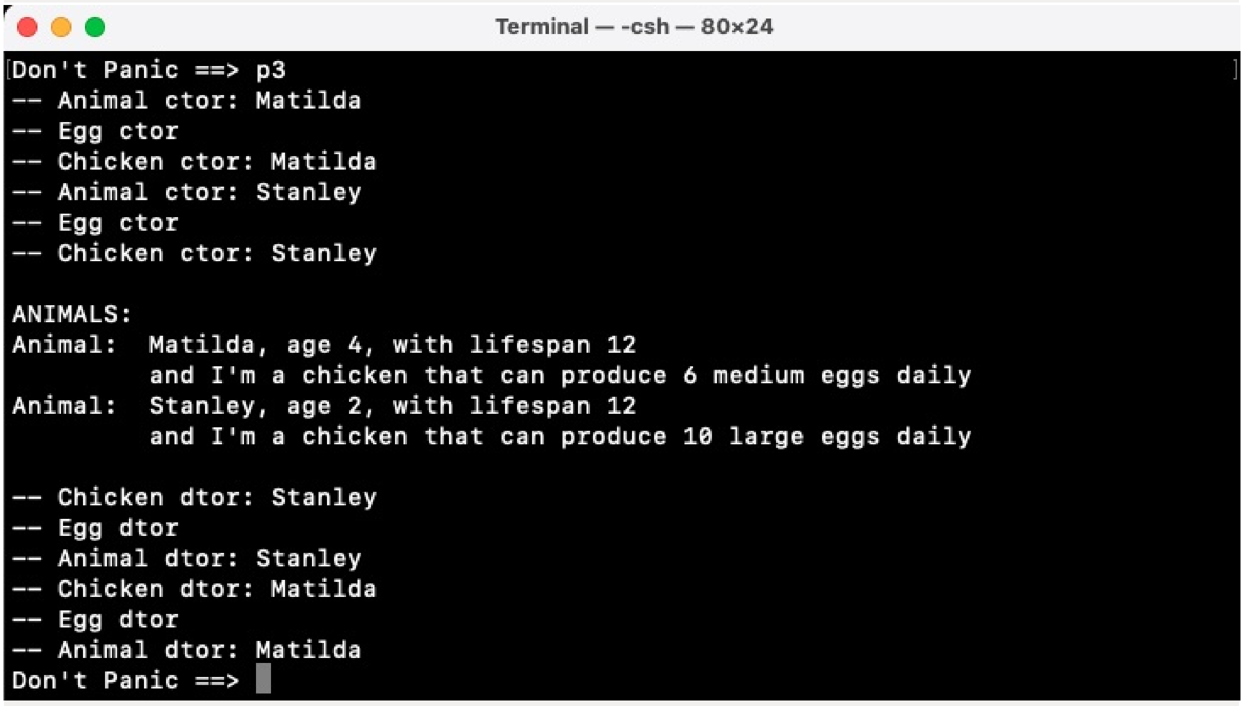
- Figure-9.3 shows that the `Chicken` class is derived from the `Animal` class.
- The `Chicken` class also has a composition relationship with the `Egg` class. From the arrow, we know that this is a unidirectional relationship, so each `Chicken` object contains an `Egg` object, but the `Egg` does not contain a `Chicken`. The multiplicity of 1 specifies that a `Chicken` object contains exactly one `Egg` object.
- Figure-9.3 indicates that the `Chicken` class has five data members: three members inherited from the `Animal` class, a private data member that stores the number of eggs that the chicken is expected to produce on a daily basis, and a containee `Egg` object.

The `Egg` class:

- The `Egg` class in Figure-9.3 contains one private data member that represents the average size of an egg, for example "medium" or "large".
- The class also provides a print member function, as well as a constructor and destructor.

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Chicken.h          *
 * * * * * * * * * * * * * * * * */
class Chicken : public Animal
{
  public:
    Chicken(string="Little Red Hen", int=0, float=0,
            int=0, string="");
    ~Chicken();
    void print() const;
  private:
    int eggCount;
    Egg eggQuality;
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Chicken.cc         *
 * * * * * * * * * * * * * * * * */
Chicken::Chicken(string n, int a, float lf, int ec, string s)
    : Animal(n,a,lf), eggCount(ec), eggQuality(s)
{
  cout<<"-- Chicken ctor: "<<getName()<<endl;
}

Chicken::~Chicken()
{
  cout<<"-- Chicken dtor: "<<getName()<<endl;
}

void Chicken::print() const
{
  Animal::print();
  cout << "         and I’m a chicken that can produce " << eggCount
  eggQuality.print(); cout << endl;
}

/* * * * * * * * * * * * * * * * *
 * Filename:  Egg.h              *
 * * * * * * * * * * * * * * * * */
class Egg
{
  public:
    Egg(string="mini");
    ~Egg();
    void print() const;
  private:
    string size;
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Egg.cc             *
 * * * * * * * * * * * * * * * * */
Egg::Egg(string s) : size(s)
{
  cout<<"-- Egg ctor"<<endl;
}
```

```
Egg::~Egg()
{
  cout<<"-- Egg dtor"<<endl;
}

void Egg::print() const
{
  cout << size << " eggs daily";
}

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc             *
 * * * * * * * * * * * * * * * * */
int main()
{
  Chicken matilda("Matilda", 4, 12, 6, "medium");
  Chicken stanley("Stanley", 2, 12, 10, "large");

  cout<<endl<<"ANIMALS:"<<endl;
  matilda.print();
  stanley.print();
  cout<<endl;

  return 0;
}
```



Terminal — -csh — 80×24

```
Don't Panic ==> p3
-- Animal ctor: Matilda
-- Egg ctor
-- Chicken ctor: Matilda
-- Animal ctor: Stanley
-- Egg ctor
-- Chicken ctor: Stanley

ANIMALS:
Animal:  Matilda, age 4, with lifespan 12
         and I'm a chicken that can produce 6 medium eggs daily
Animal:  Stanley, age 2, with lifespan 12
         and I'm a chicken that can produce 10 large eggs daily

-- Chicken dtor: Stanley
-- Egg dtor
-- Animal dtor: Stanley
-- Chicken dtor: Matilda
-- Egg dtor
-- Animal dtor: Matilda
Don't Panic ==>
```

Program-9.4: Inheritance and composition

Program purpose:

- Program-9.4 demonstrates the use of both inheritance and composition together, based on the design shown in Figure-9.3.
- The program uses the same `Animal` class defined in Program-9.1.

Lines 4-14:

- These lines show the `Chicken` class definition.
- Lines 7-8 indicate that the constructor now takes an additional string parameter to initialize the containee `Egg` object.
- Line 13 shows the declaration of an `Egg` object as a containee data member.

Lines 19-23:

- These lines show the implementation of the `Chicken` constructor.
- Line 20 shows that base class initializer syntax is used to call the `Animal` constructor with the correct parameter values; then member initializer syntax is used to both initialize the `Chicken` data member and call the containee `Egg` object's constructor with the correct parameter value. The same syntax is used to call a containee object constructor in Program-8.1.

Lines 30-35:

- These lines show the implementation of the `Chicken` printing function.
- Line 34 shows that the `Egg` printing function is called to print out the chicken object's egg information.

Lines 40-48:

- These lines show the `Egg` class definition.
- Line 43 indicates that the constructor takes a string parameter to initialize the data member.
- Line 47 declares the `size` data member.

Lines 70-81:

- These lines show the implementation of the `main()` function.
- Lines 72-73 create two `Chicken` objects. Each constructor call provides all the parameter values required to initialize the `Animal` and `Chicken` portions of each object, as well as its `Egg` containee object.
- Lines 76-77 print out the contents of both `Chicken` objects, and we see from the program output that all the data members are correctly initialized.

9.4. Types of inheritance

C++ supports a unique feature that provides a programmer with control over the inheritance of class member access specifiers, based on the type of inheritance. We discuss the different types and their usage.

9.4.1. Concepts

9.4.1.1. **What are the different types of inheritance:**

- C++ allows for three types of inheritance: public, protected, and private.
- **Only *public* inheritance** is a true is-a relationship between two classes.
- Private and protected inheritance are used as an *alternative to composition*. They are **not** a class relationship based on generalization and specialization.
- The use of private (or protected) inheritance ensures that a class has efficient access to the members of its base class, but without the violations of encapsulation that public inheritance can introduce.

9.4.1.2. **Public inheritance:**

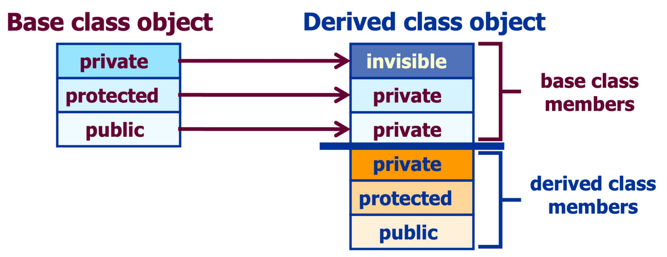
- Public inheritance is the **only** type of inheritance in C++ that implements a true is-a relationship between a superclass and a subclass.
- To use public inheritance, we specify the `public` keyword in the inheritance declaration of the derived class definition. For example, in Program-9.1, we indicate that the `Chicken` class is derived publicly from `Animal` by declaring it as:
```
    class Chicken :  public Animal {/* class content */};
```
- All the coding examples so far in this chapter have used public inheritance.

9.4.1.3. **Private and protected inheritance:**

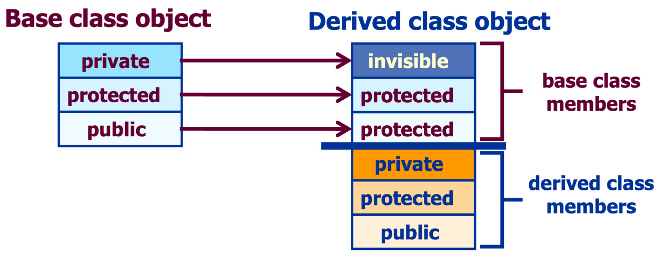
- Private and protected inheritance are **not** is-a relationships.
- They are are used to **restrict access** to the inherited base class members, in order to uphold the principle of least privilege and maintain correct encapsulation.
- Private and protected inheritance are an advanced OO programming technique that is available in C++, but not in many other OO languages.
- They are an alternative to *composition* and *delegation*.
- *Composition* was discussed in section 8.1.1.
- *Delegation* occurs when an object hands over responsibility for an operation to another object. For example, in Program-9.4 on lines 30-35, the `Chicken` printing function uses delegation twice: on line 32, it delegates responsibility for printing the `Animal` part of the chicken to the `Animal` class's printing function; then on line 34, it delegates to the `Egg` class the responsibility for printing the chicken's egg information.

9.4.1.4. **Example:**

- With *public inheritance*, private members in the base class are invisible in the derived class, and public and protected members keep the same access, as we saw in Figure-9.1a.
- Figure-9.4 shows the access specifiers of inherited members in non-public inheritance in C++.
- Figure-9.4a shows how access specifiers are inherited with *private inheritance*. We see that private members in the base class become invisible in the derived class, but public and protected members become *private* inside the derived class.
- Figure-9.4b shows how access specifiers are inherited with *protected inheritance*. Private members in the base class become invisible in the derived class, but public and protected members become *protected* inside the derived class.
- Non-public inheritance basically places a cap on the access to inherited members. With private inheritance, the maximum access specifier of inherited members is private, so other classes in the program cannot use them at all. With protected inheritance, the maximum access specifier of inherited members is protected, so only derived classes have access.



(a) Private inheritance



(b) Protected inheritance

Figure-9.4: Non-public inheritance

9.4.2. Coding example: Stack example using public inheritance

```
class Stack1 : public vector<int>
{
  public:
    void push(int num) { push_back(num); } //calls vector::push_back()
};

int main()
{
  Stack1 stack;

  for (int i=0; i<10; i++) {
    stack.push((i+1)*2);
  }

  stack[1] = 99;
  for (int i=0; i<10; i++) {
    cout << stack[i] << " ";
  }
  return 0;
}
```

Program-9.5: Stack example using public inheritance

Program purpose:

- Program-9.5 demonstrates the use of *public inheritance* as an alternative to composition and delegation, resulting in serious encapsulation issues.

NOTE: This example uses the `vector` class from the standard template library (STL), which has not yet been discussed in this textbook. It's sufficient to note that a `vector` is a collection class of same-type elements, and it provides many member functions and overloaded operators. In particular, the `push_back()` member function adds a new element to the back of the `vector`, and the subscript operator (`[]`) allows direct access to an element at a given index, similar to a primitive array.

Lines 1-5:

- These lines show the `Stack1` class definition.
- A `Stack1` object is a collection that stores integers as elements in a stack-like data structure that should behave in a strict last-in-first-out (LIFO) manner.
- Line 1 indicates that the `Stack1` class uses public inheritance to derive from the STL `vector` class that stores integers.
- The goal of using inheritance in this example is to automatically get access to a `vector` object in which to store the stack elements, and to get very efficient access to the `vector` class's member functions and overloaded operators.
- Line 4 shows the implementation of the `Stack1` class's `push()` member function, which calls the `vector` class's `push_back()` member function. Since all member functions and operators are declared with public access specifiers in the `vector` base class, the `Stack1` derived class has direct access to these members.

Lines 7-20:

- These lines show the implementation of the `main()` function.
- Line 9 declares a `Stack1` object.
- Lines 11-13 use the `Stack1` class's `push()` member function implemented on line 4 to add even numbers to the stack.
- Lines 15-18 demonstrate how the use of public inheritance in this way has unintended consequences.
- Line 15 uses the subscript operator (`[]`) on the `Stack1` object to arbitrarily set the second element to the value `99`. The syntax allows this because the `vector` class provides a public implementation of the subscript operator, which is inherited by the `Stack1` class. With public inheritance, the inherited member functions that are public in the base class remain public in the derived class. So the `main()` function is able to use them with its `Stack1` object.
- The subscript operation on line 15 violates every rule of encapsulation, because stacks are meant to be strictly LIFO structures. Directly accessing and modifying a specific element should **never** be possible. Yet our `Stack1` implementation allows it.
- Lines 16-18 show the same issue, where line 17 uses the subscript operator to print out each element. Again, this should never be allowed in a stack-like data structure.
- So public inheritance works, but it breaks encapsulation.

9.4.3. Coding example: Stack example using composition

```
 1 class Stack2
 2 {
 3   public:
 4     void push(int num) { myStuff.push_back(num); }
 5   private:
 6     vector<int> myStuff;
 7 };

 9 int main()
10 {
11   Stack2  stack;
12   for (int i=0; i<10; i++) {
13     stack.push((i+1)*2);
14   }

16 /*
17   stack.myStuff[1] = 99;
18   for (int i=0; i<10; i++) {
19     cout << stack.myStuff[i] << " ";
20   }
21 */
22   return 0;
23 }
```

Program-9.6: Stack example using composition

Program purpose:

- Program-9.6 demonstrates the same example as Program-9.5, but it uses composition and delegation instead of inheritance.

Lines 1-7:

- These lines show the `Stack2` class definition, and line 1 shows that the `Stack2` class does not use inheritance.
- Line 6 declares a private data member called `myStuff`, which is an STL `vector` object, as the underlying data structure for the class. In this composition relationship, a `Stack2` object behaves as a container, and a `vector` object is the containee that stores integers.
- Line 4 shows the implementation of the `Stack2` class's `push()` member function, which uses delegation to call the `push_back()` member function on the containee `vector` object.

Lines 9-23:

- These lines show the implementation of the `main()` function.
- Line 11 declares a `Stack2` object, and lines 12-14 use the `Stack2` class's `push()` member function implemented on line 4 to add even numbers to the stack.
- Lines 17-20 have been commented out because they do not compile. This is a very good thing, because it means that the `main()` function cannot access the `vector` object's subscript operator. That's because the `vector` object is a private data member inside the `Stack2` class.
- This implementation is much better encapsulation than the `Stack1` class in Program-9.5, but it does have a minor efficiency issue. In the implementation of `Stack2`'s `push()` member function, there is an additional step required to fetch the `myStuff` vector from memory before calling its `push_back()` member function.
- Composition and delegation work fine, but their usage results in a minor performance issue.

9.4.4. Coding example: Stack example using private inheritance

```
 1 class Stack3 : private vector<int>
 2 {
 3   public:
 4     void push(int num) { push_back(num); }
 5 };

 7 int main()
 8 {
 9   Stack3  stack;
10   for (int i=0; i<10; i++) {
11     stack.push((i+1)*2);
12   }

14 /*
15   stack[1] = 99;
16   for (int i=0; i<10; i++) {
17     cout << stack[i] << " ";
18   }
19 */
20   return 0;
21 }
```

Program-9.7: Stack example using private inheritance

Program purpose:

- Program-9.7 demonstrates the same example as Program-9.5 and Program-9.6, but using *private inheritance*.
- This technique addresses the encapsulation issues with public inheritance and maintains better efficiency over the use of composition.

Lines 1-5:

- These lines show the `Stack3` class definition.
- Line 1 shows that the `Stack3` class inherits *privately* from the STL `vector` class that stores integers.
- Line 4 shows the implementation of the `Stack3` class's `push()` member function, which calls the `vector` class's `push_back()` member function. Since all member functions and operators are declared with public access specifiers in the `vector` base class, the `Stack3` derived class has direct access to these members.

Lines 7-21:

- These lines show the implementation of the `main()` function.
- Line 9 declares a `Stack3` object, and lines 10-12 use the `Stack3` class's `push()` member function implemented on line 4 to add even numbers to the stack.
- Lines 15-18 have been commented out because they do not compile. This is a very good thing, because the `main()` function cannot access the `vector` class's subscript operator.
- Lines 15 and 17 attempt to use the subscript operator on the `Stack3` object, but this syntax *is not allowed*. With private inheritance, the `vector` class's public members become private to the `Stack3` derived class when they are inherited. So the `main()` function is unable to use the private subscript operator with its `Stack3` object.
- This implementation fixes the encapsulation issue we found in Program-9.5, and it avoids the minor efficiency issue with the composition relationship in Program-9.6.

9.5. Multiple inheritance

Multiple inheritance is a common OO design concept that can be challenging to support in any OO language. In C++, this feature is available because of the scope resolution operator.

In this section, we discuss the concepts of multiple inheritance, the challenges that can develop because of it, and their solution.

9.5.1. Concepts

9.5.1.1. **What is *multiple inheritance*:**

- *Multiple inheritance* occurs when a class inherits from more than one superclass. In other words, a class is derived from two or more base classes.
- This technique is *not* supported in all OO programming languages.
- The feature that enables multiple inheritance in C++ is the scope resolution operator. It allows for the resolution of ambiguity when a derived class has multiple base classes that contain class members named with the same identifier.

9.5.1.2. **Issues with multiple inheritance:**

- Problems arise if a derived class has multiple base classes that share a common base class, so a child class has parent classes that share a common parent.
- If a derived class has multiple base classes *that have no common parent*, it is said to have *distinct base classes*. This design causes no issues.
- If a derived class has multiple base classes *that do have a common parent*, it results in a *diamond hierarchy* design, which can be problematic.

9.5.1.3. **What is a *diamond hierarchy*:**

- A *diamond hierarchy* is formed when a derived class has multiple base classes that have a common base class.
- This creates an inheritance hierarchy that is shaped like a diamond (imagine a square rotated by 45°), as we see in Figure-9.5.
- In Figure-9.5, the `Faun` class has two base classes, `Human` and `Goat`, that share a common base class `Animal`.

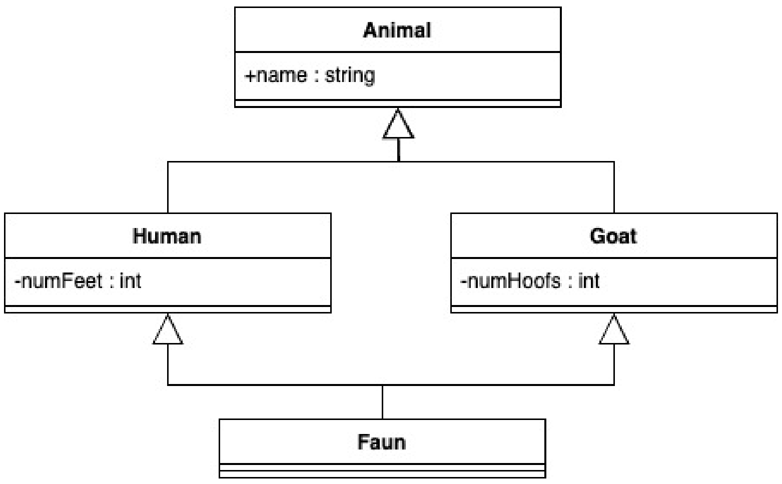


Figure-9.5: UML class diagram for Program-9.8 and Program-9.9

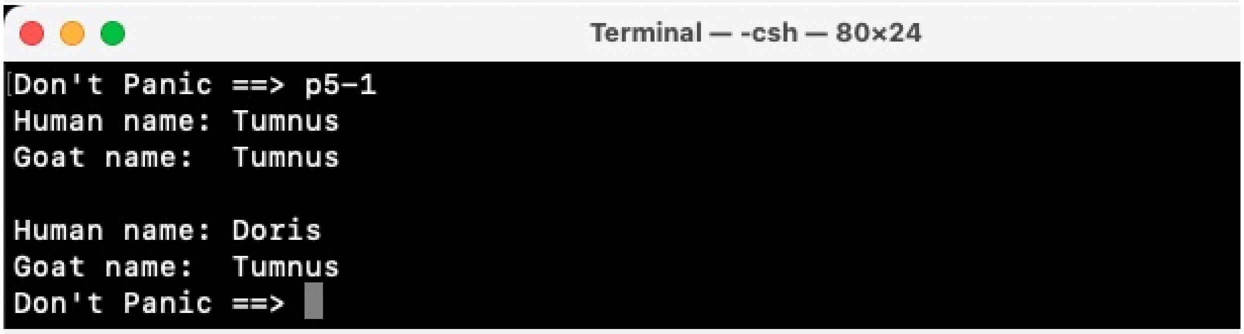
9.5.1.4. Issues with a diamond hierarchy:

- In Figure-9.5, the `Animal` class has a `name` data member that is inherited by each of its derived classes `Human` and `Goat`. Since `Faun` derives from both `Human` and `Goat`, by default it inherits *two instances* of the `name` data member: one from its `Human` base class, and another from its `Goat` base class.
- In some cases, this is not a problem. If it makes sense in an OO design for a derived class in a diamond hierarchy to have multiple instances of the same class member, then the solution is to accept the situation as *multiple inclusion base class*, as we demonstrate in Program-9.8.
- In other cases, it can be a problem for a derived class to inherit multiple instances of the same class member. In those situations, the solution is to use *virtual inheritance*, as we see in Program-9.9.
- The virtual inheritance solution ensures that the derived class, for example `Faun`, inherits *only one instance* of the `Animal` class members. This solution treats the `Faun` class as if it was directly derived from `Animal`, in addition to `Human` and `Goat`.

9.5.2. Coding example: Diamond hierarchy with multiple inclusion base class

```
class Animal
{
  public:
    Animal(string n) : name(n) { }
    string name;
};

class Human : public Animal
{
  public:
    Human(int f, string n) : numFeet(f), Animal(n) { }
  private:
    int numFeet;
};

class Goat : public Animal
{
  public:
    Goat(int h, string n) : numHoofs(h), Animal(n) { }
  private:
    int numHoofs;
};

class Faun : public Human, public Goat
{
  public:
    Faun(int f, int h, string n) : Human(f,n), Goat(h,n) { }
};

```

```
30 int main()
31 {
32   Faun mrTumnus(0,2,"Tumnus");

34 //  cout<<"name: "<< mrTumnus.name << endl;
35   cout<<"Human name: "<< mrTumnus.Human::name << endl;
36   cout<<"Goat name:  "<< mrTumnus.Goat::name << endl;

38   mrTumnus.Human::name = "Doris";
39   cout<<endl<<"Human name: "<< mrTumnus.Human::name << endl;
40   cout<<"Goat name:  "<< mrTumnus.Goat::name << endl;

42   return 0;
43 }
```



```
Don't Panic ==> p5-1
Human name: Tumnus
Goat name:  Tumnus

Human name: Doris
Goat name:  Tumnus
Don't Panic ==>
```

Program-9.8: Diamond hierarchy with multiple inclusion base class

Program purpose:

- Program-9.8 demonstrates the diamond hierarchy problem in multiple inheritance, with a multiple inclusion base class, based on the design shown in Figure-9.5.
- With the multiple inclusion base class approach, the presence of multiple instances of the same inherited class members, in this case the `Animal` class's `name` data member, is accepted as a correct design.

Lines 1-6:

- These lines show the `Animal` class definition at the top of the inheritance hierarchy.
- In addition to a constructor, the class contains one data member for the animal's name. The data member is declared with public access in the examples of this section only.

Lines 8-14:

- These lines show the `Human` class definition, which is derived from `Animal`.
- Line 13 declares one data member to indicate the number of feet.
- Line 11 shows that the constructor initializes the data member and calls the `Animal` base class constructor using the parameters provided.

Lines 16-22:

- These lines show the `Goat` class definition, which is also derived from `Animal`.
- Line 21 declares one data member to indicate the number of hoofs.
- Line 19 shows that the constructor initializes the data member and calls the `Animal` base class constructor using the parameters provided.

Lines 24-28:

- These lines show the `Faun` class definition, which is derived from both `Human` and `Goat`.
- The class contains no data members.
- Line 27 indicates that the constructor calls both the `Human` and the `Goat` base class constructors using the parameters provided.

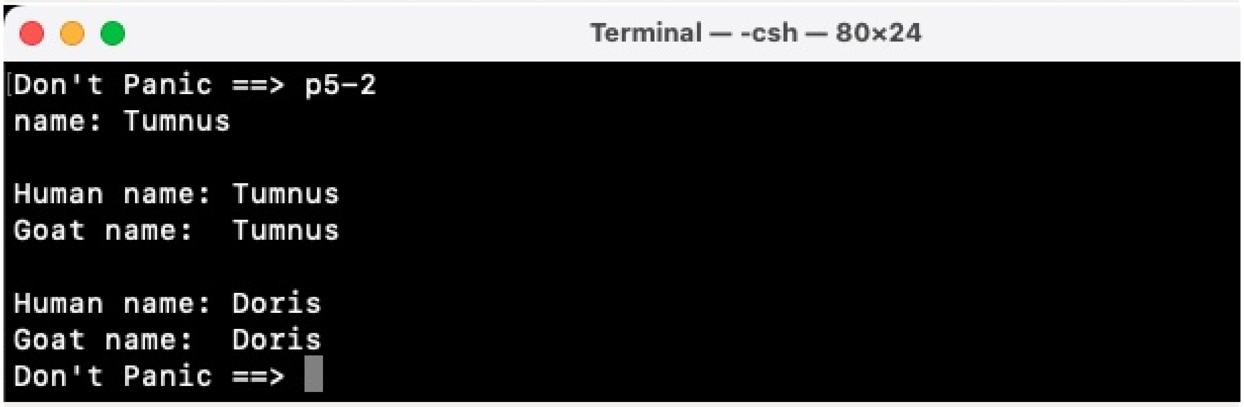
Lines 30-43:

- These lines show the implementation of the `main()` function.
- Line 32 allocates and initializes a `Faun` object in a variable called `mrTumnus`.
- Line 34 is commented out because it does not compile. It attempts to print the `Faun` object's name, but the object contains *two* `name` data members: one inherited from the `Human` class, and the other from `Goat`. Because the compiler does not know which name should be printed on line 34, the compilation fails.
- In contrast, lines 35-36 do work correctly, because we use the scope resolution operator to specify which `Faun` name should be printed. Line 35 explicitly indicates that the object's `Human` name is printed, and line 36 prints the `Goat` name.
- Line 38 changes the `Faun` object's `Human` name to `"Doris"`, which leaves its `Goat` name unchanged.
- We see from the program output of lines 39-40 that the `Faun` object does indeed have two separate data members for its name, so they can both have separate values.

9.5.3. Coding example: Diamond hierarchy with virtual inheritance

```
class Animal
{
  public:
    Animal(string n) : name(n) { }
    string name;
};

class Human : virtual public Animal
{
  public:
    Human(int f, string n) : numFeet(f), Animal(n) { }
  private:
    int numFeet;
};

class Goat : virtual public Animal
{
  public:
    Goat(int h, string n) : numHoofs(h), Animal(n) { }
  private:
    int numHoofs;
};

class Faun : public Human, public Goat
{
  public:
    Faun(int f, int h, string n) : Animal(n),
                                   Human(f,n), Goat(h,n) { }
};
```

```
30 int main()
31 {
32   Faun mrTumnus(0,2,"Tumnus");

34   cout<<"name: "<< mrTumnus.name << endl;
35   cout<<"Human name: "<< mrTumnus.Human::name << endl;
36   cout<<"Goat name:  "<< mrTumnus.Goat::name << endl;

38   mrTumnus.Human::name = "Doris";
39   cout<<endl<<"Human name: "<< mrTumnus.Human::name << endl;
40   cout<<"Goat name:  "<< mrTumnus.Goat::name << endl;

42   return 0;
43 }
```



```
Don't Panic ==> p5-2
name: Tumnus

Human name: Tumnus
Goat name:  Tumnus

Human name: Doris
Goat name:  Doris
Don't Panic ==> 
```

Program-9.9: Diamond hierarchy with virtual inheritance

Program purpose:

- Program-9.9 demonstrates the diamond hierarchy problem in multiple inheritance, with a virtual inheritance solution, based on the design shown in Figure-9.5.
- The `main()` function is identical to Program-9.8, with line 34 uncommented.
- The virtual inheritance approach eliminates the inheritance of multiple instances of the same class members and reduces them to a single instance of each inherited member.

Lines 1-6:

- These lines show the `Animal` class definition, unchanged from Program-9.8.

Lines 8-14:

- These lines show the `Human` class definition, as derived from `Animal` using *virtual* inheritance, as shown on line 8.
- The data member and constructor are unchanged from Program-9.8.

Lines 16-22:

- These lines show the `Goat` class definition, as derived from `Animal` using *virtual* inheritance, as shown on line 16.
- The data member and constructor are unchanged from Program-9.8.

Lines 24-29:

- These lines show the `Faun` class definition, which is derived from both `Human` and `Goat`.
- Line 27 indicates that, in addition to calls to both the `Human` and `Goat` base class constructors on line 28, the `Faun` constructor now needs to *also* call the `Animal` constructor directly.
- This is because, with virtual inheritance, the `Animal` class is now a *virtual base class* of `Faun`.

Lines 30-43:

- These lines show the implementation of the `main()` function.
- The function is identical to Program-9.8, except for the uncommenting of line 34.
- We see from the program output that the line 34 is now able to print the `Faun` object's name *without ambiguity*. That's because the object now only has a single instance of the inherited `name` data member, instead of two.
- With virtual inheritance, when line 38 changes the `Faun` object's `Human` name to `"Doris"`, it changes the object's only name to that value.
- When lines 39-40 print out both the object's `Human` and `Goat` names, the same unique name is printed out, as we see from the program output.

Chapter 10

Polymorphism

Runtime polymorphism is one of the most powerful OO design techniques available to programmers. It is a crucial tool for facilitating independence between classes, which in turn makes them more resilient to change. Many design patterns, which are also an important OO design tool, use polymorphic techniques. Design patterns are discussed in chapter 11.

In this chapter, we introduce the basics of runtime polymorphism and its implementation in C++, including the foundational principles of dynamic binding.

10.1. Principles

We discuss the purpose and terminology of runtime polymorphism, as well as the role it plays in the development of correctly engineered software.

10.1.1. Basics

10.1.1.1. **What is *polymorphism*:**

- In the English language, the root *poly* means "many", and *morph* means "shape".
- In OO design, *polymorphism* allows a program to:
  - define multiple functions to do the same task in different ways, and
  - choose between the different functions that perform this task
- With polymorphism, a base class can define a class interface, while the implementation of that interface may be provided by multiple derived classes.

10.1.1.2. **Two types of polymorphism:**

- *Compile-time polymorphism*:
  - this is also known as *static polymorphism*
  - in C++, it's the same as *function overloading*, which is a topic in chapter 12
- *Runtime polymorphism*:
  - this is also known as *dynamic polymorphism*, which is the focus of this chapter
  - everywhere that the term *polymorphism* is used henceforth in this textbook can be assumed to mean *runtime polymorphism*

10.1.1.3. **Polymorphism is *not* the same concept as inheritance:**

- Inheritance is a *structural* relationship between two classes and doesn't change at runtime.
- Runtime polymorphism changes the *behaviour* of the program. It selects which member function to execute at runtime, based on the object type.
- Polymorphism *uses* inheritance. It's an OO design technique that builds upon inheritance, but it's neither the equivalent nor a substitute for it.

10.1.2. Software engineering considerations

10.1.2.1. **Some important terminology for this chapter:**

- A software system is a program or application that is very large and contains many interacting components. At this level of complexity, we no longer call it simply a program.
- A *client* is the person or company that is paying us or our company to design and develop software for them. Clients provide requirements, and they receive a tested and installed software system in return.
- The *end-users* are the persons who will be using our system after it's delivered and installed. They typically work for or with the client.
- From a programmer's point of view, a ***client class*** is the class or group of classes that is *using* our code:
  - we normally write code for other programmers as part of a large system
  - our code is used by other programmers and their classes in order to achieve a result
  - those other programmers are our class users, and their classes are our *client classes*
  - a client class has nothing to do with a client; the former is a part of an OO design, and the latter is a person or company paying for software delivery

10.1.2.2. ***Change* is the arch-enemy of timely software development:**

- Clients often request changes because they want more features than originally planned.
- Software designers sometimes misunderstand the client requirements and must change the design after implementation has begun.
- Changes may be required due to external factors, for example a lack of available personnel or issues with third-party vendors or their products.
- All changes have an impact on the timeline for software delivery and may result in software that is late or incomplete.
- *Planning for changes* becomes a crucial aspect of correct software design.
- In a *modular design*, software is organized in clearly delineated independent components that communicate over well-designed minimalist interfaces. This is essential for isolating software components from each other so that they can change and evolve independently.
- Ideally, if a software system has a modular design, any changes required will be isolated to a minimal number of components, with little to no impact on the rest of the system.
- There are several important OO design techniques for creating a modular design, including the use design patterns, as we see in chapter 11. Runtime polymorphism is a crucial foundation for many of these techniques.

10.1.2.3. **Resilience to change:**

- With runtime polymorphism, we can isolate client classes from our class implementations and isolate our classes from each other.
- When classes are separated and isolated from each other as much as possible, changes can be made to one without affecting the others. Our system becomes more resilient to change.
- With polymorphism, client classes use a *base class interface* to manipulate derived class objects. Client classes *do not know* what kind of derived object they are actually using. They only know that the object is derived from a specific base class that provides a defined class interface. This technique requires that our implementation uses base class pointers to manipulate derived class objects, which is illustrated in the next coding example.

10.1.3. How runtime polymorphism works

10.1.3.1. **At the base class:**

- The base class provides a class interface, as a set of public member functions.
- Polymorphic function prototypes are defined in the class interface, but *no implementations* are provided in the base class. Only the derived classes provide these implementations.
- The base class is often *abstract*, as it is normally too generalized to be instantiated. Abstract classes are discussed later in this chapter.

10.1.3.2. **At the derived class:**

- Each derived class overrides the polymorphic member functions that comprise the base class interface.
- The derived class provides an implementation for each overridden polymorphic member function, with its own unique, specialized behaviour that is specific to the derived class.

10.1.3.3. **At the client class:**

- The client class interacts with derived class objects as if they are base class objects, using the provided base class interface:
  - the client class uses base class pointers to manipulate derived class objects
  - it calls the generalized member function at the base class, even if it has no implementation in that class
  - the function that executes is one of the overridden implementations in the derived classes
  - the client class **never** needs to know the kind of derived object that it is manipulating
  - if the code has to check the type of object, then polymorphism is *not* being used correctly
- When the client class calls a polymorphic member function, *the actual function that executes* is chosen at runtime:
  - the choice of which function to execute is done implicitly behind the scenes, based on the kind of derived object on which the member function is called
  - this selection of the correct function is called *dynamic binding*, which is discussed shortly
- It's crucial to note that polymorphism occurs **only** when the executed function is selected implicitly, at runtime, with dynamic binding. Polymorphism does *not* occur if any class makes an explicit selection, for example by using a `switch` or `if`-statement to check the type of derived object.

10.1.3.4. **Advantages of using polymorphism:**

- Polymorphism enables the generalized use of a class hierarchy.
- It isolates the client class from any changes required in the derived class implementations.
- Polymorphism promotes *independence* between classes:
  - independence of the client class from the derived classes
  - independence of the derived classes from each other
- Adding a new derived class should have *no impact* on the existing classes, neither the client class nor the existing base and derived classes.
- Independence between classes is crucial to good OO design and helps to minimize the impact of changes.

10.1.3.5. **Two unbreakable rules of polymorphism:**

- Polymorphism only works with classes related to each other by inheritance. Without it, we cannot implement polymorphism.
- Polymorphism only works with *pointers* to objects, and not the objects themselves.

10.1.3.6. **Underlying idea behind polymorphism implementation in C++:**

- Given an inheritance hierarchy, a derived class object can always be treated as if it's an instance of the base class, even if the base class is abstract.
- This works because the derived class object "is-a" kind of base class object, so we can use base class pointers with derived class object pointees.
- For example:
  - in a typical application, the client class can have a collection of what it thinks are base class objects, which it stores using base class pointers
  - but in reality, the collection contains an assortment of derived class objects
  - the client class can then invoke polymorphic functions on each object in the collection, without knowing the object type, as long as the functions are defined (but not implemented) in the base class
  - dynamic binding ensures that the correct function is called at runtime, based on the type of derived object

10.1.4. Coding example: Using pointers with base and derived class objects

The program in this example implements the UML class diagram shown in Figure-10.1.

The `Animal` class:

- Figure-10.1 shows that the `Animal` class has one protected data member that represents the animal's name.
- It also has a public member function called `sing()`, as well as a default constructor.

The `Bird` class:

- As we see in Figure-10.1, the `Bird` class is derived from the `Animal` class.
- It has no data members or member functions, except for a default constructor.

The `Chicken` class:

- Figure-10.1 shows that the `Chicken` class is derived from the `Bird` class.
- It has no data members or member functions, except for a default constructor.

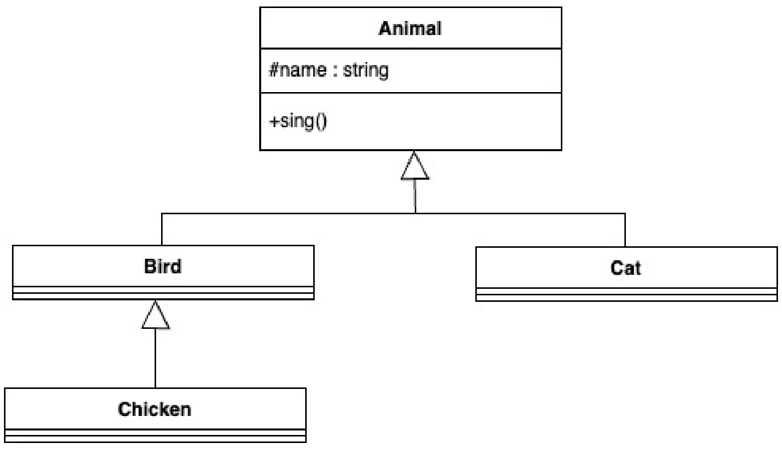


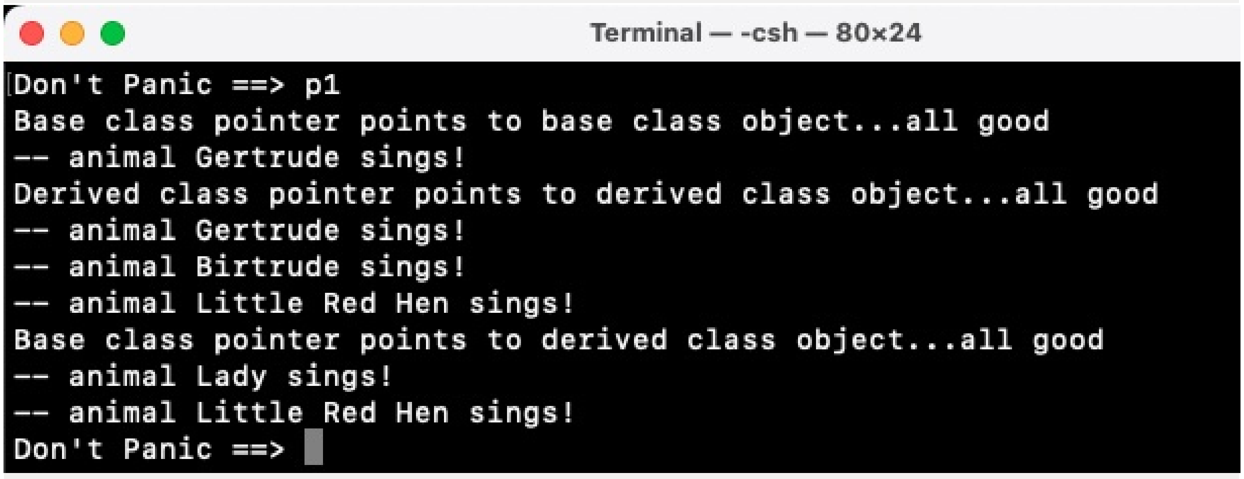
Figure-10.1: UML class diagram for Program-10.1

The `Cat` class:

- Figure-10.1 also shows that the `Cat` class is derived from the `Animal` class.
- It has no data members or member functions, except for a default constructor.

```
class Animal
{
  public:
    Animal(string n="") : name(n) { }
    void sing() { cout<< "-- animal "<<name<<" sings!"<<endl; }
  protected:
    string name;
};

class Bird : public Animal
{
  public:
    Bird(string n="") : Animal(n) { }
};

class Chicken : public Bird
{
  public:
    Chicken(string n="") : Bird(n) { }
};

class Cat : public Animal
{
  public:
    Cat(string n="") : Animal(n) { }
};

```

```
int main()
{
  Animal   gertrude("Gertrude");
  Bird     birtrude("Birtrude");
  Chicken  redHen("Little Red Hen");
  Cat      lady("Lady");

  Animal*  ap;
  Bird*    bp;
  Chicken* cp;

  cout << "Base class pointer points to base class object...";
  ap = &gertrude;
  cout << "all good" << endl;
  ap->sing();

  cout << "Derived class pointer points to derived class object...";
  bp = &birtrude;
  cp = &redHen;
  cout << "all good" << endl;

  ap->sing();
  bp->sing();
  cp->sing();

  cout << "Base class pointer points to derived class object...";
  ap = &lady;
  bp = &redHen;
  cout << "all good" << endl;

  ap->sing();
  bp->sing();

//  cout << "Derived class pointer points to base class object...";
//  cp = &gertrude;
//  cp->sing();
//  cout << "all good" << endl;

  return 0;
}
```



```
Don't Panic ==> p1
Base class pointer points to base class object...all good
-- animal Gertrude sings!
Derived class pointer points to derived class object...all good
-- animal Gertrude sings!
-- animal Birtrude sings!
-- animal Little Red Hen sings!
Base class pointer points to derived class object...all good
-- animal Lady sings!
-- animal Little Red Hen sings!
Don't Panic ==> 
```

Program-10.1: Using pointers with base and derived class objects

Program purpose:

- Program-10.1 demonstrates how pointers of specific data types may or may not be used with objects of different types, based on the design shown in Figure-10.1. Understanding which types of pointers may point to which types of objects is crucial in understanding the polymorphism examples in this chapter.
- This program does **not** use polymorphism. But it illustrates an important ingredient in its implementation, where a base class pointer can have a derived class pointee.

Lines 1-26:

- Lines 1-8 show the `Animal` class definition.
- Line 5 contains the implementation of a `sing()` member function in the `Animal` class that prints out a message to the end-user.
- Lines 10-14 show the `Bird` class definition that is derived from `Animal`.
- Lines 16-20 show the `Chicken` class definition that is derived from `Bird`.
- Lines 22-26 show the `Cat` class definition that is derived from `Animal`.
- The `Bird`, `Chicken`, and `Cat` classes all inherit the `sing()` member function implementation from `Animal`, and none of them overrides it.
- When the `sing()` member function is called on a derived class object, the implementation defined in the `Animal` base class executes.

Lines 30-37:

- Lines 30-33 declare four objects locally within the `main()` function, with one object for each type of class defined on lines 1-26.
- Lines 35-37 declare three pointers to different types of objects.
- The program demonstrates which of the three pointers can correctly point to which types of objects.

Lines 40 and 42:

- These lines show how a base class `Animal` pointer `ap` can point to a base class `Animal` object `gertrude`.
- Both pointer and pointee are of the same class, so these lines work correctly.

Lines 45-46 and 50-51:

- These lines show how derived class `Bird` and `Chicken` pointers, `bp` and `cp` respectively, can point to the derived class `Bird` and `Chicken` objects, `birtrude` and `redHen` respectively.
- Both pointers and pointees are of the same class as each other, so these lines work correctly.

Lines 54-55 and 58-59:

- These lines show how base class `Animal` and `Bird` pointers, `ap` and `bp` respectively, can have derived class objects as pointees.
- Line 54 sets an `Animal` pointer `ap` with a `Cat` object as its pointee. This works because the `Cat` class is derived from `Animal`, so every cat is a kind of animal. It's perfectly fine to use an `Animal` pointer to perform operations on the `Animal` portion of a `Cat` object.
- Line 55 sets a `Bird` pointer `bp` with a `Chicken` object as its pointee. This is also fine because `Chicken` is derived from the base class `Bird`.
- In general, a base class pointer can have any instance of its derived classes as pointee, as long as the pointer's class is a base class of the pointee object.

Lines 62-63:

- These lines are commented out because they do not compile.
- Line 62 tries to set a `Chicken` pointer `cp` with an `Animal` object `gertrude` as its pointee.
- These lines fail to compile because the `Animal` class is **not** derived from `Chicken`, and animals are not a kind of chicken.

10.1.5. Pointers and class hierarchy

		Type of object	
		base class	derived class
Type of pointer	base class	Works	Works (polymorphism)
	derived class	Doesn't compile	Works

Figure-10.2: Pointer and object data type combinations

10.1.5.1. **Base class pointers:**

- The first row of Figure-10.2 illustrates what happens when we use a base class pointer with different types of objects.
- A base class pointer with a base class object as its pointee works fine.
- A base class pointer with a derived class object as its pointee also works fine. It's an important technique that enables the implementation of runtime polymorphism.

10.1.5.2. **Derived class pointers:**

- The second row of Figure-10.2 shows what happens when we use a derived class pointer with different types of objects.
- A derived class pointer with a base class object as its pointee causes a compilation error, as we discussed in Program-10.1.
- A derived class pointer with a derived class object as its pointee works fine.

10.2. Dynamic binding

We discuss the dynamic binding of functions as a mandatory requirement to enable polymorphic behaviour in our programs.

10.2.1. Concepts

10.2.1.1. **What is a *handle* to an object:**

- A ***handle*** is the identifier that we use in a program to refer to a specific object. It can be the variable name of an object, or a pointer or reference to the object.
- For example in Program-10.1, only one instance of the `Chicken` class is allocated, but after line 55, that same object has three handles to it: the `redHen` variable, the `cp` pointer, and the `bp` pointer. All three identifiers are handles to the same object.
- When polymorphism is used, the choice of handle to an object (variable name or pointer) is a critical factor in deciding which member function is executed.

10.2.1.2. **What is *function binding*:**

- *Function binding* is the selection of the correct function to execute when a function is called.
- In most cases, there is no ambiguity. For example, if a member function is called on an object that is not part of an inheritance hierarchy, or if there are no overridden functions in the hierarchy, the choice is clear.
- However, when an overridden function is called, there are two important factors to consider:
  - the type of object handle that is used to call the member function, and
  - whether the called member function is *virtual* or *non-virtual*
- Virtual and non-virtual functions are discussed later in this chapter.

10.2.1.3. **Static function binding:**

- With *static binding*, the selection of which function to execute is made at compile time.
- This happens when the compiler has sufficient, unambiguous knowledge regarding which function must execute.
- Static binding *always* occurs if a member function is called on an object variable handle, instead of an object pointer handle.
- It can occur when a member function is called using an object pointer handle, but only if the member function is non-virtual.

10.2.1.4. **Dynamic function binding:**

- With ***dynamic binding***, the selection of which function to execute is made at runtime.
- This happens when the compiler only has ambiguous knowledge regarding which function must execute. In that case, the decision is delayed until runtime.
- Dynamic binding *never* occurs if a member function is called on an object variable handle.
- It can occur when a member function is called using an object pointer handle, but only if the member function is virtual.
- Dynamic binding is a **mandatory** condition for polymorphism to work. If a polymorphic function is called, but static binding is used, then polymorphic behaviour does **not** occur.

10.2.1.5. **Characteristics of function binding:**

- With dynamic binding, the function that is chosen for execution is selected based on the *type of object*, and **not** the type of handle.
- With static binding, the selection is always based on the type of handle.
- In C++, dynamic binding is enabled with *virtual functions*.

10.2.1.6. **Conditions for polymorphic behaviour:**

- For our programs to invoke polymorphic behaviour, we **must** ensure that dynamic binding occurs when a polymorphic member function is called.
- There are two conditions that *must* be present to trigger the use of dynamic binding:
  - we must call the polymorphic member function using an object pointer handle, and
  - the member function must be declared as *virtual*, as we see later in Program-10.3

10.2.2. Coding example: Using non-virtual functions

The program in this example implements the UML class diagram shown in Figure-10.3.

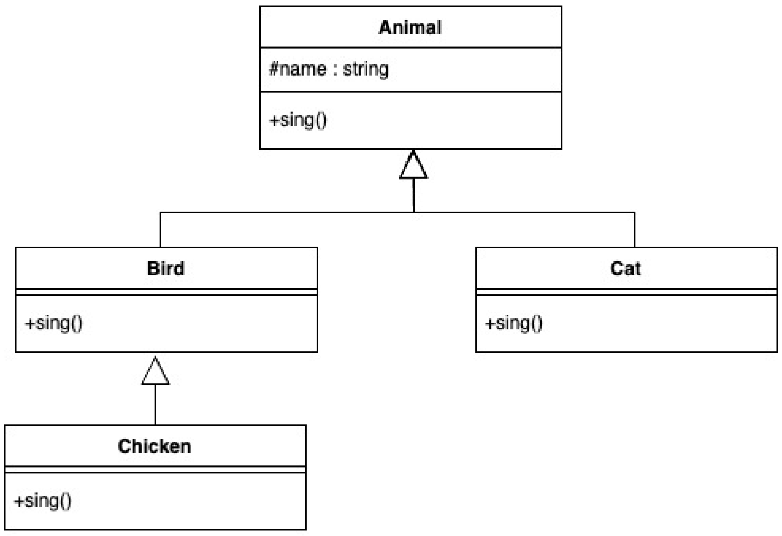


Figure-10.3: UML class diagram for Program-10.2 and Program-10.3

The `Animal` class:

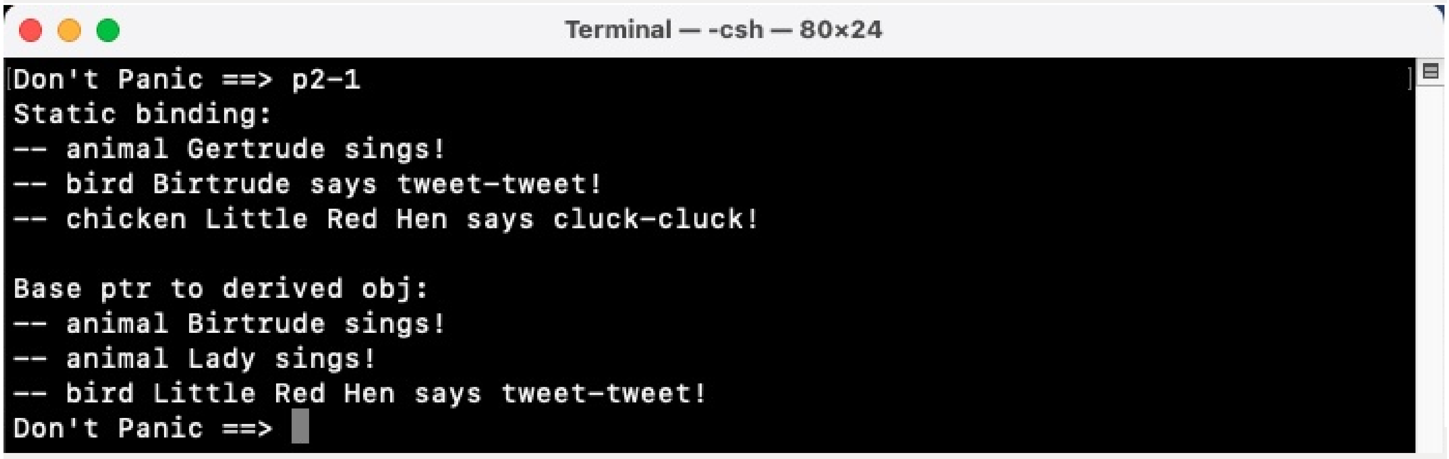
- The `Animal` class shown in Figure-10.3 is the same as in Figure-10.1.

The `Bird`, `Chicken`, and `Cat` classes:

- Figure-10.3 shows that the `Bird` and `Cat` classes are derived from the `Animal` class, and `Chicken` is derived from `Bird`.
- All three classes override the `sing()` member function inherited from their base class, and each one provides its own implementation.

```
class Animal
{
  public:
    Animal(string n="") : name(n) { }
    void sing() { cout<< "-- animal "<<name<<" sings!"<<endl; }
  protected:
    string name;
};

class Bird : public Animal
{
  public:
    Bird(string n="") : Animal(n) { }
    void sing() { cout<< "-- bird "<<name<<" says tweet-tweet!"<<endl; }
};

class Chicken : public Bird
{
  public:
    Chicken(string n="") : Bird(n) { }
    void sing() { cout<< "-- chicken "<<name<<" says cluck-cluck!"<<endl; }
};

```

```
class Cat : public Animal
{
  public:
    Cat(string n="") : Animal(n) { }
    void sing() { cout<< "-- cat "<<name<<" says meow!"<<endl; }
};

int main()
{
  Animal   gertrude("Gertrude");
  Bird     birtrude("Birtrude");
  Chicken  redHen("Little Red Hen");
  Cat      lady("Lady");

  Animal*  ap1;
  Animal*  ap2;
  Bird*    bp;
  Chicken* cp;

  cout<<"Static binding:"<<endl;
  gertrude.sing();
  birtrude.sing();
  redHen.sing();

  cout<<endl<<"Base ptr to derived obj:"<<endl;
  ap1 = &birtrude;
  ap2 = &lady;
  bp  = &redHen;
  ap1->sing();
  ap2->sing();
  bp->sing();

  return 0;
}
```



```
Don't Panic ==> p2-1
Static binding:
-- animal Gertrude sings!
-- bird Birtrude says tweet-tweet!
-- chicken Little Red Hen says cluck-cluck!

Base ptr to derived obj:
-- animal Birtrude sings!
-- animal Lady sings!
-- bird Little Red Hen says tweet-tweet!
Don't Panic ==> 
```

Program-10.2: Using non-virtual functions

Program purpose:

- Program-10.2 demonstrates the non-polymorphic behaviour that results from static binding and non-virtual functions, based on the design shown in Figure-10.3.
- Each class in the program has its own implementation of the `sing()` member function, which is overridden in every derived class.
- This program does **not** show polymorphic behaviour. It illustrates which `sing()` member function executes when it is *not* defined as virtual and it is called on a derived class object.

Lines 1-29:

- These lines show the class definitions for `Animal`, `Bird`, `Chicken`, and `Cat`.
- The `Animal` class implements the `sing()` member function on line 5.
- The `Bird`, `Chicken`, and `Cat` derived classes all inherit the `sing()` member function from their base class, and they override it with their own unique implementation on lines 14, 21, and 28 respectively.
- We see that the `sing()` member functions in all four classes are *non-virtual* because they do not use the `virtual` keyword.

Lines 33-41:

- Lines 33-36 declare four objects locally within the `main()` function, with one object for each class defined on lines 1-29.
- Lines 38-41 declare four pointers to different types of objects in the class hierarchy.

Lines 44-46:

- These lines show an example of static binding.
- It's known at compile time which `sing()` member function is called on lines 44-46 because the function is called using object variable handles, and not object pointer handles.
- With object variable handles, the compiler uses *the type of handle* to choose which `sing()` member function to call.
- On line 44, the handle used to call the function is the `gertrude` object variable, which is an `Animal` object, as declared on line 33. So the `Animal` class's `sing()` function is called.
- On line 45, the handle used to call the function is the `birtrude` object variable, which is a `Bird` object, as declared on line 34. So the `Bird` class's `sing()` function is called.
- On line 46, the handle used to call the function is the `redHen` object variable, which is a `Chicken` object, as declared on line 35. So the `Chicken` class's `sing()` function is called.
- We see from the first part of the program output which `sing()` functions execute for lines 44-46.

Lines 49-54:

- These lines also show an example of static binding.
- It's known at compile time which `sing()` member function is called on lines 52-54 because the `sing()` member functions are non-virtual.
- We see on lines 49-51 that the three base class pointers are each assigned pointees from a derived class.
- Line 49 shows that `Animal` pointer `ap1` is assigned a `Bird` object pointee.
- On line 50, an `Animal` pointer `ap2` is assigned a `Cat` object pointee.
- Line 51 shows that `Bird` pointer `bp` is assigned a `Chicken` object pointee.
- Because the `sing()` member functions are non-virtual, the compiler uses *the type of handle* to choose which `sing()` member function to call.

- On line 52, the handle used to call the function is the `ap1` pointer, which is an `Animal` pointer, as declared on line 38. So the `Animal` class's `sing()` function is called.
- On line 53, the handle used to call the function is the `ap2` pointer, which is an `Animal` pointer, as declared on line 39. So the `Animal` class's `sing()` function is called.
- On line 54, the handle used to call the function is the `bp` pointer, which is a `Bird` pointer, as declared on line 40. So the `Bird` class's `sing()` function is called.
- We see from the second part of the program output which `sing()` functions execute for lines 52-54.

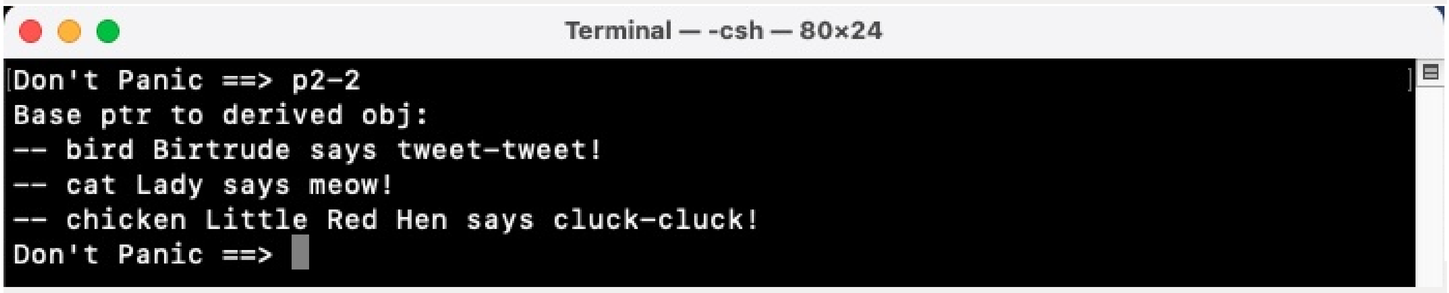
Program output:

- The first part of the program output is intuitive because the `sing()` member functions are called on object variable handles.
- This results in the `Animal` class's `sing()` function called for the `Animal` object, the `Bird` class's function called for the `Bird` object, and the `Chicken` class's function called for the `Chicken` object, which makes sense.
- The second part of the program output is less intuitive and probably not what the programmer intended. We are using base class pointers, but the pointees are all derived class objects. So we would expect the derived class `sing()` functions to be called. But because non-virtual functions always invoke static binding, it's the pointer class's `sing()` functions that are called.

10.2.3. Coding example: Using virtual functions

```
class Animal
{
  public:
    Animal(string n="") : name(n) { }
    virtual void sing() { cout<< "-- animal "<<name<<" sings!"<<endl; }
  protected:
    string name;
};

class Bird : public Animal
{
  public:
    Bird(string n="") : Animal(n) { }
    virtual void sing() { cout<< "-- bird "<<name<<" says tweet-tweet!"<<endl; }
};

class Chicken : public Bird
{
  public:
    Chicken(string n="") : Bird(n) { }
    virtual void sing() { cout<< "-- chicken "<<name<<" says cluck-cluck!"<<endl;}
};

class Cat : public Animal
{
  public:
    Cat(string n="") : Animal(n) { }
    virtual void sing() { cout<< "-- cat "<<name<<" says meow!"<<endl; }
};

```

```
31 int main()
32 {
33   Animal   gertrude("Gertrude");
34   Bird     birtrude("Birtrude");
35   Chicken  redHen("Little Red Hen");
36   Cat      lady("Lady");

38   Animal*  ap1;
39   Animal*  ap2;
40   Bird*    bp;
41   Chicken* cp;

43   cout<<"Base ptr to derived obj:"<<endl;
44   ap1 = &birtrude;
45   ap2 = &lady;
46   bp  = &redHen;
47   ap1->sing();
48   ap2->sing();
49   bp->sing();

51   return 0;
52 }
```



```
Don't Panic ==> p2-2
Base ptr to derived obj:
-- bird Birtrude says tweet-tweet!
-- cat Lady says meow!
-- chicken Little Red Hen says cluck-cluck!
Don't Panic ==>
```

Program-10.3: Using virtual functions

Program purpose:

- Program-10.3 is a variation of Program-10.2, and it also implements the class hierarchy shown in Figure-10.3.
- As in Program-10.2, each class in the program has its own implementation of the `sing()` member function, which is overridden in every derived class.
- The difference in this example is that the `sing()` member functions are defined as *virtual*.
- This program **does** show polymorphic behaviour. It illustrates which `sing()` member function executes when it is defined as virtual and it is called on a derived class object.

Lines 1-29:

- These lines show the class definitions for `Animal`, `Bird`, `Chicken`, and `Cat`.
- The `Animal` class implements the `sing()` member function on line 5.
- The `Bird`, `Chicken`, and `Cat` derived classes all inherit the `sing()` member function from their base class, and they override it with their own unique implementation on lines 14, 21, and 28 respectively.
- We see that the `sing()` member functions in all four classes are *virtual* because they use the `virtual` keyword before the function prototypes on lines 5, 14, 21, and 28.

Lines 33-41:

- Lines 33-36 declare four objects locally within the `main()` function, with one object for each class defined on lines 1-29.
- Lines 38-41 declare four pointers to different types of objects in the class hierarchy.

Lines 44-49:

- These lines show an example of dynamic binding.
- It's **not** known at compile time which implementation of the `sing()` member function is called on lines 47-49. With object pointer handles and virtual functions, the program uses *the type of object* to choose which `sing()` member function to call.
- Lines 44-46 assign derived class pointees to the three base class pointers. Line 44 assigns a `Bird` object pointee to the `Animal` pointer `ap1`; line 45 assigns a `Cat` object pointee to the `Animal` pointer `ap2`; and line 46 assigns a `Chicken` object pointee to the `Bird` pointer `bp`.
- On line 47, the handle used to call the function is the `ap1` pointer, which is an `Animal` pointer, as declared on line 38. However, because the `sing()` functions are virtual, the program looks at the *the type of object*, and not the type of handle. Since the object on which the function is called on line 47 is actually a `Bird` object, the `Bird` class's `sing()` function is called.
- On line 48, the handle used is the `ap2` pointer, which is an `Animal` pointer. But with a virtual function, the program looks at the type of object, and not the type of handle. Since the object on which the function is called on line 48 is a `Cat` object, the `Cat` class's `sing()` function is executed.
- On line 49, the handle used is the `bp` pointer, which is a `Bird` pointer. Since the object on which the function is called on line 49 is a `Chicken` object, the `Chicken` class's `sing()` function is called.
- We see from the program output which `sing()` functions execute for lines 47-49.

10.3. Virtual functions

We discuss the role of virtual functions as the C++ programming language feature used to enable the implementation of polymorphism.

10.3.1. Concepts

10.3.1.1. What is a *virtual function*:

- In C++, a *virtual function* is a member function that is subject to dynamic binding. It is selected for execution at runtime instead of at compile time.
- A virtual function is selected for execution based on *the type of the object* it is called on:
  - for example in Program-10.3 on line 47, the `sing()` member function is called on the `Animal` pointer `ap1` that is set to a pointee `Bird` object on line 44
  - because `sing()` is declared as a virtual function in this program, the executing function is selected based on *the type of object* pointee, which is a `Bird`
  - as a result, the `Bird` class's `sing()` function executes on line 47
- A non-virtual function is selected for execution based on *the type of the handle* it's called on:
  - for example in Program-10.2 on line 52, the `sing()` member function is called on the `Animal` pointer `ap1` that is set to a pointee `Bird` object on line 49
  - because `sing()` is declared as a non-virtual function in this program, the executing function is selected based on *the type of handle*, which is an `Animal` pointer
  - as a result, the `Animal` class's `sing()` function executes on line 52
- Global functions and static member functions cannot be virtual.

10.3.1.2. **Characteristics of virtual functions:**

- Once a member function is declared as virtual in a base class, the "virtual-ness" is inherited by all the derived classes, all the way down the inheritance hierarchy.
- A derived class that inherits a virtual member function *cannot* make that function a non-virtual one. It cannot "turn off" the virtual-ness.
- Based strictly on C++ syntax, the `virtual` keyword is only required at the base class and does not need to be repeated in the derived classes.
- However, based on the important software engineering principle of code readability, it is an established programming convention to repeat the `virtual` keyword in every class definition where the virtual member function is overridden. By implicitly documenting our code this way, we assist the programmers who use our class as a base class in the future.

10.3.2. Functions selected for execution

			Type of object	
			base class	derived class
NON VIRTUAL FUNCTION	Type of pointer	base class	base class function	base class function
		derived class	doesn't compile	derived class function
VIRTUAL FUNCTION	Type of pointer	base class	base class function	derived class function
		derived class	doesn't compile	derived class function

Figure-10.4: Pointer and object data type combinations with virtual functions

10.3.2.1. **With non-virtual functions:**

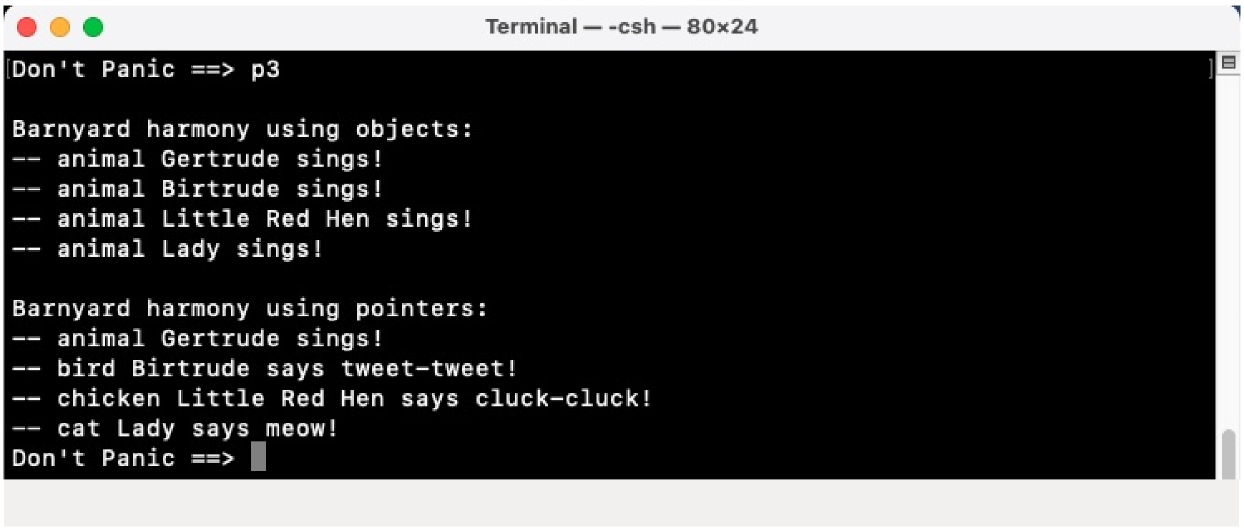
- The top two rows of Figure-10.4 illustrate which member function executes when a non-virtual function is called using different types of pointers with different types of objects.
- In all cases, with a non-virtual function, the choice of function to execute is based on the type of the object handle, in this case the type of pointer.
- Base class pointer:
  - with a base class pointer, the base class function is called regardless of the type of object pointee
- Derived class pointer:
  - a derived class pointer with a base class object pointee causes a compilation error, as we discussed in Program-10.1
  - with a derived class pointer and a derived class object pointee, the derived class function is called

10.3.2.2. With virtual functions:

- The bottom two rows of Figure-10.4 show which member function executes when a virtual function is called using different types of pointers with different types of objects.
- In all cases, with a virtual function and a pointer as the object handle, the choice of function to execute is based on the type of the object itself.
- Base class pointer:
  - with a base class pointer and a base class object pointee, the base class function is called
  - with a base class pointer and a derived class object pointee, the derived class function is called
- Derived class pointer:
  - a derived class pointer with a base class object pointee causes a compilation error, as we discussed in Program-10.1
  - with a derived class pointer and a derived class object pointee, the derived class function is called

10.3.3. Coding example: Polymorphism with virtual functions

```cpp
int main()
{
  Animal   gertrudeObj("Gertrude");
  Bird     birtrudeObj("Birtrude");
  Chicken  redHenObj("Little Red Hen");
  Cat      ladyObj("Lady");

  vector<Animal> barnyardObj;
  barnyardObj.push_back(gertrudeObj);
  barnyardObj.push_back(birtrudeObj);
  barnyardObj.push_back(redHenObj);
  barnyardObj.push_back(ladyObj);

  cout<<endl<<"Barnyard harmony using objects:"<<endl;
  for (int i=0; i<barnyardObj.size(); ++i) {
    barnyardObj[i].sing();
  }

  Animal*   gertrude = new Animal("Gertrude");
  Bird*     birtrude = new Bird("Birtrude");
  Chicken*  redHen   = new Chicken("Little Red Hen");
  Cat*      lady     = new Cat("Lady");

  vector<Animal*> barnyard;
  barnyard.push_back(gertrude);
  barnyard.push_back(birtrude);
  barnyard.push_back(redHen);
  barnyard.push_back(lady);

  cout<<endl<<"Barnyard harmony using pointers:"<<endl;
  for (int i=0; i<barnyard.size(); ++i) {
    barnyard[i]->sing();
  }

  return 0;
}
```



```
Don't Panic ==> p3

Barnyard harmony using objects:
-- animal Gertrude sings!
-- animal Birtrude sings!
-- animal Little Red Hen sings!
-- animal Lady sings!

Barnyard harmony using pointers:
-- animal Gertrude sings!
-- bird Birtrude says tweet-tweet!
-- chicken Little Red Hen says cluck-cluck!
-- cat Lady says meow!
Don't Panic ==>
```

Program-10.4: Polymorphism with virtual functions

Program purpose:

- Program-10.4 demonstrates a typical example of polymorphic behaviour, where a collection contains an assortment of different derived class objects which are treated as if they are base class objects.
- We see how a collection of objects of mixed data types behaves, first using static binding, and then with dynamic binding.
- Program-10.4 uses the same class hierarchy shown in Figure-10.3 and the classes defined in Program-10.3.
- This program uses the `vector` container from the standard template library (STL), which we discuss in chapter 15.

Lines 3-17:

- These lines show an example of a collection containing mixed-type objects, and how this results in static binding to determine which `sing()` member function executes.
- Lines 3-6 declare four objects within the `Animal` inheritance hierarchy, with one object for each type of class.
- Line 8 declares the `barnyardObj` variable as an STL `vector` that contains `Animal` objects. As we discussed in section 10.1.3, any object of a class derived from `Animal` can be treated as if it is an `Animal` object.
- Lines 9-12 add the four objects to the `barnyardObj` container.
- Lines 15-17 loop over the `barnyardObj` container and call the `sing()` member function for each of its elements.
- Even though the `sing()` member functions are defined as virtual, line 16 calls the function using an *object* handle, instead of an *object pointer* handle. This invokes static binding, as we saw in the first part of Program-10.2.
- With static binding, the called function is selected based on the type of handle. Because the `barnyardObj` container is declared as a collection of `Animal` objects on line 8, each of its elements is accessed with an `Animal` object handle.
- As a result, the `Animal` class's `sing()` function is called for each element in the container, as we see from the first part of the program output.

Lines 19-33:

- These lines show an example of a collection of mixed-type object pointers, and how this results in dynamic binding to determine which `sing()` member function executes.

- Lines 19-22 dynamically allocate four objects from the `Animal` class hierarchy, with one object for each type of class.
- Line 24 declares the `barnyard` variable as an STL `vector` of `Animal` object pointers.
- Lines 25-28 add the four object pointers to the `barnyard` container.
- Lines 31-33 loop over the `barnyard` container and call the `sing()` member function for each of its elements.
- The `sing()` member functions are defined as virtual, and the `barnyard` container is declared as a collection of `Animal` object pointers on line 24, so each of its elements is accessed with an `Animal` object pointer handle. Because of this, line 32 invokes dynamic binding, as we saw in the second part of Program-10.2.
- With dynamic binding, the called function is selected based on the type of object. So the function that is called on line 32 is based on the type of object that each container element pointee actually is.
- As a result, the `sing()` function called for each element in the container is the one corresponding to the actual type of object, as we see from the second part of the program output.

10.4. Virtual destructors

We discuss virtual destructors and their purpose in programs that use polymorphism.

10.4.1. Concepts

10.4.1.1. What is a *virtual destructor*:

- A *virtual destructor* is a class destructor that has been declared as a virtual member function.
- It ensures that the derived class destructor is called on a derived class object, even if a base class pointer is used to trigger the implicit call to the destructor.
- If a class destructor is non-virtual, static binding dictates that only the base class destructor executes, even with a derived class object. This can result in incomplete object cleanup.

10.4.1.2. With a non-virtual destructor:

- Declaring a destructor as non-virtual means that, when an object is deallocated, the selection of which destructor executes is based on *the type of handle*.
- If we use a base class pointer to deallocate a derived class object, declaring a non-virtual destructor means that only the base class destructor is called.
- In some platforms, declaring non-virtual destructors in classes that implement polymorphic behaviour may result in unpredictable behaviour.

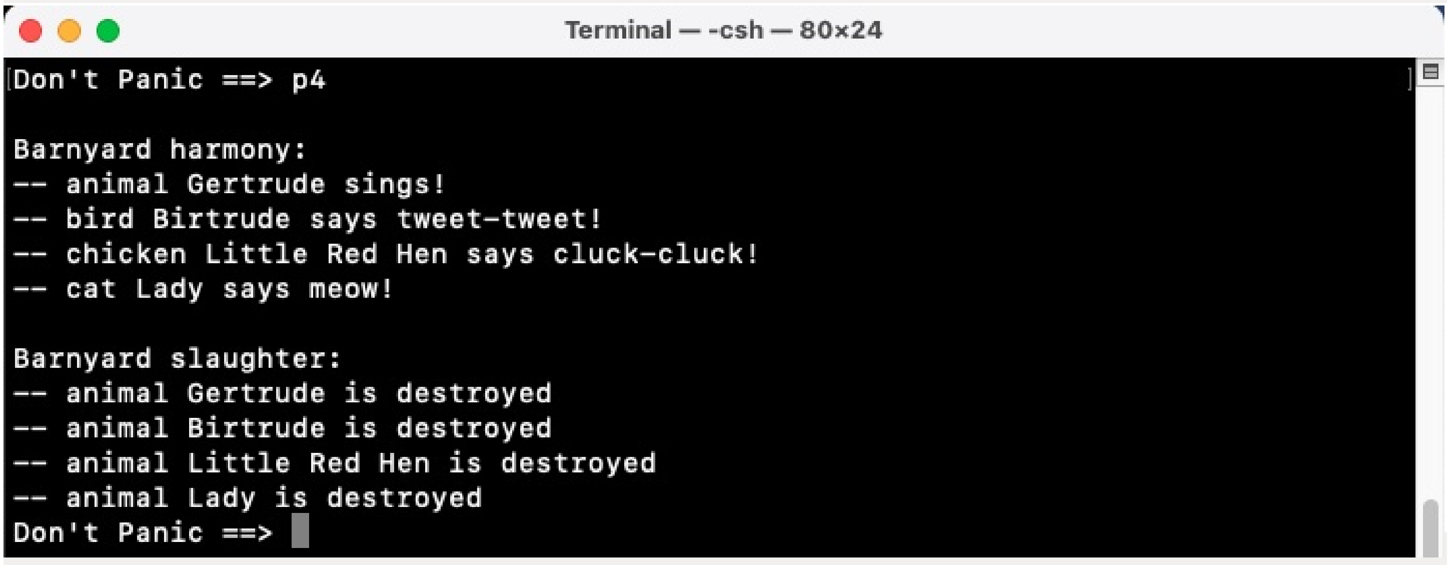
NOTE: The term *unpredictable behaviour* in library or compiler documentation is an unofficial euphemism for behaviour that is only sometimes correct, and even the people who wrote the compiler code can't predict the outcome. Overall, it's always a good rule of thumb to avoid anything to do with unpredictable behaviour.

10.4.1.3. With a virtual destructor:

- Declaring a destructor as virtual means that, when an object is deallocated, the selection of which destructor executes is based on *the type of object*, and not on the type of handle.
- If we use a base class pointer to deallocate a derived class object, having a virtual destructor means that the derived class destructor is called.
- Since an object is destroyed bottom-up, after the derived class destructor finishes executing, the base class destructor is called always automatically, as we discussed in section 9.3.

10.4.2. Coding example: Using non-virtual destructors

```
class Animal
{
  public:
    Animal(string n="") : name(n) { }
    ~Animal()            { cout<< "-- animal "<<name<<" is destroyed"<<endl; }
    virtual void sing() { cout<< "-- animal "<<name<<" sings!"<<endl; }
  protected:
    string name;
};

class Bird : public Animal
{
  public:
    Bird(string n="") : Animal(n) { }
    ~Bird()              { cout<< "-- bird "<<name<<" is destroyed"<<endl; }
    virtual void sing() { cout<< "-- bird "<<name<<" says tweet-tweet!"<<endl; }
};

class Chicken : public Bird
{
  public:
    Chicken(string n="") : Bird(n) { }
    ~Chicken()           { cout<< "-- chicken "<<name<<" is destroyed"<<endl; }
    virtual void sing() { cout<< "-- chicken "<<name<<" says cluck-cluck!"<<endl;}
};

class Cat : public Animal
{
  public:
    Cat(string n="") : Animal(n) { }
    ~Cat()               { cout<< "-- cat "<<name<<" is destroyed"<<endl; }
    virtual void sing() { cout<< "-- cat "<<name<<" says meow!"<<endl; }
};

int main()
{
  Animal*   gertrude = new Animal("Gertrude");
  Bird*     birtrude = new Bird("Birtrude");
  Chicken*  redHen   = new Chicken("Little Red Hen");
  Cat*      lady     = new Cat("Lady");

  vector<Animal*> barnyard;
  barnyard.push_back(gertrude);
  barnyard.push_back(birtrude);
  barnyard.push_back(redHen);
  barnyard.push_back(lady);

  cout<<endl<<"Barnyard harmony:"<<endl;
  for (int i=0; i<barnyard.size(); ++i) {
    barnyard[i]->sing();
  }

  cout<<endl<<"Barnyard slaughter:"<<endl;
  for (int i=0; i<barnyard.size(); ++i) {
    delete barnyard[i];
  }

  return 0;
}
```



```
Don't Panic ==> p4

Barnyard harmony:
-- animal Gertrude sings!
-- bird Birtrude says tweet-tweet!
-- chicken Little Red Hen says cluck-cluck!
-- cat Lady says meow!

Barnyard slaughter:
-- animal Gertrude is destroyed
-- animal Birtrude is destroyed
-- animal Little Red Hen is destroyed
-- animal Lady is destroyed
Don't Panic ==>
```

Program-10.5: Using non-virtual destructors

Program purpose:

- Program-10.5 demonstrates which destructors are executed when they are *not* defined as virtual and a derived class object is deallocated.
- As in Program-10.4, we are working with an STL `vector` that contains objects of different derived classes and are treated as if they are base class objects.
- Program-10.5 uses the same class hierarchy shown in Figure-10.3.

Lines 1-33:

- These lines show the class definitions for `Animal`, `Bird`, `Chicken`, and `Cat`.
- We see on lines 5, 15, 23, and 31 that a *non-virtual* destructor is implemented for each class.

Lines 37-46:

- These lines show the allocation and initialization of a collection of mixed-type object pointers.
- Lines 37-40 dynamically allocate four objects from the `Animal` class hierarchy, with one object for each type of class.
- Line 42 declares the `barnyard` variable as an STL `vector` of `Animal` object pointers.
- Lines 43-46 add the four object pointers to the `barnyard` container.

Lines 49-51:

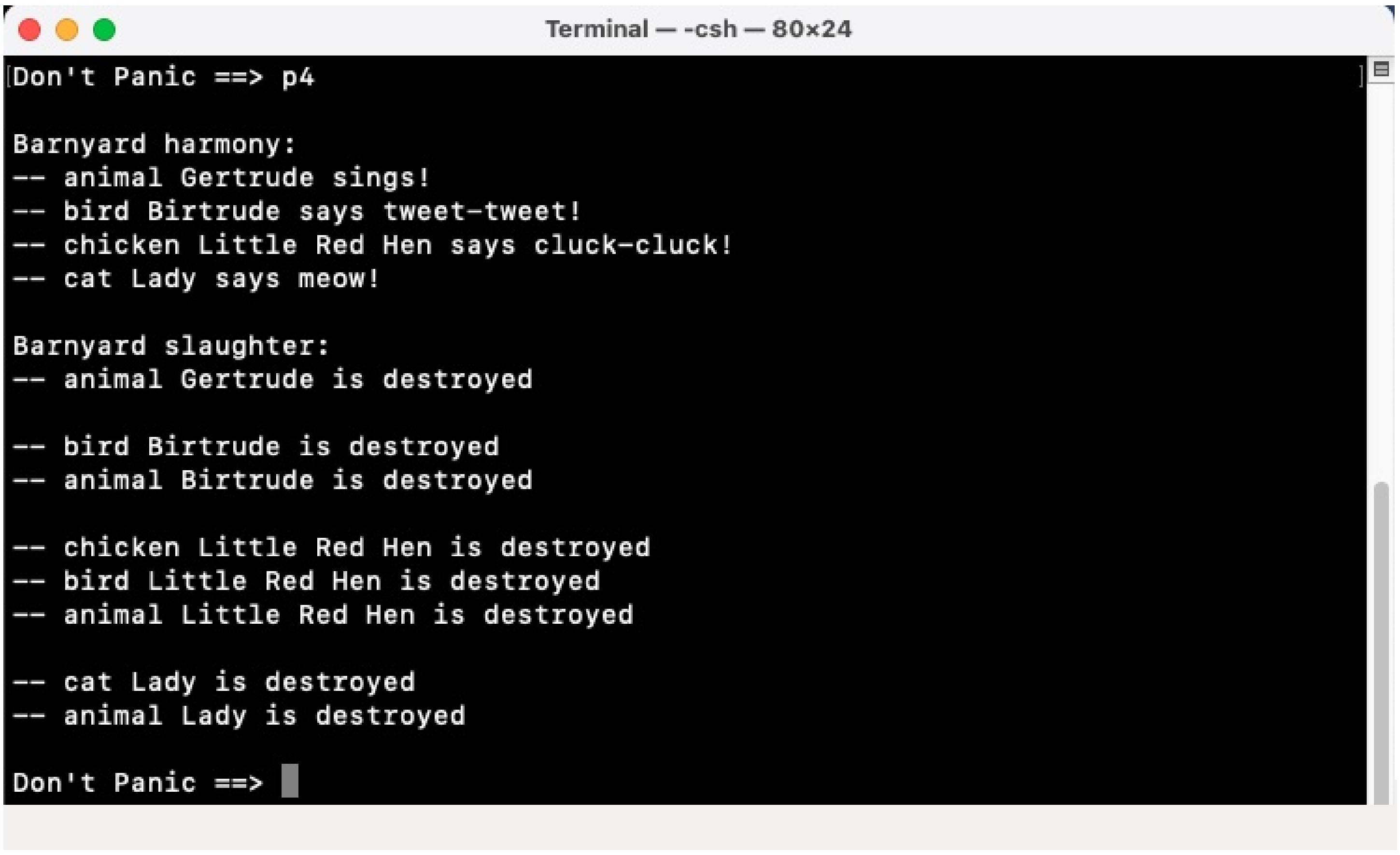
- These lines loop over the `barnyard` container and call the `sing()` member function for each of its elements.
- Because the `sing()` member functions are defined as virtual, and line 50 calls the function using an *object pointer* handle, this invokes dynamic binding, and the function is selected based on the type of object.
- As a result, the `sing()` function called for each element in the container is the one corresponding to the actual type of object, as we see from the first part of the program output.

Lines 54-56:

- These lines loop over the `barnyard` container and deallocate the memory for each of its elements. The `delete` operator automatically calls the destructor for each element.
- Because the destructors are *not* defined as virtual, line 55 invokes the destructor for each element in the container based on the type of handle, which are all `Animal` pointers.
- As a result, only the `Animal` destructor executes for each element in the container, as we see from the second part of the program output.

10.4.3. Coding example: Using virtual destructors

```cpp
class Animal
{
  public:
    Animal(string n="") : name(n) { }
    virtual ~Animal()   { cout<< "-- animal "<<name<<" is destroyed"<<endl; }
    virtual void sing() { cout<< "-- animal "<<name<<" sings!"<<endl; }
  protected:
    string name;
};

class Bird : public Animal
{
  public:
    Bird(string n="") : Animal(n) { }
    virtual ~Bird()     { cout<< "-- bird "<<name<<" is destroyed"<<endl; }
    virtual void sing() { cout<< "-- bird "<<name<<" says tweet-tweet!"<<endl; }
};

class Chicken : public Bird
{
  public:
    Chicken(string n="") : Bird(n) { }
    virtual ~Chicken()  { cout<< "-- chicken "<<name<<" is destroyed"<<endl; }
    virtual void sing() { cout<< "-- chicken "<<name<<" says cluck-cluck!"<<endl;}
};

class Cat : public Animal
{
  public:
    Cat(string n="") : Animal(n) { }
    virtual ~Cat()      { cout<< "-- cat "<<name<<" is destroyed"<<endl; }
    virtual void sing() { cout<< "-- cat "<<name<<" says meow!"<<endl; }
};
```



```
Don't Panic ==> p4

Barnyard harmony:
-- animal Gertrude sings!
-- bird Birtrude says tweet-tweet!
-- chicken Little Red Hen says cluck-cluck!
-- cat Lady says meow!

Barnyard slaughter:
-- animal Gertrude is destroyed

-- bird Birtrude is destroyed
-- animal Birtrude is destroyed

-- chicken Little Red Hen is destroyed
-- bird Little Red Hen is destroyed
-- animal Little Red Hen is destroyed

-- cat Lady is destroyed
-- animal Lady is destroyed

Don't Panic ==>
```

Program-10.6: Using virtual destructors

Program purpose:

- Program-10.6 is a variation of Program-10.5, and it illustrates which destructors are executed when they are defined as virtual and a derived class object is deallocated.
- The `main()` function is identical to Program-10.5.

Lines 1-33:

- These lines show the class definitions for `Animal`, `Bird`, `Chicken`, and `Cat`.
- We see on lines 5, 15, 23, and 31 that a *virtual* destructor is declared and implemented for each class.

Program execution:

- Lines 54-56 from Program-10.5 loop over the `barnyard` container and deallocate the memory for each of its elements. The `delete` operator automatically calls the destructor for each element.
- Because the destructors are defined as virtual, line 55 calls the destructor for each element in the container based on the type of object that each element pointee actually is.
- As a result, the correct destructors are called to clean up every part of each derived class object in the container, as we see from the second part of the program output.

10.5. Abstract classes

We discuss the principles of abstract classes in OO design and how they are implemented in C++.

10.5.1. Concepts

10.5.1.1. **What is an *abstract class*:**

- An *abstract class* is a class that is too generalized to be instantiated. In other words, a program cannot create any instances of the class.
- A class that can be instantiated is called a *concrete class*.
- When using polymorphism, it's common to design an abstract base class that provides a generalized interface, while the concrete derived classes implement individual functionality.
- In OO design, abstract classes are a key tool for promoting data abstraction by grouping together common data and behaviour inside a base class.

10.5.1.2. **Characteristics of abstract classes:**

- No objects of an abstract class can ever be created.
- Unlike some other OO languages, there is no keyword in C++ to declare a class as abstract.
- An abstract class in C++ is a class that contains at least one *pure virtual function*.
- We represent an abstract class in a UML class diagram with its class name in *italics*.

10.5.2. Pure virtual functions

10.5.2.1. **What is a *pure virtual function*:**

- A *pure virtual function* is a virtual member function that is declared in a class definition as having *no implementation*.
- This is not simply a question of declaring a member function prototype with either no corresponding implementation or an empty one. There is a special syntax that must be used to declare a member function as never having an implementation.
- For example, we declare a `dance()` member function as pure virtual in the `Animal` class definition with the following syntax: `virtual void dance() = 0;`

10.5.2.2. **Characteristics of pure virtual functions:**

- Declaring *at least one* pure virtual member function makes a class abstract.
- This means that all derived classes **must** override the pure virtual function and provide an implementation in order to be concrete.
- If a derived class does *not* override an inherited pure virtual function, then the derived class is also abstract.

10.5.3. Coding example: Abstract classes

The program in this example implements the UML class diagram shown in Figure-10.5.

The `Animal` class:

- The `Animal` class shown in Figure-10.5 contains the same data member and `sing()` member function as Figure-10.3. The class is also defined as *abstract*, with its name shown in italics.
- It contains a `dance()` member function that is also shown in italics, to denote it as a pure virtual function.

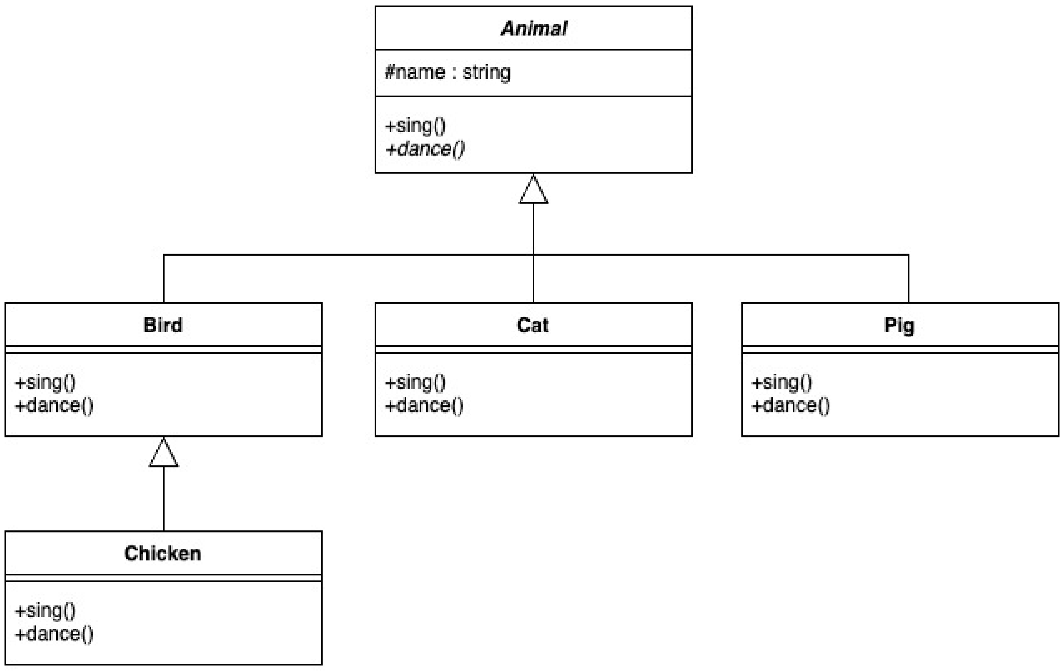


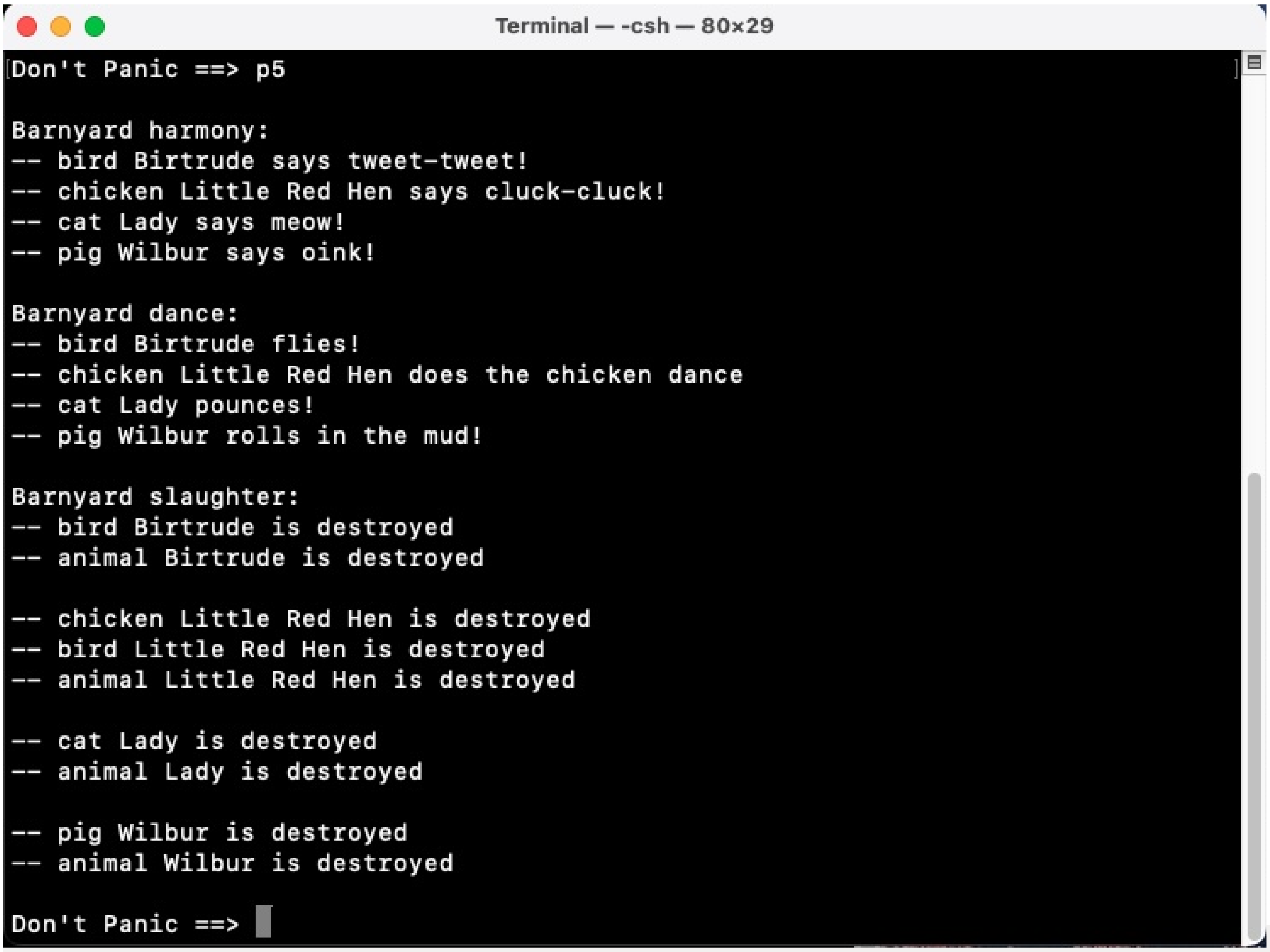
Figure-10.5: UML class diagram for Program-10.7

The `Bird`, `Cat`, `Chicken`, and `Pig` classes:

- Figure-10.5 shows that the `Bird`, `Cat`, and `Pig` classes are derived from the `Animal` class, and `Chicken` is derived from `Bird`.
- We see that all four are concrete classes, because their names are not shown in italics.
- The concrete classes also override the `dance()` member function inherited from its base class. and each one provides its own implementation. Each one also overrides and provides an implementation for the `sing()` member function.

```
class Animal
{
  public:
    Animal(string n="") : name(n) { }
    virtual ~Animal()       { cout<< "-- animal "<<name<<" is destroyed"<<endl; }
    virtual void sing()     { cout<< "-- animal "<<name<<" sings!"<<endl; }
    virtual void dance() = 0;
  protected:
    string name;
};
class Bird : public Animal
{
  public:
    Bird(string n="") : Animal(n) { }
    virtual ~Bird()       { cout<< "-- bird "<<name<<" is destroyed"<<endl; }
    virtual void sing()   { cout<< "-- bird "<<name<<" says tweet-tweet!"<<endl; }
    virtual void dance() { cout<< "-- bird "<<name<<" flies!"<<endl; }
};
class Chicken : public Bird
{
  public:
  Chicken(string n="") : Bird(n) { }
  virtual ~Chicken()   { cout<< "-- chicken "<<name<<" is destroyed"<<endl; }
  virtual void sing()  { cout<< "-- chicken "<<name<<" says cluck-cluck!"<<endl;}
  virtual void dance() {cout<< "-- chicken "<<name<<" does the chicken dance"<<endl;}
};
```

```
class Cat : public Animal
{
  public:
    Cat(string n="") : Animal(n) { }
    virtual ~Cat()        { cout<< "-- cat "<<name<<" is destroyed"<<endl; }
    virtual void sing()   { cout<< "-- cat "<<name<<" says meow!"<<endl; }
    virtual void dance() { cout<< "-- cat "<<name<<" pounces!"<<endl; }
};
class Pig : public Animal
{
  public:
    Pig(string n="") : Animal(n) { }
    virtual ~Pig()        { cout<< "-- pig "<<name<<" is destroyed"<<endl; }
    virtual void sing()   { cout<< "-- pig "<<name<<" says oink!"<<endl; }
    virtual void dance() { cout<< "-- pig "<<name<<" rolls in the mud!"<<endl; }
};

int main()
{
  Bird*     birtrude = new Bird("Birtrude");
  Chicken*  redHen   = new Chicken("Little Red Hen");
  Cat*      lady     = new Cat("Lady");
  Pig*      wilbur   = new Pig("Wilbur");

  vector<Animal*> barnyard;
  barnyard.push_back(birtrude);
  barnyard.push_back(redHen);
  barnyard.push_back(lady);
  barnyard.push_back(wilbur);

  cout<<endl<<"Barnyard harmony:"<<endl;
  for (int i=0; i<barnyard.size(); ++i) {
    barnyard[i]->sing();
  }

  cout<<endl<<"Barnyard dance:"<<endl;
  for (int i=0; i<barnyard.size(); ++i) {
    barnyard[i]->dance();
  }

  cout<<endl<<"Barnyard slaughter:"<<endl;
  for (int i=0; i<barnyard.size(); ++i) {
    delete barnyard[i];
    cout<<endl;
  }

  return 0;
}
```



```
Don't Panic ==> p5

Barnyard harmony:
-- bird Birtrude says tweet-tweet!
-- chicken Little Red Hen says cluck-cluck!
-- cat Lady says meow!
-- pig Wilbur says oink!

Barnyard dance:
-- bird Birtrude flies!
-- chicken Little Red Hen does the chicken dance
-- cat Lady pounces!
-- pig Wilbur rolls in the mud!

Barnyard slaughter:
-- bird Birtrude is destroyed
-- animal Birtrude is destroyed

-- chicken Little Red Hen is destroyed
-- bird Little Red Hen is destroyed
-- animal Little Red Hen is destroyed

-- cat Lady is destroyed
-- animal Lady is destroyed

-- pig Wilbur is destroyed
-- animal Wilbur is destroyed

Don't Panic ==>
```

Program-10.7: Abstract classes

Program purpose:

- Program-10.7 demonstrates the use of an abstract class and its concrete derived classes, based on the design shown in Figure-10.5.
- Each class in the program has its own implementation of the `sing()` member function, which is overridden in every derived class.
- Each concrete class also overrides and implements the `dance()` member function.

Lines 1-42:

- These lines show the class definitions for `Animal`, `Bird`, `Chicken`, `Cat`, and `Pig`.
- The `Animal` class implements the `sing()` member function on line 6. The `Bird`, `Chicken`, `Cat`, and `Pig` derived classes all inherit the `sing()` member function from their base class, and they override it with their own unique implementation on lines 16, 24, 32, and 40.
- Line 7 shows the declaration of the `Animal` class's pure virtual `dance()` member function. No implementation is provided for this function in the `Animal` class, and the declaration uses the special syntax.
- The `Bird`, `Chicken`, `Cat`, and `Pig` classes all override the pure virtual `dance()` member function that they inherit from their base class with their own unique implementation on lines 17, 25, 33, and 41.

Lines 44-74:

- These lines show the implementation of the `main()` function.
- The `main()` function is almost identical to Program-10.6, with the following changes.
- We no longer create an `Animal` object. Since the `Animal` class is now abstract, trying to create an instance of it would result in a compilation error.
- Line 49 dynamically allocates a new `Pig` object, and line 55 adds it to the `barnyard` container with all the other animals.
- Lines 63-65 loop over the `barnyard` container and call the `dance()` member function for each of its elements.
- Because `dance()` is defined as a virtual function and called using object pointer handles, dynamic binding is invoked. The called function is selected based on the type of object that each container element pointee actually is, and the correct messages are printed, as we see from the program output.

Timmy Tortoise

Prince Harold the Hare

10.6. Design example: A race between a Tortoise and a Hare

We illustrate two possible OO designs for a program that simulates a foot race between a Tortoise and a Hare. The first design uses neither inheritance nor polymorphism, and the second one uses both. We discuss the advantages of the latter in terms of good software engineering practice.

10.6.1. Rules of the race

10.6.1.1. **Goal:**

- The program we are designing is the simulation of a race between Timmy Tortoise and Prince Harold the Hare.
- Both runners start at the bottom of the Mount of Doom, and the first to reach the top of the mountain wins the race.

10.6.1.2. **The path:**

- Our two heroes are racing up the steep mountain on a path too narrow to run side-by-side.
- We represent the path as an array of 40 characters, with each hero’s avatar in their current position. The runners begin at position zero at the bottom of the mountain, and the first to reach position 40 at the top wins the race.
- Timmy is represented with the avatar ’`T`’ and Harold with ’`H`’.
- At the beginning of the program, the path contains a snack at every 10 positions. If a runner lands on a snack, they eat it. Once a snack is eaten, it is removed from the path.

10.6.1.3. **Runner movements:**

- The runners movements are randomized, and each one moves differently, in accordance with their nature. Tortoises are slow and steady. Hares are faster, but they tend to nap. Because the mountain path is slippery, runners sometimes slide down the mountain instead of moving upward.
- The Tortoise moves as follows:
  - 50% of the time, it plods quickly uphill by 3 positions, which takes 2 energy points
  - 30% of the time, it plods slowly uphill by 1 position, for 1 energy point
  - 20% of the time, it takes a slip downhill by 6 positions, for no energy points
- The Hare moves as follows:
  - 20% of the time, it takes a big hop uphill by 9 positions, which takes 3 energy points
  - 30% of the time, it takes a small hop uphill by 1 position, for 1 energy point
  - 10% of the time, it takes a big slip downhill by 12 positions, for no energy points
  - 20% of the time, it takes a small slip downhill by 2 positions, for no energy points
  - 20% of the time, it takes a nap and stays in the same position, for no energy points
- At every iteration of the race, the program randomly chooses the next move for each runner, from that runner's set of possible moves, in accordance with the associated probability.

10.6.1.4. **Runner energy level:**

- Each hero begins with an energy level of 25 points, and they lose points as they move up the mountain.
- If the two runners collide by landing on the same spot on the path at the same time, the Tortoise bites the Hare, and the Hare loses 3 energy points.
- If a runner eats a snack, their energy level increase back to the maximum of 25 points.
- If a runner's energy level reaches zero, that runner dies.

10.6.2. Design without inheritance and polymorphism

We consider two designs in this example. The original design is demonstrated in the UML class diagram shown in Figure-10.7, and it does *not* use inheritance or polymorphism.

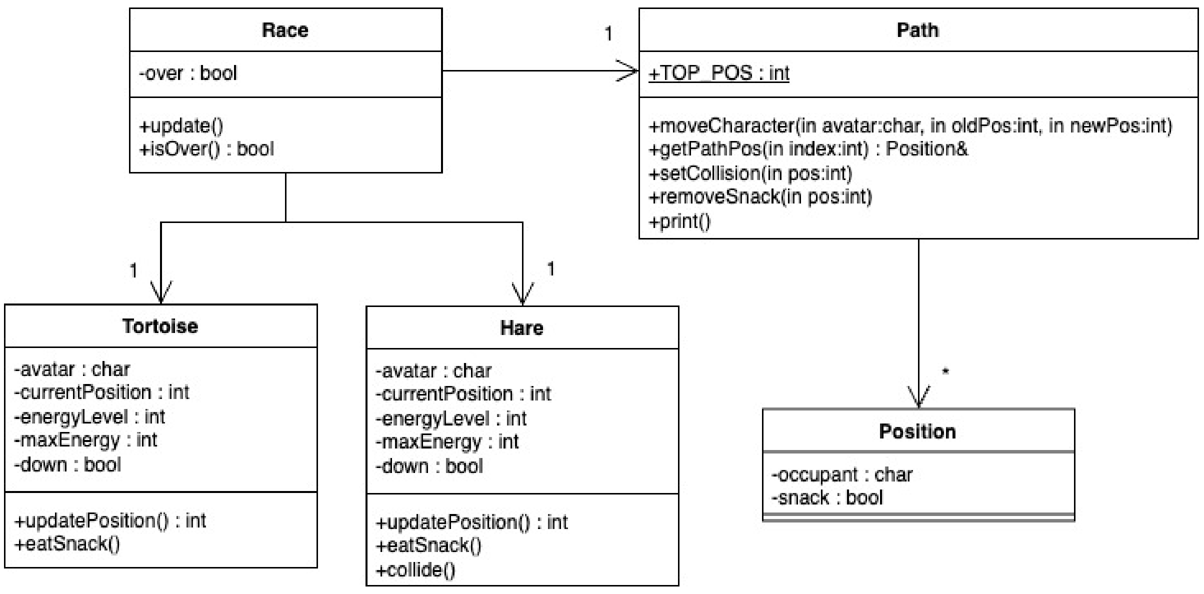


Figure-10.7: Tortoise and Hare: original design

The `Race` class:

- The `Race` class serves as the control object in this program. It is responsible for the overall control flow of the race between the Tortoise and the Hare.
- We see from Figure-10.7 that the `Race` class contains four data members: a flag that indicates whether or not the race is over, a `Path` object that represents the path up the Mount of Doom, a `Tortoise` object and a `Hare` object that represent our runners.
- In this program, the `main()` function creates a `Race` object and loops until the race is over (similar to a game loop). At every iteration, it updates the `Race` object.
- The `Race` class has a public member function called `update()`, which implements one iteration of the game loop, and an `isOver()` member function that reports if the race is finished.
- At every iteration, the `update()` member function does the following:
  - compute a new position for each runner, i.e. both Tortoise and Hare
  - move each runner's avatar on the path from their previous position to their new one
  - detect if a collision has occurred between the two runners at their new positions
    - ➢ if so, place a `'!'` character in that position on the path to indicate a collision, and have the Hare handle the collision
    - ➢ if not, check if each runner has landed on a snack; if so, the runner eats the snack and it is removed from the path
  - print out the path to the screen
  - check if the race has concluded; the race is finished if either:
    - ➢ both runners have died, or
    - ➢ one runner has reached the top of the Mount of Doom
  - if the race is over, print out a message announcing either the winner or the deaths of both runners
- The `Race` class also contains two member functions that are not shown in the UML class diagram: a default constructor and a destructor.

The `Position` class:

- The `Position` class represents a single position on the race path.
- We see from Figure-10.7 that the `Position` class contains two data members: a flag that indicates whether or not the position stores a snack, and a character that represents the current occupant of the position.
- If the position is currently occupied by a runner, the position occupant is set to that runner's avatar. If the position is empty, the occupant is simply a space character.
- The `Position` class also has other member functions that are not shown in the UML class diagram, including a default constructor, and getters and setters for both data members.

The `Path` class:

- The `Path` class represents the path used for the race.
- We see from Figure-10.7 that the `Path` class contains two data members: a static data member that stores the path's maximum position (40), and a collection of `Position` objects. There is one `Position` object for each position on the path, so the collection contains 40 elements.
- The `Path` class provides a `moveCharacter()` member function that moves a runner from one position on the path to another, to simulate that runner's movement. The function erases the runner's avatar from its previous position and places it at the new one.
- The `getPathPos()` member function returns a reference to the `Position` object stored in the positions collection at the index provided in the parameter.
- The `setCollision()` member function places a `'!'` character as the occupant at the given position where a collision has occurred.

- The `removeSnack()` member function removes a snack from a specific position.
- The `print()` member function prints out the occupant of every position in the path, as well as the current energy levels of both the Tortoise and the Hare.
- The `Path` class also contains a default constructor that sets a snack at every 10 positions.

The `Tortoise` class:

- The `Tortoise` class stores the information for the Tortoise runner.
- Figure-10.7 shows that the `Tortoise` class contains five data members: its avatar, its current position on the path as the index in the path's positions collection where the Tortoise is currently located, its current energy level, the maximum energy level that it can store, and a flag to indicate if the Tortoise is dead.
- The `Tortoise` class provides an `updatePosition()` member function that computes the Tortoise's next position on the path, based on its set of possible movements, as follows:
  - check if the Tortoise is dead or has won the race; if so, it doesn't move
  - randomly compute the Tortoise's next move from its set of possible moves
  - decrease the runner's energy level by the amount corresponding to the next move
  - compute the runner's new position based on its current position and the direction and number of positions of the next move
  - return the newly computed position as an index in the path's positions collection
- The `eatSnack()` member function resets the runner's energy level to the maximum.
- The `Tortoise` class also has other member functions that are not shown in the UML class diagram, including a default constructor, and getters for the avatar, current position, current energy level, and death flag data members.
- It also provides private member functions for setting the energy level and current position data members. These are private because they must only be used by the `Tortoise` class to perform boundary checking. Negative values are reset to zero, and a current position that exceeds the top of the mountain is reset to the maximum position.

The `Hare` class:

- The `Hare` class stores the information for the Hare runner.
- Figure-10.7 shows that the `Hare` class contains the same five data members as `Tortoise`.
- The `Hare` class also provides an `updatePosition()` member function that does the same as the equivalent in the `Tortoise` class, but it chooses a next move from the Hare's set of possible moves.
- The `eatSnack()` member function is identical to the `Tortoise` class.
- The `collide()` member function decreases the Hare's energy level by 3 points.
- The `Hare` class also has other member functions that are not shown in the UML class diagram, including a default constructor, and the same getters and setters as the `Tortoise` class.

Design discussion:

- As we can see, this is an example of a terrible design.
- There is a significant amount of code that is identical in both the `Tortoise` and `Hare` classes, resulting in code duplication.
- The `Race` class cannot treat the runners generically, so it must always be aware of which runner is a `Tortoise` and which one is a `Hare` so that it can call the correct member functions.
- If we added more types of runners that move differently, the `Race` class would constantly require updates to reflect these changes.
- This results in poor data abstraction and encapsulation, which we address in the next section.

10.6.3. Design with inheritance and polymorphism

A better design is captured in the UML class diagram shown in Figure-10.8, and it uses both inheritance or polymorphism.

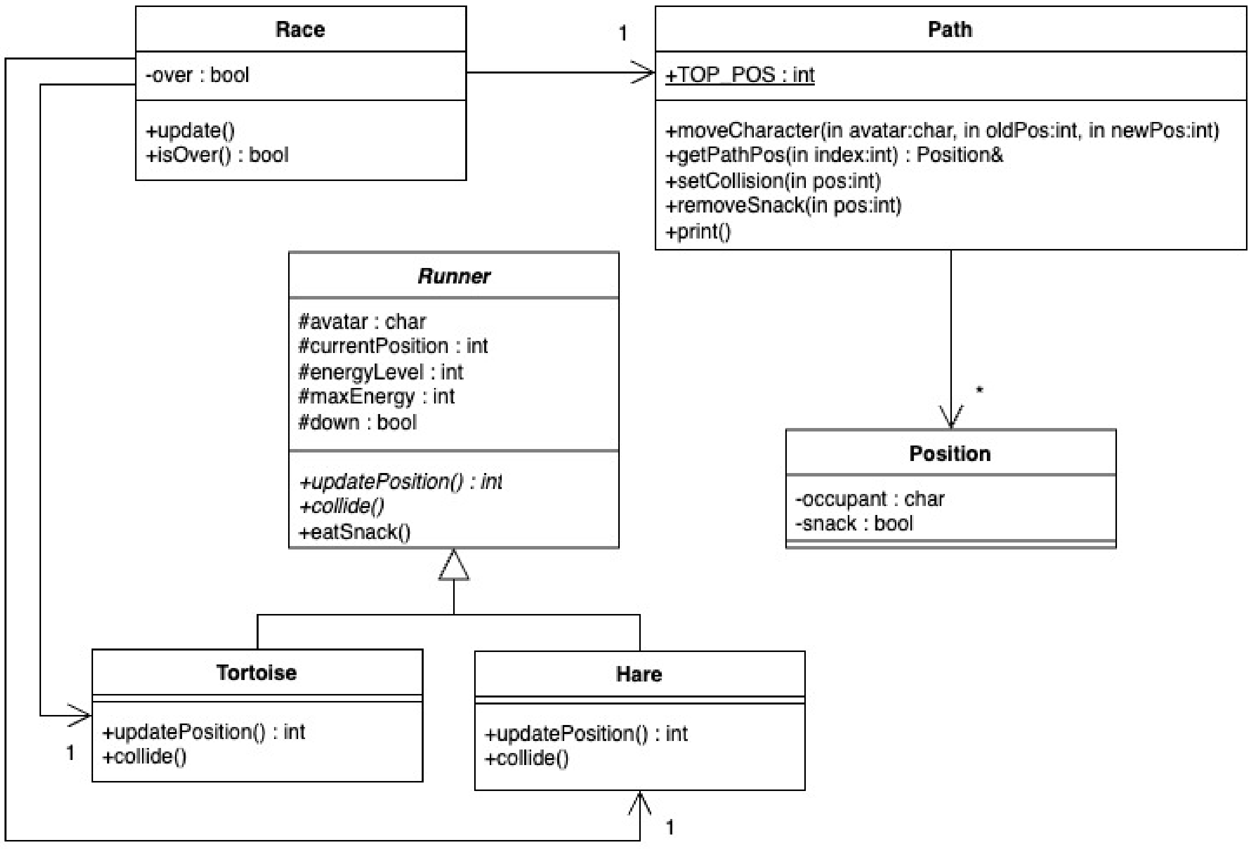


Figure-10.8: Tortoise and Hare: better design

The `Race` class:

- The `Race` class is identical to the one in section 10.6.2. It contains the same data members, as well as the same member functions with identical function implementations.
- The `update()` member function contains the same logic: it tells each runner to compute its next position; it gets the path to move the runner avatars to the new positions; it detects and deals with collisions; it checks for snacks at the new runner positions; it gets the path to print itself; and finally it checks if the race is over.

The `Path` and `Position` classes :

- The `Path` and `Position` classes are identical to the ones in section 10.6.2. They contain the same data members and member functions with identical function implementations.

The `Runner` class:

- We introduce a new `Runner` class to capture all the commonality between the `Tortoise` and `Hare` classes into an inheritance hierarchy, as shown in Figure-10.8.
- The UML class diagram in Figure-10.8 illustrates that `Runner` is an abstract class, and it is used as a base class with both `Tortoise` and `Hare` derived from it.
- The `Runner` class contains all five data members that were common to both `Tortoise` and `Hare` in section 10.6.2: the runner's avatar, its current position on the path as the index in the path's positions collection where the runner is currently located, its current energy level, the maximum energy level that it can store, and a flag to indicate if the runner is dead.

- Figure-10.8 also indicates that the five data members are now declared as protected, so that they can be accessed directly in `Runner`'s derived classes.
- The `Runner` class declares an `updatePosition()` member function as a *polymorphic* function, which is implemented in C++ as a pure virtual function. There are no moves possible for runners in general, so `Runner` cannot provide an implementation. Instead, this member function must be overridden by all concrete derived classes.
- The `collide()` member function is also declared as polymorphic. Even though collisions only affect the Hare, it's a better design to keep collisions generic. If we need additional derived classes to represent more kinds of runners in the future, some may require specialized collision functionality. Implementing this functionality inside an overridden function allows the `Race` class to be independent from how individual runners handle collisions.
- The `eatSnack()` member function is now implemented in `Runner`, as it is identical for both derived classes.
- The `Runner` class also has other member functions that are not shown in the class diagram, including a default constructor and a destructor. The getter and setter member functions that were present in both `Tortoise` and `Hare` classes are now in the `Runner` class.

The `Tortoise` class:

- In this design, the `Tortoise` class is now derived from the `Runner` class. It must implement only the member functions that are specific to the Tortoise, and it must override the polymorphic member functions declared in `Runner`.
- The `Tortoise` class overrides the `updatePosition()` member function that computes the Tortoise's next position on the path, based on its set of possible moves. This implementation is identical to the one described in section 10.6.2.
- The `Tortoise` class must also override the `collide()` member function inherited from `Runner`. Since the Tortoise does not suffer any effects when it collides with the Hare, the function has an empty implementation.
- The `Tortoise` class also provides a default constructor.

The `Hare` class:

- The `Hare` class is also now derived from the `Runner` class. It must implement only the member functions that are specific to the Hare, and it must override the polymorphic member functions declared in `Runner`.
- The `Hare` class overrides the `updatePosition()` member function that computes the Hare's next position on the path, based on its set of possible moves. This implementation is identical to the one described in section 10.6.2.
- The `Hare` class must also override the `collide()` member function inherited from `Runner`, with the same implementation as in section 10.6.2.
- The `Hare` class also provides a default constructor.

NOTE: Functional equivalency does not imply equal design quality. Two programs can have the same behaviour without their designs having equivalent software engineering qualities.

In this design, it may be tempting to provide an empty implementation of the `collide()` member function in the `Runner` class, so that it can be inherited in `Tortoise` and overridden in `Hare`. But while that option is functionally equivalent to the provided design, it is not of equal quality. It makes no sense to provide default collision behaviour at the base class because `Runner` is too generic. Declaring the member function as polymorphic in the base class forces the designers of the derived classes to think about their class behaviour. If a derived class is not affected by collisions, it's better design to force it to explicitly provide an empty implementation.

Design discussion:

- This is an example of a good design.
- The code duplication between the `Tortoise` and `Hare` classes from section 10.6.2 has been eliminated by introducing the `Runner` base class.
- The `Race` class can now treat the runners generically, and it no longer needs to be aware of which runner is a `Tortoise` and which one is a `Hare`. Because they are both a kind of `Runner`, the polymorphic member functions can be called using `Runner` pointer handles. Dynamic binding will ensure that the correct implementation executes.
- With this design, adding more types of runners is no longer problematic. As long as the correct behaviour is implemented in the derived classes, the `Race` class no longer requires updates to accommodate new runners and their behaviour.
- This is good data abstraction and encapsulation, and it upholds the principles of correct software engineering.

10.7. Behaviour classes

We examine an established technique for separating behaviour from our entity classes, based on the Strategy design pattern, which is discussed in chapter 11. Behaviour classes use polymorphism to model program behaviour as objects that can be seamlessly switched at runtime.

10.7.1. Concepts

10.7.1.1. **What is a *behaviour class*:**

- A *behaviour class* is a class that represents and encapsulates a specific behaviour.
- Behaviour classes are usually organized as an inheritance hierarchy where each derived class represents a different, specialized behaviour.

10.7.1.2. **Why use behaviour classes:**

- Representing a behaviour as an instance of a behaviour class allows a client class to easily change *at runtime* which behaviour is executed.
- Changing behaviour is done by switching to a different behaviour class object.
- Behaviour classes are crucial in making our classes resilient in the face of future changes.

10.7.2. Design example: Changing behaviour using entity classes

We consider two alternative designs for this example. The first is shown in this section, and it encapsulates polymorphic behaviour inside the entity classes. The second design is discussed in section 10.7.3, and it uses behaviour classes.

The UML class diagram shown in Figure-10.9 suggests a possible class hierarchy for game characters with a polymorphic moving behaviour. This design is similar to previous examples we've seen with polymorphism, where changing behaviour is encapsulated within the entity classes.

The `GameCharacter` class:

- The `GameCharacter` class is an abstract class that represents game characters in a zombie apocalypse game.
- In this game, the hero characters are types of students, characterized by their behaviour in the face of zombies: some run away, others fight the zombies, and some just stand there and scream. The zombies are game characters that chase the students.
- This base class declares the polymorphic `move()` member function, but it does not provide an implementation for it.

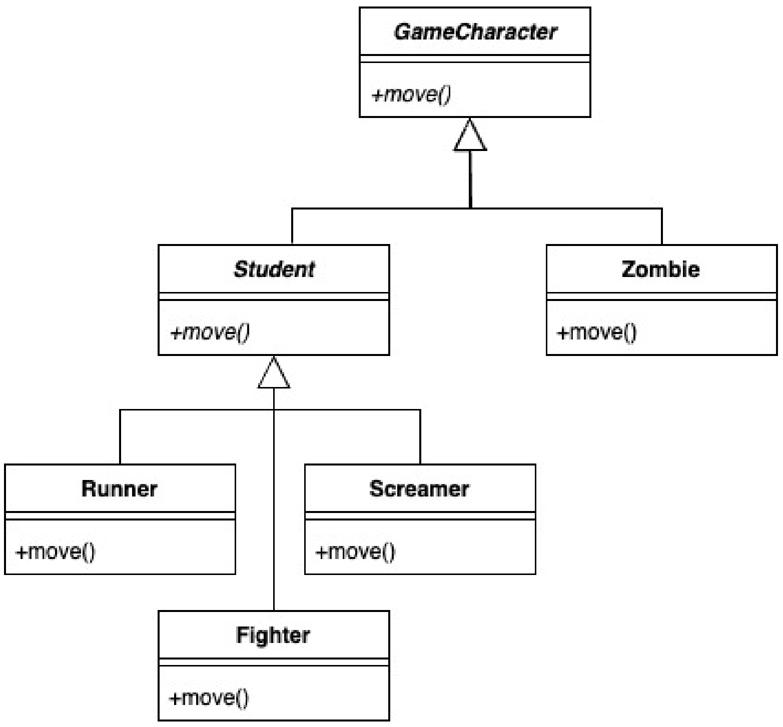


Figure-10.9: Changing behaviour using entity classes

The `Student` class:

- The `Student` class is derived from `GameCharacter`, and it is also abstract.
- It provides no implementation for the `move()` member function.

The `Zombie` class:

- The `Zombie` class is derived from `GameCharacter`, and it is a concrete class.
- It overrides the `move()` member function and provides an implementation for it that models the way that zombies chase students in the game.

The `Runner`, `Fighter`, and `Screamer` classes:

- These classes are derived from `Student`, and they are concrete classes.
- Each class overrides the `move()` member function and provides an implementation for it, based on the class's specific behaviour (running, fighting, or screaming).

Design discussion:

- This is a perfectly valid design. It uses inheritance and polymorphism, and it provides good data abstraction and encapsulation.
- However, a serious issue arises if a game character's behaviour needs to change at runtime.
- For example, when a screamer gets caught by the zombies (and you know they will be), how can their behaviour change to chasing students instead?
- Implementing a fundamental change in behaviour using a design based on entity classes is neither easy nor pretty. A far more elegant solution involves the use of behaviour classes instead, as we see in section 10.7.3.

10.7.3. Design example: Changing behaviour using behaviour classes

The UML class diagram in Figure-10.10 represents an alternative design for the apocalypse game presented in section 10.7.2. It illustrates two class hierarchies: one for the game characters, and another for the behaviour classes.

In this design, the `move()` member function in the `GameCharacter` hierarchy is **not** polymorphic, and none of the derived classes provide an implementation for it. Instead, every concrete derived class inherits a data member that's a `MoveBehaviour` object. The `GameCharacter` base class provides an implementation for the `move()` member function that simply delegates the moving behaviour to a game character's `MoveBehaviour` containee object.

The `MoveBehaviour` hierarchy consists of: an abstract base class with a polymorphic `move()` member function; and a concrete derived class for each type of behaviour, where the derived class overrides the `move()` member function and provides its own unique implementation.

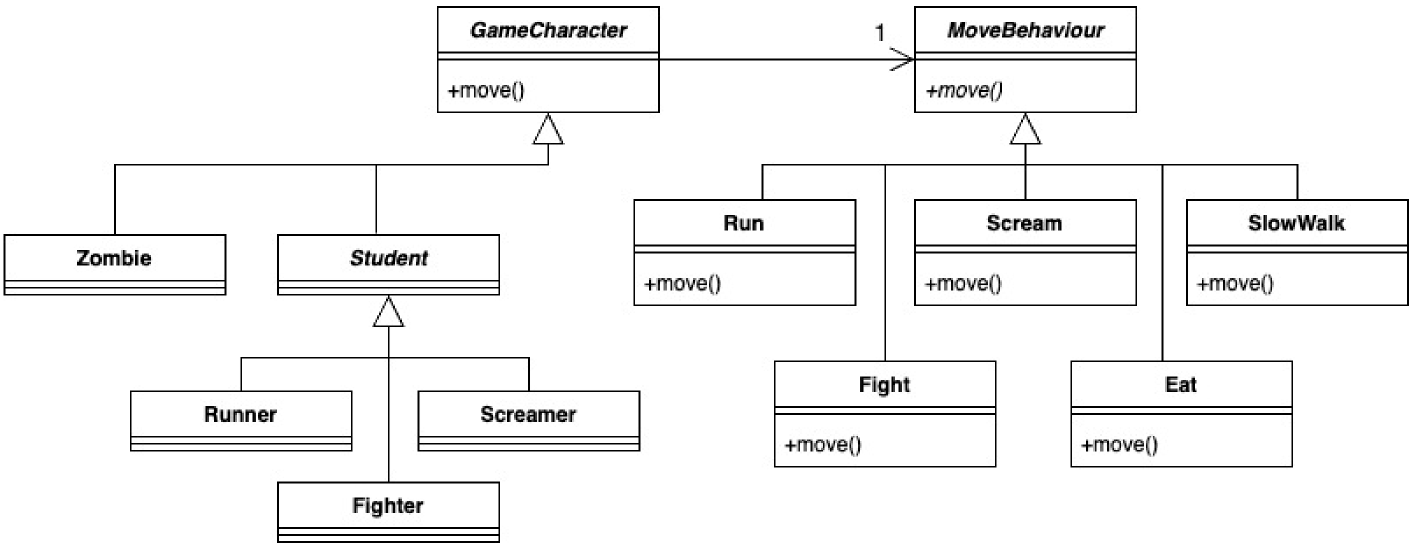


Figure-10.10: Changing behaviour using behaviour classes

The `GameCharacter` class:

- The `GameCharacter` class remains an abstract class that represents a game character in the zombie apocalypse game.
- This base class has one containee object that's an instance of one of the concrete classes derived from `MoveBehaviour`.
- `GameCharacter` provides an implementation for the `move()` member function, which simply calls the same function on its containee `MoveBehaviour` object.

The `MoveBehaviour` class:

- The `MoveBehaviour` class is an abstract class representing a moving behaviour in the game.
- This base class declares the polymorphic `move()` member function, but it does not provide an implementation for it.

The `Run`, `Fight`, `Scream`, `SlowWalk`, and `Eat` classes:

- These classes are derived from `MoveBehaviour`, and they are concrete classes.
- Each class overrides the `move()` member function and provides an implementation for it, based on the class's specific behaviour.

The `Student` class:

- The `Student` class is derived from `GameCharacter`, and it is also abstract.
- It inherits the `MoveBehaviour` data member from `GameCharacter`.
- It provides no implementation for the `move()` member function.

The `Zombie` class:

- The `Zombie` class is derived from `GameCharacter`, and it is a concrete class.
- It inherits the `MoveBehaviour` data member from `GameCharacter`.
- It does not provide an implementation for the `move()` member function, but it inherits the one provided in `GameCharacter`.
- When a `Zombie` object is created, its class constructor must initialize the `MoveBehaviour` containee data member to a new instance of a moving behaviour that makes sense for a zombie, so either slow walking or eating.

The `Runner`, `Fighter`, and `Screamer` classes:

- These classes are derived from `Student`, and they are concrete classes.
- They inherit the `MoveBehaviour` data member from `Student`.
- None of these classes provide an implementation for the `move()` member function, but they inherit the one provided in `GameCharacter`.
- When an instance of these classes is created, the class constructor must initialize its behaviour to one that makes sense for that type of student.
- The `Runner` class constructor must initialize its `MoveBehaviour` containee data member to a new instance of the `Run` behaviour class; the `Fighter` class constructor initializes its containee to a new instance of the `Fight` behaviour class; and the `Screamer` class constructor initializes it to a new instance of the `Scream` behaviour class.
- At any time during the game, a `Runner`, `Fighter`, or `Screamer` object can modify its behaviour simply by updating its containee to a different type of `MoveBehaviour` object.

Design discussion:

- Like the design shown in section 10.7.2, this one uses inheritance and polymorphism, and it provides good data abstraction and encapsulation.
- This design also addresses the issue that arises from using only entity classes, specifically how a game character's behaviour can change at runtime.
- With this design, the actual behaviour is delegated to an inheritance hierarchy of behaviour classes. Using polymorphism, the derived classes provide the implementations of their own unique style of moving behaviour.
- Changing a behaviour becomes a simple update of the containee behaviour object to a different type of behaviour.
- For example, the screamer is initialized with screaming behaviour. But when it is caught by the zombies, its behaviour can change to slow walking or eating other characters.
- There are other advantages to using behaviour classes:
  - adding a new behaviour simply requires adding a new behaviour class, with no need to change the entity classes
  - we can reuse behaviours; for example, if we want zombies to fight each other, we can reuse the fighting behaviour
  - if a cure is found, zombies can become students again

10.7.4. Coding example: Behaviour classes

The program in this example implements the UML class diagram shown in Figure-10.11. It is a variation of Program-10.7, but with the dancing implemented inside behaviour classes instead of within the entity classes.

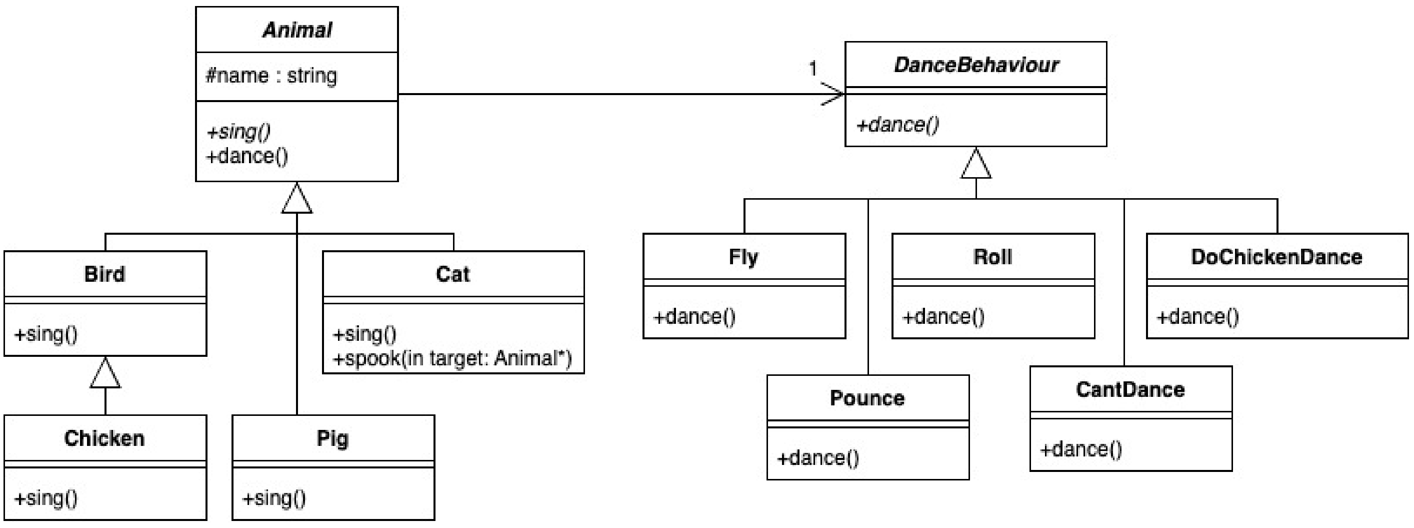


Figure-10.11: UML class diagram for Program-10.8

The `Animal` class:

- The `Animal` class shown in Figure-10.11 contains a `name` data member and a polymorphic `sing()` member function.
- This class is also defined as *abstract*, with its name represented in italics.
- The class has an additional data member, which is an instance of one of the concrete classes derived from `DanceBehaviour`. So every animal has a containee object that encapsulates the dancing behaviour of that particular animal.
- Unlike Program-10.7, the `Animal` class in this program *does* provide an implementation for the `dance()` member function. It calls the same function on its containee `DanceBehaviour` object.

The `DanceBehaviour` class:

- The `DanceBehaviour` class is an abstract class representing a dancing behaviour.
- This base class declares the polymorphic `dance()` member function, but it does not provide an implementation for it.

The `Fly`, `Pounce`, `Roll`, `CantDance`, and `DoChickenDance` classes:

- These classes are derived from `DanceBehaviour`, and they are concrete classes.
- Each class overrides the `dance()` member function and provides an implementation for it, based on the class's specific behaviour.

The `Bird`, `Chicken`, `Cat`, and `Pig` classes:

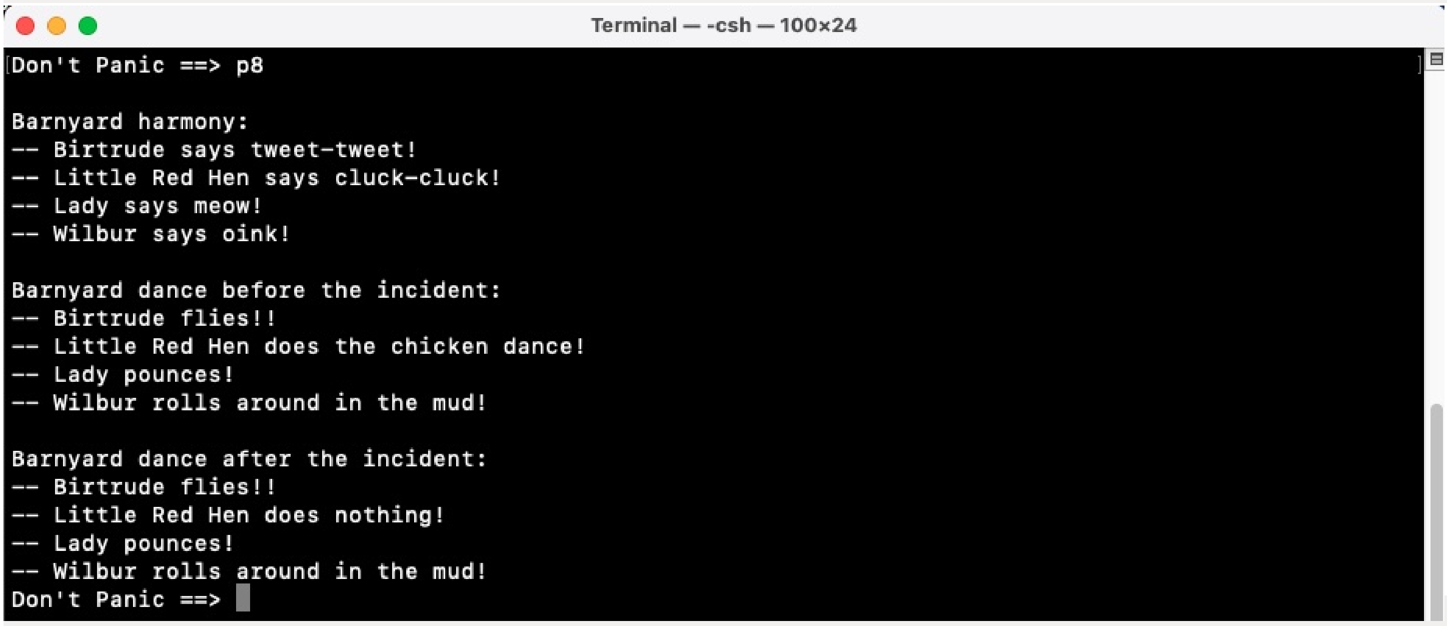
- These classes are derived from `Animal`, directly or indirectly, and they are concrete classes.
- They inherit the `DanceBehaviour` data member from their base class.
- Each class overrides the `sing()` member function and provides its own implementation.
- None of these classes provide an implementation for the `dance()` member function, but they inherit the one provided in `Animal`.
- When an instance of these classes is created, the class constructor must initialize its behaviour to one that makes sense for that type of animal.

- At any time during the program, the derived class object can modify its behaviour simply by updating its containee to a different type of `DanceBehaviour` object.
- The `Cat` class also provides a `spook()` member function. This function is called when a cat attacks another animal and that animal is injured. During their recovery, the injured animal cannot dance. So the `spook()` function resets the injured animal's dance behaviour to indicate that it can no longer dance.

```cpp
/* * * * * * * * * * * * * * * * * *
 * Filename:  Animals.h            *
 * * * * * * * * * * * * * * * * * */
class Animal
{
  public:
    Animal(string="", DanceBehaviour* b=nullptr);
    virtual void sing() = 0;
    void dance();
    void setDanceB(DanceBehaviour*);
  protected:
    string name;
    DanceBehaviour *danceBehaviour;
};

class Bird : public Animal
{
  public:
    Bird(string="", DanceBehaviour* b=new Fly);
    virtual void sing();
};

class Chicken : public Bird
{
  public:
    Chicken(string="", DanceBehaviour* b=new DoChickenDance);
    virtual void sing();
};

class Cat : public Animal
{
  public:
    Cat(string="", DanceBehaviour* b=new Pounce);
    virtual void sing();
    void spook(Animal*);
};

class Pig : public Animal
{
  public:
    Pig(string="", DanceBehaviour* b=new Roll);
    virtual void sing();
};

```

```
/* * * * * * * * * * * * * * * * * *
 * Filename:  Animals.cc          *
 * * * * * * * * * * * * * * * * */
Animal::Animal(string n, DanceBehaviour* b) : name(n), danceBehaviour(b) { }
Bird::Bird(string n, DanceBehaviour* b) : Animal(n, b) { }
Chicken::Chicken(string n, DanceBehaviour* b) : Bird(n, b) { }
Cat::Cat(string n, DanceBehaviour* b) : Animal(n, b) { }
Pig::Pig(string n, DanceBehaviour* b) : Animal(n, b) { }

void Animal::setDanceB(DanceBehaviour* b)
{
  if (danceBehaviour != nullptr) {
    delete danceBehaviour;
  }
  danceBehaviour = b;
}

void Animal::dance()
{
  cout <<"-- " <<name<< " ";
  danceBehaviour->dance();
  cout <<"!"<<endl;
}

void Bird::sing()    { cout <<"-- " <<name<< " says tweet-tweet!" << endl; }
void Chicken::sing() { cout <<"-- " <<name<< " says cluck-cluck!" << endl; }
void Cat::sing()     { cout <<"-- " <<name<< " says meow!" << endl; }
void Pig::sing()     { cout <<"-- " <<name<< " says oink!" << endl; }

void Cat::spook(Animal* a) { a->setDanceB(new CantDance); }

/* * * * * * * * * * * * * * * * * *
 * Filename:  DanceBehaviour.h    *
 * * * * * * * * * * * * * * * * */
class DanceBehaviour
{
  public:
    virtual ~DanceBehaviour() { };
    virtual void dance() = 0;
};

class CantDance :      public DanceBehaviour { public: virtual void dance(); };
class Pounce :         public DanceBehaviour { public: virtual void dance(); };
class DoChickenDance : public DanceBehaviour { public: virtual void dance(); };
class Fly :            public DanceBehaviour { public: virtual void dance(); };
class Roll :           public DanceBehaviour { public: virtual void dance(); };

/* * * * * * * * * * * * * * * * * *
 * Filename:  DanceBehaviour.cc   *
 * * * * * * * * * * * * * * * * */
void CantDance::dance()      { cout << "does nothing"; }
void Pounce::dance()         { cout << "pounces"; }
void DoChickenDance::dance() { cout << "does the chicken dance"; }
void Fly::dance()            { cout << "flies!"; }
void Roll::dance()           { cout << "rolls around in the mud"; }

```

```
/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc            *
 * * * * * * * * * * * * * * * * */
int main()
{
  Bird*     birtrude = new Bird("Birtrude");
  Chicken*  redHen   = new Chicken("Little Red Hen");
  Cat*      lady     = new Cat("Lady");
  Pig*      wilbur   = new Pig("Wilbur");

  vector<Animal*> barnyard;
  barnyard.push_back(birtrude);
  barnyard.push_back(redHen);
  barnyard.push_back(lady);
  barnyard.push_back(wilbur);

  cout<<endl<<"Barnyard harmony:"<<endl;
  for (int i=0; i<barnyard.size(); ++i) {
    barnyard[i]->sing();
  }

  cout<<endl<<"Barnyard dance before the incident:"<<endl;
  for (int i=0; i<barnyard.size(); ++i) {
    barnyard[i]->dance();
  }

  lady->spook(redHen);

  cout<<endl<<"Barnyard dance after the incident:"<<endl;
  for (int i=0; i<barnyard.size(); ++i) {
    barnyard[i]->dance();
  }

  return 0;
}
```



Terminal — -csh — 100×24

```
Don't Panic ==> p8

Barnyard harmony:
-- Birtrude says tweet-tweet!
-- Little Red Hen says cluck-cluck!
-- Lady says meow!
-- Wilbur says oink!

Barnyard dance before the incident:
-- Birtrude flies!!
-- Little Red Hen does the chicken dance!
-- Lady pounces!
-- Wilbur rolls around in the mud!

Barnyard dance after the incident:
-- Birtrude flies!!
-- Little Red Hen does nothing!
-- Lady pounces!
-- Wilbur rolls around in the mud!
Don't Panic ==>
```

Program-10.8: Behaviour classes

Program purpose:

- Program-10.8 demonstrates an implementation of runtime changes using behaviour classes.
- In this scenario, we have our usual complement of barnyard animals: Birtrude the bird, Little Red Hen the chicken, Lady the cat, and Wilbur the pig. They are all singing and dancing until the cat decides to chase the chicken, and the chicken falls off the fence and breaks its little leg. A kind veterinarian sets the chicken's leg in a cast, so it will be perfectly healed in six weeks. But for the time being, the chicken can no longer dance. We must modify the chicken's behaviour at runtime from doing the chicken dance to being unable to dance.
- Program-10.8 implements the class hierarchy shown in Figure-10.11. While Program-10.7 implements behaviour inside the entity classes of the `Animal` hierarchy, Program-10.8 encapsulates behaviour inside a second hierarchy of behaviour classes.
- While this program focuses on dancing behaviour, the behaviour classes approach could be applied to singing behaviour as well.

Lines 4-43:

- These lines show the class definitions for `Animal`, `Bird`, `Chicken`, `Cat`, and `Pig`.
- The `Animal` class is abstract, and the other four are concrete.
- Line 13 shows that the `Animal` class declares a `DanceBehaviour` containee object. This is declared as a pointer to support the use of a polymorphic function.
- Line 10 declares a setter for the dance behaviour containee.
- Line 9 declares a *non-virtual* `dance()` member function, which is given an implementation in this class.
- None of the concrete classes override the `dance()` member function.
- Line 19 shows the declaration of the `Bird` constructor. We see that the default dance behaviour of every new bird is a new `Fly` object.
- Line 26 declares the `Chicken` constructor, with the default dance behaviour of every new chicken set to a `DoChickenDance` object.
- On line 33, the `Cat` constructor specifies the default dance behaviour to be a `Pounce` object.
- Line 41 declares the `Pig` constructor, with the default dance behaviour set to a `Roll` object.

Lines 48-74:

- These lines contain the member function implementations for the `Animal` hierarchy classes.
- Lines 48-52 show the default constructor implementations. The derived class constructors call the base class constructor, which initializes the new dance behaviour containee object.
- Lines 54-60 implement the setter member function for the dance behaviour containee object. Since the object is always dynamically allocated, the previous behaviour object must be explicitly deallocated before the containee is set to a new object. Otherwise, we would create a memory leak.
- Lines 62-67 show the implementation of the `Animal` class's `dance()` member function. It prints the animal's name before delegating the printing of a specific message to the dance behaviour containee object on line 65.
- Lines 69-72 contain the concrete implementations of the `sing()` member functions that are overridden by each concrete class.
- Line 74 shows how a cat attacking another animal results in the target animal's dance behaviour changing at runtime so that it can no longer dance.

Lines 79-90:

- These lines show the class definitions for the abstract `DanceBehaviour` class and its five concrete derived classes.
- Lines 79-84 contain the `DanceBehaviour` class definition. We see from line 83 that the `dance()` member function is polymorphic and must be overridden by the concrete classes.
- Lines 86-90 show the concrete class definitions. They are all derived from `DanceBehaviour`, and each one overrides the `dance()` member function.

Lines 95-99:

- These lines contain the implementations of the overridden `dance()` member functions in the five concrete classes.
- When any animal is told to dance, line 65 invokes the `DanceBehaviour` object's overridden `dance()` member function. So depending on the type of `DanceBehaviour` object, line 65 results in the execution of one of the implementations shown on lines 95-99.

Lines 104-135:

- These lines show the implementation of the `main()` function.
- Lines 106-109 allocate and initialize four animal objects, each of a different type.
- Lines 111-115 declare a container of `Animal` pointers and add each animal to it.
- Lines 118-120 conduct the barnyard sing-along, where each animal sings in its own way.
- Lines 123-125 show the initial barnyard dance before the incident. We see from the program output that each animal dances in accordance with its default dance behaviour, as initialized in the derived class constructors.
- On line 127, the cat attacks the chicken, which results in the chicken breaking its leg and changing its dancing behaviour.
- Lines 130-132 show the barnyard dance after the incident. We see from the program output that each animal dances in accordance with its default dance behaviour, except for the chicken that can no longer dance.

Chapter 11

Design Patterns

Design patterns are an important tool in OO design. They provide consistent and reliable solutions to some common OO design issues. Each pattern is defined as a set of predefined classes, with prescribed associations between them and sometimes operations within them, in order to address a specific type of problem. Most design patterns make use of inheritance and often polymorphism to provide additional separation between the client classes and the pattern classes. [9]

In this chapter, we introduce the principles of design patterns and their general categories, and we cover a few of the more commonly used patterns.

11.1. Principles

We discuss the concepts and different types of design patterns in OO computing.

11.1.1. Basics

11.1.1.1. **What is a *design pattern*:**

- In general, a *pattern* is the regular and consistent way in which something is repeated.
- A *design pattern* in computing is a solution to a commonly occurring OO programming problem. It's an established way of organizing classes in order to solve one of these problems.
- Design patterns are specific to OO design and programming. They have no equivalent in other programming paradigms like imperative, functional, or logic.

11.1.1.2. **How are design patterns used:**

- There are several established, well-defined design patterns.
- Each pattern is meant to solve a *very specific* computing problem. It is a specialized tool meant to be used exclusively for the purpose for which it was designed.
- A design pattern should never be applied to a problem it was not designed to solve. A good OO design only uses the design patterns that are known to fit with the situation.
- Each design pattern dictates the precise usage of:
  - inheritance and polymorphism
  - delegation through composition
  - specific operations to be implemented

11.1.1.3. **The authoritative textbook on design patterns:**

- All official design patterns are defined in the following textbook [9]: *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994.
- The authors are collectively and colloquially known as the "Gang of Four".
- The design pattern definitions in this textbook uses the term client class in the same way we used it in section 10.1.2. A client class is a class that is *using* the classes in the design pattern.

11.1.1.4. **What are the types of design patterns:**

- There are three types of design patterns defined in the Gang of Four textbook:
  - creational
  - structural
  - behavioural
- *Architectural patterns* also exist, but they are not true design patterns. They work on a larger scale and usually involve the organization of large groups of classes called subsystems and/or components. Architectural patterns are outside the scope of this textbook.

11.1.2. Types of design patterns

11.1.2.1. **Creational design patterns:**

- Creational design patterns are used for specifying *how objects are created*, including which objects are created and which objects create other objects.
- Examples of creational design patterns include Factory, Abstract Factory, and Singleton.

11.1.2.2. **Structural design patterns:**

- Structural design patterns specify *how objects are associated with each other*, usually through inheritance and composition.
- Examples of structural design patterns include Façade, Bridge, Decorator, and Proxy.

11.1.2.3. **Behavioural design patterns:**

- Behavioural design patterns specify *how objects communicate with each other*, specifically which objects call what operations on which other objects.
- Examples of behavioural design patterns include Observer, Strategy, and Visitor.

11.1.2.4. **Architectural patterns are *not* design patterns:**

- Architectural patterns determine *how objects are grouped together* into subsystems and/or components, which are groups of classes that belong together functionally.
- Examples of architectural patterns include client-server, peer-to-peer, model-view-controller (MVC), layered, and a few others.

11.2. Selected design patterns

We illustrate and discuss a sample of the design patterns defined in the Gang of Four textbook.

11.2.1. Façade

11.2.1.1. **What is the *Façade* design pattern:**

- *Façade* is a structural design pattern. Its name comes from a French word that means an artificial outward appearance.
- With this pattern, we implement a Façade class that serves as a *gateway* to other classes.
- We don't usually name this class Façade, because that's its job, not its name.

11.2.1.2. **How is Façade used:**

- Façade simplifies the interface to a group of complex classes:
  - the client class calls the simplified operations defined in the Façade class
  - the Façade class takes care of calling the correct operation(s) on the actual class(es), by using delegation
- The operations on the actual classes are still available to the client class, but it's simpler to use the Façade class interface instead.
- The Façade class serves to encapsulate the details of the actual classes that provide the operations.

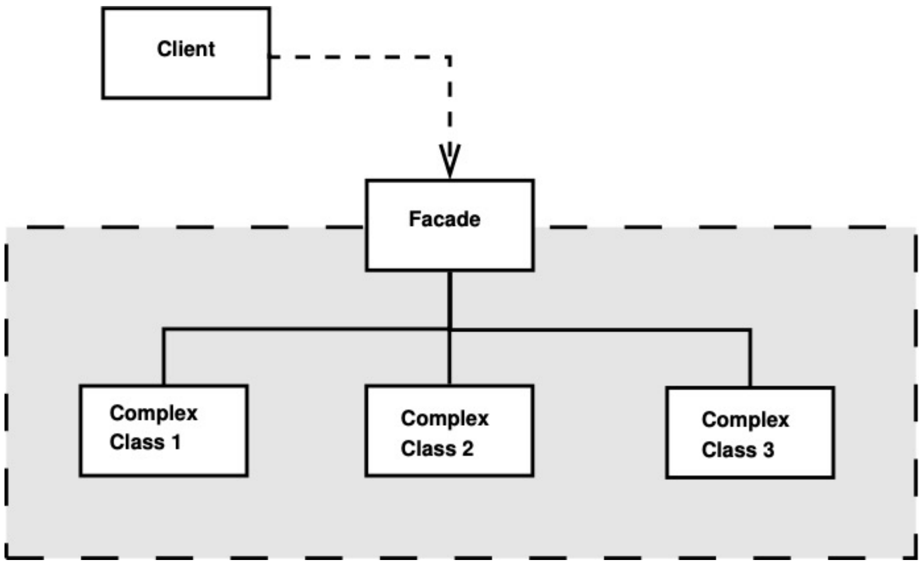


Figure-11.1: Façade design pattern

11.2.1.3. **Structure of the Façade design pattern:**

- Figure-11.1 illustrates the layout of the Façade design pattern.
- We see that the client class uses the operations on the Façade class, which has its own relationships with the complex classes that do the actual work.
- In a UML class diagram, a dashed line represents a "uses" relationship, where one class uses the services of another, for example calling its operations.

11.2.1.4. **Example of Façade:**

- Figure-11.2 shows an example of the Façade design pattern.
- In this example, a user interface (UI) is the client class, and the Façade class is a control class responsible for implementing a one-click-ordering feature on an online vendor web site.
- The UI can still access the individual classes that do the work of collecting user profile, billing, and shipping information. But these individual steps are simplified with the one-click-ordering feature, which gathers all the same information from the same classes.
- The end result is a higher level of abstraction for the UI, which can remain isolated from any future changes in the way that profile, billing, and shipping information is used.

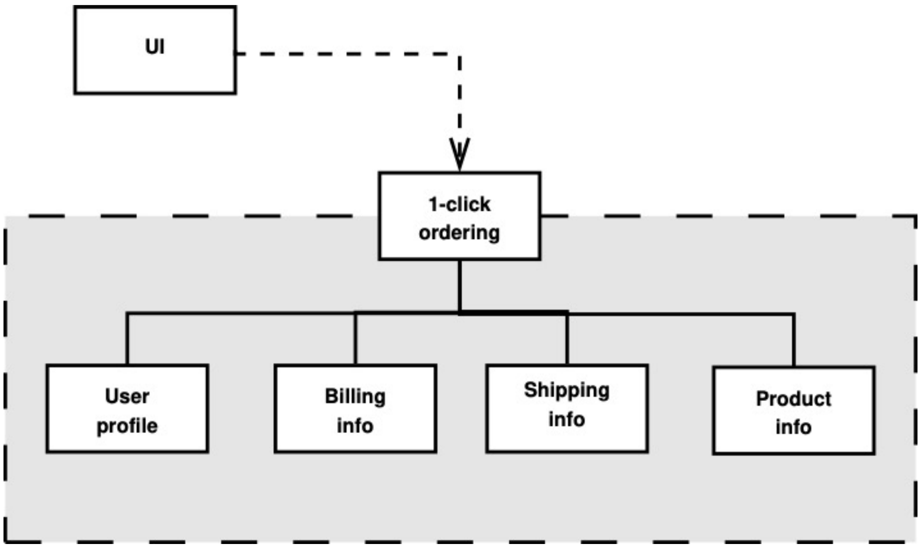


Figure-11.2: Example of Façade design pattern

11.2.2. Observer

11.2.2.1. **What is the *Observer* design pattern:**

- *Observer* is a behavioural design pattern. It specifies the set of classes and operations that must be implemented as part of this pattern.
- It allows observer objects to track changes in the state of a subject object and react to those changes.
- A subject's *state* is information about the subject on which an observer's behaviour is dependent. If the subject's state changes, its observers must be notified so that they can update their behaviour accordingly.
- The exact nature of observer objects, and subject objects and their state is entirely dependent on the particular system being designed.
- The purpose of the Observer design pattern is to facilitate some measure of independence between classes that have an inherent dependence on each other. Observer and subject classes can still change and evolve separately with minimal impact on one another, as long as their interactions are limited to those prescribed by the Observer pattern.

11.2.2.2. **How is Observer used:**

- Observer provides a generic way for classes to react to the changing state of other classes in a way that maintains their independence from each other.
- A class hierarchy of *subject* classes must be implemented:
  - the subject maintains a collection of the observers that are following its current state
  - the subject notifies all its observers any time that its state changes
- A class hierarchy of *observer* classes must be implemented:
  - the observer subscribes to notifications from the subject that it's interested in
  - the observer updates itself every time it's notified of a change in the subject's state
- In some systems, subject/observer classes are also called publisher/subscriber.

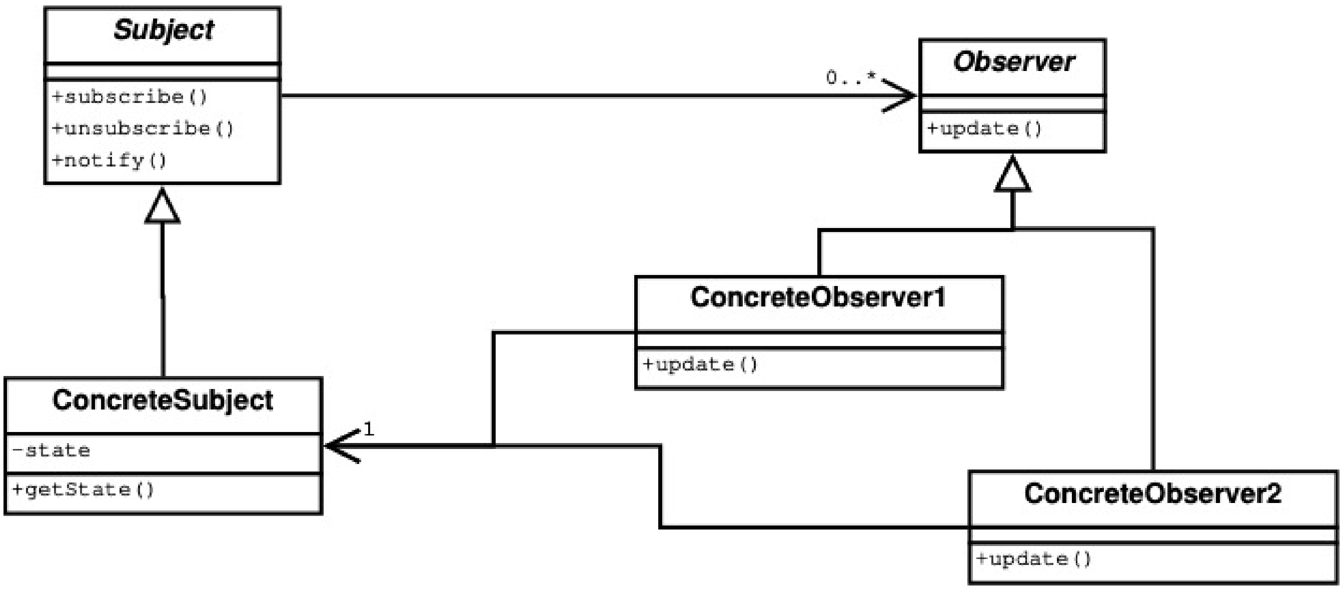


Figure-11.3: Observer design pattern

11.2.2.3. **Structure of the Observer design pattern:**

- Figure-11.3 illustrates the layout of the Observer design pattern.
- The subject base class is abstract, and it must maintain, as a containee, a collection of the observer objects that are interested in tracking a subject's state.
- This subject base class also must provide the operations necessary for observers to subscribe and unsubscribe from a subject's collection of observers. In addition, it must provide a notification operation that loops over the subject's collection of observers and calls the update operation on each one.
- Concrete subject classes store their state and make it available to observers.
- The observer base class is abstract, and it must provide an update operation that's called when a subject notifies its observers of a change in state.
- Concrete observer classes may contain a reference to the subject that they are tracking, in order to query the subject for its updated state.

11.2.2.4. **Example of Observer:**

- Figure-11.4 shows an example of the Observer design pattern.
- In this example, the subject class hierarchy represents a variety of online games accessible through a web browser. There is one concrete subject, which is a chess game.
- The subject base class contains a collection of observers, each of which is a different type of view of an online game in progress.
- The observer base class represents the different types of views that may be available for participants to interact with an online game.
- We see two different types of views as concrete observer classes: the player view and the spectator view. The two views are very different from each other, since they provide different interfaces to the game being played.
- When a chess player makes a new move, the UI for both the players and the spectators watching the game must be updated accordingly. The notification operation allows the game subject to tell its view observers that a change occurred, so that they can update their UI.

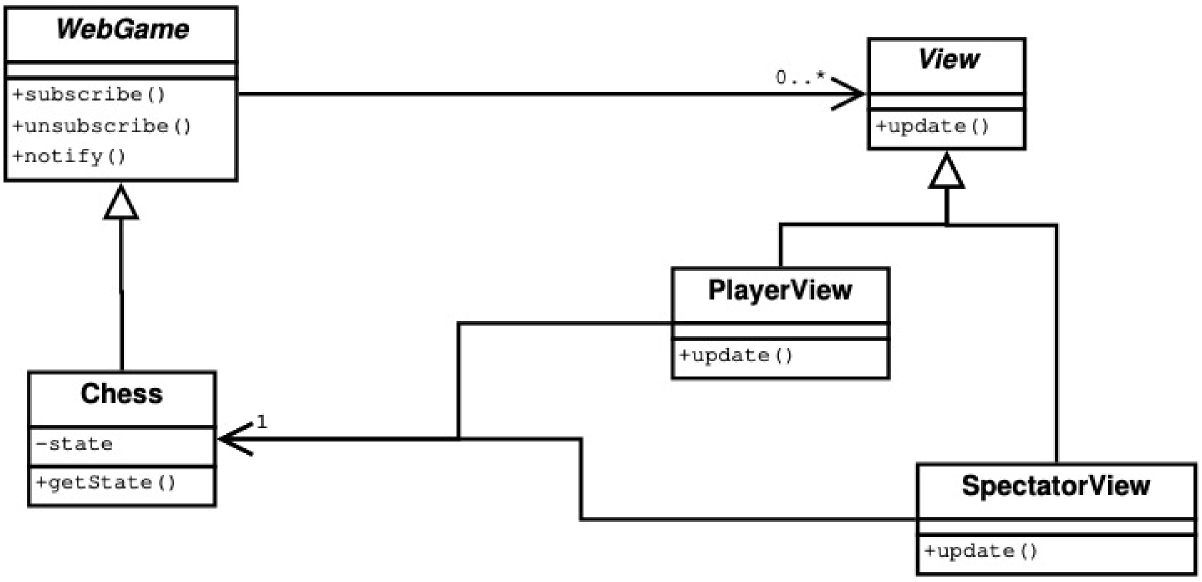


Figure-11.4: Example of Observer design pattern

11.2.3. Strategy

11.2.3.1. **What is the *Strategy* design pattern:**

- *Strategy* is a behavioural design pattern. It defines an abstract interface for a family of algorithms.
- It can also provide an interface for a set of behaviours, as we discussed in section 10.7.

11.2.3.2. **How is Strategy used:**

- Strategy provides a class interface at the base class class, and it encapsulates each type of algorithm inside a derived class.
- Program behaviour can change at runtime because each type of behaviour is encapsulated within an object. Switching from one type of derived class object to another at runtime, as needed, changes which concrete implementation of the algorithm is executed.
- The behaviour classes in section 10.7 are an example of the Strategy design pattern.

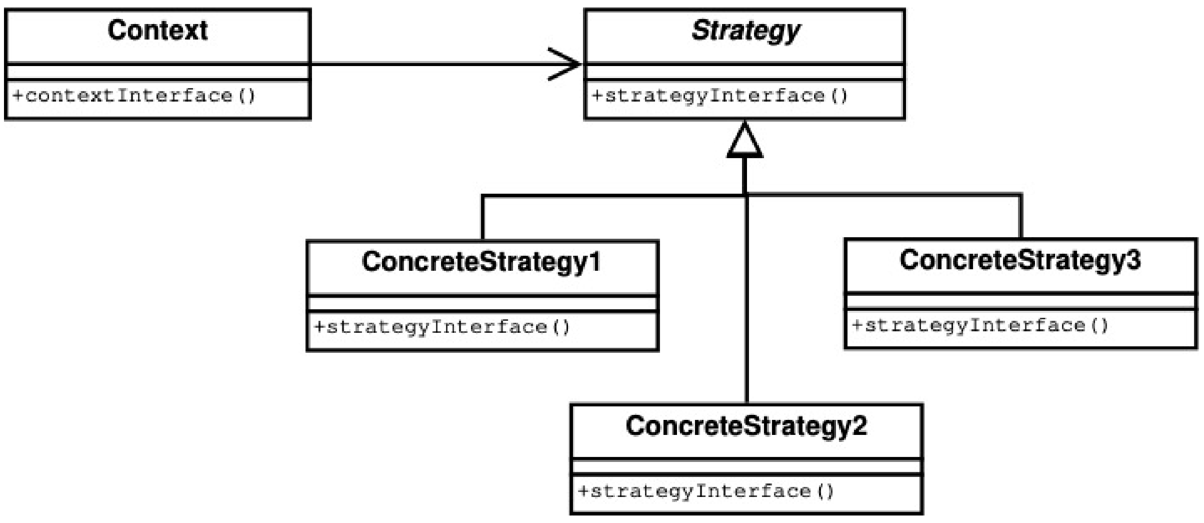


Figure-11.5: Strategy design pattern

11.2.3.3. **Structure of the Strategy design pattern:**

- Figure-11.5 illustrates the layout of the Strategy design pattern.
- A client class, possibly a control object, provides the context or conditions for deciding to switch algorithms or behaviours at runtime.
- An abstract base class provides the interface to the Strategy class hierarchy.
- Each type of concrete derived class overrides and implements the polymorphic operations defined in the base class interface.

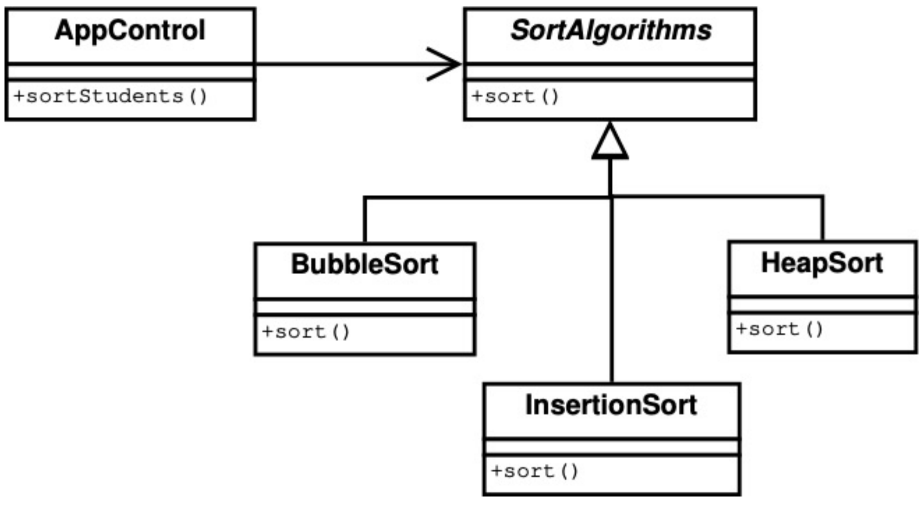


Figure-11.6: Example of Strategy design pattern

11.2.3.4. **Example of Strategy:**

- Figure-11.6 shows an example of the Strategy design pattern.
- In this example, an application control object decides which sorting algorithm to use and when to switch to a different one.
- The class hierarchy of sorting algorithms defines the sorting operation in the base class interface.
- Each type of algorithm overrides the sorting operation and implements it in its own way.

11.2.4. Factory

11.2.4.1. What is the *Factory* design pattern:

- *Factory* is a creational design pattern. It encapsulates the creation of derived class objects, which separates the object creation from the client class.
- The object creation is performed by a Factory object, and the created objects belong to a Product class hierarchy.

11.2.4.2. How is Factory used:

- A class hierarchy of products is defined. It consists of an abstract base class that provides a generalized interface, and concrete derived classes that override and provide different implementations for the base class interface operations.
- The client class delegates the creation of derived class objects to a Factory object.
- The Factory creates a new object and returns it to the client class. The client doesn't know which kind of derived object was created, but it does know that it's a type derived from the Product base class.
- The client class interacts with the new derived object through the polymorphic operations declared in the base class interface and overridden by the derived classes.

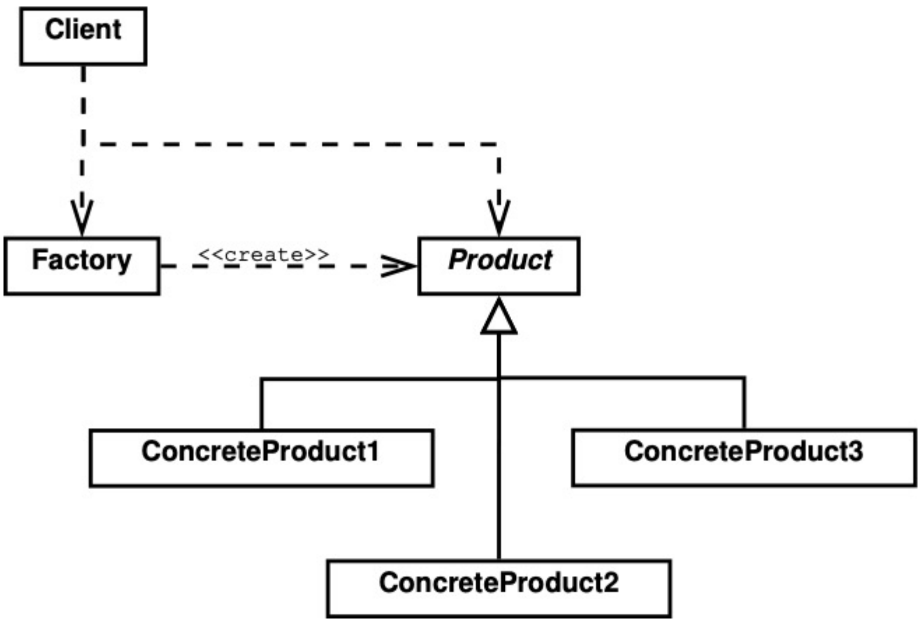


Figure-11.7: Factory design pattern

11.2.4.3. Structure of the Factory design pattern:

- Figure-11.7 illustrates the layout of the Factory design pattern.
- The client class interacts with both a Factory object and some concrete Product objects.
- The Factory object creates different types of products, all derived from the Product abstract base class, and returns these Product objects to the client.
- The client class manipulates the Product objects by calling the polymorphic operations declared in the Product base class interface and overridden in its concrete derived classes.

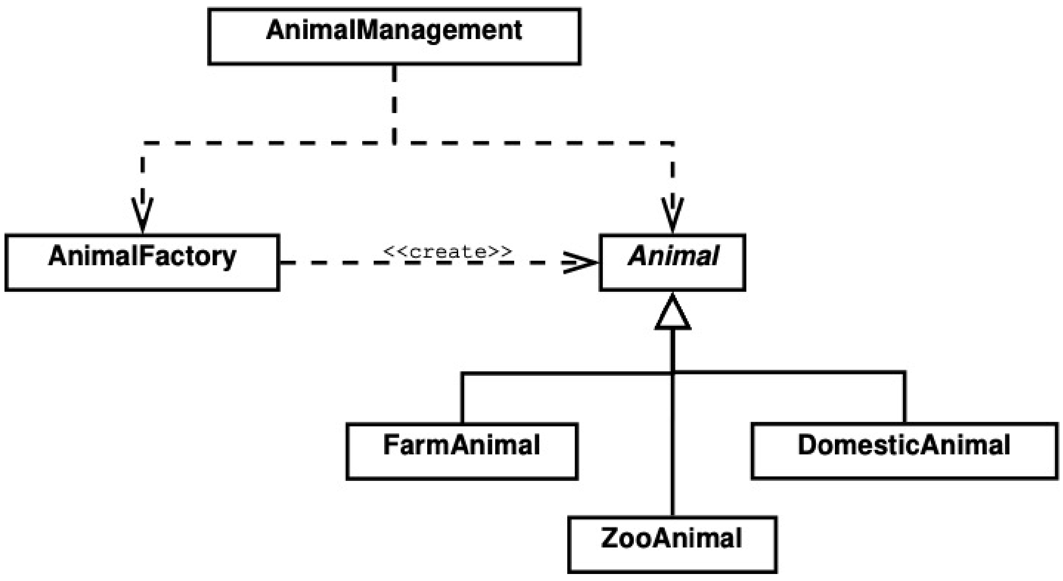


Figure-11.8: Example of Factory design pattern

11.2.4.4. **Example of Factory:**

- Figure-11.8 shows an example of the Factory design pattern.
- In this example, the client class is the control object for an animal management system.
- An animal factory creates concrete animal objects on behalf of the client class.
- The role of the animal base class is to provide a class interface with operations for the client class to manipulate any kind of animal.
- The client class has no knowledge of the specific type of animals that it's managing. It only knows that they are some kind of animal.

11.2.5. Anti-patterns

11.2.5.1. **What is an *anti-pattern*:**

- An *anti-pattern* is a common bad programming habit. It's a solution to a commonly occurring computing problem that appears to be effective but ends up violating the principles of good software engineering.
- There are too many of these to count.

11.2.5.2. **The God object:**

- One type of anti-pattern is called *the Blob* or *the God object*. It consists of one class that contains most of the program functionality.
- This is a risk when designing control objects that don't delegate sufficient functionality to the entity classes.
- It can also be a danger when implementing the Façade design pattern. It's easy for the Façade class to get overloaded with too much functionality.

11.2.5.3. **The Big Ball of Mud:**

- Another type of anti-pattern is called *the Big Ball of Mud*, where a program is implemented with no actual design or architecture.
- It's the modern-day equivalent of spaghetti code, an unstructured mess of code that was more common with older programming languages that provided fewer control structures.

Chapter 12

Overloading

Overloading allows a programmer to give multiple meanings to the same language construct. For example, in C++, we can overload functions and operators. This is a useful feature for ensuring that our code is streamlined, concise, and self-documenting.

In this chapter, we introduce both types of overloading in C++: function overloading and operator overloading. We discuss the different types of overloaded operators and the advantages of using this technique in our classes.

12.1. Function overloading

We discuss the basics of function overloading, in this case global functions. The same principles apply to member functions as well.

12.1.1. Concepts

12.1.1.1. **What is *overloading*:**

- In computing, *overloading* is the practice of providing multiple definitions for something. C++ allows the overloading of functions and operators.
- We overload a global function or a member function by reusing the same function name, but with different parameters.
- Operators can be overloaded to work with the classes that we define.

12.1.1.2. **What are *overloaded functions*:**

- *Overloaded functions* are functions that have the same name as each other, but with different functionality.
- Both global functions and member functions can be overloaded.

12.1.1.3. **Characteristics of overloaded functions:**

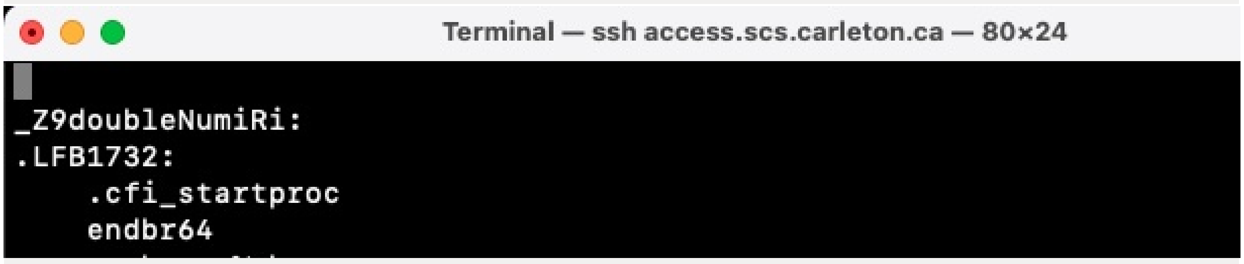
- Overloaded functions have the same name, but they must be different in the order or data types of their parameters. Otherwise, the compiler cannot resolve the ambiguity when the function is called.
- They must have a unique signature. It's insufficient for two overloaded functions of the same name to have a unique prototype.
- By convention, overloaded functions should be used only for functionally related tasks.

12.1.1.4. **How does function overloading work:**

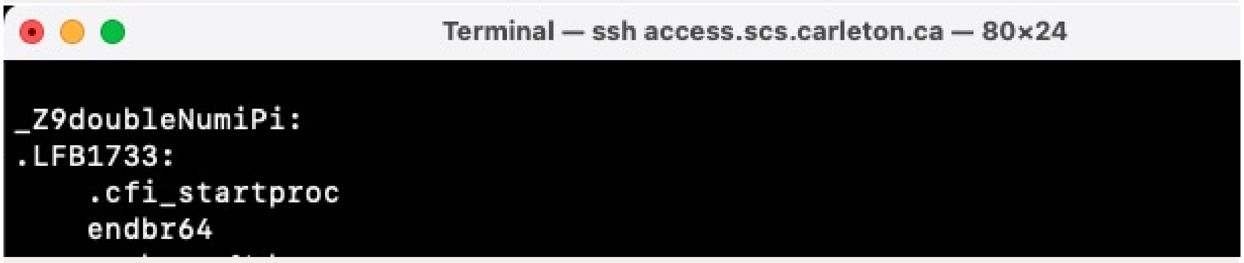
- The compiler converts our overloaded function name into a unique name for each function. This is done by *mangling* the function name.
- Mangling changes the function name to a combination of the original function name and the ordered parameter data types. The result is a unique name for each function.
- When the compiler processes a call to an overloaded function, it chooses which function should execute, based on the order of the parameters and their data types.

12.1.2. Coding example: Function overloading

```
 1 bool checkNum(int);
 2 void doubleNum(int, int&);
 3 void doubleNum(int, int*);

 5 int main()
 6 {
 7   bool inputOk = false;
 8   int  num, result1, result2;

10 /* User input loop is not shown */
11   doubleNum(num, result1);
12   doubleNum(num, &result2);
13 /* The rest of the program is not shown */
14 }
```



```
_Z9doubleNumiRi:
.LFB1732:
    .cfi_startproc
    endbr64
```



```
_Z9doubleNumiPi:
.LFB1733:
    .cfi_startproc
    endbr64
```

Program-12.1: Function overloading

Program purpose:

- Program-12.1 is identical to Program-4.2, so only the relevant portions are shown.
- In this program, the `doubleNum()` global function is overloaded with two versions. The corresponding assembly language shows how the function name is mangled to produce a unique name for each version.

Line 2:

- This line declares the `doubleNum()` function that takes an integer and an integer reference as parameters.
- We see from the first part of the assembly language how the function name has been mangled for this version of the function.

- If we ignore the `_Z9` prefix, we see that the function name has been changed to `doubleNumiRi`. The function name `doubleNum` is followed by the two parameter data types: `i` for integer and `Ri` for reference to an integer.

Line 3:

- This line declares the `doubleNum()` function that takes an integer and an integer pointer as parameters.
- We see from the second part of the assembly language how the function name has been mangled for this version of the function.
- Again ignoring the prefix, we see that the function name has been changed to `doubleNumiPi`. The function name `doubleNum` is followed by the two parameter data types: `i` for integer and `Pi` for pointer to an integer.

Line 11:

- This line calls the `doubleNum()` function with two integers as parameters (`num` and `result1`).
- Because the second parameter is an integer and not a pointer, the compiler decides that it's an integer passed into the function as a reference.
- So the compiler translates line 11 to a call to the version of `doubleNum()` that takes an integer and an integer reference as parameters.

Line 12:

- This line calls the `doubleNum()` function with an integer and the address of an integer variable as parameters (`num` and `&result2`).
- Because the second parameter is an address, the compiler sees that it's a pointer to an existing integer variable.
- So the compiler translates line 12 to a call to the version of `doubleNum()` that takes an integer and an integer pointer as parameters.

12.2. Operator overloading

We discuss the basics concepts of operator overloading, and its implementation in C++.

12.2.1. Concepts

12.2.1.1. **What is an *overloaded operator*:**

- An *overloaded operator* is an operator that works with our user-defined data types, specifically the classes that we design and implement.
- When we design a class in C++, we must decide which operators (if any) should be implemented to work with an object of this class as an operand.
- If we think of an operator as a function, then an overloaded operator is a function written by the class developer, instead of built into the programming language.
- We cannot rewrite the implementation of operators for primitive data types, like integers. We can only implement operators for our own classes.
- The operators we implement are considered "overloaded" because they are already defined for primitive data types. By providing an implementation for an operator's behaviour with our own data types, we are overloading that operator with additional definitions.

12.2.1.2. **Example of an overloaded operator:**

- In partial Program-12.2, assuming that the `Student` class is defined, what does line 2 mean? How do we compare two `Student` objects with the greater-than (>) operator?
- The beauty of overloaded operators is that the class developer decides what the greater-than operator does. We choose whether the operator compares student names, or student numbers, or GPAs, and we write the corresponding code.

```
1 Student matilda, joe;
2 if (matilda > joe) {
3   cout << "Matilda wins!" << endl;
4 }
5 else {
6   cout << "Joe wins!" << endl;
7 }
```

Program-12.2: Using overloaded operators

12.2.1.3. **Why overload an operator:**

- There are many good reasons why the C++ language provides this feature.
- Language consistency: We can use operators with primitive data types as operands, so it makes sense to do the same with instances of our own classes.
- Code readability: The use of recognizable operators makes the code more streamlined and easier to read, as long as the operators work in the same way they do for primitive types.
- Because we can: We should never ignore the "coolness" factor when designing a new feature. The authors of C++ figured out how to implement this feature, so why not provide it?

12.2.1.4. **Limitations to overloading operators:**

- We cannot change the implementation of operators for primitive data types.
- We cannot create new operators.
- We cannot change an operator’s arity, precedence, or associativity.
- We cannot overload non-overloadable operators. These include the dot, scope resolution, and conditional operators, and a few more.

12.2.1.5. **Overloaded operators in the `string` class:**

- The `string` class defined in the C++ standard library already provides many overloaded operators.
- These include: the assignment operator (=) that copies a literal or variable value into a string variable, relational operators that compare two strings, the subscript operator that accesses a single character in a string, stream insertion and extraction operators that perform standard I/O with strings, and many more.

12.2.2. Implicit and explicit overloading

12.2.2.1. **Implicitly overloaded operators:**

- Implicitly overloaded operators are operators that are provided for our classes by the C++ language, even if we do not implement them.
- They include the assignment operator (`=`), the address-of operator (`&`), and the sequencing operator (`,`).

12.2.2.2. **Explicitly overloaded operators:**

- Explicitly overloaded operators are operators that we implement for our classes ourselves. We as the class developer decide which operators make sense for our classes.
- Almost all operators can be overloaded, but there are exceptions.
- Each operator must be overloaded *separately*. There are no freebies.
- For example, if we implement the addition (`+`) and assignment operators (`=`), we do **not** automatically get the addition-assignment operator (`+=`). These are considered to be three *different* operators, although correct design principles dictate that their implementations should call each other, wherever possible.

NOTE: We discuss the role of the `this` object in overloaded operator implementations later in this chapter. However, it's worth nothing that this concept is nearly identical in Java. In a member function, the `this` object is the object *on which the member function is called*. In Java, the `this` object is an object reference, but in C++, it's an object **pointer**. To access the object from within a member function in C++, we must *dereference* the `this` pointer.

12.2.3. Approach to overloading

12.2.3.1. **How do we overload an operator:**

- In general, an operator is a *function*, and its operands are its *parameters*.
- Most operators can be implemented as either a global function or a member function, but not both at the same time. However, correct encapsulation principles dictate that we should implement an overloaded operator as a member function, wherever possible.
- When an operator is implemented as a member function:
  - the operator is called on *the left-hand side (LHS) operand*
  - so inside the overloaded operator function, the LHS operand is the `this` object
  - with a binary operator, the right-hand side (RHS) operand is passed into the function as a parameter
- If an overloaded operator is implemented as a global function, all operands are parameters.

12.2.3.2. **Overloaded operator function name:**

- The name of an overloaded operator function consists of the keyword `operator`, followed by the operator symbol.
- For example, the function name of an overloaded equality operator would be: `operator==`

12.2.3.3. **Overloaded operator function parameters:**

- The parameters of an overloaded operator function depend of the operator's arity.
- With a binary operator, an operator member function is called with the LHS operand as the `this` object and the RHS operand passed in to the function as a parameter.
- With a unary operator, an operator member function is called with the LHS operand as the `this` object and no parameters.

12.2.3.4. **Overloaded operator function example:**

- Assume that:
  - we have defined a `Time` class
  - the `Time` class implements a member function with the prototype: `bool operator==(Time&)`
  - our program declares two `Time` objects called `now` and `then`
  - the two objects are compared using the overloaded equality operator using the syntax: `(now == then)`
- Because the LHS operand (`now`) is an instance of the `Time` class, its `operator==()` member function is automatically called.
- Inside the member function, `now` is the `this` object because it's the LHS operand.
- The `then` object is passed in to the member function as a parameter.
- So the expression `(now == then)` is converted internally by the compiler to the function call: `now.operator==(then)`

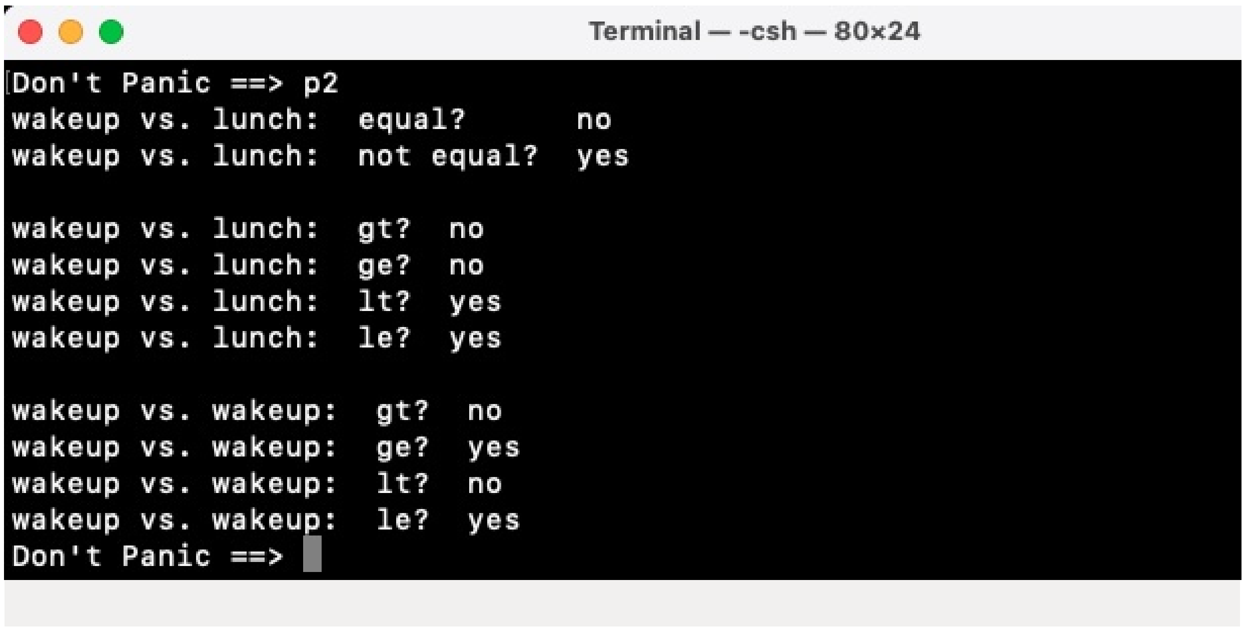
12.2.3.5. **Overloaded operator function return type:**

- It's imperative that we pay close attention to an overloaded operator's return type.
- Every built-in operator for every primitive data type in C, C++, and Java returns *a non-void value*. Therefore, our overloaded operator implementations should **never** have `void` as a return type.
- When unsure, think about how the same operator works with integers. If we implement an equality operator (`==`), we know that comparing two integers returns a boolean value. So our implementation of the same operator for our classes should also return a boolean.
- For the operators that don't return a boolean, we must make sure that we enable *cascading*, so that operators can be chained together within the same expression. We discuss cascading later in this chapter.

12.2.4. Coding example: Overloaded relational operators

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Time.h              *
 * * * * * * * * * * * * * * * * */
class Time
{
  public:
    Time(int=0, int=0, int=0);
    void setTime(int, int, int);
    void print() const;
    bool operator==(Time&) const;
    bool operator!=(Time&);
    bool operator<(Time&)  const;
    bool operator<=(Time&) const;
    bool operator>(Time&)  const;
    bool operator>=(Time&) const;
  private:
    int hours;
    int minutes;
    int seconds;
    int convertToSecs() const;
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Time.cc             *
 * * * * * * * * * * * * * * * * */
Time::Time(int h, int m, int s)
{
  setTime(h, m, s);
}

void Time::setTime(int h,int m,int s)
{
  hours   = ( h >= 0 && h < 24) ? h : 0;
  minutes = ( m >= 0 && m < 60) ? m : 0;
  seconds = ( s >= 0 && s < 60) ? s : 0;
}

int Time::convertToSecs() const
{
  return (hours*3600 + minutes*60 + seconds);
}

void Time::print() const
{
  cout<<setfill('0')<<setw(2)<<hours<<":"
      <<setfill('0')<<setw(2)<<minutes<<":"
      <<setfill('0')<<setw(2)<<seconds<<endl;
}

bool Time::operator==(Time& t) const
{
  return ( convertToSecs() == t.convertToSecs() );
}

```

```
bool Time::operator!=(Time& t)
{
  return !(*this == t);
}

bool Time::operator<(Time& t) const
{
  return ( convertToSecs() < t.convertToSecs() );
}

bool Time::operator<=(Time& t) const
{
  return ( (*this < t) || (*this == t) );
}

bool Time::operator>(Time& t) const
{
  return !(*this <= t);
}

bool Time::operator>=(Time& t) const
{
  return !(*this < t);
}

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc             *
 * * * * * * * * * * * * * * * * */
int main()
{
  Time wakeup(5,45,0);
  Time lunch(12,0,0);

  cout<<"wakeup vs. lunch:  equal?      "
      << ( (wakeup == lunch) ? "yes" : "no" ) <<endl;
  cout<<"wakeup vs. lunch:  not equal?  "
      << ( (wakeup != lunch) ? "yes" : "no" ) <<endl<<endl;
  cout<<"wakeup vs. lunch:  gt?  "
      << ( (wakeup >  lunch) ? "yes" : "no" ) <<endl;
  cout<<"wakeup vs. lunch:  ge?  "
      << ( (wakeup >= lunch) ? "yes" : "no" ) <<endl;
  cout<<"wakeup vs. lunch:  lt?  "
      << ( (wakeup <  lunch) ? "yes" : "no" ) <<endl;
  cout<<"wakeup vs. lunch:  le?  "
      << ( (wakeup <= lunch) ? "yes" : "no" ) <<endl<<endl;

  cout<<"wakeup vs. wakeup:  gt?  "
      << ( (wakeup >  wakeup) ? "yes" : "no" ) <<endl;
  cout<<"wakeup vs. wakeup:  ge?  "
      << ( (wakeup >= wakeup) ? "yes" : "no" ) <<endl;
  cout<<"wakeup vs. wakeup:  lt?  "
      << ( (wakeup <  wakeup) ? "yes" : "no" ) <<endl;
  cout<<"wakeup vs. wakeup:  le?  "
      << ( (wakeup <= wakeup) ? "yes" : "no" ) <<endl;
  return 0;
}
```



```
Don't Panic ==> p2
wakeup vs. lunch:  equal?     no
wakeup vs. lunch:  not equal?  yes

wakeup vs. lunch:  gt?  no
wakeup vs. lunch:  ge?  no
wakeup vs. lunch:  lt?  yes
wakeup vs. lunch:  le?  yes

wakeup vs. wakeup:  gt?  no
wakeup vs. wakeup:  ge?  yes
wakeup vs. wakeup:  lt?  no
wakeup vs. wakeup:  le?  yes
Don't Panic ==> 
```

Program-12.3: Overloaded relational operators

Program purpose:

- Program-12.3 demonstrates the implementation of several overloaded relational operators as member functions of the `Time` class.

Lines 4-21:

- These lines show the `Time` class definition.
- Lines 17-19 indicate that the class has three data members that represent the current time in a 24-hour clock: the hours, the minutes, and the seconds.
- Lines 7-9 declare some basic member functions: a default constructor, a setter, and a printing function.
- Line 20 is a private helper function that converts a `Time` object's hours, minutes and seconds to a total number of seconds and returns that value.
- Lines 10-15 declare the function prototypes for the six overloaded relational operators.
- On line 10, we see that the equality operator member function is called `operator==`, it takes a single `Time` object reference as parameter, and it returns a boolean, just as the equality operator does for integers.
- Lines 11-15 show the rest of the operator prototypes with their unique names. They all take a `Time` object reference as parameter, and they return a boolean.
- **Remember** that objects can access all the members of another object of the same class, even the private ones, as we saw in the discussion on copy constructors in section 3.7. We do **not** need to provide getters to access the members of the parameter object.

NOTE: Overloaded operator function prototypes are neither arbitrary nor negotiable. Object parameters are always passed as a reference. We do **not** pass objects by value because that makes a copy of the object, which is inefficient and a waste of computational resources. We do **not** pass objects using pointers because it results in very messy syntax that negates the entire point of overloading operators for improved code readability.

Lines 26-48:

- These lines contain the basic member function implementations for the `Time` class.
- Lines 26-29 show the default constructor.
- Lines 31-36 set the `Time` object to the given parameters after validating each one.

- Lines 38-41 convert the data members of the `Time` object to a total number of seconds.
- Lines 43-48 show how a `Time` object is printed out.

Lines 50-78:

- These lines contain the member function implementations of the `Time` class's overloaded relational operators.
- We know that relational operators are all binary operators. So in the body of each member function implementation, the LHS operand is the `this` object, and the RHS operand is the function parameter.
- Lines 50-53 implement the overloaded equality operator. Both LHS and RHS operands are converted to a total number of seconds, and the two resulting integers are compared for equality.
- The first call to `convertToSecs()` on line 52 converts the `this` object (the LHS operand) to seconds, and the second call does the same for the parameter (the RHS operand).
- In accordance with good design principles, we strive to reuse code by calling existing functions as much as possible within these operator implementations. With relational operators in general, it's a standard rule-of-thumb that only the equality and less-than operators must be written from scratch. All others can be implemented by calling combinations of these two.
- Lines 55-58 show the implementation of the inequality operator that simply calls the equality operator that's implemented on lines 50-53.
- Line 57 calls the equality operator with the same operands used for the call to the inequality operator. The dereferenced `this` object is again used as the LHS operand and the given parameter as the RHS.
- Lines 60-63 implement the less-than operator from scratch by converting both operands to seconds and using the integer less-than operator for comparison.
- Lines 65-68 show the implementation of the less-than-or-equal-to operator that calls both the less-than operator implemented on lines 60-63 and the equality operator implemented on lines 50-53.
- Lines 70-73 implement the greater-than operator that calls the less-than-or-equal-to operator that's implemented on lines 65-68.
- Lines 75-78 show the implementation of the greater-than-or-equal-to operator that calls the less-than operator that's implemented on lines 60-63.

PRO TIP: Notice how the function implementations on lines 50 through 78 **do not** contain any `if`-statements. When a function returns a boolean, it is *never* elegant to structure the code as follows: `if (<condition> == true) {return true;} else {return false;}`
It is always better to simply use: `return <condition>;`

For example, in the equality operator on line 52, the logic can be understood as "return true if both operands convert to the same total number of seconds, and false otherwise".

Lines 83-110:

- These lines show the implementation of the `main()` function.
- Lines 85-86 declare two `Time` objects with variable names `wakeup` and `lunch`. The wakeup time is initialized to 6:45:00 am, and lunch time to 12:00:00 pm.
- Lines 88-99 compare the two `Time` objects, `wakeup` and `lunch`, using the corresponding overloaded operators implemented on lines 50-78. Either "yes" or "no" is printed out, depending on the resulting comparison.
- Lines 101-108 compare the `wakeup` object with itself, using the same overloaded operators.
- We see from the program output that the overloaded operators return the correct results.

12.3. Cascading

We discuss the use of a non-void return type for overloaded operator implementations in C++, and how it is necessary for language consistency.

12.3.1. Concepts

12.3.1.1. **You must remember `this`:**

- As in the Java programming language, C++ uses the `this` keyword inside the implementation of a member function to refer to the object on which the function is called.
- The crucial difference in C++ is that `this` is a **pointer**, and not a reference. Any explicit use of `this` in the body of the member function *must* dereference the pointer in order to access the corresponding object.
- The `this` object is passed in to all non-static member functions as an *implicit parameter*.
- In each member function, the `this` object can be used either implicitly or explicitly.
- For example in Program-12.3:
  - the `setTime()` member function on lines 31-36 uses the `Time` data members *implicitly*
  - the function sets the values of `hours`, `minutes`, and `seconds` without specifying that these are data members of the `this` object; their membership in the `Time` class is *implied*, rather than explicitly stated
  - the corresponding explicit use of the `this` object would have lines 33-35 setting the data members using `this->hours`, `this->minutes`, and `this->seconds`, respectively
- Experienced programmers only use the `this` object explicitly in portions of the program where it's needed for better clarity or readability. It's normal practice to use `this` implicitly most of the time.

12.3.1.2. **What is *cascading*:**

- *Cascading* is the chaining together of multiple operations within a single expression.
- For example in Program-2.1, recall that line 8 had four operators combined together in the same statement: `z = y + 2 * x - 3;`
- In that program, all three variables `x`, `y`, and `z` were declared as integers, so the built-in operators were used.
- But what if `x`, `y`, and `z` were all `Time` objects? We would need to implement the four overloaded operators ourselves, but we would also need to think very carefully about their return values. If the operators returned `void`, we would only be able to use one of them in an expression, because `void` is not a valid operand to any operator.
- In order to use multiple operators in the same expression, we must enable cascading in our overloaded operator implementations.

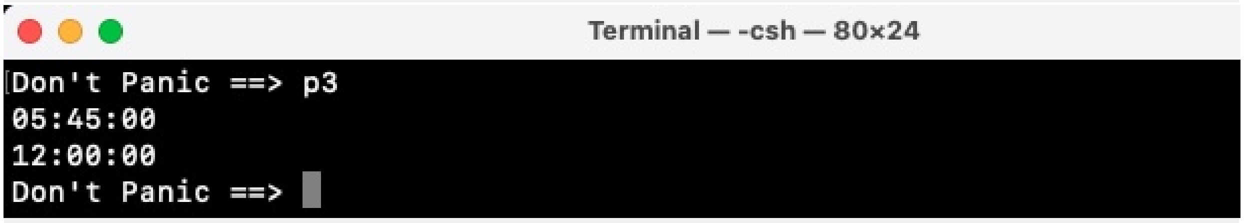
12.3.1.3. **How does cascading work:**

- We enable cascading in our member functions by having them return the correct value.
- Every overloaded operator that is implemented as a member function uses the `this` object as its LHS operand. So if we want to chain together multiple operators in the same expression, we need to use the `this` object as the return value.
- If a member function does its work and returns the `this` object, then we can call another member function on that same returned object within the same expression.
- We can use this technique in any member function, including overloaded operators.

12.3.2. Coding example: Cascading member functions

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Time.h             *
 * * * * * * * * * * * * * * * * */
class Time
{
  public:
    Time(int=0, int=0, int=0);
    void print() const;
    Time& setTime(int, int, int);
    Time& setHours(int);
    Time& setMinutes(int);
    Time& setSeconds(int);
  private:
    int hours;
    int minutes;
    int seconds;
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Time.cc            *
 * * * * * * * * * * * * * * * * */
Time::Time(int h, int m, int s)
{
  setTime(h, m, s);
}

Time& Time::setTime(int h,int m,int s)
{
  setHours(h);
  setMinutes(m);
  setSeconds(s);
  return *this;
}

Time& Time::setHours(int h)
{
  hours = ( h >= 0 && h < 24) ? h : 0;
  return *this;
}

Time& Time::setMinutes(int m)
{
  minutes = ( m >= 0 && m < 60) ? m : 0;
  return *this;
}

Time& Time::setSeconds(int s)
{
  seconds = ( s >= 0 && s < 60) ? s : 0;
  return *this;
}
/*  Time::print() function is not shown  */

```

```
54 /* * * * * * * * * * * * * * * * *
55  * Filename:  main.cc             *
56  * * * * * * * * * * * * * * * * */
57 int main()
58 {
59   Time wakeup, lunch;

61   wakeup.setHours(5).setMinutes(45).setSeconds(0);
62   wakeup.print();
63   lunch.setTime(12,0,0).print();

65   return 0;
66 }
```



```
Don't Panic ==> p3
05:45:00
12:00:00
Don't Panic ==>
```

Program-12.4: Cascading member functions

Program purpose:

- Program-12.4 demonstrates how member functions are implemented to enable cascading.

Lines 4-17:

- These lines show the `Time` class definition.
- Lines 9-12 declare the four member functions that enable cascading: `setTime()`, `setHours()`, `setMinutes()`, and `setSeconds()`.
- We know that cascading is enabled because each of these member functions returns a `Time` object on which other operations can be performed.

Lines 27-51:

- These lines contain the implementation of the four setter member functions that enable cascading.
- Lines 32, 38, 44, and 50 show that, to enable cascading, we must return a value on which other operations can be performed. By returning the dereferenced `this` object, these lines allow another member function to be called on the exact same object.
- It's crucial that the `this` object is returned as a `Time` *reference*, and *not* an object. If another member function is called, it must be on the **same** `Time` object, and not simply a copy of it.

Lines 57-66:

- These lines show the implementation of the `main()` function.
- Line 61 demonstrates multiple member function calls on the *same object*, stored in the `wakeup` variable.
- Because the dot operator has left-to-right associativity, the first member function to execute on line 61 is `setHours()`. It sets the `hours` of the `wakeup` object to 5 and returns the updated object. It cannot return a copy, otherwise the changes would occur on different `Time` objects.
- The next member function to execute on line 61 is `setMinutes()`. It takes the `wakeup` object with the `hours` previously set to 5, it sets the same object's `minutes` to 45, and then it returns the newly updated object.

- Next on line 61 is the call to `setSeconds()` on the same object with the updated `hours` and `minutes`.
- If cascading was not enabled and these member functions returned nothing (`void`), line 61 would have to be split into multiple statements.
- Line 63 demonstrates how cascading is used to call the `setTime()` and `print()` member functions in the same statement.

12.4. Operators as functions

We discuss the implementation of overloaded operators as global or member functions.

12.4.1. Concepts

12.4.1.1. **Types of functions for overloading operators:**

- Overloaded operators can be implemented as either member functions or global functions.
- They cannot be implemented as both because the compiler considers them equivalent.
- If a class developer provides both a member function and a global function for the same class and operator, the compiler will be unable to resolve the ambiguity when the operator is invoked, and compilation will fail.

12.4.1.2. **Overloading an operator as a *member function*:**

- As we saw in Program-12.3 with operators implemented as member functions, the LHS operand is used as the `this` object inside the member function implementation.
- For binary operators, the member function takes the RHS operand as a parameter.
- For unary operators, the member function takes no parameters.
- It is **good** software engineering to implement overloaded operators as member functions, wherever possible. However, as we discuss shortly, there are operators that cannot be implemented as member functions.

12.4.1.3. **Overloading an operator as a *global function*:**

- Global functions have no `this` object, so all operands are passed in as parameters.
- For binary operators, the global function takes both operands as parameters.
- For unary operators, the global function takes the single operand as parameter.
- In some cases, the global function must be made a friend function of a class in order to access the class members.
- For good software engineering, the use of friendship in C++ should be strictly limited to cases where operators *cannot* be overloaded as member functions.

12.4.1.4. **Operators that *must* be implemented as member functions:**

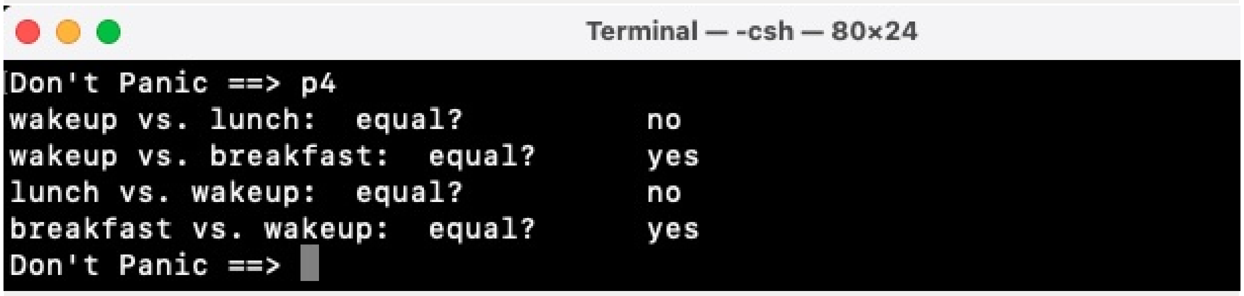
- C++ dictates that some operators can *only* be overloaded as member functions.
- These include the typecast (`()`), subscript (`[]`), and arrow (`->`) operators.
- All overloaded operator member function implementations must enable cascading, in order for our code to meet the minimum expectations of our class users.

12.4.1.5. Operators that *must* be implemented as global functions:

- C++ dictates that some operators can *only* be overloaded as global functions.
- There are two cases where we *cannot* implement an operator as a member function:
  - when the LHS operand is a primitive data type, and
  - when the LHS operand is an instance of a library class
- The first case, where the LHS operand is a primitive data type, can occur when we need to enable commutativity, as we see shortly in Program-12.5. Since primitive data types are not classes, they cannot have member functions.
- The second case, where the LHS is an instance of a library class, includes the stream insertion (<<) and stream extraction (>>) operators, as we discuss later in this chapter. Since we do not have access to the C++ standard library source code, we cannot modify its classes to add new member functions.
- All overloaded operator global function implementations must enable cascading.

12.4.2. Coding example: Operators as functions

```cpp
/* * * * * * * * * * * * * * * * *
 * Filename:  Time.h              *
 * * * * * * * * * * * * * * * * */
class Time
{
  public:
    Time(int=0, int=0, int=0);
    void setTime(int, int, int);
    void print() const;
    bool operator==(int) const;
  private:
    int hours;
    int minutes;
    int seconds;
    int convertToSecs() const;
};
bool operator==(int, Time&);

/* * * * * * * * * * * * * * * * *
 * Filename:  Time.cc             *
 * * * * * * * * * * * * * * * * */
Time::Time(int h, int m, int s)
{
  setTime(h, m, s);
}

bool Time::operator==(int s) const
{
  return ( convertToSecs() == s );
}

bool operator==(int s, Time& t)
{
  return ( t == s );
}
/* The basic Time member functions are not shown */

/* * * * * * * * * * * * * * * * * *
 * Filename:  main.cc              *
 * * * * * * * * * * * * * * * * * */
int main()
{
  Time wakeup(5,45);
  int  lunch     = 43200;  // noon
  int  breakfast = 20700;  // 5:45

  cout<<"wakeup vs. lunch:  equal?          "
      << ( (wakeup == lunch) ? "yes" : "no" ) <<endl;
  cout<<"wakeup vs. breakfast:  equal?      "
      << ( (wakeup == breakfast) ? "yes" : "no" ) <<endl;

  cout<<"lunch vs. wakeup:  equal?          "
      << ( (lunch == wakeup) ? "yes" : "no" ) <<endl;
  cout<<"breakfast vs. wakeup:  equal?      "
      << ( (breakfast == wakeup) ? "yes" : "no" ) <<endl;
  return 0;
}
```



```
Don't Panic ==> p4
wakeup vs. lunch:  equal?          no
wakeup vs. breakfast:  equal?      yes
lunch vs. wakeup:  equal?          no
breakfast vs. wakeup:  equal?      yes
Don't Panic ==> 
```

Program-12.5: Operators as functions

Program purpose:

- Program-12.5 demonstrates some operators overloaded as member functions, and others as global functions.
- In this program, we implement two versions of the overloaded equality (`==`) operator that compares a `Time` object with an integer corresponding to a total number of seconds. The goal is to determine whether or not the `Time` object, when converted to the equivalent total number of seconds, is equal to the integer operand.
- The first version of the equality operator takes the `Time` object as the LHS operand and the integer as the RHS. Because the LHS operand is an object, we implement this functionality as a member function.
- The second version of the equality operator takes the integer as the LHS operand and the `Time` object as the RHS. Because the LHS operand is a primitive data type, this functionality cannot be implemented as a member function, so we use a global function instead.
- We implement both versions of the operator in order to provide commutativity. If `t` is a `Time` object and `s` is an integer denoting a total number of seconds, a programmer using our `Time` class should be able to compare either `(t == s)` or `(s == t)` and obtain the same result.
- Some basic member function implementations of the `Time` class are not shown, but they are identical to Program-12.3.

Lines 4-17:

- Lines 4-16 show the `Time` class definition.
- On line 10, we declare the function prototype for the version of the equality operator that takes a `Time` object as the LHS operand and an integer as the RHS.
- Line 17 declares the global function prototype for the version of the equality operator that takes an integer as the LHS operand and a `Time` object as the RHS. This line is necessary as a forward reference to inform the compiler of a global function that may be called in the code before the compiler can process its implementation.
- Strictly speaking, line 17 is not required to be positioned inside the `Time` class header file. But since this global function is closely related to the `Time` class, it is convention to place it in the same file as the class definition.

Lines 27-30:

- These lines show the implementation of the first version of the overloaded equality operator.
- Line 27 indicates that this is a member function by using the scope resolution operator to bind the function name to the `Time` class.
- Line 29 converts the `this` object, as the LHS operand, to a total number of seconds. It then uses the built-in integer equality operator to compare that number of seconds to the RHS operand found in the given parameter.

Lines 32-35:

- These lines show the implementation of the second overloaded equality operator.
- Line 32 indicates that this is a global function by *not* using the scope resolution operator to bind the function name to any class.
- We see that both the integer and the `Time` object operands are passed in as parameters.
- Line 34 uses correct design principles by calling the version of the equality operator that we already implemented. By simply switching the order of the operands, line 34 calls the first version of the equality operator that's implemented on lines 27-30.
- In principle, this global function implementation is not required to be positioned in the `Time` class source file. But since the function is closely related to the `Time` class, it is convention to place it in the same file as the class's member function implementations.

Lines 41-57:

- These lines show the implementation of the `main()` function.
- Line 43 declares a `Time` object called `wakeup`, which is set to 5:45 am.
- Lines 44-45 declare two integers that represent a total number of seconds. The `lunch` variable is set to the seconds equivalent of 12:00 pm (noon), and `breakfast` is initialized to the total seconds corresponding to 5:45 am.
- Line 48 uses the equality operator to compare the `Time` object on the LHS with the `lunch` integer on the RHS. The compiler converts this to a call to the member function implemented on lines 27-30.
- Line 50 also compares the `Time` object on the LHS with an integer on the RHS, so this is converted to a call to the same member function.
- Line 53 uses the equality operator to compare the `lunch` integer on the LHS with the `Time` object on the RHS. The compiler converts this to a call to the global function implemented on lines 32-35.
- Line 55 also compares an integer on the LHS with the `Time` object on the RHS, so this is converted to a call to the same global function.

12.5. Overloaded stream operators

We discuss the implementation of the stream insertion and extraction operators in C++.

12.5.1. Concepts

12.5.1.1. **What are the *stream operators*:**

- The *stream insertion operator* (`<<`) adds a sequence of bytes to an output stream. The output stream can be one of the standard streams or an output file.
- The *stream extraction operator* (`>>`) reads a sequence of bytes from an input stream. The input stream can be the standard input stream or an input file.
- Both are binary operators, and they are already overloaded for primitive data types and most library classes, including the `string` class.
- To use the stream insertion and extraction operators with instances of our own classes, we must overload and implement them as *global functions*.
- When the stream operators are implemented, the LHS operand is always an output stream for insertion and an input stream for extraction. We cannot implement them as member functions, because we are not permitted to modify the `ostream` and `istream` library classes.
- Both stream operator implementations **must** enable cascading. Our class users must be able to print out or read in any combination of values and objects within the same expression, exactly the way they can with primitive data types.

12.5.1.2. **Implementation of the stream insertion operator:**

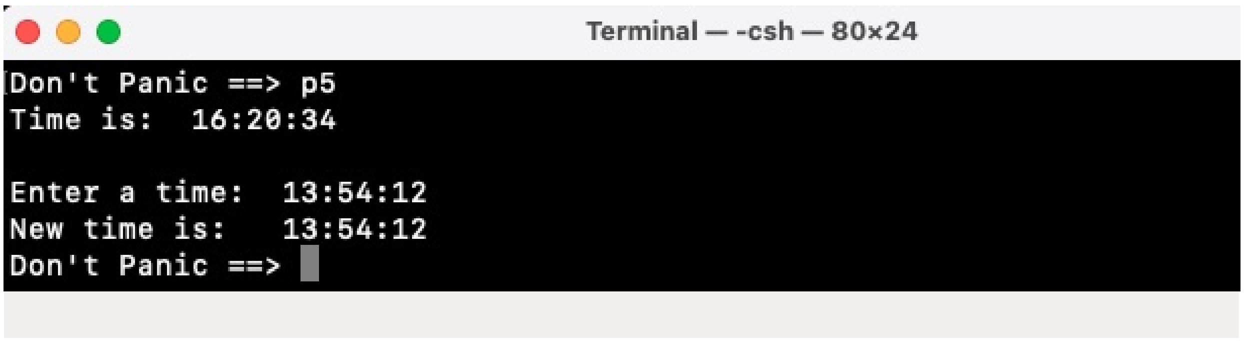
- When we use the stream insertion operator, the LHS operand is the `ostream` object where we wish to send our output, for example the `cout` object for standard output.
- The LHS operand is passed into the global function implementation of the operator as the first parameter, as an `ostream` object reference.
- The RHS operand is the object to print out, and it's passed into the operator implementation as the second parameter, as an object reference.
- It may be necessary to make the operator implementation a friend function in the class of the object to be output, if the function needs access to the object's private and/or protected members. In this case, it's better to use friendship than to violate the principle of least privilege by providing multiple getters.

12.5.1.3. **Implementation of the stream extraction operator:**

- When we use the stream extraction operator, the LHS operand is the `istream` object from which we wish to read our input, for example the `cin` object for standard input.
- The LHS operand is passed into the global function implementation of the operator as the first parameter, as an `istream` object reference.
- The RHS operand is the object into which the input is stored, and it's passed into the operator implementation as the second parameter, as an object reference.
- It may be necessary to make the operator implementation a friend function in the class of the object to be input.

12.5.2. Coding example: Overloaded stream operators

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Time.h               *
 * * * * * * * * * * * * * * * * */
class Time
{
  friend ostream& operator<<(ostream&, Time&);
  friend istream& operator>>(istream&, Time&);
  public:
    Time(int=0, int=0, int=0);
    void setTime(int, int, int);
  private:
    int hours;
    int minutes;
    int seconds;
    int convertToSecs() const;
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Time.cc              *
 * * * * * * * * * * * * * * * * */
/* The basic Time member functions are not shown */

ostream& operator<<(ostream& output, Time& t)
{
  output<<setfill('0')<<setw(2)<<t.hours<<":"
        <<setfill('0')<<setw(2)<<t.minutes<<":"
        <<setfill('0')<<setw(2)<<t.seconds;
  return output;
}

istream& operator>>(istream& input, Time& t)
{
  int h, m, s;
  input>>setw(2)>>h;
  input.ignore();
  input>>setw(2)>>m;
  input.ignore();
  input>>setw(2)>>s;
  t.setTime(h,m,s);
  return input;
}

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc              *
 * * * * * * * * * * * * * * * * */
int main()
{
  Time theTime(16, 20, 34);
  cout << "Time is:  " << theTime << endl << endl;

  cout << "Enter a time:  ";
  cin >> theTime;
  cout << "New time is:   "<< theTime << endl;
  return 0;
}
```



```
Don't Panic ==> p5
Time is:  16:20:34

Enter a time:  13:54:12
New time is:   13:54:12
Don't Panic ==> 
```

Program-12.6: Overloaded stream operators

Program purpose:

- Program-12.6 demonstrates how the overloaded stream insertion and extraction operators are implemented for the `Time` class.
- Some basic member function implementations of the `Time` class are not shown, but they are identical to Program-12.3.

Lines 4-16:

- These lines contain the `Time` class definition.
- Lines 6-7 show that the `Time` class grants friendship to the two stream operator global functions. This is necessary for these functions to access a `Time` object's data members.
- Lines 6-7 also serve as forward references to inform the compiler of these global functions and their prototypes.
- The `print()` member function that was present in previous implementations of the `Time` class has been removed. Moving forward, all future coding examples with the `Time` class use the stream insertion operator to print out a `Time` object.

Lines 23-29:

- These lines show the implementation of the stream insertion operator global function.
- Line 23 declares the LHS operand as an `ostream` object reference passed in as a parameter called `output`, and the RHS operand as a `Time` object reference called `t`. We also see that the return type enables cascading.
- Lines 25-27 are nearly identical to the previous implementation of the `print()` member function. But in this case, the hours, minutes, and seconds printed belong to the `t` parameter.
- Line 25 also shows that we do **not** print out the object directly to the `cout` object. Our implementation must be generalized for use with *any* output stream, including output files. The exact output stream is provided in the first parameter.
- Line 28 shows how cascading is enabled. Returning the same output stream object (as an `ostream` reference) allows a subsequent call for stream insertion within the same expression.

Lines 31-41:

- These lines show the implementation of the stream extraction operator global function.
- Line 31 declares the LHS operand as an `istream` object reference passed in as a parameter called `input`, and the RHS operand as a `Time` object reference called `t`. We also see that the return type enables cascading.
- Line 33 declares three temporary variables where we store the user input before the `Time` object `t` is initialized with these values.
- Lines 34-38 show that we do **not** read the object directly from the `cin` object. Our implementation must be generalized for use with *any* input stream, including input files. The exact input stream is provided in the first parameter.

- Lines 34-38 use the C++ `iomanip` library to assist in parsing the user input.
- Line 34 reads two digits for the hours and stores the entered value in variable `h`. Then line 35 skips over the colon (`:`) that the user enters to separate the hours and minutes.
- Line 36 reads the next two digits for the minutes and stores the entered value in variable `m`. Then line 37 skips over the colon entered to separate the minutes and seconds.
- Line 38 reads the next two digits for the seconds and stores the entered value in variable `s`.
- Line 39 calls the `Time` class's `setTime()` member function to set the data members of the parameter object `t` to the user-entered values.
- Line 40 shows how cascading is enabled. Returning the same input stream object allows a subsequent call for stream extraction within the same expression.

Lines 48-49:

- These lines show how the stream insertion operator is called from the `main()` function.
- Line 48 declares and initializes a `Time` object called `theTime`.
- Line 49 prints out the `theTime` variable using the stream insertion operator.
- We see that line 49 actually uses the stream insertion operator four times. Only the second call uses the implementation on lines 23-29.
- The first use of the stream insertion operator on line 49 is to print to `cout` the literal string `"Time is:"`. This calls the overloaded operator that's implemented for strings, which returns the same `cout` object to facilitate cascading.
- The second use of stream insertion on that line is to print the `theTime` object to the `cout` object returned from the first use of stream insertion. This second use of stream insertion calls the operator we implemented on lines 23-29.
- Because our operator implementation correctly enables cascading, the third use of stream insertion on line 49 allows a new line to be printed out after the `Time` object.

Line 52-53:

- Line 52 shows how the stream extraction operator is called from the `main()` function.
- This line reads user input into the `theTime` variable by calling the stream extraction operator that's implemented on lines 31-41.
- Lines 34-38 read the hours, minutes, and seconds into local variables.
- Then line 39 sets the data members of `main()`'s `theTime` object to these user-entered values.
- Line 53 again uses the stream insertion operator that we implemented on lines 23-29 to print out the correct value of the `theTime` object, as we see from the program output.

12.6. Overloaded arithmetic operators

We discuss some issues that arise from the implementation of overloaded arithmetic operators.

12.6.1. Concepts

12.6.1.1. **Characteristics of arithmetic operators:**

- In previous coding examples, we overloaded the relational operators, which are fairly straightforward to implement.
- Arithmetic operators, however, require more complex thinking in terms of implementation.
- For example, how do we implement the addition operator (+) for the `Time` class? What about addition-assignment (+=)? It makes sense that the two implementations have some commonalities. Can they share code, or can one call the other?

12.6.1.2. **The addition-assignment operator:**

- The addition-assignment operator (`+=`) is, perhaps surprisingly, a simpler implementation than addition (`+`).
- Let's consider how addition-assignment works first for integers, and then for `Time` objects.

12.6.1.3. **The addition-assignment operator for *integers*:**

- Assume that `x` and `y` are both declared as integer variables and initialized to valid values.
- The expression `(x += y)` has variable `x` as the LHS operand and `y` as the RHS.
- To write the code for this operator, the following logic applies:
  - use integer addition to add together the current values of `x` and `y`, and
  - store the resulting sum into variable `x`
- This implementation requires no intermediate values or temporary variables.

12.6.1.4. **The addition-assignment operator for *`Time` objects*:**

- Let's consider how we can apply the integer addition-assignment logic to `Time` objects.
- Assume that `x` and `y` are both declared as `Time` objects and initialized to valid values.
- The expression `(x += y)` has `Time` object `x` as the LHS operand and `y` as the RHS.
- If we implement this overloaded addition-assignment operator as a member function of the `Time` class, `x` becomes the `this` object, and `y` is passed in as a parameter.
- The logic inside the member function works as follows:
  - convert `x` to a corresponding total number of seconds, and do the same for `y`
  - use integer addition to add together the total number of seconds for both `x` and `y`
  - convert the resulting sum into an equivalent number of hours, minutes, and seconds
  - replace the current values in `Time` object `x` with the values computed in the previous step
- Again, this implementation requires no intermediate values or temporary objects.

12.6.1.5. **The addition operator:**

- The addition operator (`+`) is more complicated than it first seems.
- Let's consider how addition works first for integers, and then for `Time` objects.

12.6.1.6. **The addition operator for *integers*:**

- Assume that `x`, `y`, and `z` are all declared as integer variables and initialized to valid values.
- The expression `(z = x + y)` actually contains two operators: assignment and addition.
- It's crucial to note that both operators are implemented *separately*, and their implementations are independent from each other.
- Based on operator precedence, the addition operator `(x + y)` is evaluated first.
- The expression `(x + y)` has variable `x` as the LHS operand and `y` as the RHS.
- To write the code for this operator, the following logic applies:
  - use integer addition to add together the current values of `x` and `y`, and
  - return the resulting sum as the addition operator's return value
- It's important to note that neither variable `x` nor `y` is modified by the addition operation.

- Also, the addition operator is not concerned with how the resulting sum is used. That value is simply returned from the operation, so that it can be used as operand to another operator.
- In the expression `(z = x + y)`, the assignment operator (`=`) is called with variable `z` as the LHS operand and the resulting sum of `x` and `y` as the RHS.

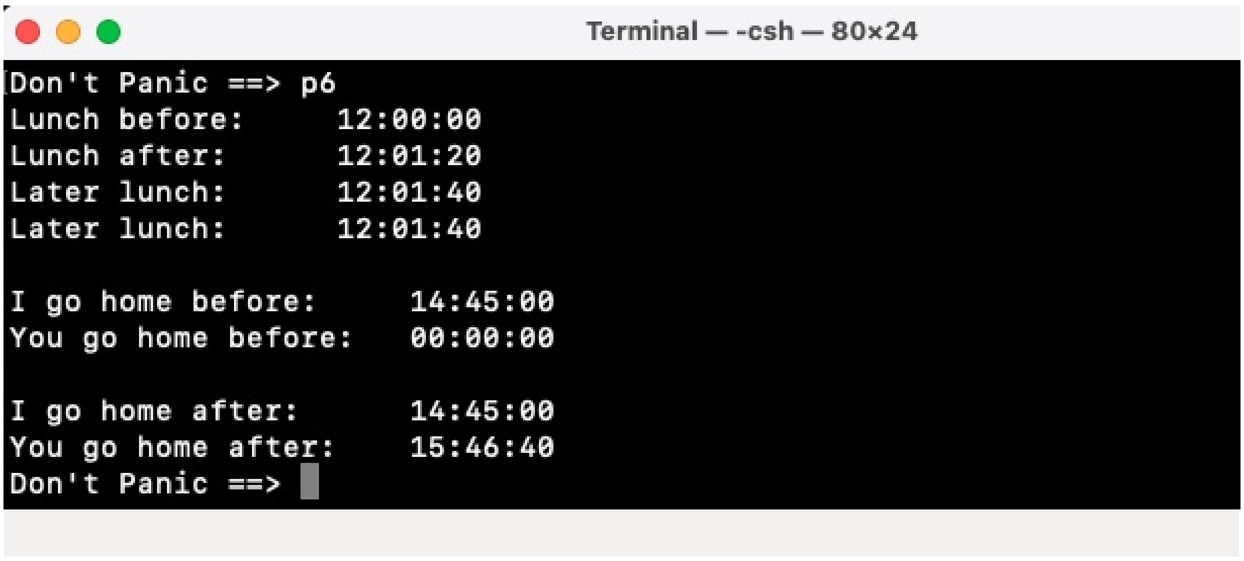
12.6.1.7. The addition operator for *`Time` objects*:

- Assume that `x`, `y`, and `z` are all declared as `Time` objects and initialized to valid values.
- Given the expression `(z = x + y)`, the addition operator `(x + y)` is evaluated first. It has `Time` object `x` as the LHS operand and `y` as the RHS.
- If we implement this overloaded addition operator as a member function of the `Time` class, `x` becomes the `this` object, and `y` is passed in as a parameter.
- The logic inside the member function works as follows:
  - convert `x` to a corresponding total number of seconds, and do the same for `y`
  - use integer addition to add together the total number of seconds for both `x` and `y`
  - convert the resulting sum into an equivalent number of hours, minutes, and seconds
  - store these values into *a temporary locally declared `Time` object*
  - return this temporary `Time` object *by value* as the addition operator's return value
- It's important to note that neither `Time` object `x` nor `y` is modified by the addition operation. The implementation requires a new `Time` object because an addition operation *does not modify its operands*. It simply returns something new that didn't exist before.
- The new `Time` object **must** be returned by value. We can **never** return a pointer into a stack frame that has been popped off the function call stack, because the corresponding memory is no longer allocated.
- We also note that the first three steps of the above logic are identical to the addition-assignment operation. Therefore, a correct implementation of the addition operator must *call the addition-assignment operator* to do a portion of the work.
- In the expression `(z = x + y)`, the assignment operator (`=`) is called with `Time` object `z` as the LHS operand and the new `Time` object returned from the addition of `x` and `y` as the RHS.

12.6.2. Coding example: Overloaded arithmetic operators

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Time.h              *
 * * * * * * * * * * * * * * * * */
class Time
{
  friend ostream& operator<<(ostream&, Time&);
  public:
    Time(int=0, int=0, int=0);
    void setTime(int, int, int);
    Time& operator=(const Time&);
    Time& operator+=(const int);
    Time  operator+(const int);
  private:
    int   hours;
    int   minutes;
    int   seconds;
    int   convertToSecs() const;
    void  setTime(int);
};
```

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Time.cc            *
 * * * * * * * * * * * * * * * * */
/* The basic Time member functions are not shown */

void Time::setTime(int s)
{
  hours   = s / 3600;
  minutes = (s % 3600) / 60;
  seconds = (s % 3600) % 60;
}

Time& Time::operator=(const Time& t)
{
  hours   = t.hours;
  minutes = t.minutes;
  seconds = t.seconds;
  return *this;
}

Time& Time::operator+=(const int s)
{
  setTime(convertToSecs() + s);
  return *this;
}

Time Time::operator+(const int s)
{
  Time t;
  t = *this;
  t += s;
  return t;
}

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc            *
 * * * * * * * * * * * * * * * * */
int main()
{
  Time lunch(12);
  cout << "Lunch before:      " << lunch << endl;
  lunch += 80;
  cout << "Lunch after:       " << lunch << endl;
  cout << "Later lunch:       " << (lunch += 20) << endl;
  cout << "Later lunch:       " << lunch << endl;

  Time iGoHome(14,45);
  Time youGoHome;
  cout << endl << "I go home before:      " << iGoHome   << endl;
  cout <<         "You go home before:    " << youGoHome << endl;

  youGoHome = iGoHome + 3700;
  cout << endl << "I go home after:       " << iGoHome   << endl;
  cout <<         "You go home after:     " << youGoHome << endl;
  return 0;
}
```



```
Don't Panic ==> p6
Lunch before:      12:00:00
Lunch after:       12:01:20
Later lunch:       12:01:40
Later lunch:       12:01:40

I go home before:      14:45:00
You go home before:    00:00:00

I go home after:       14:45:00
You go home after:     15:46:40
Don't Panic ==> 
```

Program-12.7: Overloaded arithmetic operators

Program purpose:

- Program-12.7 demonstrates how the overloaded assignment, addition-assignment, and addition operators are implemented as member functions of the `Time` class.
- The assignment (=) operator copies into the LHS operand every member of the `Time` object provided as the RHS operand.
- The addition-assignment (+=) operator adds to the LHS operand the total number of seconds in the RHS operand, and it updates the LHS operand with the resulting sum.
- The addition (+) operator adds to the LHS operand the total number of seconds in the RHS operand, and it returns a new `Time` object initialized with the resulting sum.
- Some basic member function implementations of the `Time` class are not shown, but they are identical to Program-12.3. The stream insertion operator is implemented in Program-12.6.

Lines 4-19:

- These lines contain the `Time` class definition.
- Lines 10, 11, and 12 declare the member function prototypes for the assignment, addition-assignment and addition operators, respectively.
- Lines 10-11 show that the assignment and addition-assignment operators enable cascading by returning the same LHS operand object.
- Line 12 indicates that a new `Time` object is returned *by value* as the addition result.
- Line 18 declares a second version of the `setTime()` member function that takes one integer as parameter. The function sets the data members of the `this` object to the hours, minutes, and seconds corresponding to the parameter value.

Lines 25-30:

- These lines implement the second version of the `setTime()` member function.
- The function separates the total number of seconds in the parameter into the corresponding hours, minutes, and seconds, and it sets the `this` object to these values.

Lines 32-38:

- These lines show the implementation of the overloaded assignment operator.
- The parameter represents the RHS operand, which is the source of the copy, and the `this` object is the LHS operand and the destination of the copy.
- Line 37 enables cascading.

Lines 40-44:

- These lines show the implementation of the overloaded addition-assignment operator.
- Line 42 converts the `this` object to its corresponding total number of seconds, and it adds to this number the total number of seconds in the given parameter. Then it sets the `this` object to the equivalent of the resulting sum by calling the `setTime()` member function implemented on lines 25-30.
- Line 43 enables cascading.

Lines 46-52:

- These lines show the implementation of the overloaded addition operator.
- Line 48 declares a local `Time` object `t` to store the addition result.
- Line 49 calls the assignment operator implemented on lines 32-38 to copy the values from the `this` object into the local `t` object.
- Lines 48-49 could be combined as `Time t = *this;` which is an initialization and calls the copy constructor. Here, we chose a separate declaration and assignment operation instead.
- Line 50 calls the addition-assignment operator implemented on lines 40-44 to add to the `t` object the number of seconds in the parameter, and store the resulting sum into `t`.
- Line 51 returns the updated `t` object and enables cascading.

Lines 57-75:

- These lines contain the implementation of the `main()` function.
- Line 59 declares a local `Time` object called `lunch` and sets it to 12:00 pm (noon).
- Line 61 adds 80 seconds (equivalent to 1 minute and 20 seconds) to the `lunch` object by calling the addition-assignment operator implemented on lines 40-44.
- Line 63 verifies the cascading of the addition-assignment operator by using its return value as an operand for stream insertion.
- Lines 66-67 declare two other `Time` objects, called `iGoHome` and `youGoHome`.
- Line 71 sets the `youGoHome` object to the resulting sum of the current values in the `iGoHome` object and 3700 seconds (equivalent to 1 hour, 1 minute and 40 seconds).
- Line 71 first calls the addition operator implemented on lines 46-52 to add the current values in the `iGoHome` object and 3700 seconds, *without modifying* the `iGoHome` object. A new `Time` object containing the resulting sum is returned from the addition operator.
- Then line 71 calls the assignment operator implemented on lines 32-38 to copy the values from the temporary `Time` object returned by the addition operator into the `youGoHome` object.
- The program output demonstrates the correct functioning of the new overloaded operators.

12.7. Collection class operators

We continue discussing the uses of overloaded operators in our programs. Here, we demonstrate some useful operators that can be implemented in collection classes.

12.7.1. Concepts

12.7.1.1. What are *collection class operators*:

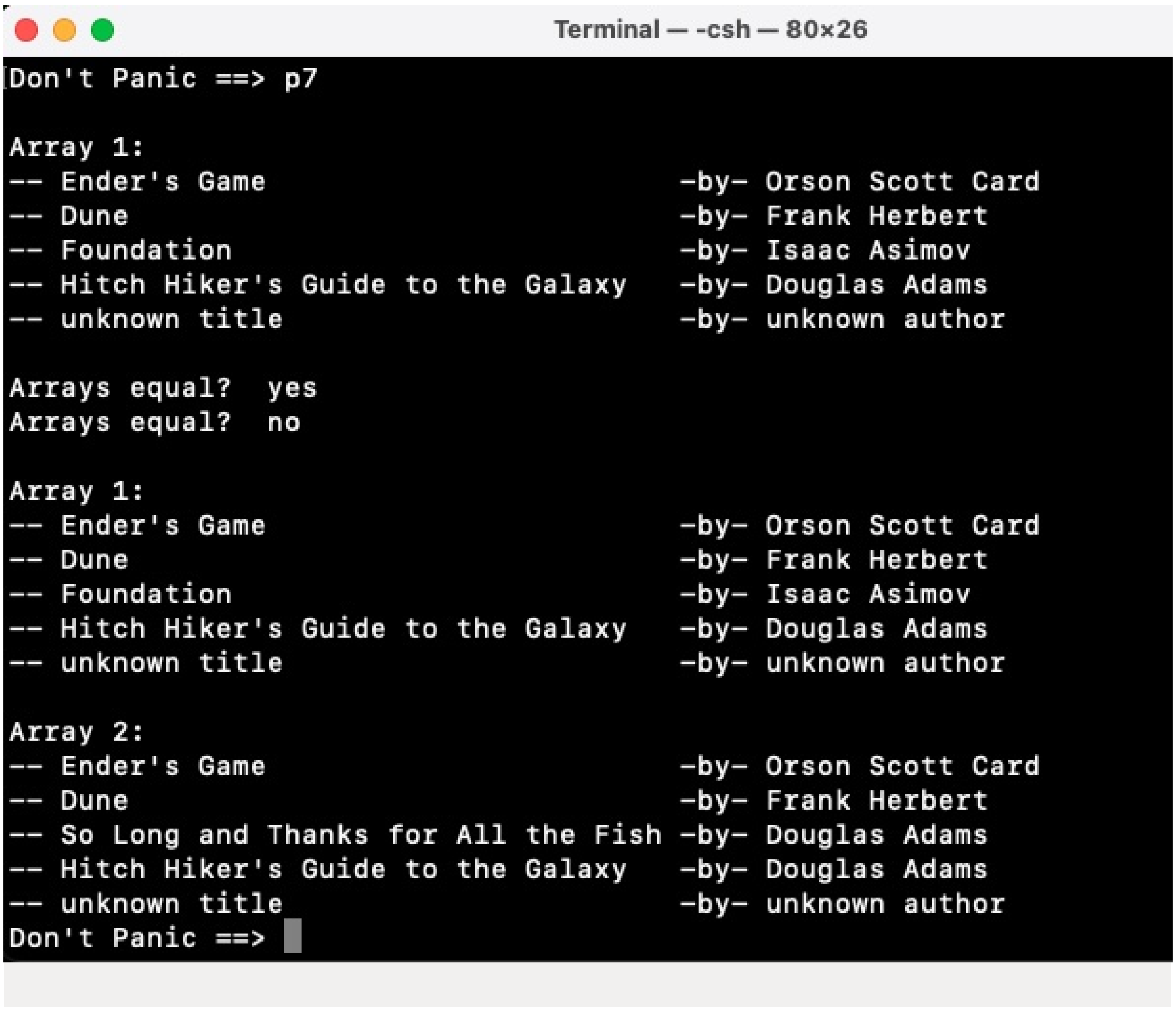
- *Collection class operators* are operators used with instances of collection classes.
- For example, the subscript operator (`[]`) can be used to access an element at a specific index in the collection; the addition and subtraction (`+=` and `-=`) operators can add or remove an element from the collection; relational operators can compare two collections; stream insertion, extraction, and many other operators can also be implemented, as needed.

12.7.2. Coding example: Collection class operators

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Book.h              *
 * * * * * * * * * * * * * * * * */
class Book
{
  friend ostream& operator<<(ostream&, Book&);
  public:
    Book(string="unknown title", string="unknown author");
    Book& operator=(const Book&);
    bool  operator!=(Book&) const;
  private:
    string title;
    string author;
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Book.cc             *
 * * * * * * * * * * * * * * * * */
Book::Book(string t, string a) : title(t), author(a) { }

ostream& operator<<(ostream& output, Book& b)
{
  output << "-- " << setw(35) << left << b.title
         << " -by- " << right << b.author;
  return output;
}

Book& Book::operator=(const Book& b)
{
  title  = b.title;
  author = b.author;
  return *this;
}

bool Book::operator!=(Book& b) const
{
  return ( (title != b.title) || (author != b.author) );
}

/* * * * * * * * * * * * * * * * *
 * Filename:  BookArray.h         *
 * * * * * * * * * * * * * * * * */
class BookArray
{
  friend ostream& operator<<(ostream&, BookArray&);
  public:
    BookArray(int=10);
    ~BookArray();
    Book& operator[](int);
    BookArray& operator=(BookArray&);
    bool operator==(BookArray&) const;
  private:
    int   capacity;
    Book* elements;
};
```

```
/* * * * * * * * * * * * * * * * * *
 * Filename:  BookArray.cc        *
 * * * * * * * * * * * * * * * * * */
BookArray::BookArray(int c)
{
  if (c < 0) {
    cerr<<"Invalid capacity"<<endl;
    exit(1);
  }

  capacity = c;
  elements = new Book[capacity];
}

BookArray::~BookArray()
{
  delete [] elements;
}

BookArray& BookArray::operator=(BookArray& arr)
{
  if (&arr == this) {
    return *this;
  }

  if (capacity != arr.capacity) {
    delete [] elements;
    capacity = arr.capacity;
    elements = new Book[capacity];
  }

  for (int i=0; i<capacity; ++i) {
    elements[i] = arr[i];
  }

  return *this;
}

bool BookArray::operator==(BookArray& arr) const
{
  if (capacity != arr.capacity) {
    return false;
  }

  for (int i=0; i<capacity; ++i) {
    if (elements[i] != arr.elements[i]) {
      return false;
    }
  }

  return true;
}

```

```
Book& BookArray::operator[](int s)
{
  if (s<0 || s >= capacity) {
    cerr<<"Overflow"<<endl;
    exit(1);
  }

  return elements[s];
}

ostream& operator<<(ostream& output, BookArray& arr)
{
  for (int i=0; i<arr.capacity; ++i) {
    output << arr.elements[i] << endl;
  }
  return output;
}

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc            *
 * * * * * * * * * * * * * * * * */
int main()
{
  BookArray arr1(5);
  BookArray arr2;

  Book b1("Ender's Game", "Orson Scott Card");
  Book b2("Dune", "Frank Herbert");
  Book b3("Foundation", "Isaac Asimov");
  Book b4("Hitch Hiker's Guide to the Galaxy", "Douglas Adams");
  Book b5("So Long and Thanks for All the Fish", "Douglas Adams");

  arr1[0] = b1;
  arr1[1] = b2;
  arr1[2] = b3;
  arr1[3] = b4;
  cout << endl << "Array 1:" << endl << arr1 << endl;

  arr2 = arr1;
  cout << "Arrays equal?  "<< ((arr1 == arr2) ? "yes" : "no") << endl;

  arr2[2] = b5;
  cout << "Arrays equal?  "<< ((arr1 == arr2) ? "yes" : "no") << endl;

  cout << endl << "Array 1:" << endl << arr1;
  cout << endl << "Array 2:" << endl << arr2;

  return 0;
}
```



```
Don't Panic ==> p7

Array 1:
-- Ender's Game                        -by- Orson Scott Card
-- Dune                                -by- Frank Herbert
-- Foundation                          -by- Isaac Asimov
-- Hitch Hiker's Guide to the Galaxy   -by- Douglas Adams
-- unknown title                       -by- unknown author

Arrays equal?  yes
Arrays equal?  no

Array 1:
-- Ender's Game                        -by- Orson Scott Card
-- Dune                                -by- Frank Herbert
-- Foundation                          -by- Isaac Asimov
-- Hitch Hiker's Guide to the Galaxy   -by- Douglas Adams
-- unknown title                       -by- unknown author

Array 2:
-- Ender's Game                        -by- Orson Scott Card
-- Dune                                -by- Frank Herbert
-- So Long and Thanks for All the Fish -by- Douglas Adams
-- Hitch Hiker's Guide to the Galaxy   -by- Douglas Adams
-- unknown title                       -by- unknown author
Don't Panic ==> 
```

Program-12.8: Collection class operators

Program purpose:

- Program-12.8 demonstrates the implementation of overloaded collection class operators.
- The program defines a `BookArray` collection class that contains `Book` objects as elements.
- The underlying collection inside the collection class is a dynamically allocated array of objects, as discussed in section 4.4 and Program-4.7.

Lines 4-14:

- These lines contain the `Book` class definition.
- Lines 12-13 declare two private data members for the title and author of a book.
- Line 6 defines a stream insertion operator, line 8 a default constructor, line 9 an overloaded assignment operator (`=`), and line 10 an inequality operator (`!=`).

Lines 19-38:

- These lines show the `Book` class member function implementations.
- Line 19 implements a simple default constructor.
- Lines 21-26 show an overloaded stream insertion operator that prints out both data members.
- Lines 28-33 implement an overloaded assignment operator that copies the contents of a provided book into the `this` object.
- Lines 35-38 show an overloaded inequality operator, where two books are considered unequal if either their titles or their authors differ.

Lines 43-55:

- These lines contain the `BookArray` class definition.
- Lines 53-54 declare two private data members: the maximum capacity of the array, and the dynamically allocated array that stores the `Book` elements. The dynamic allocation of the elements array must occur inside the constructor.
- Lines 47-48 define a default constructor and a destructor, respectively. The constructor parameter specifies a maximum capacity.
- Line 49 declares an overloaded subscript operator that takes an integer index as parameter, line 50 defines an assignment operator, and line 51 an equality operator.

Lines 59-73:

- These lines show the `BookArray` class constructor and destructor implementations.
- Lines 59-68 show the default constructor, where lines 61-64 validate the capacity parameter, line 66 sets the capacity data member from the parameter, and line 67 dynamically allocates the elements array with the specified capacity.
- Lines 70-73 implement the destructor that deallocates the elements array.

Lines 75-92:

- These lines implement the `BookArray` overloaded assignment operator.
- Lines 77-79 perform an initial sanity check that the two operands are not the exact same object. If the two `BookArray` objects are the same one, the function simply returns.
- Lines 81-85 check whether the LHS operand's existing elements array has the correct capacity to accommodate the elements from the RHS operand. If the two operand capacities don't match, the existing LHS array is deallocated on line 82 and reallocated on line 84 to the same capacity as the RHS array.
- Lines 87-89 copy every `Book` element from the RHS array to the LHS one. Line 88 calls the `Book` class's overloaded assignment operator implemented on lines 28-33.
- Line 91 enables cascading.

Lines 94-107:

- These lines show the `BookArray` overloaded equality operator.
- Lines 96-98 check if the LHS and RHS operands have different capacities. If they do, they cannot be equal, and the function returns.
- Lines 100-104 loop over the elements of the two `BookArray` objects and compare the books at the same index, using the `Book` class's inequality operator implemented on lines 35-38.
- The first mismatched book indicates that the two `BookArray` objects are not equal. If we reach the end of the elements arrays without finding a mismatch, then they must be equal.

Lines 109-117:

- These lines implement the `BookArray` overloaded subscript operator.
- Lines 111-114 check that the index specified in the parameter is valid. Because the operator function returns a `Book` reference, there is no value to be returned if the index is invalid. We must terminate the program instead, as we see on line 113.
- Line 116 returns a reference to the `Book` element found at the given index.

Lines 119-125:

- These lines show the `BookArray` overloaded stream insertion operator.
- Lines 121-123 loop over the elements array, and line 122 prints each book element by calling the `Book` class's overloaded stream insertion operator implemented on lines 21-26.

Lines 132-139:

- These lines declare the `Book` and `BookArray` objects in the `main()` function.
- Line 132 declares a `BookArray` object called `arr1` with a capacity of five books.
- Line 133 declares a second `BookArray` object called `arr2` that is initialized with a default capacity of 10 books, as specified by the default argument to the `BookArray` constructor.
- Lines 135-139 declare five `Book` objects.

Lines 141-145:

- Each of the statements on lines 141-144 does the following:
  - first, it calls the `BookArray` subscript operator implemented on lines 109-117 to retrieve the `Book` reference at the given index of in the `arr1` object
  - then, it calls the `Book` class's assignment operator, implemented on lines 28-33, to copy the RHS operand `Book` object into the `Book` reference returned in the previous step
- Line 145 prints out the `arr1` object by calling the `BookArray` stream insertion operator implemented on lines 119-125.

Lines 147-151:

- Line 147 assigns the contents of the `arr1` object into `arr2` by calling the `BookArray` assignment operator implemented on lines 75-92. The `arr2` elements array is reallocated to a capacity of five elements to match `arr1`, and each `Book` element in `arr1` is copied into `arr2`.
- Line 148 checks that both `BookArray` objects are the same by calling the `BookArray` equality operator implemented on lines 94-107. At this point, both have the same elements.
- Line 150 replaces the `Book` element at index 2 of `arr2` with a different book.
- When equality is tested again on line 151, we see from the program output that the two `BookArray` objects are no longer equal.

Lines 153-154:

- These lines print out both `BookArray` objects.
- The program output shows that both `BookArray` objects contain the same books, except for the third element that was changed on line 150.

12.8. Increment and decrement operators

The increment and decrement operators raise a new issue with overloaded operator implementations. As we saw in Program-2.1, the two operators have a prefix and a postfix version, with very different behaviours. We need to take those behaviours into account in our own implementations.

12.8.1. Basics

12.8.1.1. **What is special about *increment/decrement operators*:**

- Like most operators, the increment (`++`) and decrement (`--`) operators can be overloaded as either member functions or global functions.
- The added complication is that both operators have a *prefix* version and a *postfix* version, and both exhibit different behaviours.
- When the increment or decrement operation is executed *as its own separate statement* in the code, there is no observable difference between the prefix and postfix versions, although there are efficiency considerations that are discussed shortly.
- The difference in behaviour between prefix and postfix is with the operator's *return value*, and it's observed only when the operation is *part of an expression with other operators*.

12.8.1.2. **Prefix operator return value:**

- The prefix version of increment/decrement (for example, `++i` or `--i`) increments or decrements the operand and *returns the operand's new value*, i.e. the operand value *after* the operation.
- If prefix increment/decrement is *the only operator in a statement*, its return value is **ignored**. So the behaviour is the same, whether the prefix or postfix version is used. For example:
  - given the statements: `int i = 10; ++i;`
  - in this case, the value of variable `i` is updated from `10` to `11`, and nothing more happens
  - the same applies when the operation is in the loop advancing statement (the third part) of a `for`-loop header: `for (int i=0; i<MAX; ++i)`
  - in the above case, the increment operation is again the only operator in a statement
- If prefix increment/decrement is *one of multiple operators in the same expression*, its return value is the **new operand value**. As a result, the new value is used as an operand to the next operator executing in the same expression. For example:
  - given the statements: `int i = 10; cout << ++i;`
  - in this case, the increment is part of an expression with two operators, and the increment operator's return value is the RHS operand for stream insertion
  - since the value returned by the prefix operator is the new operand value, the stream insertion operation prints `11` to the screen
- The prefix decrement operator has the same return value behaviour as prefix increment.

12.8.1.3. **Postfix operator return value:**

- The postfix version of increment/decrement (for example, `i++` or `i--`) increments or decrements the operand and *returns the operand's original value*, i.e. the operand value *before* the operation.
- If postfix increment/decrement is *the only operator in a statement*, its return value is also **ignored**. So the behaviour is the same, whether the prefix or postfix version is used.
- If postfix increment/decrement is *one of multiple operators in the same expression*, it does update the operand, but its return value is the **original operand value**. As a result, the original value is used as an operand to the next operator executing in the same expression. For example:
  - given the statements: `int i = 10; cout << i++;`
  - in this case, the increment is part of an expression with two operators, and the increment operator's return value is the RHS operand for stream insertion
  - the increment operator still increments the operand `i` from `10` to `11`
  - since the value returned by the postfix operator is the original operand value, the stream insertion operation prints `10` to the screen
- The postfix decrement operator has the same return value behaviour as postfix increment.

12.8.1.4. **Increment/decrement operator function prototypes:**

- Because an operator only has one function name (for example, `operator++` for increment), there is a complication in differentiating between the prefix and postfix implementations.
- Every function in C++ must have a unique signature. Since the function names are identical, the designers of C++ introduced an integer *dummy parameter* for postfix operator functions.
- A dummy parameter has no actual use. It's simply a placeholder that provides the postfix version of an operator function with a different signature from the prefix version.

- The value of the dummy parameter is meaningless and should always be ignored.
- For example:
  - there are four possible function prototypes for the `Time` class's increment operator
  - `Time& Time::operator++()` is the member function prototype for the *prefix* version
  - `Time Time::operator++(int)` is the member function prototype for the *postfix* version
  - `Time& operator++(Time&)` is the global function prototype for the *prefix* version
  - `Time operator++(Time&,int)` is the global function prototype for the *postfix* version
- We already know that the prefix operator returns a reference to the updated operand object. We'll see shortly why the postfix operator must return an actual object by value.

12.8.1.5. **Prefix operator implementation:**

- The prefix increment/decrement operators for our classes have a simple implementation:
  - update the operand object by incrementing or decrementing it
  - return a reference to that same updated operand object
- If we implement the prefix operator as a member function, the operand becomes the `this` object inside the function, and the function takes no parameters.
- If we implement the prefix operator as a global function, the operand is passed in to the function as an object reference.

12.8.1.6. **Postfix operator implementation:**

- The postfix increment/decrement operators for our classes have a slightly more complicated implementation, since we need to make a copy of the *original* operand value:
  - make a local copy of the operand object with its original value; we may need to implement a copy constructor to do this
  - update the operand object by incrementing or decrementing it
  - return *by value* the local copy object as the operand's original value
- If we implement the postfix operator as a member function, the operand becomes the `this` object inside the function, and the function takes a dummy integer parameter.
- If we implement the postfix operator as a global function, the function takes two parameters: the operand as an object reference and a dummy integer parameter.

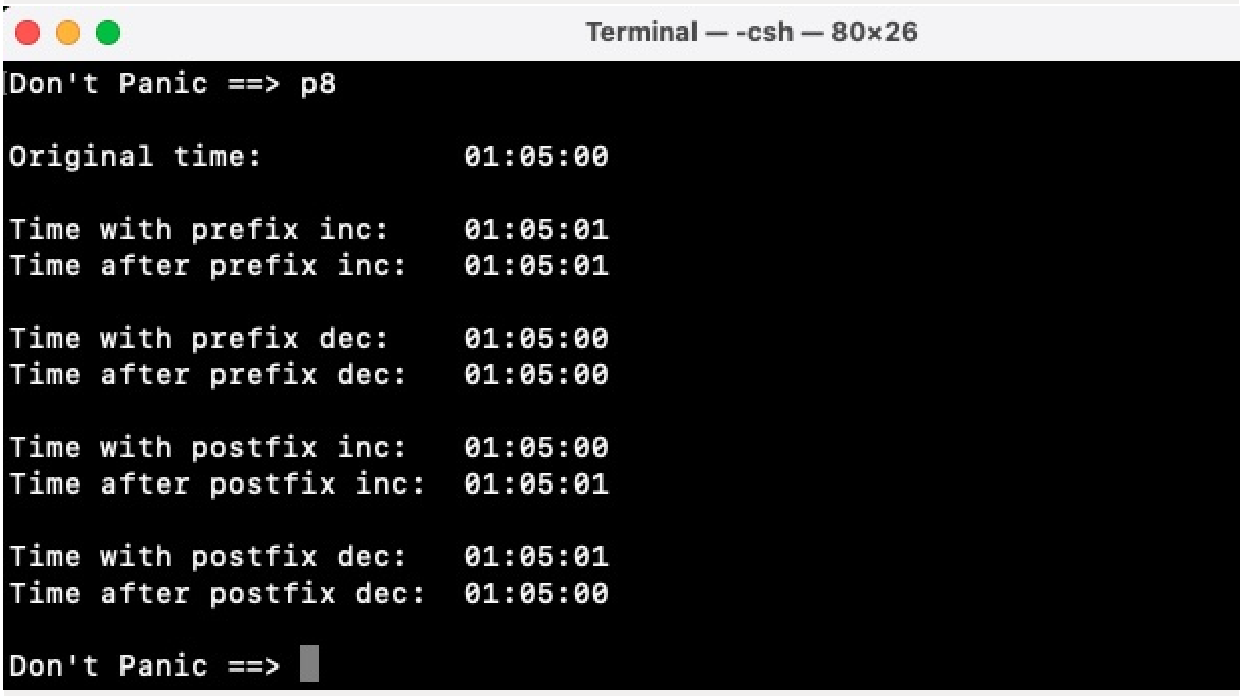
NOTE: Once again, we note that we can **never** use a pointer into a called function's stack frame as a return value. Once that stack frame is popped off the function call stack, its memory is no longer allocated. Attempting to access that memory through a pointer will result in a segmentation fault and program crash.

PRO TIP: We can see why the postfix implementation of an operator is slower than the prefix one. Having to make a copy of an object is a drain on computational resources, especially if the object is a large one or the operator is called repeatedly, for example as the advancing statement of a `for`-loop. While the use of prefix or postfix operators for primitive data type variables are fairly equivalent, it's a good programming habit to favour prefix operators wherever possible.

12.8.2. Coding example: Increment and decrement operators

```
/* * * * * * * * * * * * * * * * *
 * Filename:  Time.h              *
 * * * * * * * * * * * * * * * * */
class Time
{
  friend ostream& operator<<(ostream&, const Time&);
  public:
    Time(int=0, int=0, int=0);
    Time(const Time&);
    void setTime(int, int, int);
    Time& operator++();
    Time& operator--();
    Time  operator++(int);
    Time  operator--(int);
  private:
    int   hours;
    int   minutes;
    int   seconds;
    int   convertToSecs() const;
    void  setTime(int);
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Time.cc             *
 * * * * * * * * * * * * * * * * */
Time::Time(const Time& t) { setTime(t.hours, t.minutes, t.seconds); }

Time& Time::operator++()
{
  setTime(convertToSecs() + 1);
  return *this;
}

Time& Time::operator--()
{
  setTime(convertToSecs() - 1);
  return *this;
}

Time Time::operator++(int)
{
  Time tmp = *this;
  ++(*this);
  return tmp;
}

Time Time::operator--(int)
{
  Time tmp = *this;
  --(*this);
  return tmp;
}

/* The basic Time member functions are not shown */

```

```
56 /* * * * * * * * * * * * * * * * *
57  * Filename:  main.cc              *
58  * * * * * * * * * * * * * * * * */
59 int main()
60 {
61   cout <<endl<<"Original time:           "<< now <<endl;

63   cout << endl << "Time with prefix inc:    " << ++now << endl;
64   cout <<         "Time after prefix inc:   " << now   << endl;
65   cout << endl << "Time with prefix dec:    " << --now << endl;
66   cout <<         "Time after prefix dec:   " << now   << endl;

68   cout << endl << "Time with postfix inc:   " << now++ << endl;
69   cout <<         "Time after postfix inc:  " << now   << endl;
70   cout << endl << "Time with postfix dec:   " << now-- << endl;
71   cout <<         "Time after postfix dec:  " << now   << endl<<endl;

73   return 0;
74 }
```



```
Don't Panic ==> p8

Original time:           01:05:00

Time with prefix inc:    01:05:01
Time after prefix inc:   01:05:01

Time with prefix dec:    01:05:00
Time after prefix dec:   01:05:00

Time with postfix inc:   01:05:00
Time after postfix inc:  01:05:01

Time with postfix dec:   01:05:01
Time after postfix dec:  01:05:00

Don't Panic ==> 
```

Program-12.9: Increment and decrement operators

Program purpose:

- Program-12.9 demonstrates how overloaded prefix and postfix, increment and decrement operators are implemented as member functions of the `Time` class.
- Some basic member function implementations of the `Time` class are not shown, but they are identical to Program-12.7.

Lines 4-21:

- These lines contain the `Time` class definition.
- Line 9 declares a copy constructor.
- Lines 11-14 show the member function prototypes for the overloaded operators. Line 11 declares the prefix increment operator, line 12 the prefix decrement, line 13 the postfix increment, and line 14 the postfix decrement.

Line 26:

- This line shows the implementation of the new copy constructor required in this program.

Lines 28-32:

- These lines show the implementation of the prefix increment operator.
- From the absence of parameters, we know that this is the prefix version of the operator.
- Line 30 demonstrates the logic of the operation. The operand found in the `this` object is converted to a total number of seconds. That total is incremented by one second, and the operand's data members are reset to the hours, minutes, and seconds equivalent to the incremented total number of seconds.
- Line 31 returns the updated operand and enables cascading.

Lines 34-38:

- These lines show the implementation of the prefix decrement operator.
- From the absence of parameters, we know that this is the prefix version of the operator.
- Line 36 demonstrates the logic of the operation. The operand found in the `this` object is converted to a total number of seconds and decremented by one second, and the operand's data members are reset to the values equivalent to the decremented total number of seconds.
- Line 37 returns the updated operand and enables cascading.

Lines 40-45:

- These lines show the implementation of the postfix increment operator.
- From the single integer parameter, we know that this is the postfix version of the operator.
- Line 42 declares a local `Time` object, which is initialized as a copy of the operand found in the `this` object by calling the copy constructor implemented on line 26. This copy is necessary because a postfix operator is required to return the *original* value of the operand object.
- Line 43 demonstrates good code reuse by calling the prefix increment operator, implemented on lines 28-32, which performs the increment operation on the same operand object. Because `this` is a pointer, it must be dereferenced in order to access its pointee object.
- Line 44 returns the local copy of the original operand, by value, and it enables cascading.

Lines 47-52:

- These lines show the implementation of the postfix decrement operator.
- From the single integer parameter, we know that this is the postfix version of the operator.
- Line 49 declares a local `Time` object, which is initialized as a copy of the operand found in the `this` object by calling the copy constructor implemented on line 26.
- Line 50 calls the prefix decrement operator, implemented on lines 34-38, which performs the decrement operation on the same operand object.
- Line 51 returns the local copy of the original operand, by value, and it enables cascading.

Lines 59-74:

- These lines show the implementation of the `main()` function.
- Line 63 calls the prefix increment operator implemented on lines 28-32.
- Line 65 calls the prefix decrement operator implemented on lines 34-38.
- Line 68 calls the postfix increment operator implemented on lines 40-45. The program output shows the original value printed on line 68 and the updated value on line 69.
- Line 70 calls the postfix decrement operator implemented on lines 47-52. The program output shows the original value printed on line 70 and the updated value on line 71.

Chapter 13

Templates

Templates are an important tool in promoting code reuse in our programs. Sometimes called *generics* in other programming languages, templates in C++ allow programmers to treat data types as a kind of variable inside our functions and classes.

In this chapter, we discuss the use of templates in general and the implementation of function templates and class templates in C++.

13.1. Principles

We discuss the role of code reuse in OO design and the mechanisms for code reuse in C++.

13.1.1. Code reuse in OO design

13.1.1.1. **Main principles of OO design:**

- As discussed in previous chapters, designing programs for *data abstraction* focuses on the separation of the abstract properties of classes from their concrete implementations.
- There are two important techniques for designing our programs for *code reuse*:
  - make use of existing code, for example by reusing our own or library-provided classes
  - design our classes so that they are reusable, either in our own programs or others

13.1.1.2. **Designing for code reuse:**

- This section focuses on designing our classes so that they are reusable within the same program or other ones.
- *Generic programming* techniques. including *templates* in C++, help make our classes reusable. The goal is to code once and reuse the same code multiple times.

13.1.2. Mechanisms for code reuse

13.1.2.1. **Code reuse with functions:**

- Designing a modular function makes it reusable.
- We design and implement the function, then we call it with different parameter values.
- By varying the parameter values, the code effectively reuses the function by calling it in different ways.
- With functions, parameter values can vary, but the parameter data types are fixed.

13.1.2.2. **Code reuse with templates:**

- Using a template also makes our code reusable, because it works with all kinds of data types.
- A template can be either a function or a class, and it uses a data type as a kind of *parameter* to the template.
- The template code is written once, and then it can be used with *any data type* as parameter.

13.2. Function templates

We discuss how functions are implemented in C++ independently of data types.

13.2.1. Concepts

13.2.1.1. **What is a *function template*:**

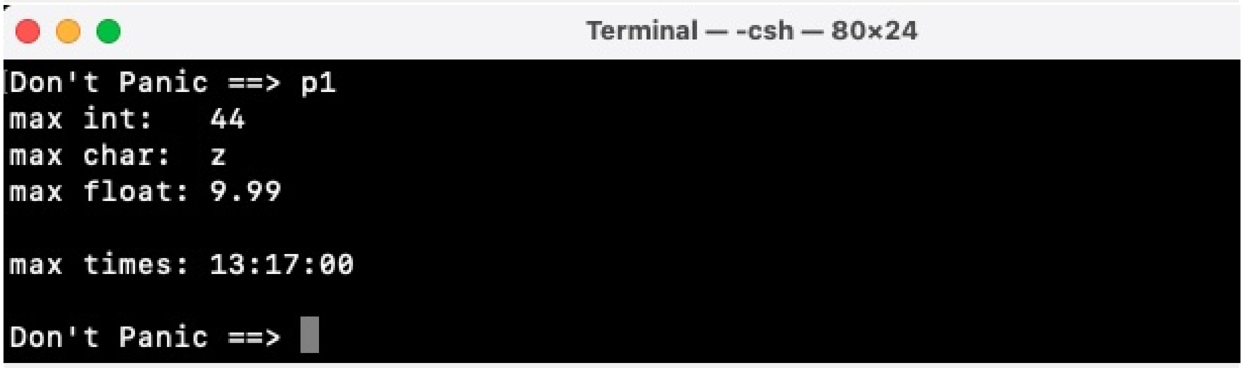
- A ***function template*** is a global or member function that works with any data type.
- The function template definition specifies the function implementation, but with one or more data types as a kind of "variable" or parameter to the template. We call this the *template parameter*.
- It's common to use a single uppercase letter, like `T` or `V`, to denote the template parameter.
- The template parameter can be reused anywhere inside the function template, for example as a data type for parameters, return type, local variables, and so on.
- For example, a `max()` function that compares three values passed in as parameters and returns the maximum has identical logic whether we are comparing three integers, three floating point numbers, or three `Time` objects. If we make `max()` a function template with template parameter `T`, then all three parameters and the return value can be declared as data type `T`, and the same function implementation can be reused for any data type.
- Both global and member functions can be templates. However, a member function template must be contained within a class template.

13.2.1.2. **How function templates work:**

- The work required for the implementation of function templates is distributed between the programmer and the compiler.
- The programmer writes the function implementation using a template parameter in place of a data type. Then they call the function in the program, using a specific data type.
- At compile time, the compiler detects when a function template is *called*. If a function template is never called, then it is never compiled.
- A function template must be called using a specific data type. The compiler then generates a new *specialization* of the template with that data type hard-coded into the implementation. The generated specialization is then compiled with the rest of the code.
- For every call to a function template with a different data type, the compiler generates a new specialization that's hard-coded for that data type. Specializations are *only* generated for the data types with which the function is called inside the program.
- A function template may be overloaded with a non-template version of the same function.

13.2.2. Coding example: Function templates

```
int main()
{
  cout << "max int:    " << max(33, 22, 44)          << endl;
  cout << "max char:  " << max('Z', 'z', 'c')       << endl;
  cout << "max float: " << max(3.14f, 9.99f, 3.45f) << endl;
  cout << endl;

  Time t1(11,22);
  Time t2(13,17);
  Time t3(9,55);
  cout << "max times: " << max(t1,t2,t3) << endl << endl;

  return 0;
}

template <typename T>
T max(T v1, T v2, T v3)
{
  T maxValue = v1;

  if (v2 > maxValue) {
    maxValue = v2;
  }

  if (v3 > maxValue) {
    maxValue = v3;
  }

  return maxValue;
}
```



```
Don't Panic ==> p1
max int:   44
max char:  z
max float: 9.99

max times: 13:17:00

Don't Panic ==>
```

Program-13.1: Function templates

Program purpose:

- Program-13.1 demonstrates the implementation of a global function template.
- The `Time` class is identical to Program-12.7, with the addition of an overloaded greater-than (>) operator.
- The program declares a global function template called `max()` that takes three values as parameters, compares them, and returns the maximum value.

Lines 16-30:

- These lines contain the `max()` global function template that compares three values passed in as parameters and returns the maximum as the return value.
- Line 16 announces that a template definition follows. It also declares that, within the template definition, the name of the template parameter is `T`.
- Line 17 shows the function prototype, which declares all three parameters and the return type with the template parameter data type `T`.
- We see from lines 19-27 that three member functions are required of any class that's used as the template parameter data type. Line 19 generates a call to a copy constructor; lines 21 and 25 call the overloaded greater-than operator; and lines 22 and 26 call the overloaded assignment operator.
- Line 19 declares a local variable of the template parameter data type to store the current maximum value. If `T` is a class, this line calls its copy constructor.
- Lines 21-27 show the logic that determines the maximum value.
- Line 29 returns the maximum of the three parameter values.

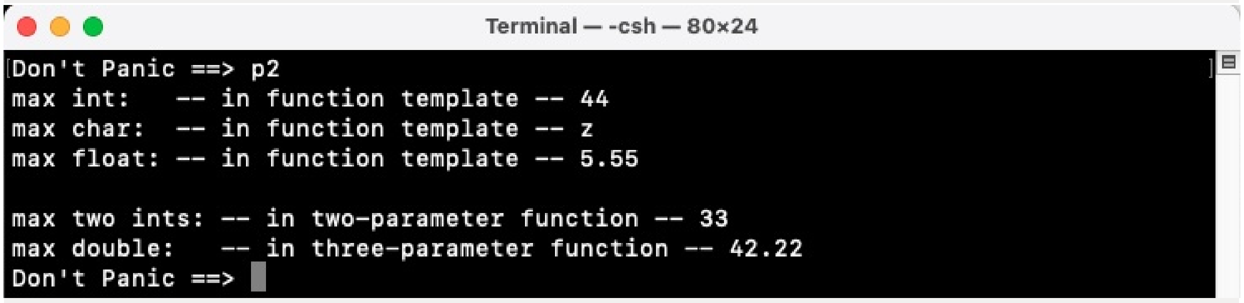
Lines 1-14:

- These lines contain the implementation of the `main()` function.
- At compile time, line 3 generates a specialization of the `max()` function for the integer data type; line 4 generates a second specialization for characters; line 5 creates a third specialization for floats; and line 11 generates a fourth version of `max()` for `Time` objects.
- The program output shows that the maximum is computed correctly for all four data types.

13.2.3. Coding example: Overloaded function templates

```cpp
int main()
{
  cout << "max int:    " << max(33, 22, 44)           << endl;
  cout << "max char:   " << max('Z', 'z', 'c')        << endl;
  cout << "max float:  " << max(2.22f, 5.55f, 3.88f) << endl<<endl;

  cout << "max two ints: " << max(33, 22)             << endl;
  cout << "max double:   " << max(2.22, 5.55, 3.88) << endl;
  return 0;
}

template <typename T>
T max(T v1, T v2, T v3)
{
  cout << "-- in function template -- ";
  T maxValue = v1;

  if (v2 > maxValue) {
    maxValue = v2;
  }

  if (v3 > maxValue) {
    maxValue = v3;
  }

  return maxValue;
}

```

```
29 int max(int v1, int v2)
30 {
31   cout << "-- in two-parameter function -- ";
32   if (v1 > v2) {
33     return v1;
34   }
35   else {
36     return v2;
37   }
38 }

40 double max(double d1, double d2, double d3)
41 {
42   cout << "-- in three-parameter function -- ";
43   return 42.22;
44 }
```



```
Don't Panic ==> p2
max int:   -- in function template -- 44
max char:  -- in function template -- z
max float: -- in function template -- 5.55

max two ints: -- in two-parameter function -- 33
max double:   -- in three-parameter function -- 42.22
Don't Panic ==>
```

Program-13.2: Overloaded function templates

Program purpose:

- Program-13.2 demonstrates a function template overloaded with a non-template version.
- The program declares a global function template called `max()` that takes three values as parameters, compares them, and returns the maximum.
- It also implements an overloaded non-template version of `max()` that takes two integer parameters, as well as an overloaded non-template version that takes three doubles. We see that, if the compiler has a choice between a function template and a non-template version, the non-template function is called.

Lines 12-27:

- These lines contain the implementation of the `max()` global function template. The code is identical to Program-13.1, with the addition of a printed debugging statement.

Lines 29-44:

- Lines 29-38 contain the implementation of a non-template version of `max()` that takes two integer parameters and returns the maximum.
- Lines 40-44 show the implementation of a non-template version of `max()` that takes three doubles and returns a constant.

Lines 1-10:

- These lines contain the implementation of the `main()` function.
- Lines 3, 4, and 5 generate specializations of the function template for the integer, character, and floating point number data types, respectively.
- Line 7 does *not* generate a template specialization because it only takes two parameters. Instead, it calls the two-parameter `max()` function shown on lines 29-38.

- Line 8 also does *not* generate a template specialization. It takes three parameters and could potentially generate a new template specialization for the double data type. However, because a non-template version of `max()` that takes three doubles is provided on lines 40-44, the non-template function is called instead.

13.3. Class templates

We discuss how classes are implemented in C++ independently of data types.

13.3.1. Concepts

13.3.1.1. **What is a *class template*:**

- A ***class template*** is a class that works with any data type. It is sometimes also called a *parameterized type*.
- The class template definition specifies all the class members, including data members and member functions, but with one or more data types as a kind of "variable" or parameter to the template.
- The template parameter can be reused inside the class template for any class member, for example as a data type for data members, and member function parameters or return types or local variables.
- For example, an `Array` class that stores same-type elements has identical behaviour whether the elements are all integers, floating point numbers, or `Time` objects. If we make `Array` a class template with template parameter `T`, then the elements can be declared as data type `T`, and the same member function implementations can be reused for any data type.

13.3.1.2. **Characteristics of a class template:**

- Making a class into a class template changes the nature of the class name itself. Since our code is packaged so that member function implementations are outside the class definition, the member functions become function templates in order to be defined as belonging to a class template.
- For example, outside the `Array` class definition, the class name is no longer simply `Array`. With `T` as the template parameter, the class name becomes `Array<T>` everywhere. Member function implementations that are packaged into a source file must indicate that they belong to the `Array<T>` class, and not `Array`.
- Class templates are commonly used to implement collection classes.

13.3.1.3. **How class templates work:**

- The work required for the implementation of class templates is distributed between the programmer and the compiler.
- The programmer writes the class definition and its member function implementations using a template parameter in place of a data type. Then they create an instance of the class in the program, using a specific data type.
- At compile time, the compiler detects when a class template is *instantiated*, i.e. when an instance of the class is created.
- A class template must be instantiated using a specific data type. The compiler then creates a *specialization* of the template with that data type hard-coded into the class definition and its member function implementations. The generated specialization is then compiled with the rest of the code.

- For every instance of a class template that's instantiated with a different data type, the compiler generates a new specialization that's hard-coded for that data type. Specializations are *only* generated for the data types with which the class is instantiated inside the program.

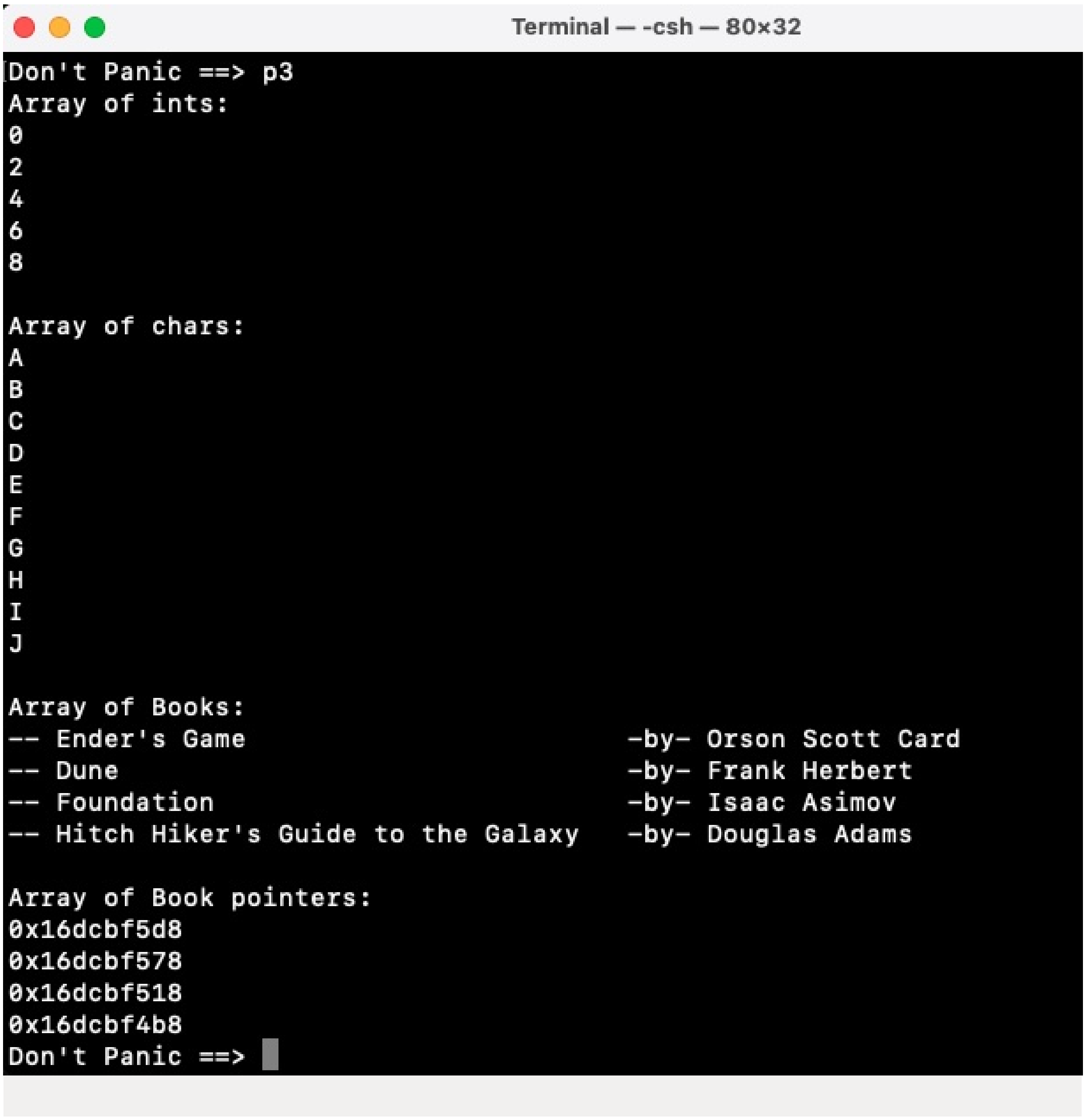
13.3.1.4. **Class template variations:**

- A class template may be defined with multiple template parameters.
- A default data type can be specified for a template parameter, in the same way that we declare default arguments for functions, as we discussed in section 3.3.
- If a class defines a static member, each specialization gets its own copy of that member.

13.3.2. Coding example: Class templates

```
/* * * * * * * * * * * * * * * * * *
 * Filename:  Array.h              *
 * * * * * * * * * * * * * * * * * */
template <class T>
class Array
{
  template <class V>
  friend ostream& operator<<(ostream&, const Array<V>&);
  public:
    Array(int=10);
    ~Array();
    T&  operator[](int);
  private:
    int   capacity;
    T*    elements;
};

template <class T>
Array<T>::Array(int c)
{
  if (c < 0) {
    cerr<<"Invalid capacity"<<endl;
    exit(1);
  }
  capacity = c;
  elements = new T[capacity];
}

template <class T>
Array<T>::~Array()
{
  delete [] elements;
}

template <class T>
T& Array<T>::operator[](int s)
{
  if (s<0 || s >= capacity) {
    cerr<<"Overflow"<<endl;
    exit(1);
  }
  return elements[s];
}
```

```
template <class T>
ostream& operator<<(ostream& output, const Array<T>& arr)
{
  for (int i=0; i<arr.capacity; ++i) {
    output << arr.elements[i] << endl;
  }
  return output;
}

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc            *
 * * * * * * * * * * * * * * * * */
int main()
{
  Array<int>  arrInts(5);
  Array<char> arrChars;
  Array<Book> arrBooks(4);

  Book b1("Ender's Game", "Orson Scott Card");
  Book b2("Dune", "Frank Herbert");
  Book b3("Foundation", "Isaac Asimov");
  Book b4("Hitch Hiker's Guide to the Galaxy", "Douglas Adams");

  for (int i=0; i<5; ++i) {
    arrInts[i] = i*2;
  }
  cout<<"Array of ints:"<<endl;
  cout<< arrInts << endl;

  for (int i=0; i<10; ++i) {
    arrChars[i] = i+65;
  }
  cout<<"Array of chars:"<<endl;
  cout<< arrChars << endl;

  arrBooks[0] = b1;
  arrBooks[1] = b2;
  arrBooks[2] = b3;
  arrBooks[3] = b4;
  cout<<"Array of Books:"<<endl;
  cout << arrBooks << endl;

  Array<Book*> arrPtrs(4);
  arrPtrs[0] = &b1;
  arrPtrs[1] = &b2;
  arrPtrs[2] = &b3;
  arrPtrs[3] = &b4;
  cout<<"Array of Book pointers:"<<endl;
  cout << arrPtrs;

  return 0;
}
```



```
Don't Panic ==> p3
Array of ints:
0
2
4
6
8

Array of chars:
A
B
C
D
E
F
G
H
I
J

Array of Books:
-- Ender's Game                           -by- Orson Scott Card
-- Dune                                   -by- Frank Herbert
-- Foundation                             -by- Isaac Asimov
-- Hitch Hiker's Guide to the Galaxy      -by- Douglas Adams

Array of Book pointers:
0x16dcbf5d8
0x16dcbf578
0x16dcbf518
0x16dcbf4b8
Don't Panic ==> 
```

Program-13.3: Class templates

Program purpose:

- Program-13.3 demonstrates the implementation of a class template.
- The program defines a collection class template called `Array` that uses a dynamically allocated array to store elements of the template parameter data type.
- The `Array` class is a template version of the `BookArray` class in Program-12.8, with many of the same member functions and overloaded operators.
- The `Book` class used in this program is identical to Program-12.8.

Lines 4-16:

- These lines contain the `Array` class template definition, with line 4 defining it as a template.
- Lines 7-8 grant friendship to the overloaded stream insertion operator global function, which must be a function template to manipulate an instance of the `Array<T>` class template. Within the scope of the friendship declaration, we must use a different template parameter (`V` instead of `T`), in order to not "hide" the overall class template parameter `T`.
- Lines 10 and 11 declare the default constructor and destructor, respectively.

- Line 12 defines the overloaded subscript operator, which returns a reference to the array element of type `T` at the given index.
- Lines 14 and 15 declare the maximum array capacity and the underlying container in which the class stores its elements, respectively, as data members. The elements are stored in a dynamically allocated primitive array, with elements of the template parameter data type `T`.

Lines 18-27:

- These lines show the implementation of the `Array<T>` constructor.
- Line 18 declares the member function as a function template.
- Line 19 shows that the constructor belongs to the `Array<T>` class (remember that outside the class definition, the `Array` class does not exist).
- Line 26 dynamically allocates the memory for the underlying container, as a primitive array of elements of type `T`.
- The overall logic of the constructor is identical to Program-12.8.

Lines 29-51:

- These lines show the implementation of the remainder of the `Array<T>` member functions.
- Lines 29-33 implement the destructor, which is identical to Program-12.8.
- Lines 35-43 and lines 44-51 do the same for the overloaded subscript and stream insertion operators, respectively.

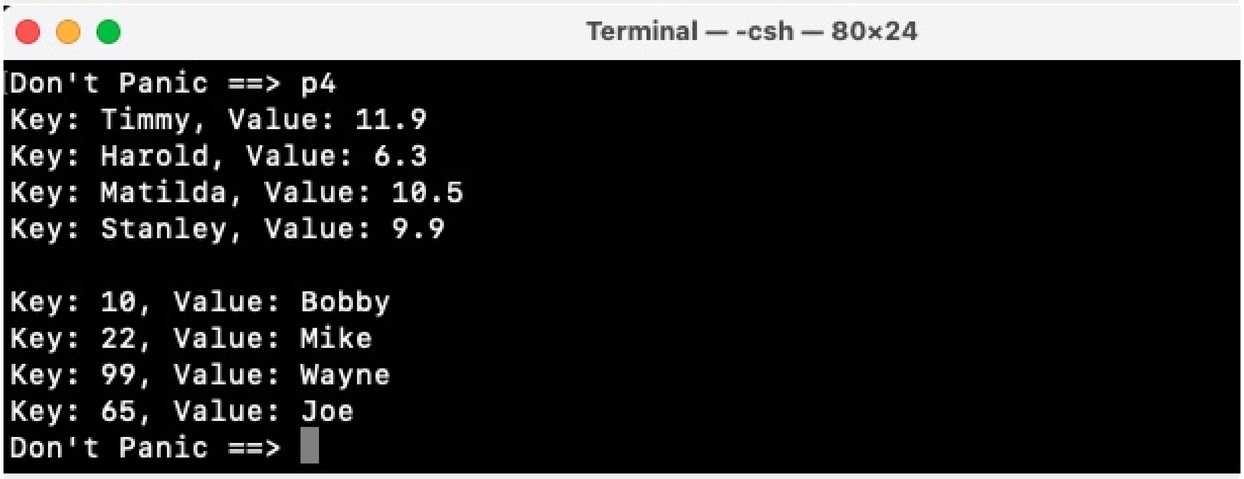
Lines 56-95:

- These lines contain the implementation of the `main()` function.
- Line 58 creates an instance of the `Array<T>` class, called `arrInts`, with template parameter `T` set to the integer data type. In the constructor on line 26, the primitive array of integers is dynamically allocated with a capacity of 5 elements.
- Line 59 declares another instance of the `Array<T>` class, called `arrChars`, with template parameter `T` set to the character data type, with a default capacity of 10 elements as declared on line 10.
- Line 60 creates a third instance of the `Array<T>` class, `arrBooks`, with template parameter `T` set to the `Book` class data type. In the constructor on line 26, the primitive array of `Book` objects is dynamically allocated with a capacity of 4 elements.
- Lines 62-65 allocate and initialize four `Book` objects.
- Lines 67-69 populate the `arrInts` object with five even numbers (and zero) by using the `Array<T>` class's overloaded subscript operator implemented on lines 35-43.
- Line 71 prints out the contents of the integer array by calling the `Array<T>` class's overloaded stream insertion operator shown on lines 44-51. As we see from the program output, the five integers are printed out to the screen.
- Lines 73-75 populate the `arrChars` object with ten uppercase characters using the `Array<T>` class's overloaded subscript operator.
- Line 77 prints out the contents of the character array with the `Array<T>` class's overloaded stream insertion operator. The program output shows the ten characters printed out.
- Lines 79-82 populate the `arrBooks` object with the four existing `Book` objects by using the `Array<T>` subscript operator.
- Line 84 prints out the books array with the `Array<T>` stream insertion operator. As we see from the program output, the four `Book` objects are printed out.
- Line 86 declares an instance of the `Array<T>` class, called `arrPtrs`, with template parameter `T` set to the `Book` pointer data type. In the constructor on line 26, the primitive array of `Book` pointers is dynamically allocated with a capacity of 4 elements.

- Lines 87-90 populate the `arrPtrs` object with the addresses of the four existing `Book` objects by using the `Array<T>` subscript operator.
- Line 92 prints out the contents of the book pointers array by calling the `Array<T>` stream insertion operator. The program output shows the addresses of the four `Book` objects.

13.3.3. Coding example: Class template variations

```cpp
/* * * * * * * * * * * * * * * * *
 * Filename:  MapPair.h           *
 * * * * * * * * * * * * * * * * */
template <class T=int, class V=string>
class MapPair
{
  template <class A, class B>
  friend ostream& operator<<(ostream&, MapPair<A,B>&);
  public:
    void set(T, V);
  private:
    T key;
    V value;
};

template <class T, class V>
void MapPair<T,V>::set(T k, V v)
{
  key   = k;
  value = v;
}

template <class T, class V>
ostream& operator<<(ostream& output, MapPair<T,V>& p)
{
  output << "Key: " << p.key << ", Value: " << p.value << endl;
  return output;
}

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc             *
 * * * * * * * * * * * * * * * * */
#define MAX_SIZE  4

int main()
{
  MapPair<string, float> grades[MAX_SIZE];
  grades[0].set("Timmy", 11.9);
  grades[1].set("Harold", 6.3);
  grades[2].set("Matilda", 10.5);
  grades[3].set("Stanley", 9.9);

  for (int i=0; i<MAX_SIZE; ++i) {
    cout<<grades[i];
  }
  cout<<endl;

48    MapPair<> playerNums[MAX_SIZE];
49    playerNums[0].set(10, "Bobby");
50    playerNums[1].set(22, "Mike");
51    playerNums[2].set(99, "Wayne");
52    playerNums[3].set(65, "Joe");

54    for (int i=0; i<MAX_SIZE; ++i) {
55      cout << playerNums[i];
56    }

58    return 0;
59  }
```



```
Don't Panic ==> p4
Key: Timmy, Value: 11.9
Key: Harold, Value: 6.3
Key: Matilda, Value: 10.5
Key: Stanley, Value: 9.9

Key: 10, Value: Bobby
Key: 22, Value: Mike
Key: 99, Value: Wayne
Key: 65, Value: Joe
Don't Panic ==> 
```

Program-13.4: Class template variations

Program purpose:

- Program-13.4 demonstrates two variations of class templates: multiple template parameters and default template parameter types.
- The program defines a class template called `MapPair` that stores a key-value pair, similar to the elements of a map data structure. We see how a class template can have two template parameters, and how these parameters may have default values.

Lines 4-14:

- These lines contain the `MapPair` class template definition.
- Line 4 reveals a lot of information about this class template, including the definition of the class as a template. It also shows that `MapPair` has two template parameters: `T` and `V`.
- In addition, line 4 defines default types for each template parameter. By default, `T` has a data type of integer, and `V` is a string. The syntax is nearly identical to that used in the declaration of default arguments to functions, as we saw in section 3.3.
- Lines 7-8 grant friendship to the overloaded steam insertion operator global function.
- Line 10 defines a setter function for the key and value data members.
- Lines 12-13 show the declaration of the key-value pair as data members. We see that the `key` data member is declared with the data type indicated by template parameter `T`, and `value` has the data type specified by template parameter `V`.

Lines 16-28:

- Lines 16-21 show the implementation of the key-value setter member function.
- Lines 23-28 implement the stream insertion operator that prints out the key-value pair.

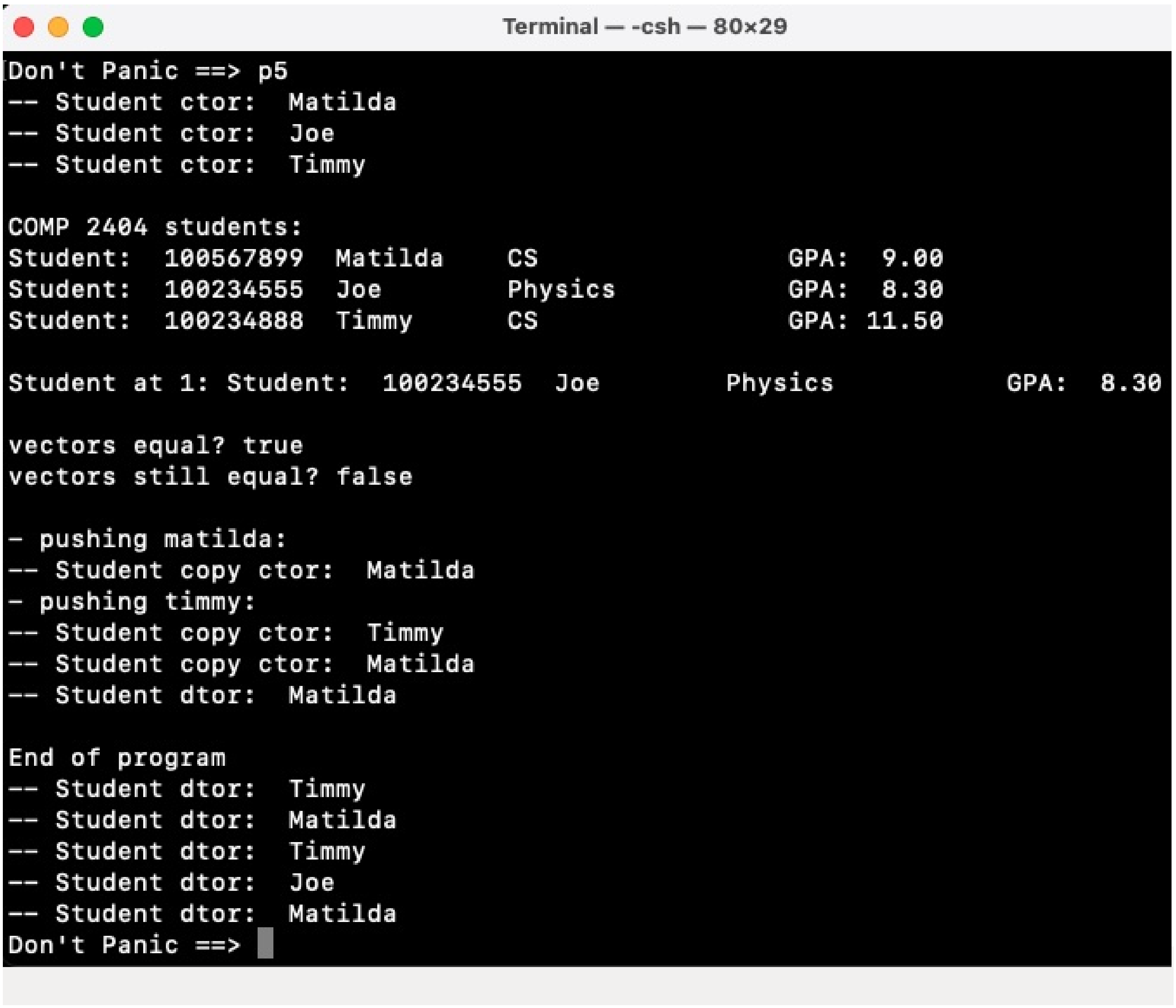
Lines 35-59:

- These lines contain the implementation of the `main()` function.
- Line 37 declares a primitive array of four `MapPair` objects, called `grades`. Each `MapPair` object in the array contains a key with a string data type and a value with a float data type.
- Lines 38-41 populate the `grades` array with key-value pairs by calling the `MapPair` class's setter member function implemented on lines 16-21.
- Lines 43-45 loop over the `grades` array and print out each pair by calling the `MapPair` class's overloaded stream insertion operator implemented on lines 23-28.
- Line 48 declares another array of four `MapPair` objects, called `playerNums`. In this array, each `MapPair` object uses the default template parameter data types for the key-value pair, as defined on line 4. So the key uses an integer data type and the value a string data type.
- Lines 49-52 populate the `playerNums` array with key-value pairs by calling the `MapPair` class's setter member function.
- Lines 54-56 loop over the `playerNums` array and print out each pair by calling the `MapPair` class's stream insertion operator.

13.3.4. Coding example: STL `vector` class template

```cpp
/* * * * * * * * * * * * * * * * *
 * Filename:  Student.h          *
 * * * * * * * * * * * * * * * * */
class Student
{
  friend ostream& operator<<(ostream&, Student&);
  public:
    Student(string="000000000", string="No name", string="No major", float=0.0f);
    Student(const Student&);
    ~Student();
  private:
    const string number;
    string name;
    string majorPgm;
    float  gpa;
};

/* * * * * * * * * * * * * * * * *
 * Filename:  Student.cc         *
 * * * * * * * * * * * * * * * * */
Student::Student(string s1, string s2, string s3, float g)
    : number(s1), name(s2), majorPgm(s3), gpa(g)
{
  cout<<"-- Student ctor:  "<< name <<endl;
}

Student::Student(const Student& stu)
    : number(stu.number), name(stu.name), majorPgm(stu.majorPgm), gpa(stu.gpa)
{
  cout<<"-- Student copy ctor:  "<< name <<endl;
}

Student::~Student()
{
  cout<<"-- Student dtor:  "<< name <<endl;
}
```

```cpp
ostream& operator<<(ostream& output, Student& stu)
{
  output<<"Student:  " << stu.number << "  " << left << setw(10)
        << stu.name << " " << setw(15) << stu.majorPgm << "   GPA: "
        << fixed << setprecision(2) << setw(5) << right << stu.gpa << endl;
  return output;
}

/* * * * * * * * * * * * * * * * *
 * Filename:  main.cc            *
 * * * * * * * * * * * * * * * * */
int main()
{
  Student matilda("100567899", "Matilda", "CS", 9.0f);
  Student joe("100234555", "Joe", "Physics", 8.3f);
  Student timmy("100234888", "Timmy", "CS", 11.5f);

  vector<Student*> comp2404;
  comp2404.push_back(&matilda);
  comp2404.push_back(&joe);
  comp2404.push_back(&timmy);

  cout<<endl<<"COMP 2404 students:"<<endl;
  for (int i=0; i<comp2404.size(); ++i) {
    cout << *(comp2404[i]);
  }

  cout<<endl<<"Student at 1: ";
  cout<< *(comp2404.at(1)) << endl;

  vector<Student*> comp2406(comp2404);
  cout<<"vectors equal? " << (comp2404==comp2406?"true":"false") << endl;
  comp2406.pop_back();
  cout<<"vectors still equal? "<<(comp2404==comp2406?"true":"false")<<endl<<endl;

  vector<Student> vect2;
  cout<< "- pushing matilda:" << endl;
  vect2.push_back(matilda);
  cout<< "- pushing timmy:" << endl;
  vect2.push_back(timmy);

  cout<< endl << "End of program" << endl;
  return 0;
}
```



```
Don't Panic ==> p5
-- Student ctor:  Matilda
-- Student ctor:  Joe
-- Student ctor:  Timmy

COMP 2404 students:
Student:  100567899  Matilda    CS               GPA:   9.00
Student:  100234555  Joe        Physics          GPA:   8.30
Student:  100234888  Timmy      CS               GPA:  11.50

Student at 1: Student:  100234555  Joe        Physics          GPA:   8.30

vectors equal? true
vectors still equal? false

- pushing matilda:
-- Student copy ctor:  Matilda
- pushing timmy:
-- Student copy ctor:  Timmy
-- Student copy ctor:  Matilda
-- Student dtor:  Matilda

End of program
-- Student dtor:  Timmy
-- Student dtor:  Matilda
-- Student dtor:  Timmy
-- Student dtor:  Joe
-- Student dtor:  Matilda
Don't Panic ==> 
```

Program-13.5: STL `vector` class template

Program purpose:

- Program-13.5 demonstrates the use of the standard template library (STL) `vector` class, which is a collection class that is also a class template.
- The program defines some `vector` objects and showcases some of the overloaded operators and member functions provided with the class.

Lines 4-43:

- These lines contain the `Student` class definition and member function implementations.
- The class is very similar to the one we saw in Program-8.4, with the addition of a copy constructor and a destructor, as well as the replacement of the print member function with an overloaded stream insertion operator.
- For clarity purposes, the default constructor, the copy constructor, and the destructor print out the name of the student. This assists in highlighting one of the major problems with the STL `vector` class.

Lines 50-62:

- These lines show the first part of the `main()` function implementation.
- Lines 50-52 allocate and initialize three `Student` objects. We see from the program output that the default constructor is called for each one.
- Line 54 declares an STL `vector` object, called `comp2404`, to contain `Student` object *pointers*. Collections are always empty when they are declared, and the STL `vector` is no exception.
- Lines 55-57 add the addresses of the three existing `Student` objects to the end of the `vector`. We see from the program output that inserting pointers into a `vector` does *not* cause any `Student` objects to be created or copied.
- Lines 60-62 print out the contents of the `vector`. Because the `vector` elements are pointers, each element must be dereferenced in order to print out the `Student` information.
- Line 61 calls the `Student` stream insertion operator implemented on lines 37-43.

Lines 64-70:

- These lines demonstrate some member functions implemented in the STL `vector` class.
- Line 65 shows the use of the `at()` member function. It works similarly to the subscript operator, and it returns the element at the given index. However, it's much safer to use. While the subscript operator is limited in its bounds checking, the `at()` member function throws an exception if an out-of-bounds index is provided. Exception handling is discussed chapter 14.
- Line 67 uses the STL `vector`'s copy constructor to make a copy of `comp2404` into a new `vector` called `comp2406`. Because the `vector` elements are pointers, a shallow copy is performed. As a result, no objects are copied, but the elements of both `vector`s point to the same `Student` objects declared on lines 50-52.
- Line 68 compares the two `vector`s using the overloaded equality operator, and the program output shows that the two are equal.
- Line 69 removes the last element in the `comp2406 vector`.
- When line 70 compares the two `vector`s again, the program output shows that they are no longer equal.

Lines 72-76:

- These lines demonstrate one of the major problems with the STL `vector` class.
- Line 72 declares an STL `vector` object, called `vect2`, to contain `Student` *objects*, and not object pointers.
- When `matilda` is added to the new `vector` on line 74, we see from the program output that a *copy* of the `matilda` object is created.
- When `timmy` is added to the `vector` on line 76, the program output shows that both `matilda` and `timmy` are copied. So now there are three instances of `matilda` in the program and two instances of `timmy`. Immediately afterwards, one instance of `matilda` is destroyed.
- So what's going on here? Initially, a `vector` is created with a capacity of one element. When we add `matilda` on line 74, the object is copied into the `vector`, which is then full.
- When `timmy` is added on line 76, there is no room for it. So a new `vector` is created, with double the capacity (so two elements), and the existing elements and the new one are copied into this new `vector`. When the old `vector` is deallocated, its copy of `matilda` is no longer required, and it is destroyed.
- Storing objects in an STL `vector` is a cumbersome affair. Elements are copied when they are added to the `vector`, and again every time the `vector` capacity is increased. If the objects are somewhat large, computational efficiency will be negatively impacted.
- The moral of the story: STL containers are great to use, as long as we store primitive data types or pointers, and not objects.

Part IV

Essential C++ Techniques

Chapter 14

Exception Handling

Exception handling is a mechanism in C++ that provides an alternate control flow when a serious error is encountered by the program at runtime.

In this chapter, we introduce the principles of exception handling and its mechanism in C++. We also illustrate some side effects of bypassing the regular program control structures.

14.1. Principles

We discuss the principles of software robustness and error handling in our programs.

14.1.1. Software robustness

14.1.1.1. **What is *software robustness*:**

- *Software robustness* is the degree to which a program can keep running despite the presence of faults.
- A *fault* in a piece of software is a defect, also known as a *bug*. It's something, either internal or external to the program, that causes it to crash.
- No program is perfect! We should never assume that ours have no bugs.

14.1.1.2. **Types of faults:**

- There are different kinds of faults that could result in a program crash.
- The end-user may enter bad input:
  - they may enter the wrong data type, for example a string instead of a numeric value
  - they could enter input in an unexpected format, for example too many or too few values, or input that the program doesn't expect
  - they could abruptly terminate the program, for example with Ctrl+C
- The programmer may inadvertently introduce software bugs:
  - array bounds may not be checked, leading to a segmentation fault and program crash
  - null or garbage pointers may be dereferenced, also leading to segmentation faults
  - memory leaks may accumulate and heap space may run out
  - there could be many other software defects that are not caught during testing

14.1.1.3. **Fault prevention:**

- *Fault prevention* is the process of writing software in a way that minimizes the number of faults that are introduced.
- Good ways to prevent faults including following the rules of good software engineering and OO design.
- Code reviews, where a team of programmers go over a critical piece of software line-by-line, is also a good way to ensure quality and detect design errors that may result in faults.

14.1.1.4. **Fault detection:**

- *Fault detection* is the process of discovering faults before the end-user does.
- Software faults are most often detected by *debugging* and *testing*.
- *Debugging* is an informal activity, where the programmer verifies the correct functioning of a portion of the program. Debugging is neither organized nor planned.
- *Testing* is a crucial stage of the software development life-cycle. Testers put together a test plan that addresses all the client requirements. A *test plan* consists of a suite of test cases that verify the correct functioning of a specific unit of software, under a variety of scenarios.
- The testing activity must include verifying the success paths and the failure paths through the software execution.
- It must also ensure that the different software components communicate correctly with each other, and that the system works together as a whole.
- The goal of testing is to uncover faults before the software reaches the client and end-user.

14.1.1.5. **Fault tolerance:**

- *Fault tolerance* is a program's ability to keep running in the presence of faults, by detecting and handling them at runtime.
- We ensure fault tolerance in our programs with thorough error handling, including inline error handling and exception handling.

14.1.2. Error handling

14.1.2.1. **What is *error handling*:**

- *Error handling* is the process of making sure that, once an error is detected, it does not interrupt the program's execution.
- It ensures that the error is either fixed or dealt with, so that its consequences have minimal impact on the rest of the program and do not interrupt its normal operation.
- The opposite of error handling is allowing the program to terminate unexpectedly and crash.
- Possible ways to handle errors include correcting the error and allowing the program to continue execution, or notifying the end-user of the problem, or terminating the program in a controlled manner.
- For example, the end-user may enter a numeric value outside the expected range. Checking the entered value and finding it out-of-range is an example of *error detection*. The corrective measures that are taken are an example of *error handling*. In this case, it could include prompting the user again until they enter a correct value, or replacing the incorrect value with a default correct one, before continuing with the normal operation of the program.

14.1.2.2. **Dealing with errors:**

- There are two approaches for dealing with errors detected during a program's execution: inline error handling and exception handling.
- With *inline error handling*, the error handling logic is intermixed with the program logic.
- With *exception handling*, errors are detected in one part of the program, but they are handled in a different part.

14.1.2.3. **What is *inline error handling*:**

- *Inline error handling* is a technique for basic error handling. It consists of alternating the program's regular instructions with verifying after each step whether an error occurred. If it does, the error handling logic ensures that the program's execution can continue normally.
- A typical example of inline error handling is replacing an out-of-range numeric input with a valid value within the `if`-statement that checks the entered value for correctness.
- Inline error handling is good for addressing minor errors. However, it can make the program more difficult to read, maintain, and debug.
- More crucially, there are many situations where inline error handling is simply inadequate. For example, the part of the program that detects the error may not have sufficient information to fully handle the error. In those situations, exception handling is the better choice.

14.1.2.4. **What is an *exception*:**

- An *exception* is an error that occurs infrequently during a program's execution.
- Exceptions are *not* the same as regular errors. They are an exceptional situation that occurs only occasionally in a program but can have a major impact on its execution.

14.1.2.5. **What is *exception handling*:**

- *Exception handling* (EH) is a more sophisticated technique for error handling. It is normally used to address exceptions, and not common errors.
- One of the main features of EH is that it creates a *separation* between error detection (where the error is found) and error handling (where the error is dealt with).
- The ability to delegate error handling to a different part of the program from error detection is often essential for maintaining correct design encapsulation.
- The separation of error handling from error detection requires a mechanism for *error reporting*. Error detection finds the error, error reporting flags it to the EH mechanism as an error, and error handling does the work to deal with it.
- When an exception is reported, the EH mechanism triggers an alternative control flow structure. As a result, the regular function-call-and-return process discussed in section 4.1.2 is bypassed.
- This alternative control flow structure can have unexpected consequences if our code is not designed with this possibility in mind. The related topic of stack unwinding is discussed later in this chapter.

14.2. EH mechanism

We discuss the overall exception handling mechanism and its components in C++.

14.2.1. Basic components

14.2.1.1. **Components of EH mechanism:**

- Three components work together to support the reporting and handling of an exception using the EH mechanism: the `try` block, the `throw` statement, and the `catch` block.
- The `try` block performs the *error detection* part of the work, before an exception is generated.
- The `throw` statement does the *error reporting* and generates an exception.
- The `catch` block performs the *error handling*, after an exception is generated.

14.2.1.2. **The `try` block:**

- The `try` block contains code that may potentially detect an error and generate an exception.
- If we think that a portion of our code, or a function called within it, may need to report an exception at some point, we must enclose that code within a `try` block.
- Each `try` block must be immediately followed by a set of one or more `catch` blocks.

14.2.1.3. **The `catch` block:**

- The `catch` block contains code that handles an exception that is reported inside a `try` block.
- Every `try` block must be followed by one or more `catch` blocks, each one handling a different type of exception that can be generated by the `try` block.
- Each `catch` block takes a single parameter. Every exception is generated with a parameter, which we call the *exception parameter*. The first `catch` block that takes a parameter of the matching data type is the block that handles the exception.
- Only one `catch` block handles a generated exception, and once the `catch` block has finished executing, the exception is considered resolved. The program control flow continues at the first statement immediately following the set of `catch` blocks.

14.2.1.4. **The `throw` statement**:

- The `throw` statement is used to generate an exception to report an error.
- It must be enclosed within a `try` block. The `try` block may be situated directly in the same function as the `throw` statement, or indirectly in a calling function.
- A `throw` statement takes one exception parameter with a data type that's used to select a matching `catch` block.
- A `throw` statement *immediately terminates the `try` block in which it's located*. There are no function returns, and any statements following the `throw` are ignored. The EH mechanism performs a jump operation from the `throw` statement directly to one of the `catch` blocks that follow the terminated `try` block.
- If a `throw` statement generates an exception but is not located inside a `try` block, or if the `throw` parameter does not match any of the `catch` block parameter data types, then the exception cannot be caught or handled. An uncaught exception causes the program to terminate immediately.

14.2.1.5. **Exception specifications:**

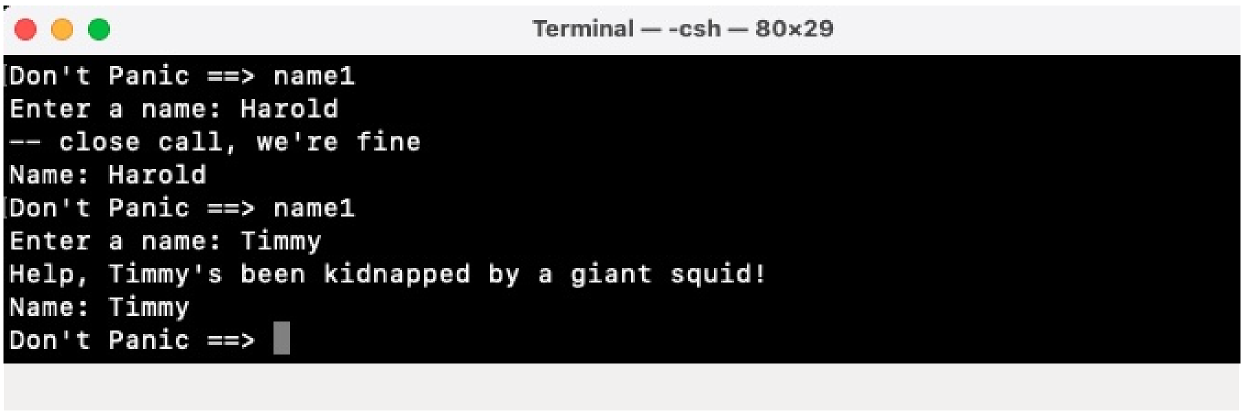
- The *exception specification* of a function defines the set of parameter data types that a `throw` statement inside that function may use, when it generates an exception.
- When we provide a function implementation, we can choose to declare which exception parameter data types the function is allowed to throw.
- The purpose of an exception specification is to communicate to our class users which kinds of exceptions their code should be prepared to handle.
- An exception specification is declared between the function prototype and the function body.
- For example:
  - if we define a `DivByZeroError` class, a function `foo()` that may generate an exception with a parameter that's either an integer or a `DivByZeroError` object would appear as:
    ```
    int foo(int x) throw(int, DivByZeroError) { /*function body*/ }
    ```
  - a function that may not generate any exceptions at all would appear as:
    ```
    int foo(int x) throw() { /*function body*/ }
    ```
  - a function that may generate exceptions of any data type would appear as:
    ```
    int foo(int x) { /*function body*/ }
    ```
- If an exception is generated with a parameter data type that's disallowed in the exception specification, the `terminate()` library function is called, and the program terminates.

14.2.1.6. **The C++ standard library `exception` class:**

- The C++ standard library provides an `exception` class that can be used as a base class for our own user-defined exception classes.
- The `exception` class constructor takes a string parameter that's used to describe the error.
- The `what()` member function is used on an `exception` object to retrieve the string.

14.2.2. Coding example: Exception in the same function

```
int main()
{
  string someName = "abracadabra";
  someName = enterName();
  cout << "Name: " << someName << endl;
  return 0;
}

string enterName()
{
  string s;
  cout<<"Enter a name: ";
  cin >> s;
  try {
    if (s == "Timmy") {
      throw "Help, Timmy's been kidnapped by a giant squid!";
    }
    cout << "-- close call, we're fine" << endl;
  }
  catch(const char* error) {
    cout << error << endl;
  }
  return s;
}
```


Terminal — -csh — 80×29

```
Don't Panic ==> name1
Enter a name: Harold
-- close call, we're fine
Name: Harold
Don't Panic ==> name1
Enter a name: Timmy
Help, Timmy's been kidnapped by a giant squid!
Name: Timmy
Don't Panic ==> 
```

Program-14.1: Exception in the same function

Program purpose:

- Program-14.1 shows the first of three variations of the same program. In this version, an exception is generated in the same function where the `try` block is located.
- The program uses the `enterName()` function to prompt the user to enter a name. If the name is `"Timmy"`, it throws an exception. Otherwise, the program continues normally.

Lines 1-7:

- These lines show the implementation of the `main()` function.
- Line 3 declares a string variable called `someName` and sets it to an initial value.
- Line 4 calls the `enterName()` function to read a name from the end-user, and it stores the returned value into the `someName` variable.
- Line 5 prints out the `someName` variable.

Lines 9-24:

- These lines show the implementation of the `enterName()` function.
- Lines 12-13 prompt the user to enter a name, which is stored in local variable `s`.
- Lines 14-19 contain the `try` block where we suspect an exception might be generated.
- Line 15 checks if the entered name is `"Timmy"`.
- If so, line 16 throws an exception using a literal string as parameter. Literal strings in C and C++ are considered to be a constant character pointer data type (`const char*`), so that's the parameter data type with which the exception is generated.
- Lines 20-22 show the `catch` block that's associated with the `try` block on lines 14-19. It takes a constant character pointer data type as parameter, so it can catch the exception generated on line 16.
- If an exception is generated on line 16, it immediately terminates the `try` block. The control flow jumps directly from line 16 to the `catch` block on line 20, then line 21 prints out the value of the parameter.
- In this case, line 18 is never executed because the remainder of the `try` block after line 16 is completely ignored.
- Line 23 returns the user-entered name to the calling function.

Program execution:

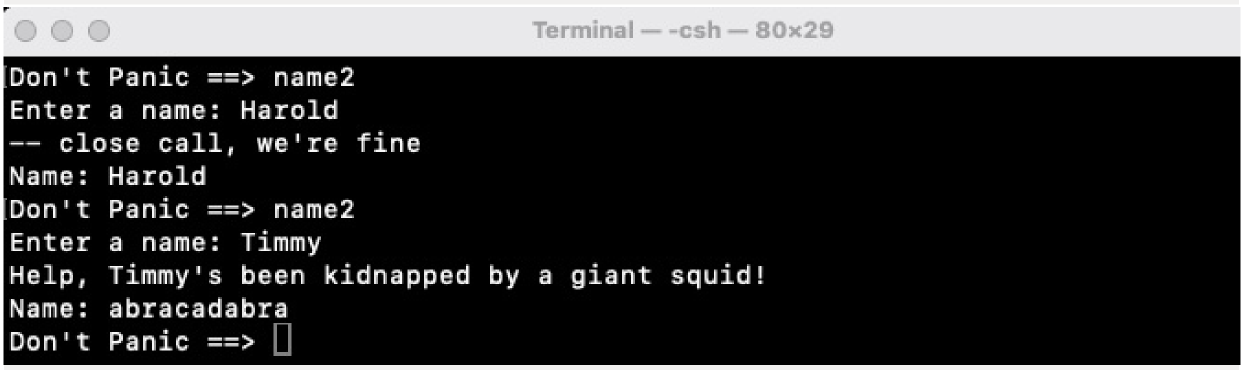
- The program output shows two executions of the program.
- In the first execution:
  - the user enters the name `"Harold"` on line 13; since the condition on line 15 is false, no exception is generated; the control flow continues on to execute line 18 and then line 23

  - the name entered by the end-user on line 13 is returned to the `main()` function on line 23, and the returned name is printed out on line 5

- In the second execution:
  - the user enters the name `"Timmy"` on line 13; since the condition on line 15 is true, an exception is generated on line 16; the control flow jumps to the `catch` block on line 20, since it's defined with a parameter data type that matches the `throw` statement parameter
  - the `catch` block is executed, and the parameter value is printed on line 21; then the control flow continues on line 23 after the `catch` block
  - the name entered by the end-user on line 13 is returned to the `main()` function on line 23, and the returned name is printed out on line 5

14.2.3. Coding example: Exception in a directly called function

```
int main()
{
  string someName = "abracadabra";

  try {
    someName = enterName();
    cout <<"-- close call, we're fine"<<endl;
  }
  catch(const char* error) {
    cout << error <<endl;
  }

  cout << "Name: " << someName << endl;

  return 0;
}

string enterName()
{
  string s;
  cout<<"Enter a name: ";
  cin >> s;

  if (s == "Timmy") {
    throw "Help, Timmy's been kidnapped by a giant squid!";
  }

  return s;
}
```



```
Don't Panic ==> name2
Enter a name: Harold
-- close call, we're fine
Name: Harold
Don't Panic ==> name2
Enter a name: Timmy
Help, Timmy's been kidnapped by a giant squid!
Name: abracadabra
Don't Panic ==> 
```

Program-14.2: Exception in a directly called function

Program purpose:

- Program-14.2 shows the second of three variations of the same program. In this version, an exception is generated in a function that's directly called from inside the `try` block.
- The program uses the `enterName()` function to prompt the user to enter a name. If the name is `"Timmy"`, it throws an exception. Otherwise, the program continues normally.

Lines 1-16:

- These lines show the implementation of the `main()` function.
- Line 3 declares a string variable called `someName` and sets it to an initial value.
- Lines 5-8 contain the `try` block where we suspect an exception might be generated.
- Line 6 inside the `try` block calls the `enterName()` function to read a name from the end-user, and it stores the returned value into the `someName` variable.
- Lines 9-11 show the `catch` block that's associated with the `try` block on lines 5-8. It takes a constant character pointer data type as parameter, so it can catch the exception generated elsewhere in the program.
- Line 13 prints out the `someName` variable.

Lines 18-29:

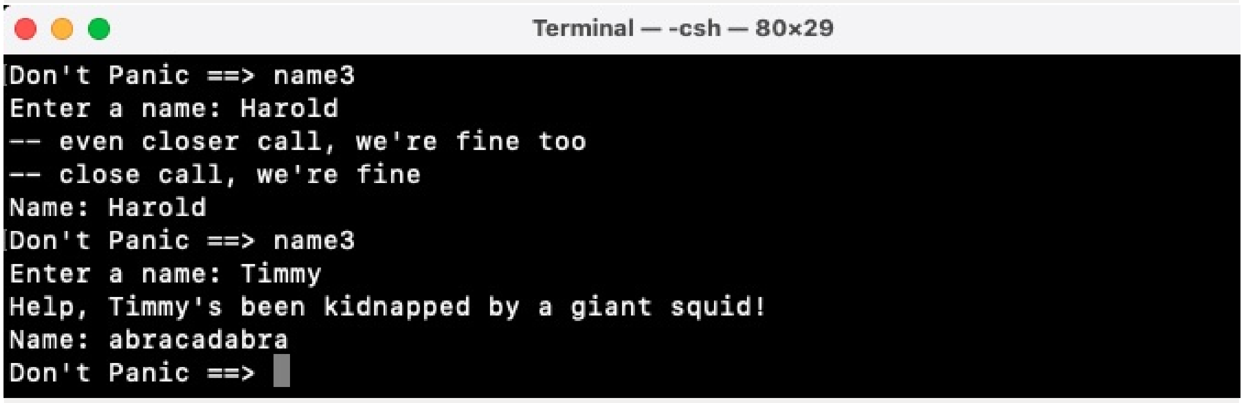
- These lines show the implementation of the `enterName()` function.
- Lines 21-22 prompt the user to enter a name, which is stored in local variable `s`.
- Line 24 checks if the entered name is `"Timmy"`.
- If so, line 25 throws an exception using a literal string as parameter.
- If an exception is generated, it immediately terminates the `try` block. The control flow jumps directly from line 25 to the `catch` block on line 9, then line 10 prints out the value of the parameter.
- In this case, lines 28 and 7 are never executed, because the remainder of both the `enterName()` function and the `try` block after the function call on line 6 are completely ignored.
- If no exception is generated, line 28 returns the user-entered name to the calling function.

Program execution:

- The program output shows two executions of the program.
- In the first execution:
  - the user enters the name `"Harold"` on line 22; since the condition on line 24 is false, no exception is generated; the control flow continues on to execute line 28
  - the name entered by the end-user on line 22 is returned to the `main()` function on line 28, and the returned name is printed out on line 13
- In the second execution:
  - the user enters the name `"Timmy"` on line 22; since the condition on line 24 is true, an exception is generated on line 25; the control flow jumps to the `catch` block on line 9, since it's defined with a parameter data type that matches the `throw` statement parameter
  - the `catch` block is executed, and the parameter value is printed on line 10; then the control flow continues on line 13 after the `catch` block
  - because line 28 is never executed, the name entered by the end-user on line 22 is not returned to the `main()` function; instead, line 13 prints out the `someName` variable value that was initialized on line 3

14.2.4. Coding example: Exception in an indirectly called function

```cpp
int main()
{
  string someName = "abracadabra";

  try {
    someName = enterName();
    cout << "-- close call, we're fine" << endl;
  }
  catch(const char* error) {
    cout << error <<endl;
  }

  cout << "Name: " << someName << endl;

  return 0;
}

string enterName()
{
  string s;
  cout<<"Enter a name: ";
  cin >> s;

  checkName(s);
  cout << "-- even closer call, we're fine too" << endl;

  return s;
}

void checkName(string name)
{
  if (name == "Timmy") {
    throw "Help, Timmy's been kidnapped by a giant squid!";
  }
}
```



```
Don't Panic ==> name3
Enter a name: Harold
-- even closer call, we're fine too
-- close call, we're fine
Name: Harold
Don't Panic ==> name3
Enter a name: Timmy
Help, Timmy's been kidnapped by a giant squid!
Name: abracadabra
Don't Panic ==>
```

Program-14.3: Exception in an indirectly called function

Program purpose:

- Program-14.3 shows the third of three variations of the same program. In this version, an exception is generated in a function that's indirectly called from inside the `try` block.
- The program uses the `enterName()` function to prompt the user to enter a name. The `enterName()` function calls `checkName()` to verify the entered name. If the name is `"Timmy"`, it throws an exception. Otherwise, the program continues normally.

Lines 1-16:

- These lines show the implementation of the `main()` function. It is identical to Program-14.2, and the same explanations apply.

Lines 18-28:

- These lines show the implementation of the `enterName()` function.
- Lines 21-22 prompt the user to enter a name, which is stored in local variable `s`.
- Line 24 calls the `checkName()` function with the user-entered name as a parameter to check if we have found Timmy.
- Line 27 returns the user-entered name to the calling function.

Lines 30-35:

- These lines show the implementation of the `checkName()` function.
- Line 32 checks if the user-entered name is `"Timmy"`.
- If so, line 33 throws an exception using a literal string as parameter.
- If an exception is generated on line 33, it immediately terminates the `try` block. The control flow jumps directly from line 33 to the `catch` block on line 9, then line 10 prints out the value of the parameter.
- In this case, lines 25, 27, and 7 are ever executed, because the remainder of both the `enterName()` function after the function call on line 24 and the rest of the `try` block after line 6 are completely ignored.

Program execution:

- The program output shows two executions of the program.
- In the first execution:
  - the user enters the name `"Harold"` on line 22; since the condition on line 32 is false, no exception is generated; the control flow continues on to execute lines 25 and 27
  - the name entered by the end-user on line 22 is returned to the `main()` function on line 27, and the returned name is printed out on line 13
- In the second execution:
  - the user enters the name `"Timmy"` on line 22; since the condition on line 32 is true, an exception is generated on line 33; the control flow jumps to the `catch` block on line 9, since it's defined with a parameter data type that matches the `throw` statement parameter
  - the `catch` block is executed, and the parameter value is printed on line 10; then the control flow continues on line 13 after the `catch` block
  - because line 27 is never executed, the name entered by the end-user on line 22 is not returned to the `main()` function; instead, line 13 prints out the `someName` variable value that was initialized on line 3

14.2.5. Structuring the EH code

14.2.5.1. Positioning the `try` block:

- Deciding on what parts of a program to include inside a `try` block may have significant consequences for the program's behaviour.
- It's crucial to remember that all the code following a `throw` statement, or a call to a function that contains a `throw` statement, whether directly or indirectly, may never execute.
- There is no one-size-fits-all solution to selecting what code to include in a `try` block. But it's an important decision in structuring our code to get the correct behaviour.

14.2.5.2. Example: placing a loop inside a `try` block:

- Let's assume that we have a `for`-loop that processes a collection of objects.
- If we put the entire loop inside a `try` block, then any exception that's generated will terminate the entire loop. This means that all further iterations of the loop will *not* execute, and any remaining elements in the collection will not be processed.
- In some programs, that's exactly the behaviour that we want. But we have to make sure we understand why this happens.

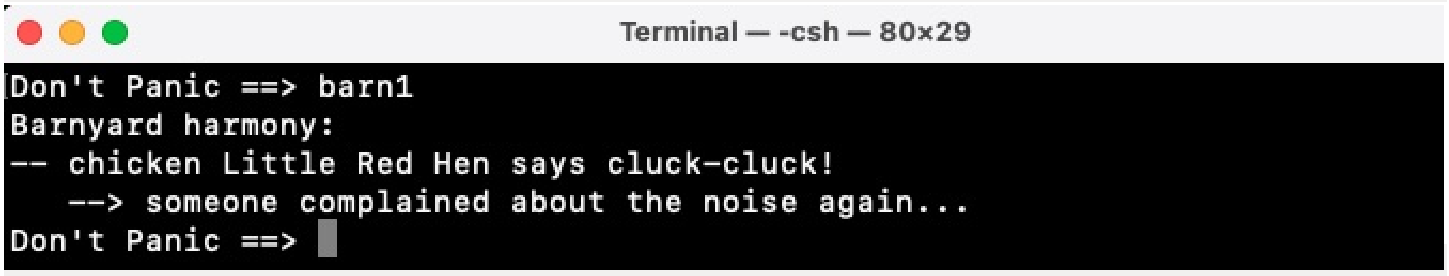
14.2.5.3. Example: placing a statement inside a `try` block:

- Let's assume again that we have a `for`-loop that processes a collection of objects.
- If we put a `try` block *inside* the loop, then any exception that's generated will terminate only the current iteration, and *not* the entire loop. This means that, after the exception is handled by a `catch` block, all the remaining elements in the collection will be processed.
- Again, in some programs, that's the desired behaviour. We have to make smart choices about where to position the `try` block so that we get the program behaviour we want.

14.2.6. Coding example: Exception that terminates a loop

```cpp
class Animal
{
  public:
    Animal(string n="Fluffy") : name(n) { }
    virtual ~Animal() { }
    virtual void sing() = 0;
  protected: string name;
};

class Bird : public Animal
{
  public:
    Bird(string n="") : Animal(n) { }
    virtual ~Bird() { }
    virtual void sing()  {cout<< "-- bird "<<name<<" says tweet-tweet!"<<endl;}
};

class Chicken : public Bird
{
  public:
    Chicken(string n="") : Bird(n) { }
    virtual ~Chicken() { }
    virtual void sing()  {cout<< "-- chicken "<<name<<" says cluck-cluck!"<<endl;}
};
```

```
class Cat : public Animal
{
  public:
    Cat(string n="") : Animal(n) { }
    virtual ~Cat() { }
    virtual void sing()  {cout<< "-- cat "<<name<<" says meow!"<<endl;}
};

class Pig : public Animal
{
  public:
    Pig(string n="") : Animal(n) { }
    virtual ~Pig() { }
    virtual void sing()  {throw "Pigs don’t sing !!!";}
};

int main()
{
  Chicken*  redHen   = new Chicken("Little Red Hen");
  Pig*      wilbur   = new Pig("Wilbur");
  Cat*      lady     = new Cat("Lady");

  vector<Animal*> barnyard;
  barnyard.push_back(redHen);
  barnyard.push_back(wilbur);
  barnyard.push_back(lady);

  cout << "Barnyard harmony:" << endl;
  try {
    for (int i=0; i<barnyard.size(); ++i) {
      barnyard[i]->sing();
    }
  }
  catch(const char* error) {
    cout << "   --> someone complained about the noise again... " << endl;
  }

  for (int i=0; i<barnyard.size(); ++i) {
    delete barnyard[i];
  }

  return 0;
}
```



```
Don't Panic ==> barn1
Barnyard harmony:
-- chicken Little Red Hen says cluck-cluck!
   --> someone complained about the noise again...
Don't Panic ==> 
```

Program-14.4: Exception that terminates a loop

Program purpose:

- Program-14.4 shows the first of two variations of the same program. Both versions loop over an animals collection and call the `sing()` member function for each element. The member function for the second element generates an exception.
- In this version, the entire `for`-loop is included inside the `try` block.

Lines 1-39:

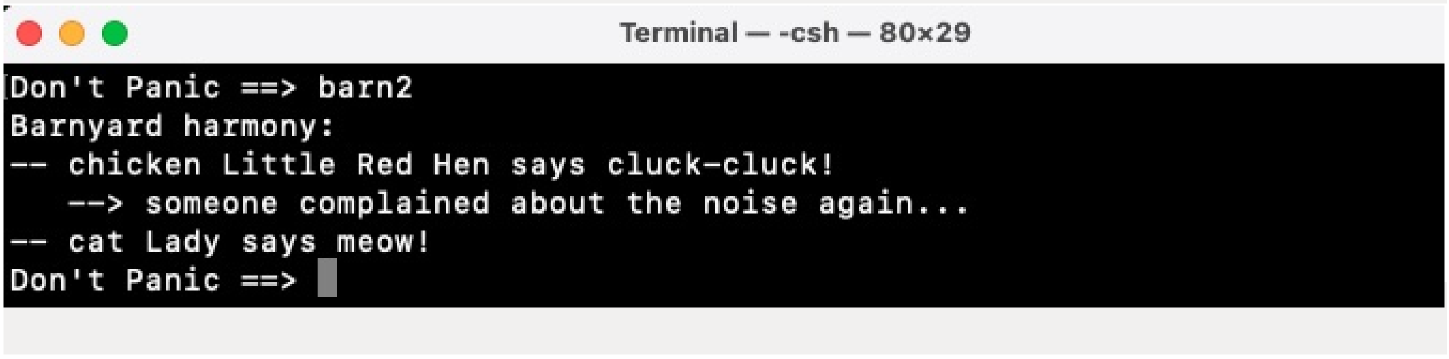
- These lines show the class definitions for `Animal`, `Bird`, `Chicken`, `Cat`, and `Pig`.
- The `Animal` base class is abstract, and the others are concrete and derived from `Animal`.
- Each concrete class provides an implementation for the `sing()` member function.
- Line 38 shows that the `Pig` class's `sing()` member function throws an exception.

Lines 41-67:

- These lines show the implementation of the `main()` function.
- Lines 43-45 create three concrete animal objects. Line 47 declares an STL `vector` of `Animal` pointers, and lines 48-50 add the objects to the `vector`.
- Lines 53-57 contain the `try` block where we suspect an exception might be generated.
- Inside the `try` block, on lines 54-56, we see the loop over the animals collection that calls the `sing()` member function for each element on line 55.
- The first element in the `vector` is a `Chicken` object, and we see from the program output that its `sing()` member function is called and executes correctly.
- The second element in the `vector` is a `Pig` object. Its `sing()` member function is called and generates an exception on line 38.
- When the exception is generated, it immediately terminates the `try` block. The control flow jumps directly from line 38 to the `catch` block on line 58, then line 59 prints out a message.
- Once the `catch` block finishes executing on line 60, the control flow continues to line 62.
- Because of the exception generated while processing the `Pig` object, the loop on lines 54-56 never continues to the next iteration that processes the `Cat` object, as we see from the program output.

14.2.7. Coding example: Exception that terminates one iteration

```cpp
int main()
{
  Chicken*   redHen    = new Chicken("Little Red Hen");
  Pig*       wilbur    = new Pig("Wilbur");
  Cat*       lady      = new Cat("Lady");

  vector<Animal*> barnyard;
  barnyard.push_back(redHen);
  barnyard.push_back(wilbur);
  barnyard.push_back(lady);

  cout << "Barnyard harmony:" << endl;
  for (int i=0; i<barnyard.size(); ++i) {
    try { barnyard[i]->sing(); }
    catch(const char* error) {
      cout << "   --> someone complained about the noise again... " << endl;
    }
  }

20   for (int i=0; i<barnyard.size(); ++i)
21     delete barnyard[i];

23   return 0;
24 }
```



```
Don't Panic ==> barn2
Barnyard harmony:
-- chicken Little Red Hen says cluck-cluck!
   --> someone complained about the noise again...
-- cat Lady says meow!
Don't Panic ==>
```

Program-14.5: Exception that terminates one iteration

Program purpose:

- Program-14.5 shows the second of two variations of the same program. Both versions loop over an animals collection and call the `sing()` member function for each element. The member function for the second element generates an exception.
- In this version, the `try` block is positioned inside the `for`-loop, and its only statement is the call to the `sing()` member function.
- The class definitions are identical to Program-14.4.

Lines 3-10:

- These lines are identical to Program-14.4. They create three concrete animal objects, declare an STL `vector` of `Animal` pointers, and add the animal objects to the `vector`.

Lines 13-18:

- These lines contain the `for`-loop that calls the `sing()` member function for each element.
- Line 14 shows the `try` block where we suspect an exception might be generated. It contains a single statement: the call to the `sing()` member function for the current `vector` element.
- The first element in the `vector` is a `Chicken` object, and we see from the program output that its `sing()` member function is called and executes correctly.
- The second element in the `vector` is a `Pig` object. Its `sing()` member function is called and generates an exception, as we see from line 38 of Program-14.4.
- The exception immediately terminates the `try` block. The control flow jumps directly from the `throw` statement to the `catch` block on line 15, then line 16 prints out a message.
- Once the `catch` block finishes executing on line 17, the control flow continues to the next iteration of the `for`-loop.
- Despite the exception generated while processing the `Pig` object, the loop on lines 13-18 is *not* terminated. It continues to the next iteration that processes the next `vector` element.
- The third element in the `vector` is a `Cat` object, and we see from the program output that its `sing()` member function is called and executes correctly.

14.3. EH features

We discuss some features of the EH mechanism, including the use of multiple `catch` blocks and re-throwing an exception.

14.3.1. Multiple catch blocks

14.3.1.1. **The `throw` statement parameter:**

- A new exception is generated using a `throw` statement with one exception parameter.
- This parameter is necessary in order to select the correct `catch` block to handle the exception.

14.3.1.2. **Selecting a `catch` block:**

- A `try` block must be followed by one or more `catch` blocks, each one taking an exception parameter of a different data type.
- When an exception is generated, the data type of the exception parameter is compared with the `catch` block parameter types. The `catch` blocks are checked linearly, starting at the first one following the `try` block.
- The first `catch` block that takes a parameter matching the exception parameter data type is the one selected to handle the exception.
- If no `catch` block matches the exception parameter data type, the exception cannot be handled, and the program terminates.

14.3.1.3. **Executing a `catch` block:**

- A single `catch` block is selected to handle an exception.
- Once the `catch` block has finished executing, the exception is considered resolved, and all the other `catch` blocks are ignored.
- The program control flow resumes at the first instruction following the last `catch` block.

14.3.1.4. **The catch-all block:**

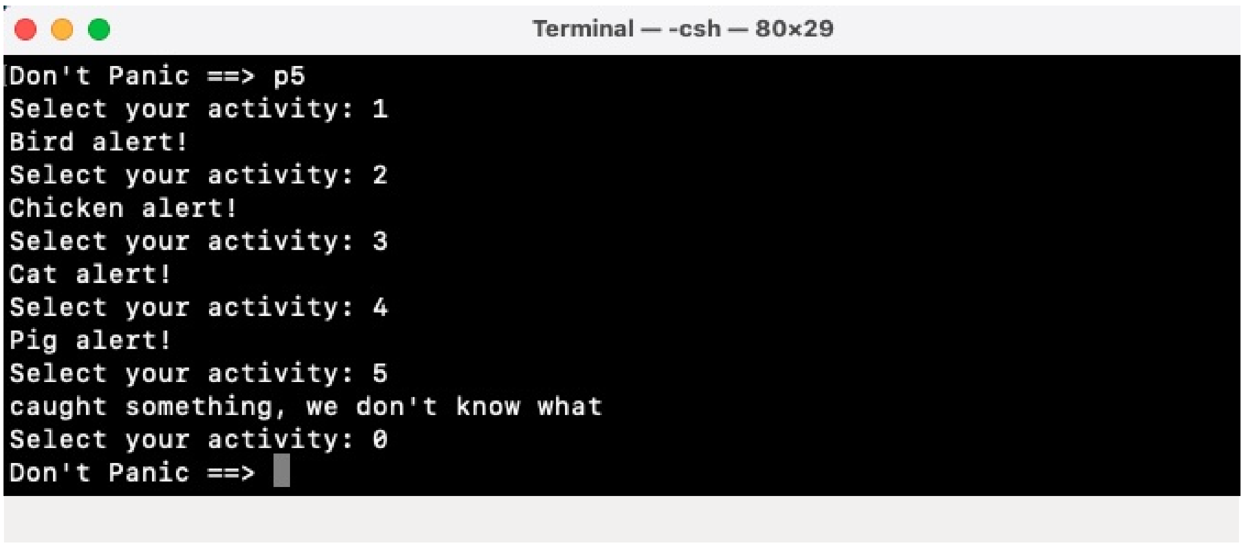
- A *catch-all block* can handle an exception parameter of any data type.
- It is declared using an ellipsis (three consecutive dots) as a parameter: `catch(...){ }`
- Because the parameter of a catch-all block cannot be given a name, it cannot be used in the body of the `catch` block.

14.3.1.5. **Order of `catch` blocks:**

- The *order* of the `catch` blocks is important.
- For example, if a catch-all block is positioned first, it will always be selected because all exception parameter data types will match it. So a catch-all block should always be positioned last.
- In programs with an inheritance hierarchy of exception classes, the order also matters:
  - if a `catch` block with a base class parameter is placed first, it will be selected for any exception parameter that matches either the base class or any of its derived classes
  - that's because the derived class objects are considered to be a kind of base class object
  - a `catch` block with a derived class parameter should always be positioned *before* a `catch` block with a base class parameter

14.3.2. Coding example: Multiple catch blocks

```cpp
int main()
{
  Bird    birtrude;
  Chicken redHen;
  Cat     lady;
  Pig     wilbur;
  int     choice;

  while (1) {
    cout << "Select your activity: ";
    cin  >> choice;

    try {
      if (choice == 0) {
        break;
      }
      else if (choice == 1) {
        throw birtrude;
      }
      else if (choice == 2) {
        throw redHen;
      }
      else if (choice == 3) {
        throw lady;
      }
      else if (choice == 4) {
        throw wilbur;
      }
      else {
        throw "wrong choice";
      }
    }
    catch(Chicken& c) {
      cout<<"Chicken alert!"<<endl;
    }
    catch(Bird& b) {
      cout<<"Bird alert!"<<endl;
    }
    catch(Cat& c) {
      cout<<"Cat alert!"<<endl;
    }
    catch(Pig& p) {
      cout<<"Pig alert!"<<endl;
    }
    catch(...) {
      cout<<"caught something, we don't know what"<<endl;
    }
  }

  return 0;
}
```


Terminal — -csh — 80×29

```
Don't Panic ==> p5
Select your activity: 1
Bird alert!
Select your activity: 2
Chicken alert!
Select your activity: 3
Cat alert!
Select your activity: 4
Pig alert!
Select your activity: 5
caught something, we don't know what
Select your activity: 0
Don't Panic ==>
```

Program-14.6: Multiple catch blocks

Program purpose:

- Program-14.6 demonstrates the use of multiple `catch` blocks for the same `try` block.
- The program prompts the end-user to select an option that generates different exceptions.
- The class definitions are identical to Program-14.4.

Lines 9-48:

- These lines show an infinite loop that repeatedly prompts the end-user for a selection, and each selection results in a different exception being generated.

Lines 13-32:

- These lines contain the `try` block where we suspect an exception might be generated.
- Depending on the user's numeric selection at each iteration of the loop, an exception with different exception parameter data types is generated within the `try` block.

Lines 33-47:

- These lines show the five `catch` blocks that are associated with the `try` block on lines 13-32. Each `catch` block takes a different type of exception parameter.
- The `catch` block that takes a `Bird` parameter on lines 36-38 is placed *after* the `catch` block that takes an object of its derived class `Chicken` as parameter on lines 33-35. If the `Bird` parameter `catch` block was positioned first, all exceptions generated with a `Chicken` parameter would be handled by the `Bird` parameter `catch` block.
- Lines 45-47 show the catch-all block that handles an exception with a parameter data type that does not match any of the other `catch` blocks.

Program execution:

- We see from the program output that each type of exception is generated and handled.
- Each exception is caught by the first `catch` block with a data type matching the exception parameter. The `catch` block executes and handles the exception.
- Once the selected `catch` block terminates execution, the exception is considered resolved. No other `catch` blocks are tested for a match, and none of them execute.
- After an exception is handled, the control flow continues, in this case from the `while`-loop closing brace on line 48 back to the loop header on line 9 for the next iteration.

14.3.3. Re-throwing an exception

14.3.3.1. Handling an exception multiple times:

- Once a generated exception has been handled, it is considered resolved, and the program continues executing. But what if multiple parts of the program need to handle the exception?
- For example:
  - assume that we have a `try` block in function `a()`; within that `try` block, function `a()` calls function `b()`, which calls function `c()`, and `c()` generates an exception
  - the exception generated in function `c()` terminates the `try` block in function `a()` and jumps to a matching `catch` block in `a()`; in the process, the stack frames for both `b()` and `c()` are popped off the function call stack, and any remaining code in both functions is bypassed
  - if there was cleanup required in functions `b()` and `c()`, for example memory to be deallocated or files to be closed, before jumping back to `a()`, it would never occur, and some important resources may become unreachable
- In the scenario described above, we need a mechanism to handle the exception multiple times, so that cleanup can be performed in different parts of the program.
- We call this mechanism an exception *re-throw*.
- The concept of *function cleanup* is similar to the object cleanup that a class destructor performs. When a program allocates resources, for example dynamically allocating memory or opening files, the resources must be released when they are no longer needed. This is called the ***cleanup***, whether it's done explicitly by a function, or implicitly by a destructor.

14.3.3.2. How a re-throw works:

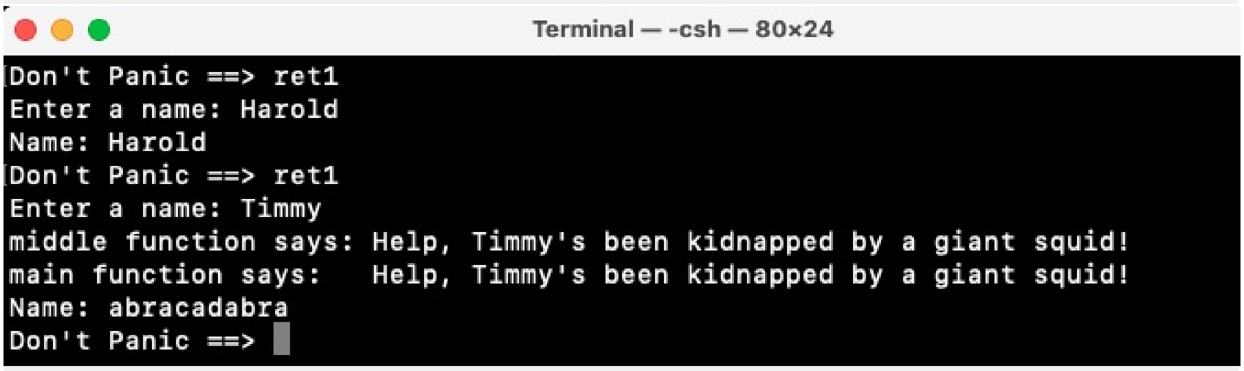
- If we need to handle an exception in multiple called functions, we need `try` and `catch` blocks in each of the functions.
- For example:
  - assume that function `a()` has a `try` block that contains a call to function `b()`; and `b()` has a `try` block that calls function `c()`; and `c()` has a `try` block that generates an exception
  - in this case, the `throw` statement in `c()` only terminates the `try` block in that function, and the control flow jumps to a matching `catch` block in `c()`, which handles the exception
  - if function `b()` also needs to handle the error, then the `catch` block in `c()` must *re-throw* the exception
  - this re-throw is considered a different exception and is treated accordingly; it terminates the `try` block in function `b()`, and the control flow jumps to a matching `catch` block in `b()`
  - similarly, if we want function `a()` to also handle the exception, then the `catch` block in `b()` must again *re-throw* the exception, which terminates the `try` block in function `a()`, and the control flow jumps to a matching `catch` block in `a()`
- By giving every called function the opportunity to clean up after itself, each one is able to deallocate the resources that they have reserved, and correct encapsulation is preserved.

14.3.3.3. Approaches for re-throwing an exception:

- An exception re-throw always occurs by using a new `throw` statement inside a `catch` block.
- There are two options for re-throwing an exception: we can re-throw the same exception, or throw a brand new one.
- When a `catch` block re-throws the same exception, its `throw` statement automatically reuses the same exception parameter.
- When a `catch` block throw a new exception, a new exception parameter must be specified.

14.3.4. Coding example: Re-throwing the same exception

```
int main()
{
  string someName = "abracadabra";

  try {
    someName = enterName();
  }
  catch(const char* error) {
    cout << "main function says:    " << error << endl;
  }

  cout << "Name: " << someName << endl;

  return 0;
}

string enterName()
{
  string s;
  cout << "Enter a name: ";
  cin  >> s;

  try {
    checkName(s);
  }
  catch(const char* err) {
    cout << "middle function says: " << err << endl;
    throw;
  }

  return s;
}

void checkName(string name)
{
  if (name == "Timmy") {
    throw "Help, Timmy's been kidnapped by a giant squid!";
  }
}
```



```
Don't Panic ==> ret1
Enter a name: Harold
Name: Harold
Don't Panic ==> ret1
Enter a name: Timmy
middle function says: Help, Timmy's been kidnapped by a giant squid!
main function says:   Help, Timmy's been kidnapped by a giant squid!
Name: abracadabra
Don't Panic ==>
```

Program-14.7: Re-throwing the same exception

Program purpose:

- Program-14.7 shows the first of two variations of the previous examples Program-14.1 through Program-14.3. In this version, the same exception is re-thrown.
- The program generates an exception in the `checkName()` function. The exception is caught and handled in a `catch` block in the calling function `enterName()`, which then re-throws the same exception, which is caught and handled in `main()`.

Lines 1-15:

- These lines show the implementation of the `main()` function.
- Line 3 declares a string variable called `someName` and sets it to an initial value.
- Lines 5-7 contain the `try` block where we suspect an exception might be generated.
- Line 6 inside the `try` block calls the `enterName()` function to read a name from the end-user, and it stores the returned value into the `someName` variable.
- Lines 8-10 show the `catch` block that's associated with the `try` block on lines 5-7. It catches an exception that's generated from the `try` block or its called function.
- If an exception is generated in `enterName()` with a constant character pointer data type as parameter, it is caught and handled in the `catch` block on lines 8-10, and line 9 prints out the value of the exception parameter.
- Line 12 prints out the `someName` variable.

Lines 17-32:

- These lines show the implementation of the `enterName()` function.
- Lines 20-21 prompt the user to enter a name, which is stored in local variable `s`.
- From within a `try` block on lines 23-25, line 24 calls the `checkName()` function with the user-entered name as a parameter to check if we have found Timmy.
- The corresponding `catch` block on lines 26-29 catches an exception that's generated from the `try` block or its called function.
- If an exception is generated in `checkName()` with a constant character pointer data type as parameter, it is caught and handled in the `catch` block on lines 26-29. Line 27 prints out the value of the exception parameter, and line 28 re-throws the same exception, with the same exception parameter.
- The re-throw on line 28 terminates the `try` block in the calling function `main()`, and the control flow jumps directly from line 28 to the `catch` block on line 8. As a result, line 31 is never executed, because the remainder of the `enterName()` function after the re-throw on line 28 is completely ignored.
- If no exception is generated, line 31 returns the user-entered name to the calling function.

Lines 34-39:

- These lines show the implementation of the `checkName()` function.
- Line 36 checks if the user-entered name is `"Timmy"`.
- If so, line 37 throws an exception using a literal string as parameter.
- If an exception is generated on line 37, it immediately terminates the `try` block in the calling function `enterName()`. The control flow jumps from line 37 to the `catch` block on line 26.

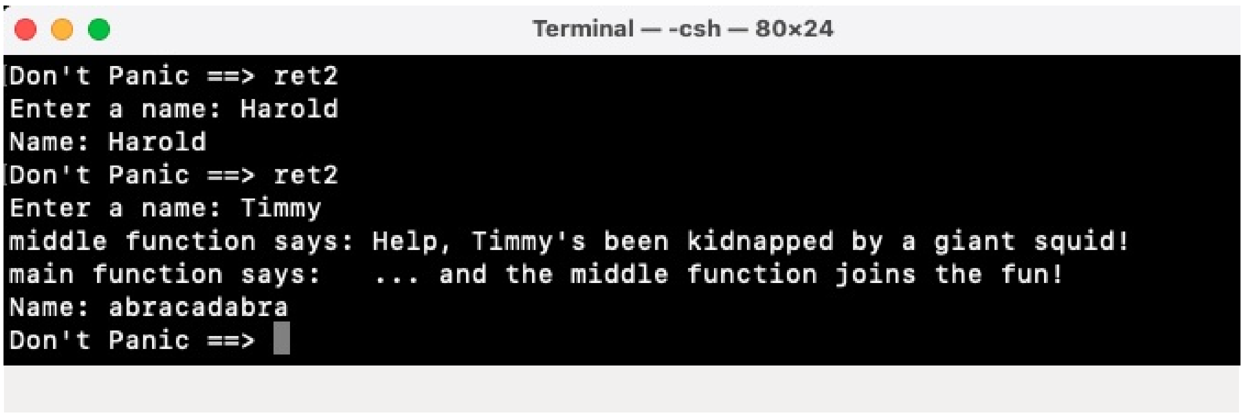
Program execution:

- The program output shows two executions of the program.
- In the first execution:
  - the user enters the name `"Harold"` on line 21; since the condition on line 36 is false, no exception is generated; the control flow continues on to execute line 31

  - the name entered by the end-user on line 21 is returned to the `main()` function on line 31, and the returned name is printed out on line 12
- In the second execution:
  - the user enters the name `"Timmy"` on line 21; since the condition on line 36 is true, an exception is generated on line 37; the control flow jumps to the `catch` block on line 26
  - the `catch` block is executed, and the exception parameter value is printed on line 27; then the same exception is re-thrown on line 28, and the control flow jumps to the `catch` block on line 8
  - the `catch` block is executed, and the exception parameter value is printed on line 9; then the control flow continues on line 12 after the `catch` block
  - because line 31 is never executed, the name entered by the end-user on line 21 is not returned to the `main()` function; instead, line 12 prints out the `someName` variable value that was initialized on line 3

14.3.5. Coding example: Re-throwing a new exception

```
 1 int main()
 2 {
 3   string someName = "abracadabra";

 5   try {
 6     someName = enterName();
 7   }
 8   catch(const char* error) {
 9     cout << "main function says:   " << error << endl;
10   }

12   cout << "Name: " << someName << endl;

14   return 0;
15 }

17 string enterName()
18 {
19   string s;
20   cout << "Enter a name: ";
21   cin  >> s;

23   try {
24     checkName(s);
25   }
26   catch(const char* err) {
27     cout << "middle function says: " << err << endl;
28     throw "... and the middle function joins the fun!";
29   }

31   return s;
32 }

34 void checkName(string name)
35 {
36   if (name == "Timmy") {
37     throw "Help, Timmy's been kidnapped by a giant squid!";
38   }
39 }
```



```
Don't Panic ==> ret2
Enter a name: Harold
Name: Harold
Don't Panic ==> ret2
Enter a name: Timmy
middle function says: Help, Timmy's been kidnapped by a giant squid!
main function says:   ... and the middle function joins the fun!
Name: abracadabra
Don't Panic ==> 
```

Program-14.8: Re-throwing a new exception

Program purpose:

- Program-14.8 shows the second of two variations of the previous examples Program-14.1 through Program-14.3. In this version, a new exception is re-thrown.
- The program generates an exception in the `checkName()` function. The exception is caught and handled in a `catch` block in the calling function `enterName()`, which then re-throws a different exception, which is caught and handled in `main()`.
- Both the `main()` and `checkName()` function implementations are identical to Program-14.7.

Lines 17-32:

- These lines show the implementation of the `enterName()` function, and it's nearly identical to Program-14.7.
- From within a `try` block on lines 23-25, line 24 calls the `checkName()` function with the user-entered name as a parameter to check if we have found Timmy.
- The corresponding `catch` block on lines 26-29 catches an exception that's generated from the `try` block or its called function.
- If an exception is generated in `checkName()`, it is caught and handled in the `catch` block on lines 26-29. Line 27 prints out the value of the exception parameter, and line 28 re-throws *a new exception*.
- The re-throw on line 28 terminates the `try` block in the calling function `main()`, and the control flow jumps directly from line 28 to the `catch` block on line 8.
- If no exception is generated, line 31 returns the user-entered name to the calling function.

Program execution:

- The program output shows two executions of the program.
- The first execution is identical to Program-14.7. The user enters the name `"Harold"`, and no exception is generated.
- In the second execution, the user enters the name `"Timmy"`, and an exception is generated on line 37. The control flow jumps to the `catch` block on line 26, and the exception parameter is printed on line 27. Then *a new exception with a new exception parameter* is re-thrown on line 28, and the control flow jumps to the `catch` block on line 8. The new exception parameter is printed on line 9, then the control flow continues on line 12 after the `catch` block.
- If we compare the program output from both Program-14.7 and Program-14.8, we see that they are different. When the exception parameter is printed on line 9 in the `main()` function of both programs, it's the exception generated from the `enterName()` function on line 28. In Program-14.7, the same exception parameter value is printed in both `catch` blocks on lines 27 and 9. In Program-14.8, a different exception parameter value is printed on line 9.

14.4. Stack unwinding

We discuss approaches for dealing with the function cleanup issues that arise when EH is used.

14.4.1. Concepts

14.4.1.1. **What is *stack unwinding*:**

- *Stack unwinding* is the process of dealing with the bypassed cleanup of called functions, from the function where an exception is generated through to the one where it's handled, when an exception transfers the program control flow from one part of the program to another.
- There are different approaches to ensure that the stack unwinding is done correctly, without loss of data or locking of resources. We discuss these approaches in this section.

14.4.1.2. **Characteristics of stack unwinding:**

- Stack unwinding is initiated when an exception is generated using a `throw` statement, which immediately terminates the `try` block in which it's contained.
- The `try` block may be in the same function where the exception is generated, or it may be in the calling function, or that function's calling function, up to any number of functions in the chain of function calls between the exception generation and the terminated `try` block.
- As soon as the exception is generated, every function in that chain is terminated. Their stack frames are popped off the function call stack, and the local variables inside them are destroyed. This can result in the loss of important data, or the locking of some essential resources because cleanup is not performed.

14.4.2. Approaches to resource cleanup

14.4.2.1. **Issues during stack unwinding:**

- The use of EH, which transfers control directly from a `throw` statement to a `catch` block, bypasses the regular function-call-and-return control structure of a program.
- As a result, functions do not get the chance to clean up after themselves. Local variables are lost, dynamically allocated memory may remain allocated, and the program can end up in an inconsistent state.
- There are strategies that can help with function cleanup, and we look at three of them next.

14.4.2.2. **Using an error object as the exception parameter:**

- With this approach, we create an error class that holds all the information requiring cleanup.
- The error class contains as many data members as there are individual resources that must be deallocated or released.
- When an exception is generated, a new instance of the error class is created and initialized with pointers to the specific resources to be cleaned up. The new error object is then used as the exception parameter.
- The `catch` block performs the cleanup required by each data member of the error object.
- This approach exemplifies bad encapsulation. It requires the `catch` block to have knowledge of data across multiple functions and objects, thus violating the principle of least privilege.

14.4.2.3. **Designing self-cleaning functions:**

- With this approach, every function is responsible for its own cleanup, so each one is implemented with its own `try`-`catch` block pair.

- If the function that generates an exception requires cleanup, then its `catch` block handles the exception by performing the required cleanup, and then re-throwing the exception.
- If each function in the call chain does the same, it can clean up its own resources and re-throw the exception, so that its calling function has the same opportunity.
- This is an example of good encapsulation, since each function only requires knowledge about its own resources.

14.4.2.4. **Designing self-cleaning objects:**

- With this approach, every object is responsible for its own cleanup, so each class provides a destructor that performs the required work.
- Destructors are always called on scope exit, including when `return()` or `exit()` or `throw()` are invoked. So object cleanup is performed automatically, as long as the destructors are correctly implemented.
- This is an example of good encapsulation, since each object only requires knowledge about its own resources.

14.4.3. **Coding example: Cleanup using error objects**

```
class Error
{
  public:
    Error(int* a) : arr(a) { }
    void cleanup() { delete [] arr; }
  private:  int* arr;
};

int main()
{
  try { initArray(); }
  catch(Error* err) {
    cout << "-- Input error! " << endl;
    err->cleanup();
    delete err;
  }
  return 0;
}

void initArray()
{
  int size;
  cout << "Enter the number of elements: ";
  cin  >> size;

  int* myArray = new int[size];
  cout<<"Enter the elements: " <<endl;
  for (int i=0; i<size; ++i) {
    cin >> myArray[i];
  }

  if (!cin.good()) {
    Error* tmpError = new Error(myArray);
    throw tmpError;
  }

37   cout<<endl<<"Array:"<<endl;
38   for (int i=0; i<size; ++i) {
39     cout << myArray[i] << " ";
40   }
41   cout<<endl;

43   delete [] myArray;
44 }
```

Program-14.9: Cleanup using error objects

Program purpose:

- Program-14.9 shows the first of three variations of a program that populates and prints an integer array with values entered by the end-user.
- In this version, the `initArray()` function dynamically allocates an integer array and populates it with user-entered values. If the user enters a non-numeric value, an exception is generated using an error object. The `main()` function handles the error by cleaning up the dynamically allocated array, which is contained in the error object.
- Because all three versions of this program have identical behaviour, from the user's point of view, we only show the program output for the third variation.

Lines 1-7:

- These lines show the `Error` class definition.
- Line 6 declares a data member that stores a dynamically allocated array requiring cleanup.
- Line 4 shows a constructor that initializes the data member from a parameter.
- Line 5 implements the `cleanup()` member function that deallocates the array stored in the data member.

Lines 9-18:

- These lines show the implementation of the `main()` function.
- Line 11 contains a `try` block that calls the `initArray()` function.
- Lines 12-16 show the `catch` block that handles an exception by printing an error message and calling the `Error` object parameter's `cleanup()` member function.

Lines 20-44:

- These lines show the implementation of the `initArray()` function.
- Line 24 reads the number of values to be entered by the end-user. Line 26 dynamically allocates an integer array, and lines 28-30 populate it with user-entered values.
- Line 32 tests the standard input stream object `cin` to see if an error occurred. If the user enters a non-numeric value, line 33 creates a new `Error` object and initializes it with the dynamically allocated array that requires cleanup. Line 34 generates an exception, using the new `Error` object as parameter.
- If an exception is generated on line 34, the control flow jumps to line 12, and the exception parameter is used to deallocate the array by calling its `cleanup()` member function.
- If no exception is generated on line 34, the array values are printed out on lines 38-40, and the array is deallocated on line 43.

14.4.4. Coding example: Cleanup with self-cleaning functions

```cpp
int main()
{
  try { initArray(); }
  catch(...) {
    cout << "-- Input error! " << endl;
  }
  return 0;
}

void initArray()
{
  int size;
  cout << "Enter the number of elements: ";
  cin  >> size;

  int* myArray = new int[size];
  cout<<"Enter the elements: " <<endl;
  try {
    for (int i=0; i<size; ++i) {
       myArray[i] = enterInt();
    }
  }
  catch(...) {
    delete [] myArray;
    throw;
  }

  cout<<endl<<"Array:"<<endl;
  for (int i=0; i<size; ++i) {
    cout << myArray[i] << " ";
  }
  cout<<endl;

  delete [] myArray;
}

int enterInt()
{
  int element;
  cin >> element;

  if (!cin.good()) {
    throw("Invalid input");
  }

  return element;
}
```

Program-14.10: Cleanup with self-cleaning functions

Program purpose:

- Program-14.10 shows the second of three variations of a program that populates and prints an integer array with values entered by the end-user.

- In this version, the `initArray()` function dynamically allocates an array and repeatedly calls the `enterInt()` function to read each array value from the end-user. If the user enters a non-numeric value, `enterInt()` generates an exception. Because `initArray()` performs the dynamic allocation of the array, the same function handles the exception by cleaning it up. It then re-throws the same exception for `main()` to handle as well.

Lines 1-8:

- These lines show the implementation of the `main()` function.
- Line 3 contains a `try` block that calls the `initArray()` function.
- Lines 4-6 show the `catch` block that handles an exception by printing out an error message.

Lines 10-35:

- These lines show the implementation of the `initArray()` function.
- Line 14 reads the number of values to be entered by the end-user, and line 16 dynamically allocates an integer array.
- Lines 18-22 contain a `try` block with a `for`-loop that calls the `enterInt()` function on line 20 to read a value from the end-user.
- Lines 23-26 show the `catch` block that handles an exception generated in the `enterInt()` function. Line 24 deallocates the array, and line 25 re-throws the same exception, which causes the control flow to jump to the `catch` block on line 4.
- If no exception is generated in `enterInt()`, the array values are printed out on lines 29-31, and the array is deallocated on line 34.

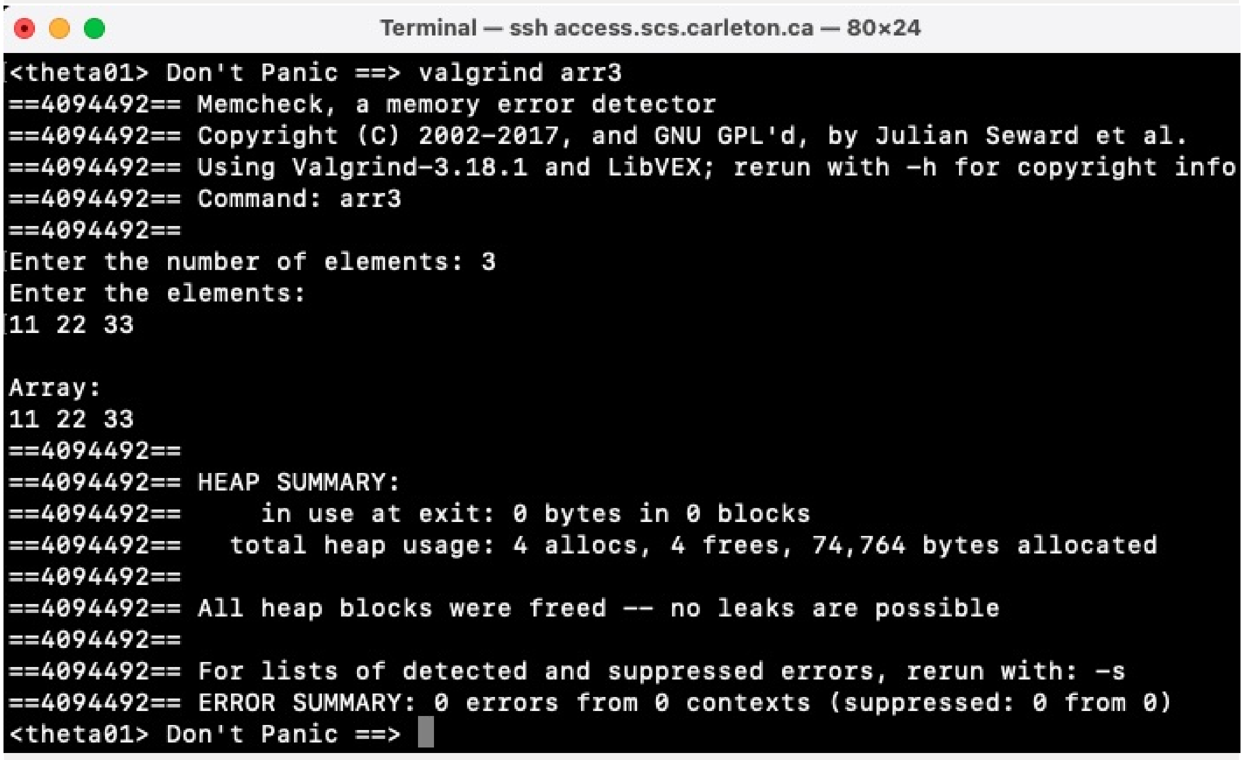
Lines 37-47:

- These lines show the implementation of the `enterInt()` function.
- Line 40 reads in a value from the end-user.
- Line 42 tests the standard input stream object `cin` to see if an error occurred. If the user enters a non-numeric value, line 43 generates an exception, and the control flow jumps to the `catch` block on line 23.
- If no exception is generated on line 43, line 46 returns the user-entered value to `initArray()`.

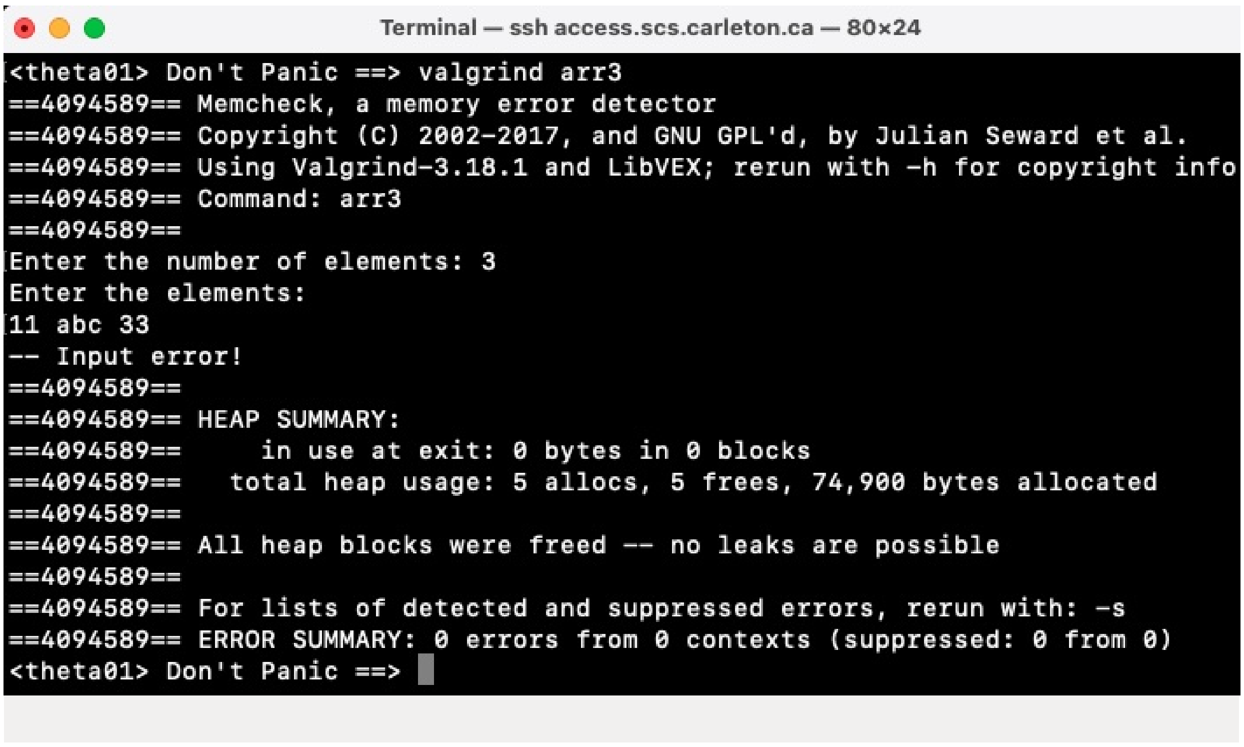
14.4.5. Coding example: Cleanup with self-cleaning objects

```cpp
class IntArray
{
  public:
    IntArray(int s=10) : size(s) { arr = new int[size]; }
    ~IntArray()                  { delete [] arr;       }
    int& operator[](int s)       { return arr[s];       }
  private:
    int  size;
    int* arr;
};

int main()
{
  try { initArray(); }
  catch(...) {
    cout << "-- Input error! " << endl;
  }
  return 0;
}
```

```
void initArray()
{
  int size;
  cout << "Enter the number of elements: ";
  cin  >> size;

  IntArray myArray(size);
  cout<<"Enter the elements: " <<endl;
  for (int i=0; i<size; ++i) {
    myArray[i] = enterInt();
  }

  cout<<endl<<"Array:"<<endl;
  for (int i=0; i<size; ++i) {
    cout << myArray[i] << " ";
  }
  cout<<endl;
}

int enterInt()
{
  int element;
  cin >> element;

  if (!cin.good()) {
    throw("Invalid input");
  }

  return element;
}
```



```
<theta01> Don't Panic ==> valgrind arr3
==4094492== Memcheck, a memory error detector
==4094492== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4094492== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4094492== Command: arr3
==4094492==
Enter the number of elements: 3
Enter the elements:
11 22 33

Array:
11 22 33
==4094492==
==4094492== HEAP SUMMARY:
==4094492==     in use at exit: 0 bytes in 0 blocks
==4094492==   total heap usage: 4 allocs, 4 frees, 74,764 bytes allocated
==4094492==
==4094492== All heap blocks were freed -- no leaks are possible
==4094492==
==4094492== For lists of detected and suppressed errors, rerun with: -s
==4094492== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
<theta01> Don't Panic ==>
```



```
<theta01> Don't Panic ==> valgrind arr3
==4094589== Memcheck, a memory error detector
==4094589== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4094589== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4094589== Command: arr3
==4094589==
Enter the number of elements: 3
Enter the elements:
11 abc 33
-- Input error!
==4094589==
==4094589== HEAP SUMMARY:
==4094589==     in use at exit: 0 bytes in 0 blocks
==4094589==   total heap usage: 5 allocs, 5 frees, 74,900 bytes allocated
==4094589==
==4094589== All heap blocks were freed -- no leaks are possible
==4094589==
==4094589== For lists of detected and suppressed errors, rerun with: -s
==4094589== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
<theta01> Don't Panic ==> 
```

Program-14.11: Cleanup with self-cleaning objects

Program purpose:

- Program-14.11 shows the third of three variations of a program that populates and prints an integer array with values entered by the end-user.
- In this version, the `initArray()` function declares an `IntArray` object to store the user-entered values and repeatedly calls the `enterInt()` function to read each array value from the end-user. If the user enters a non-numeric value, `enterInt()` generates an exception. Because the `IntArray` object's constructor performs the dynamic allocation of the underlying array, the same object's destructor cleans it up.

Lines 1-10:

- These lines show the `IntArray` class definition.
- Line 9 declares the data member that stores a dynamically allocated array of integers.
- Line 4 implements a constructor that dynamically allocates the underlying array.
- Line 5 contains the destructor that deallocates the array.
- Line 6 shows the overloaded subscript operator used to initialize and print array elements.

Lines 12-19:

- These lines implement the `main()` function, which is identical to Program-14.10.
- Line 14 contains a `try` block that calls the `initArray()` function.
- Lines 15-17 show the `catch` block that handles an exception by printing out an error message.

Lines 20-37:

- These lines show the implementation of the `initArray()` function.
- Line 26 declares an `IntArray` object that's used to store the user-entered values. The object's constructor, shown on line 4, dynamically allocates the underlying array.
- Lines 28-30 show the `for`-loop that calls the `enterInt()` function on line 29 to read a value from the end-user. The array values are then printed out on lines 33-35.
- As we see, this function performs *no error handling* and no explicit cleanup at all.

Lines 39-49:

- These lines implement the `enterInt()` function, which is identical to Program-14.10.
- If the user enters a non-numeric value, line 45 generates an exception, and the control flow jumps to the `catch` block on line 15.
- If an exception is generated, the stack frame for the `initArray()` function is popped off the function call stack, but its `IntArray` object destructor is still called. So the `throw` statement on line 45 jumps the control flow to line 15, but it also generates an automatic call to the destructor of the `IntArray` object declared on line 26. The object's destructor, shown on line 5, deallocates the underlying array.
- If no exception is generated on line 45, line 48 returns the user-entered value to `initArray()`.

Program output:

- The program output shows two executions of the program, each running with the `valgrind` utility so that we can see there are no memory leaks.
- In the first execution, a valid numeric value is entered in each of the three calls to `enterInt()` on line 29. The array values are printed on lines 33-35.
- In the second execution, the user enters the value `"abc"` as the second element. Since the condition on line 44 is true, an exception is generated on line 45, and the control flow jumps to the `catch` block on line 15. The `initArray()` function terminates, but its object destructor is still called automatically. The destructor for the `IntArray` object declared on line 26 executes as shown on line 5, and it deallocates the underlying array.
- We see from the program output that the two executions result in no memory leaks, which indicates that the cleanup is performed correctly, both in the execution with valid values and in the execution that generates an exception.

Chapter 15

The Standard Template Library (STL)

The standard template library (STL) provides several useful tools to assist with programming in C++. These include class templates that store collections and manipulate their elements, and global functions that operate on these class templates. [10]

In this chapter, we introduce the main principles and components of the STL.

15.1. Principles

We discuss the three main components of the STL and how they interact.

15.1.1. Basics

15.1.1.1. **What is the *standard template library (STL)*:**

- The *standard template library (STL)* is a set of class templates and the global functions, called *algorithms*, that operate on these classes.
- It provides several collection classes, called *container classes*, that implement many member functions and overloaded operators.

15.1.1.2. **Characteristics of the STL:**

- The STL containers and algorithms can be helpful tools, if used correctly. However, there are some caveats to their usage.
- The STL can be somewhat non-intuitive. Many of its containers and algorithms require the use of *iterators*, which are discussed later in this chapter. Iterators add a layer of complexity that can seem intimidating at first.
- The STL can also significantly degrade a program's computational performance if we are not mindful of how the container classes work behind the scenes, as we saw in Program-13.5.

15.1.2. Main components

15.1.2.1. **Main components of the STL:**

- The STL is comprised of three components: containers, iterators, and algorithms.
- *Containers* are the STL collection classes used to store elements of the same data type. There are three kinds of STL containers: sequence containers, associative containers, and container adapters. They are discussed later in this chapter.
- *Iterators* are used for traversing STL containers and accessing their elements. They are also necessary for most STL algorithms to operate on the containers.

- *Algorithms* are global functions that perform operations on STL containers, typically through the use of iterators.

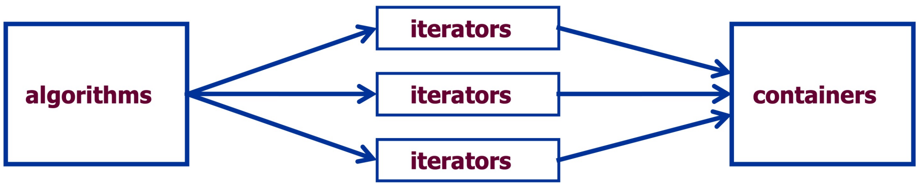


Figure-15.1: STL component interactions

15.1.2.2. **Interactions between STL components:**

- Figure-15.1 illustrates how STL containers, iterators and algorithms interact with each other.
- We see that STL algorithms do not access containers and their elements directly. Instead, they use iterators to do so.
- This was a deliberate decision made by the designers of the STL. The separation between algorithms and containers ensures their independence and allows each one to change and evolve without impacting the other.
- The functioning of an STL algorithm remains independent from the type of container on which it operates.

15.1.3. Coding example: Traversing STL containers

```cpp
#include <vector>
#include <list>

int main()
{
  Student matilda("100567899", "Matilda", "CS", 9.0f);
  Student joe(     "100789111", "Joe", "Physics", 8.3f);
  Student stanley("100456123", "Stanley", "Geography", 5.6f);
  Student amy(     "100123444", "Amy", "Math", 10.8f);

  vector<Student> stuVect;
  cout << "Pushing students on student vector:" << endl;
  stuVect.push_back(matilda);
  stuVect.push_back(joe);
  stuVect.push_back(stanley);
  stuVect.push_back(amy);

  cout<<endl<<"Vector students:"<<endl;
  for (int i=0; i<stuVect.size(); ++i) {
    cout<< stuVect[i];
  }
  cout<<endl;

  list<Student> stuList;
  cout << "Pushing students on student list:" << endl;
  stuList.push_back(matilda);
  stuList.push_back(amy);
  stuList.push_back(stanley);
  stuList.push_back(joe);

31 /*
32   cout<<endl<<"List students:"<<endl;
33   for (int i=0; i<stuList.size(); ++i) {
34     cout<< stuList[i];
35   }
36 */
37   cout<< endl << "End of program" << endl;
38   return 0;
39 }
```

Program-15.1: Traversing STL containers

Program purpose:

- Program-15.1 demonstrates the basic use of the STL `vector` and `list` classes, as a variation of Program-13.5.
- The program defines a `vector` and a `list` of `Student` objects. We see that STL `vector` elements can be accessed directly with the subscript operator, and `list` elements cannot.
- The `Student` class used in this program is identical to Program-13.5.

Lines 6-22:

- These lines demonstrate the basic use of the STL `vector` container.
- Lines 6-9 allocate and initialize four `Student` objects.
- Line 11 declares an STL `vector` of `Student` object elements. The use of the STL `vector` class requires the inclusion of the corresponding library header file, as shown on line 1.
- Lines 13-16 add the four `Student` objects to the back of the `vector`.
- Lines 19-21 print the contents of the `vector` by traversing it and calling the `Student` class's overloaded stream insertion operator on each of its elements.
- We see on line 20 that each element is accessed using the `vector` class's subscript operator.

Lines 24-35:

- These lines demonstrate the basic use of the STL `list` container.
- Line 24 declares an STL `list` of `Student` object elements. The use of the STL `list` class requires the inclusion of the corresponding library header file, as shown on line 2.
- Lines 26-29 add the four `Student` objects to the back of the `list`.
- Lines 32-35 are commented out because they do *not* compile. They attempt to print the contents of the `list` by traversing it and printing each of its elements.
- Line 34 does not compile because it accesses each `list` element using the subscript operator. However, the STL `list` container *does not* provide a subscript operator. Instead, we must use iterators.

15.2. STL Iterators

We discuss STL iterators as an essential component in using both STL containers and algorithms.

15.2.1. Concepts

15.2.1.1. **What is an *STL iterator*:**

- An *STL iterator* is a class template with instances that allow our program to access STL containers and use STL algorithms on them.
- An iterator is *conceptually* similar to a pointer. But it is **not** a pointer.

- The iterator classes provide overloaded operators that match many of the same operators commonly used on pointers.

15.2.1.2. **Characteristics of iterators:**

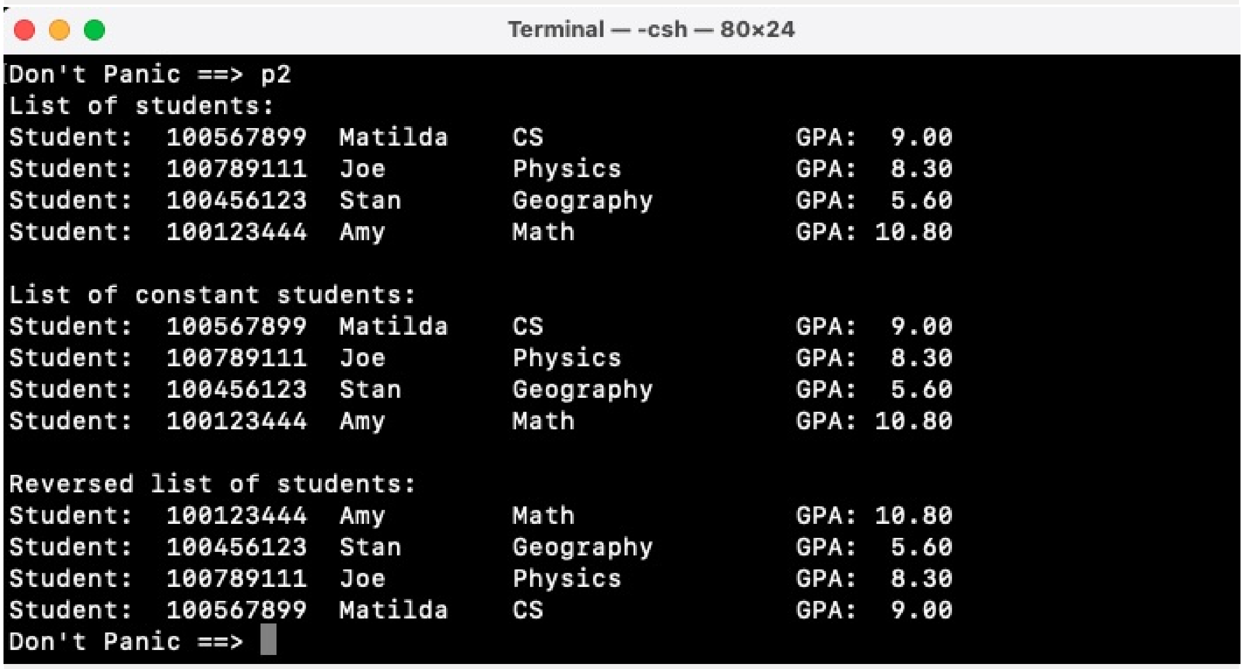
- An iterator object can be used to traverse an STL container that does not support the overloaded subscript operator, for example the STL `list`.
- Of the three types of STL containers, only sequence containers and associative containers support the use of iterators. Container adapters do not. The different types of STL containers are discussed in section 15.3.
- Iterators are also used as parameters to many STL algorithms, as discussed in section 15.4.

15.2.1.3. **Types of iterators:**

- Different classes of iterators can be used to traverse an STL container in a forward direction (from the first element to the last), or in the backward direction (from the last element to the first). These iterators are called *forward iterators* and *reverse iterators*, respectively.
- Forward iterators include the `iterator` class, as well as the `const_iterator` class that does not allow modifications to the container elements.
- Reverse iterators include the `reverse_iterator` and `const_reverse_iterator` classes. The latter does not allow modifications to the container elements.

15.2.2. Coding example: Iterators

```
int main()
{
  Student matilda("100567899", "Matilda", "CS", 9.0f);
  Student joe(    "100789111", "Joe", "Physics", 8.3f);
  Student stanley("100456123", "Stanley", "Geography", 5.6f);
  Student amy(    "100123444", "Amy", "Math", 10.8f);

  list<Student> stuList;
  stuList.push_back(matilda);
  stuList.push_back(joe);
  stuList.push_back(stanley);
  stuList.push_back(amy);

/*  Forward iterator  */
  list<Student>::iterator itr;
  cout << "List of students:" << endl;
  for ( itr  = stuList.begin();
        itr != stuList.end();
        ++itr ) {
    if (itr->getName() == "Stanley") {
      itr->setName("Stan");
    }
    cout<< *itr;
  }

26 /*  Constant forward iterator  */
27   list<Student>::const_iterator cItr;
28   cout<<endl<<"List of constant students:"<<endl;
29   for ( cItr  = stuList.begin(); cItr != stuList.end(); ++cItr ) {
30 /*
31     if (cItr->getName() == "Stanley") {
32       cItr->setName("Stan");
33     }
34 */
35     cout<< *cItr;
36   }

38 /*  Reverse iterator  */
39   list<Student>::reverse_iterator revItr;
40   cout<<endl<<"Reversed list of students:"<<endl;
41   for (revItr=stuList.rbegin(); revItr!=stuList.rend(); ++revItr) {
42     cout<< *revItr;
43   }

45   return 0;
46 }
```



```
Don't Panic ==> p2
List of students:
Student:  100567899  Matilda    CS                GPA:  9.00
Student:  100789111  Joe        Physics           GPA:  8.30
Student:  100456123  Stan       Geography         GPA:  5.60
Student:  100123444  Amy        Math              GPA: 10.80

List of constant students:
Student:  100567899  Matilda    CS                GPA:  9.00
Student:  100789111  Joe        Physics           GPA:  8.30
Student:  100456123  Stan       Geography         GPA:  5.60
Student:  100123444  Amy        Math              GPA: 10.80

Reversed list of students:
Student:  100123444  Amy        Math              GPA: 10.80
Student:  100456123  Stan       Geography         GPA:  5.60
Student:  100789111  Joe        Physics           GPA:  8.30
Student:  100567899  Matilda    CS                GPA:  9.00
Don't Panic ==> 
```

Program-15.2: Iterators

Program purpose:

- Program-15.2 demonstrates the use of STL `iterator` class templates.
- The program defines an STL `list` of `Student` objects. We use three different types of iterators to traverse the `list` and print its elements.
- The `Student` class used in this program is identical to Program-13.5.

Lines 3-12:

- Lines 3-6 allocate and initialize four `Student` objects.
- Line 8 declares an STL `list` of `Student` object elements.
- Lines 9-12 add the four `Student` objects to the back of the `list`.

Lines 15-24:

- Line 15 declares a forward iterator, called `itr`, as an STL `iterator` object that operates on an STL `list` of `Student` objects.
- The `itr` iterator is used as the looping variable in a `for`-loop on lines 17-24 that traverses the `list` and prints out each of its elements.
- Line 17 shows the first part of the loop header. It initializes the `itr` looping variable by calling the STL `list`'s `begin()` member function, which returns an iterator that points to the first element in the `list`. The returned iterator is assigned to the `itr` looping variable using the `iterator` class's overloaded assignment operator.
- Line 18 shows the second part of the `for`-loop header. It checks the loop's breaking condition by calling the STL `list`'s `end()` member function, which returns an iterator that points **just past** the last element in the `list`. This returned iterator is compared to the current value of the `itr` looping variable using the `iterator` class's overloaded inequality operator. If the two iterators are equal, the `for`-loop terminates.
- Line 19 shows the third part of the `for`-loop header. It advances the `itr` looping variable by calling the `iterator` class's overloaded prefix increment operator. This operator moves the looping variable to the next element in the `list`.
- Lines 20-22 change the name of a student. This demonstrates that a forward iterator does allow changes to the container elements. The `iterator` class's overloaded arrow operator is used to call the `Student` public member functions on the current element.
- Line 23 calls the `iterator` class's overloaded dereferencing operator to access the current element in the `list`. Using similar syntax to pointers, dereferencing the `itr` looping variable returns the corresponding `Student` object. Once the current `list` element is accessed, it's printed to the screen using the `Student` class's overloaded stream insertion operator.
- We see from the program output that the list of students is printed in the forward direction, in same order in which the elements are stored in the list.

Lines 27-36:

- Line 27 declares a constant forward iterator, called `cItr`, as an STL `const_iterator` object that operates on an STL `list` of `Student` objects.
- The `cItr` iterator is used as the looping variable in a `for`-loop on lines 29-36 that traverses the `list` and prints out each of its elements.
- Lines 31-33 are commented out because they do *not* compile. These lines attempt to change the name of a student, but because line 32 uses a constant iterator, changes to the container elements are not allowed.
- Line 35 calls the `const_iterator` class's overloaded dereferencing operator to access the current element in the `list`. Once the current `list` element is accessed, it's printed to the screen using the `Student` class's overloaded stream insertion operator.

Lines 39-43:

- Line 39 declares a reverse iterator, called `revItr`, as an STL `reverse_iterator` object that operates on an STL `list` of `Student` objects.
- The `revItr` iterator is used as the looping variable in a `for`-loop on lines 41-43 that traverses the `list` and prints out its elements in reverse order.
- In the `for`-loop header on line 41, the STL `list`'s `rbegin()` member function returns an iterator that points to the last element in the `list`, and `rend()` returns an iterator that points **just before** the first element. The `reverse_iterator` class's overloaded prefix increment operator moves the iterator to the previous element in the `list`.
- Line 42 calls the iterator's dereferencing operator to print the current element in the `list`.
- We see from the program output that the list of students is printed in the backward direction, in the reverse order that the elements are stored in the list.

15.2.3. Categories of iterators

15.2.3.1. What are the *categories of iterators*:

- There are four categories of iterators, each with its own capabilities. In order of weakest to strongest, they are: input/output iterators, forward iterators, bidirectional iterators, and random access iterators.
- Each category possesses the properties and capabilities of all the weaker categories. For example, bidirectional iterators have all the capabilities of input/output and forward iterators, but none of those of random access iterators.
- The iterator category supported by an STL container determines which STL algorithms can be used on that container.

15.2.3.2. Capabilities of the different categories:

- *Input/output iterators* only work on I/O streams. This allows the use STL algorithms on I/O streams as if they are containers.
- *Forward iterators* only work on containers in the forward direction. If an STL container can only be accessed in the forward direction, the STL algorithms that require forward iterators can be used on it. The STL algorithms that require bidirectional or random access iterators cannot be used.
- *Bidirectional iterators* work on containers in both the forward and reverse direction. For example, the STL `list` supports bidirectional iterators, as we saw in Program-15.2.
- *Random access iterators* allow direct access to any element in a container, usually through the subscript operator. This is the strongest category of iterators, and all STL algorithms can be used on containers that support these iterators. For example, the STL `vector` supports random access iterators, but the STL `list` does not.

15.2.4. Operations on iterators

15.2.4.1. All iterators support the following:

- The dereferencing (`*`) operator;
- The increment (`++`) operator;
- The assignment (`=`) operator; and
- The equality and inequality (`==` and `!=`) operators.

15.2.4.2. Forward, bidirectional, and random access iterators support:

- The `begin()` member function, which returns an iterator that points to the first element in the container; and
- The `end()` member function, which returns an iterator that points **just past** the last element in the container. Because this member function does *not* return an iterator to a valid element, we should *never* dereference its returned value. If we do, the program will likely crash.

15.2.4.3. Bidirectional and random access iterators support:

- The `rbegin()` member function, which returns an iterator that points to the last element in the container;
- The `rend()` member function, which returns an iterator that points **just before** the first element in the container; and
- The decrement (`--`) operator.

15.2.4.4. **Random access iterators support:**

- The subscript (`[]`) operator;
- The relational (<, >, <=, >=) operators;
- The addition (+) and addition-assignment (+=) operators; and
- The subtraction (−) and subtraction-assignment (−=) operators.
- Addition and subtraction operations behave in the same way as C/C++ pointer arithmetic.

15.2.4.5. **For optimal performance with iterators:**

- The prefix increment and decrement operators should be used instead of postfix. As discussed in chapter 12, the postfix increment/decrement operators are slower because they must create temporary copies of objects.
- The loop ending value should be stored before the loop. For example, if the iterator returned by the `end()` member function is saved into a variable, the loop header won't have to call this member function with every iteration.

15.3. STL Containers

We discuss the different types of STL containers and their properties.

15.3.1. Concepts

15.3.1.1. **What is an *STL container*:**

- An *STL container* is a collection class template that's provided in the STL. It's a data structure that contains a collection of same-type elements.
- All the elements in an STL container are of the same data type as each other. Because the container is a class template, the elements' data type can be any defined type.
- Many member functions and overloaded operators are implemented for each type of STL container.
- There are three types of STL containers: sequence containers, associative containers, and container adapters.

15.3.1.2. **Characteristics of STL containers:**

- All STL containers provide:
  - basic member functions, including a default constructor, a copy constructor, a destructor, and an implementation of the assignment operator
  - insertion and deletion member functions, for example `insert()`, `delete()`, `clear()`
  - size-related member functions, for example `size()`, `empty()`, `max-size()`
  - some overloaded relational operators
- In addition to the above, sequence and associative containers also provide:
  - member functions for iteration using STL iterators, for example `begin()`, `end()`
- To store objects in an STL container, the class developers must provide:
  - a copy constructor and an overloaded assignment operator
  - for some STL algorithms, the overloaded equality (==) and less-than (<) operators

15.3.2. Streams as containers

15.3.2.1. What is a *stream*:

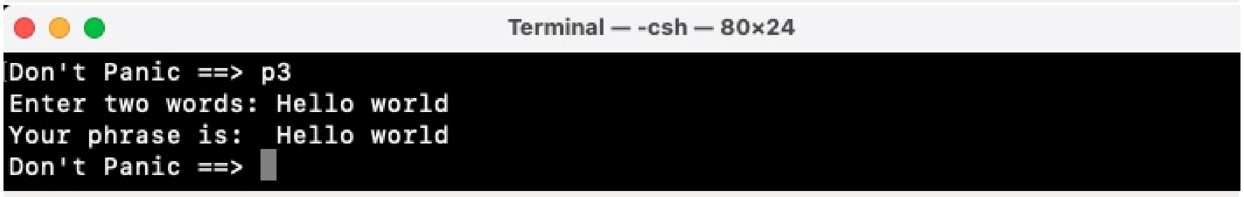
- A *stream* is a linear sequence of bytes, for example console I/O, files, devices, and others.
- Streams are discussed in the chapter 16.

15.3.2.2. Using streams as containers:

- Some STL algorithms, for example the `copy()` algorithm, can be used with a stream by treating it as a container.
- Streams only work with input/output iterators.

15.3.3. Coding example: Streams as containers

```
 1 #include <iterator>

 3 int main()
 4 {
 5   ostream_iterator<string> outItr(cout);
 6   *outItr = "Enter two words: ";

 8   istream_iterator<string> inItr(cin);
 9   string w1, w2;

11   w1 = *inItr;
12   ++inItr;
13   w2 = *inItr;

15   *outItr = "Your phrase is:  ";
16   *outItr = w1 + " " + w2;
17   *outItr = "\n";

19   return 0;
20 }
```


```
Don't Panic ==> p3
Enter two words: Hello world
Your phrase is:  Hello world
Don't Panic ==>
```

Program-15.3: Streams as containers

Program purpose:

- Program-15.3 demonstrates the use of STL input/output iterators on the standard I/O streams.
- The program uses I/O iterators to read some values from the end-user and print them out.

Lines 5-6:

- These lines declare an output iterator and use it to prompt the end-user.
- Line 5 declares an output stream iterator, called `outItr`, as an STL `ostream_iterator` object that operates on strings. The iterator is initialized with the `cout` object by using it as parameter to the `ostream_iterator` constructor.

- Line 6 uses the `ostream_iterator` class's overloaded dereferencing operator to print a literal string to the standard output.

Lines 8-13:

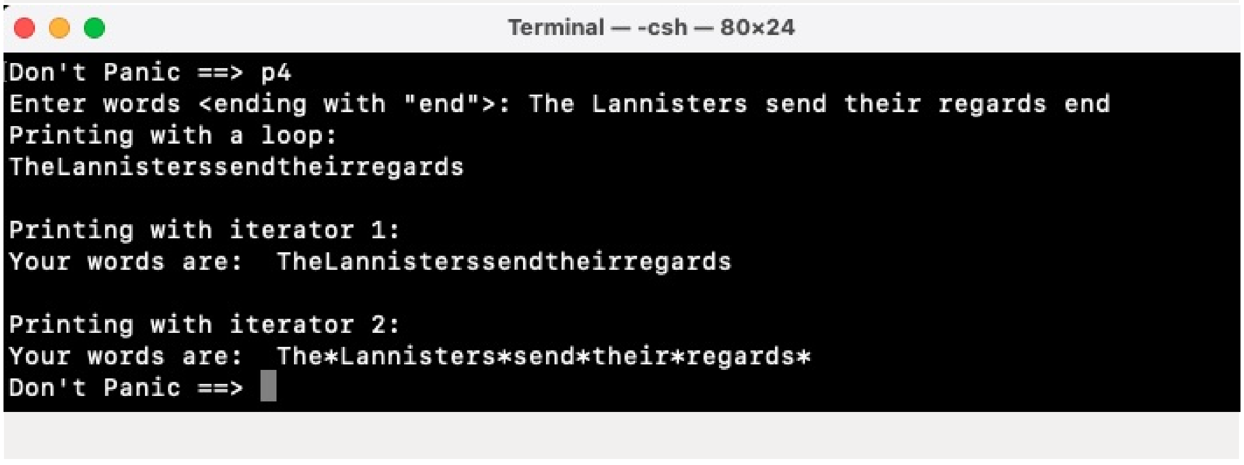
- These lines declare an input iterator and use it to read two strings from the end-user.
- Line 8 declares an input stream iterator, called `inItr`, as an STL `istream_iterator` object that operates on strings. The iterator is initialized with the `cin` object by using it as parameter to the `istream_iterator` constructor.
- Line 11 uses the `istream_iterator` class's overloaded dereferencing operator to read a string from the end-user. The entered string is stored into the `w1` variable.
- Line 12 calls the `istream_iterator` class's overloaded prefix increment operator to advance the iterator past the string that was just read.
- Line 13 again uses the `istream_iterator` dereferencing operator to read another string, which is stored into the `w2` variable.

Lines 15-17:

- These lines use the output iterator to print out the strings entered by the user on lines 11-13.
- Line 15 uses the `ostream_iterator` class's overloaded dereferencing operator to print a literal string to the screen.
- Line 16 calls the same operator to print out the contents of the two string variables, followed by a newline character on line 17.

15.3.4. Coding example: Copying containers to a stream

```
#include <vector>
#include <iterator>
#include <algorithm>

int main()
{
  vector<string> words;
  string str;

  cout<<"Enter words <ending with \"end\">: ";
  cin>>str;
  while (str != "end") {
    words.push_back(str);
    cin >> str;
  }

  cout << "Printing with a loop:" << endl;
  for (int i=0; i<words.size(); ++i) {
    cout << words[i];
  }
  cout << endl << endl;

  cout << "Printing with iterator 1:" << endl;
  ostream_iterator<string> outItr(cout);
  cout<<"Your words are:  ";
  copy(words.begin(), words.end(), outItr);
  cout << endl << endl;

29   cout << "Printing with iterator 2:" << endl;
30   ostream_iterator<string> outItr2(cout, "*");
31   cout<<"Your words are:  ";
32   copy(words.begin(), words.end(), outItr2);
33   cout << endl;

35   return 0;
36 }
```



```
Don't Panic ==> p4
Enter words <ending with "end">: The Lannisters send their regards end
Printing with a loop:
TheLannisterssendtheirregards

Printing with iterator 1:
Your words are:  TheLannisterssendtheirregards

Printing with iterator 2:
Your words are:  The*Lannisters*send*their*regards*
Don't Panic ==> 
```

Program-15.4: Copying containers to a stream

Program purpose:

- Program-15.4 demonstrates the use of the STL `copy()` algorithm with I/O iterators to print out the entire contents of an STL container to the screen.

Lines 10-15:

- These lines prompt the user to enter a sequence of space-separated words, up to a sentinel value of "end".
- Each user-entered string is stored in an STL `vector`, called `words`, as shown on line 13.

Lines 17-21:

- These lines take the usual approach to print out the contents of a collection, by iterating over the `words` vector and printing each element.
- The program output shows that each word is printed, but without any delimiters in between.

Lines 23-27:

- These lines show the first usage of the STL `copy()` algorithm and an output stream iterator to print out the entire contents of a `vector` in a single statement.
- Line 24 declares an output stream iterator, called `outItr`, as an STL `ostream_iterator` object that operates on strings. The iterator is initialized with the `cout` object by using it as parameter to the `ostream_iterator` constructor.
- Line 26 uses the STL `copy()` algorithm, with the output stream iterator `outItr`, to print out the entire contents of the `words` vector to the screen, from the first element to the last.
- The `copy()` algorithm takes three iterators as parameters. The first two parameters specify the source of the copy, as a pair of iterators that indicate the range of container elements to be copied. The third parameter is an iterator to the destination of the copy, in this case an output stream iterator.
- Line 26 requires that the `vector` elements implement a stream insertion operator. Because the `string` class does overload this operator, the `copy()` algorithm can be used to print out an STL container of `string`s. The program output shows the printed words without delimiters.

Lines 29-33:

- These lines show the second usage of the STL `copy()` algorithm and an output stream iterator to print out the entire contents of a `vector` in a single statement.
- Line 30 declares a second output stream iterator, called `outItr2`, as an `ostream_iterator` object that operates on strings, and it's initialized with the `cout` object. In this example, we indicate a second parameter to the constructor: the delimiter string to be output between the individual `vector` elements, in this case an asterisk (`*`).
- Line 32 uses the STL `copy()` algorithm, with the second output stream iterator `outItr2`, to print out the `words` vector to the screen. The program output shows that each word is printed, with an asterisk as delimiter between the words.

15.3.5. Sequence containers

15.3.5.1. What is a *sequence container*:

- A *sequence container* is an STL container that retains the order of its elements.
- If a class user stores the elements inside a container in a specific order, the container must maintain them in the required order.
- The STL provides several sequence containers, including: `vector`, `list`, and `deque`.
- Each sequence container implements a number of useful member functions, including `front`, `back`, `push_back`, `pop_back`, and many others.
- The ***back*** of a sequence container is its end, after the last element. The ***front*** of a container is its beginning, before its first element.

15.3.5.2. Characteristics of an STL `vector`:

- Storage:
  - elements are stored *contiguously*, which means all together sequentially in memory
  - the `vector` grows as needed, as more elements are added
  - it allows direct access to any element, with the overloaded subscript operator (`[]`) or the `at()` member function
- Insertion and deletion:
  - insertion and deletion of elements at the back of the `vector` is very efficient
  - insertion and deletion anywhere else causes the elements to be copied, which is inefficient
- Iterators: `vector`s support random access iterators

15.3.5.3. Characteristics of an STL `list`:

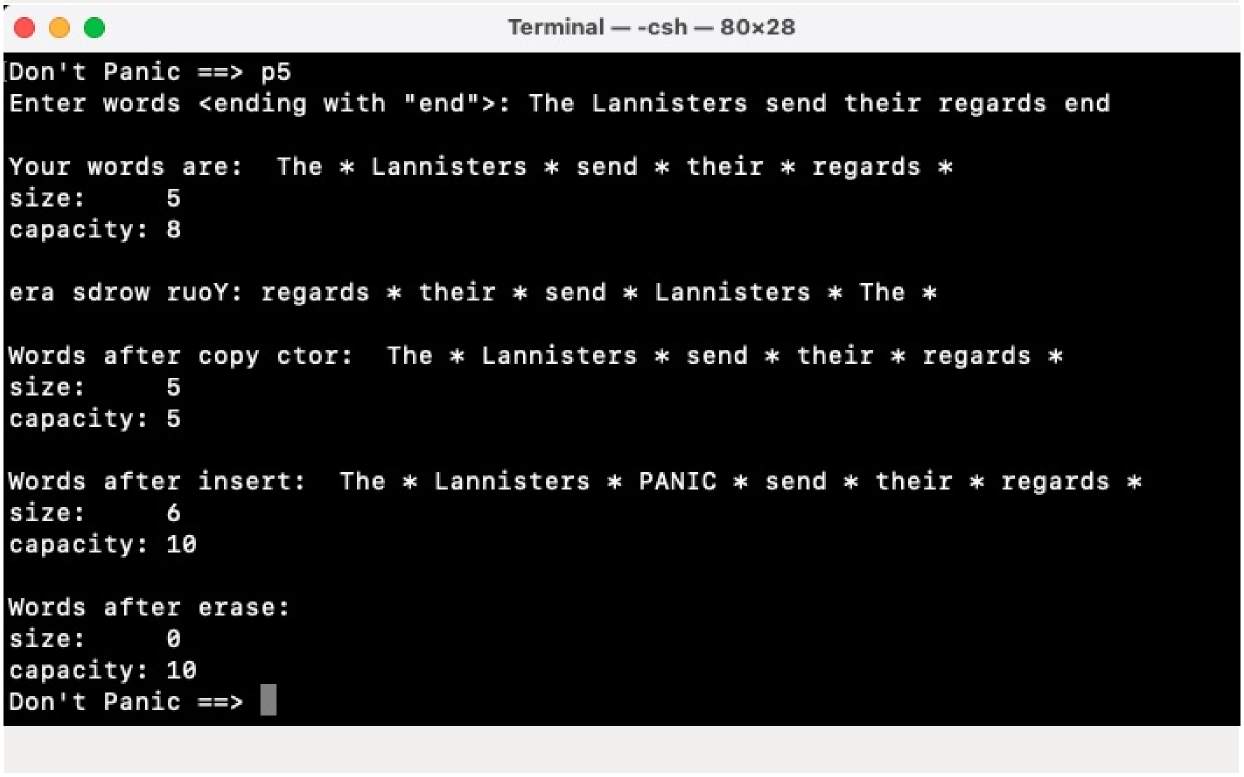
- Storage:
  - an STL `list` is implemented as a doubly linked list
  - elements are *not* stored contiguously in memory
  - the `list` grows as needed, as more elements are added
  - it does *not* allow direct access to its elements
- Insertion and deletion:
  - insertion and deletion of elements anywhere in the `list` is efficient
- Iterators:
  - `list`s support bidirectional iterators
  - they do *not* support random access iterators

15.3.5.4. **Characteristics of an STL `deque`:**

- A *deque* is a double-ended queue. It's a data structure meant to be added to and removed from at the front and at the back of the container.
- Storage:
  - elements are *not* stored contiguously in memory
  - the `deque` grows as needed, as more elements are added
  - it allows direct access to any element, with the overloaded subscript operator or the `at()` member function
- Insertion and deletion:
  - insertion and deletion of elements at the back or the front of the `deque` is very efficient
  - insertion and deletion anywhere else is more efficient than an STL `vector`, but less efficient than an STL `list`
- Iterators: `deque`s support random access iterators

15.3.6. Coding example: STL `vector`s

```
int main()
{
  vector<string> words1;
  string str;

  cout<<"Enter words <ending with \"end\">: ";
  cin>>str;
  while (str != "end") {
    words1.push_back(str);
    cin>>str;
  }
  cout<<endl;

  ostream_iterator<string> outItr(cout, " * ");

  cout<<"Your words are:  ";
  copy(words1.begin(), words1.end(), outItr);  cout<<endl;
  cout<<"size:     "<<words1.size()<<endl;
  cout<<"capacity: "<<words1.capacity()<<endl<<endl;

  cout<<"era sdrow ruoY: ";
  copy(words1.rbegin(), words1.rend(), outItr);  cout<<endl<<endl;

  vector<string> words2 = words1;
  cout<<"Words after copy ctor:  ";
  copy(words2.begin(), words2.end(), outItr);  cout<<endl;
  cout<<"size:     "<<words2.size()<<endl;
  cout<<"capacity: "<<words2.capacity()<<endl<<endl;

  words2.insert(words2.begin()+2, "PANIC");
  cout<<"Words after insert:  ";
  copy(words2.begin(), words2.end(), outItr);  cout<<endl;
  cout<<"size:     "<<words2.size()<<endl;
  cout<<"capacity: "<<words2.capacity()<<endl<<endl;

36    words2.erase(words2.begin(), words2.end());
37    cout<<"Words after erase:  ";
38    copy(words2.begin(), words2.end(), outItr);  cout<<endl;
39    cout<<"size:      "<<words2.size()<<endl;
40    cout<<"capacity: "<<words2.capacity()<<endl;

42    return 0;
43  }
```



```
Don't Panic ==> p5
Enter words <ending with "end">: The Lannisters send their regards end

Your words are:  The * Lannisters * send * their * regards *
size:     5
capacity: 8

era sdrow ruoY: regards * their * send * Lannisters * The *

Words after copy ctor:  The * Lannisters * send * their * regards *
size:     5
capacity: 5

Words after insert:  The * Lannisters * PANIC * send * their * regards *
size:     6
capacity: 10

Words after erase:
size:     0
capacity: 10
Don't Panic ==> 
```

Program-15.5: STL `vector`s

Program purpose:

- Program-15.5 demonstrates some commonly used member functions of the STL `vector`.

Lines 6-11:

- These lines prompt the user to enter a sequence of space-separated words, up to a sentinel value of "end".
- Each word is stored in an STL `vector` of strings, called `words1`, as shown on line 9.

Lines 14-22:

- These lines print out the `words1` vector, both in the forward and reverse directions.
- Line 14 declares an output stream iterator, called `outItr`, as an STL `ostream_iterator` object that operates on strings, and it's initialized with the `cout` object. We use the second constructor parameter to define the delimiter string (`" * "`) to be output between the individual elements.
- Line 17 uses the STL `copy()` algorithm, with the output stream iterator `outItr`, to print out the entire contents of `words1` to the screen. The iterator parameters specify that the entire `vector`, from the first element to the last, is copied to the output stream.
- Lines 18-19 show the difference between the size and the capacity of an STL container. Its *size* specifies the current number of elements that are stored in the container, and its *capacity* is the number of elements that it can accommodate before being full.

- The program output from lines 18-19 shows that the size of `words1` is 5 elements, and its maximum capacity is 8. Remember from Program-13.5 that a `vector`'s capacity begins at 1 and doubles every time an element is added to a full container.
- Line 22 uses the STL `copy()` algorithm, with the output stream iterator `outItr`, to print out the entire contents of `words1` to the screen, in reverse order.

Lines 24-28:

- These lines copy `words1` into a new `vector` and print it out.
- Line 24 uses the `vector` copy constructor to initialize a new `vector` called `words2` from the contents of `words1`.
- Line 26 uses the STL `copy()` algorithm to print out the entire contents of `words2` to the screen.
- The program output from lines 27-28 shows that `words2` is initialized on line 24 with the current capacity of `words1`, which is 5 elements.

Lines 30-34:

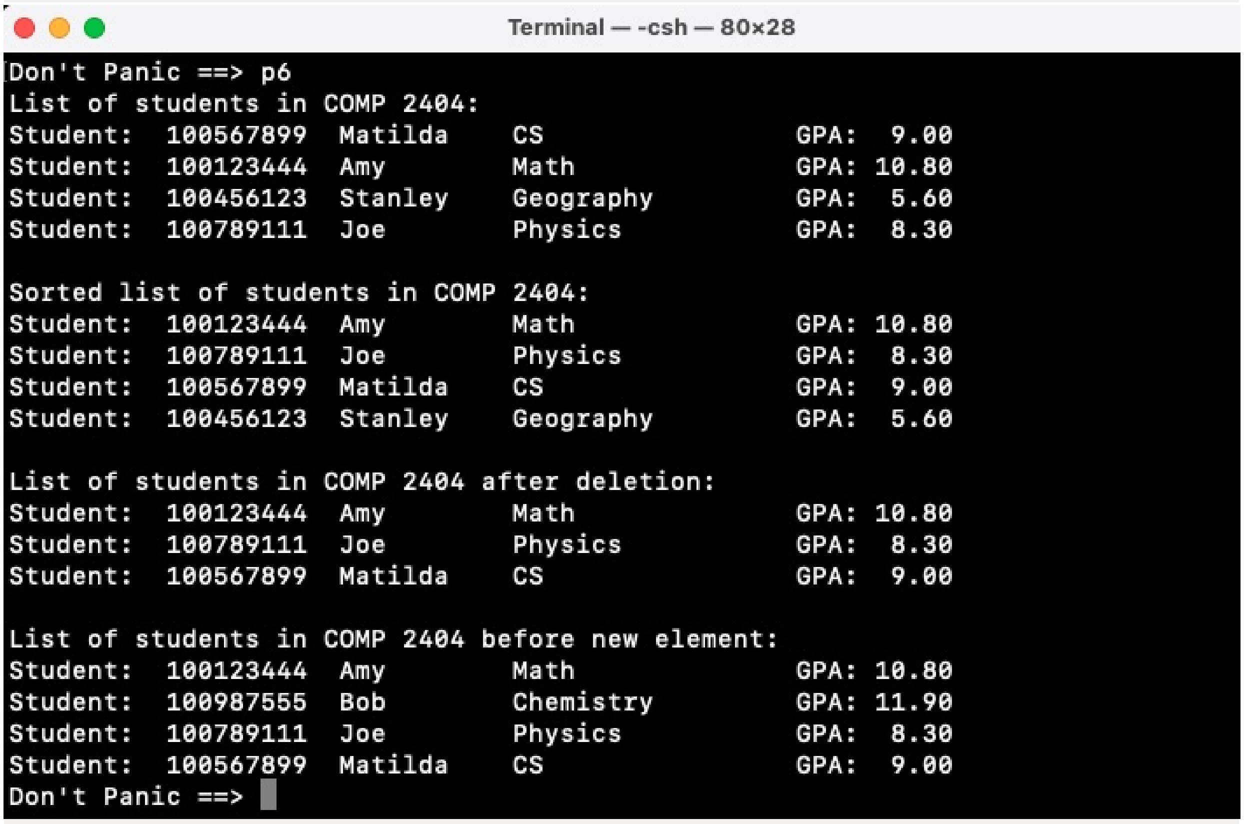
- These lines insert a new word in the middle of a `vector` and print it out.
- Line 30 uses the `vector` class's `insert()` member function to insert a new word in the third position of `words2`. The member function takes an iterator as its first parameter, to indicate the insertion point in the `vector`, and the new element as its second parameter.
- Because a `vector` supports random access iterators, we can use an addition operation on line 30 to specify the insertion point at 2 elements after the first one.
- Line 32 uses the STL `copy()` algorithm to print out the entire contents of `words2` to the screen.
- The program output from lines 33-34 shows that, in order to add a sixth element to `words2`, the `vector`'s capacity was doubled from 5 to 10.

Lines 36-40:

- These lines remove the entire contents of a `vector`.
- Line 36 uses the `vector` class's `erase()` member function to delete all the elements in `words2`. The member function takes two iterators as parameters, indicating where the deletion should begin and end.
- Line 38 uses the STL `copy()` algorithm to print out the empty `words2` vector to the screen.
- The program output from lines 39-40 shows that `words2` now contains no elements, but its capacity remains unchanged.

15.3.7. Coding example: STL `lists`

```
 1 int main()
 2 {
 3   Student matilda("100567899", "Matilda", "CS", 9.0f);
 4   Student joe(    "100789111", "Joe", "Physics", 8.3f);
 5   Student stanley("100456123", "Stanley", "Geography", 5.6f);
 6   Student amy(    "100123444", "Amy", "Math", 10.8f);
 7   Student bob(    "100987555", "Bob", "Chemistry", 11.9f);

 9   list<Student> comp2404;
10   ostream_iterator<Student> outItr(cout);
11   comp2404.push_back(matilda);
12   comp2404.push_back(amy);
13   comp2404.push_back(stanley);
14   comp2404.push_back(joe);
15
16    cout<<"List of students in COMP 2404:"<<endl;
17    copy(comp2404.begin(), comp2404.end(), outItr);

19    comp2404.sort();
20    cout<<endl<<"Sorted list of students in COMP 2404:"<<endl;
21    copy(comp2404.begin(), comp2404.end(), outItr);

23    comp2404.erase(--comp2404.end());
24    cout<<endl<<"List of students in COMP 2404 after cull:"<<endl;
25    copy(comp2404.begin(), comp2404.end(), outItr);

27    comp2404.push_back(bob);
28    comp2404.sort();
29    cout<<endl<<"List of students in COMP 2404 before merge:"<<endl;
30    copy(comp2404.begin(), comp2404.end(), outItr);

32    return 0;
33  }
```



```
Don't Panic ==> p6
List of students in COMP 2404:
Student:  100567899  Matilda    CS                  GPA:  9.00
Student:  100123444  Amy        Math                GPA: 10.80
Student:  100456123  Stanley    Geography           GPA:  5.60
Student:  100789111  Joe        Physics             GPA:  8.30

Sorted list of students in COMP 2404:
Student:  100123444  Amy        Math                GPA: 10.80
Student:  100789111  Joe        Physics             GPA:  8.30
Student:  100567899  Matilda    CS                  GPA:  9.00
Student:  100456123  Stanley    Geography           GPA:  5.60

List of students in COMP 2404 after deletion:
Student:  100123444  Amy        Math                GPA: 10.80
Student:  100789111  Joe        Physics             GPA:  8.30
Student:  100567899  Matilda    CS                  GPA:  9.00

List of students in COMP 2404 before new element:
Student:  100123444  Amy        Math                GPA: 10.80
Student:  100987555  Bob        Chemistry           GPA: 11.90
Student:  100789111  Joe        Physics             GPA:  8.30
Student:  100567899  Matilda    CS                  GPA:  9.00
Don't Panic ==> 
```

Program-15.6: STL `list`s

Program purpose:

- Program-15.6 demonstrates some commonly used member functions for the STL `list`.
- The `Student` class used in this program is identical to Program-13.5, with the addition of the overloaded equality and less-than operators. Both operators compare the `Student` names.

Lines 3-14:

- These lines declare and initialize some `Student` objects, as well as an STL `list` to store them, and an output stream iterator to print them.
- Lines 3-7 allocate and initialize five `Student` objects.
- Line 9 declares an STL `list` of `Student` objects, called `comp2404`.

- Line 10 declares an output stream iterator, called `outItr`, as an STL `ostream_iterator` object that operates on `Student` objects, and it's initialized with the `cout` object.
- Lines 11-14 add four of the `Student` objects to the `comp2404` list.

Lines 16-21:

- These lines sort the `list` and print it out.
- Line 17 uses the STL `copy()` algorithm to print out the entire contents of `comp2404` to the screen. This requires the elements inside the `list` to implement the overloaded stream insertion operator, which the `Student` class does.
- The program output from line 17 shows that the `Student` objects are in the order in which they were added to the `list` on lines 11-14.
- Line 19 calls the STL `list` class's `sort()` member function to reorder the elements inside `comp2404`. This member function requires the elements inside the `list` to implement the overloaded less-than and equality operators, which the `Student` class does.
- Line 21 uses the STL `copy()` algorithm again to print out the contents of `comp2404`.
- The program output from line 21 shows that the `Student` elements are now ordered by name.

Lines 23-25:

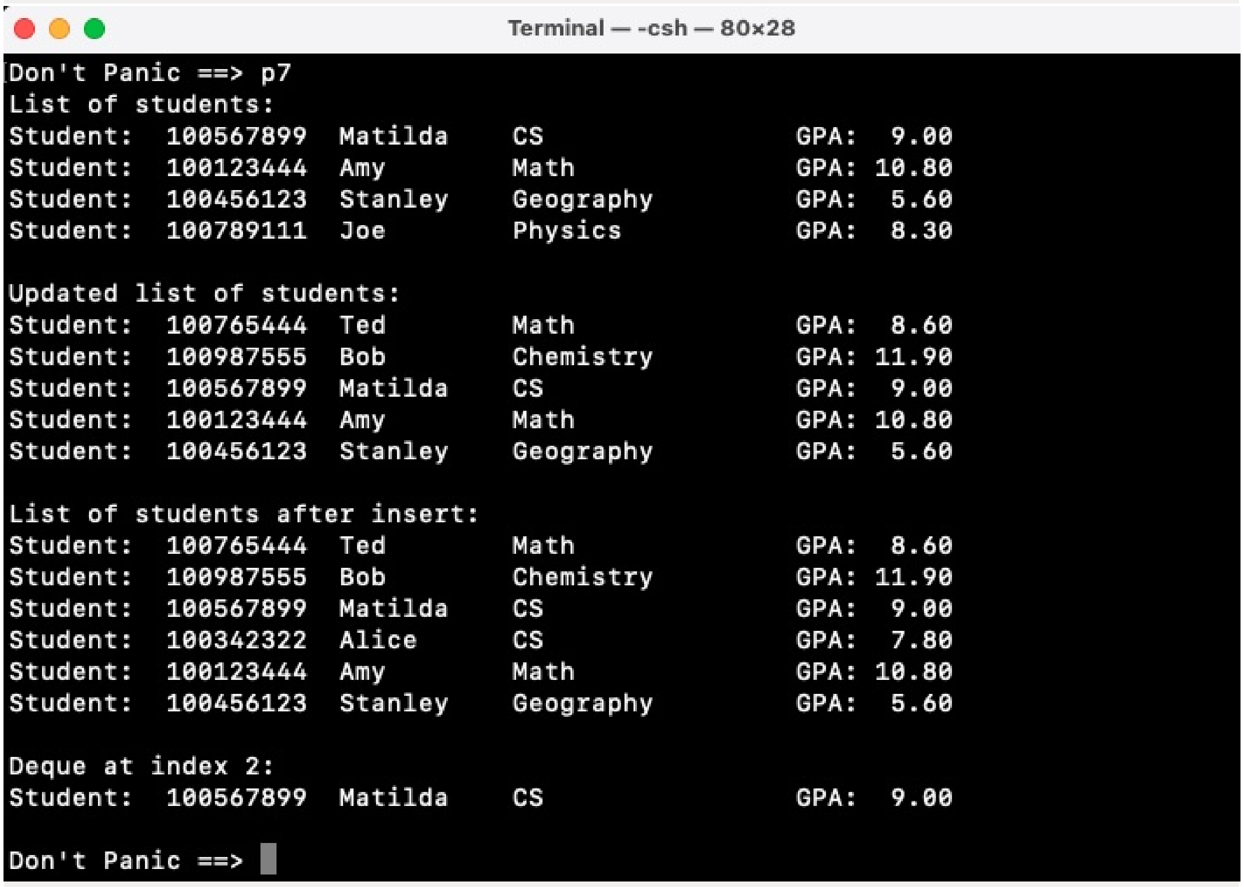
- These lines remove the last element in the `list`.
- Line 23 calls the `list` class's `erase()` member function to remove the last element in `comp2404`. The member function takes an iterator as parameter to indicate where the deletion should begin. Because no second parameter is used to specify where the deletion should end, the member function deletes every element from the starting point to the end of the `list`.
- The iterator's overloaded decrement operator is used on line 23 so that the deletion begins with the last element. Because an STL `list` does not support random access iterators, we cannot use a subtraction operation. However, the decrement operator is implemented for the `list` class's bidirectional iterators.
- Line 25 uses the STL `copy()` algorithm to print out the contents of `comp2404` to the screen.
- The program output from line 25 shows that the last element has been removed.

Lines 27-30:

- These lines add a new element to the `list`, and sort it.
- Line 27 adds a new `Student` object to the back of the `list`.
- Line 28 calls the `list` class's `sort()` member function to reorder the elements by name.
- Line 30 uses the STL `copy()` algorithm to print out the contents of `comp2404` to the screen.
- The program output from line 30 shows that the new element has been added, and the list has been sorted again.

15.3.8. Coding example: STL deques

```
int main()
{
  Student matilda("100567899", "Matilda", "CS", 9.0f);
  Student joe(    "100789111", "Joe", "Physics", 8.3f);
  Student stanley("100456123", "Stanley", "Geography", 5.6f);
  Student amy(    "100123444", "Amy", "Math", 10.8f);
  Student bob(    "100987555", "Bob", "Chemistry", 11.9f);
  Student alice(  "100342322", "Alice", "CS", 7.8f);
  Student ted(    "100765444", "Ted", "Math", 8.6f);

  deque<Student> stuDeque;
  ostream_iterator<Student> outItr(cout);
  stuDeque.push_back(matilda);
  stuDeque.push_back(amy);
  stuDeque.push_back(stanley);
  stuDeque.push_back(joe);

  cout << "List of students:" << endl;
  copy(stuDeque.begin(), stuDeque.end(), outItr);

  stuDeque.push_front(bob);
  stuDeque.push_front(ted);
  stuDeque.pop_back();
  cout << endl << "Updated list of students:" << endl;
  copy(stuDeque.begin(), stuDeque.end(), outItr);

  stuDeque.insert(stuDeque.end()-2, alice);
  cout << endl << "List of students after insert:" << endl;
  copy(stuDeque.begin(), stuDeque.end(), outItr);

  cout << endl << "Deque at index 2: " << endl << stuDeque[2] << endl;

  return 0;
}
```



```
Don't Panic ==> p7
List of students:
Student:  100567899  Matilda    CS                  GPA:  9.00
Student:  100123444  Amy        Math                GPA: 10.80
Student:  100456123  Stanley    Geography           GPA:  5.60
Student:  100789111  Joe        Physics             GPA:  8.30

Updated list of students:
Student:  100765444  Ted        Math                GPA:  8.60
Student:  100987555  Bob        Chemistry           GPA: 11.90
Student:  100567899  Matilda    CS                  GPA:  9.00
Student:  100123444  Amy        Math                GPA: 10.80
Student:  100456123  Stanley    Geography           GPA:  5.60

List of students after insert:
Student:  100765444  Ted        Math                GPA:  8.60
Student:  100987555  Bob        Chemistry           GPA: 11.90
Student:  100567899  Matilda    CS                  GPA:  9.00
Student:  100342322  Alice      CS                  GPA:  7.80
Student:  100123444  Amy        Math                GPA: 10.80
Student:  100456123  Stanley    Geography           GPA:  5.60

Deque at index 2:
Student:  100567899  Matilda    CS                  GPA:  9.00

Don't Panic ==> 
```

Program-15.7: STL `deque`s

Program purpose:

- Program-15.7 demonstrates some commonly used member functions for the STL `deque`.
- The `Student` class used in this program is identical to Program-13.5.

Lines 3-16:

- These lines declare and initialize some `Student` objects, as well as an STL `deque` to store them, and an output stream iterator to print them.
- Lines 3-9 allocate and initialize seven `Student` objects.
- Line 11 declares an STL `deque` of `Student` objects, called `stuDeque`, and lines 13-16 add four of the `Student` objects to it.
- Line 12 declares an output stream iterator as an STL `ostream_iterator` object that operates on `Student` objects, and it's initialized with the `cout` object for standard output.

Lines 18-25:

- These lines add and remove elements from the ends of the `deque`.
- Line 19 uses the STL `copy()` algorithm to print out the entire contents of `stuDeque` to the screen. This requires the elements inside the `deque` to implement the overloaded stream insertion operator, which the `Student` class does.
- The program output from line 19 shows that the `Student` objects are in the order in which they were added to the `deque` on lines 13-16.
- Lines 21-23 call the `deque` class's `push_front()` and `pop_back()` member functions to add two elements to the front of `stuDeque` and remove one element from the back.
- Line 25 uses the STL `copy()` algorithm again to print out the contents of `stuDeque`.
- The program output from line 25 shows two new elements at the front and the last element removed.

Lines 27-31:

- These lines insert a new element in the middle of the `deque`, and we see the subscript operator at work.
- Line 27 uses the `deque` class's `insert()` member function to insert a new `Student` in the third position from the end of `stuDeque`. The member function takes an iterator as its first parameter, to indicate the insertion point, and the new element as its second parameter.
- The iterator's overloaded subtraction operator is used on line 27 so that the insertion is made at two elements from the end of `stuDeque`. An STL `deque` supports random access iterators, so we can use a subtraction operation.
- Line 29 uses the STL `copy()` algorithm again to print out the contents of `stuDeque`.
- The program output from line 29 shows the new element inserted in the correct position.
- Line 31 uses the `deque` class's overloaded subscript operator to access the element at index `2`. The `Student` object's overloaded stream insertion operator is used to print the element.

15.3.9. Associative containers

15.3.9.1. **What is an *associative container*:**

- An *associative container* is an STL container that stores elements using *keys*.
- The keys in an associative container are stored in a user-specified order, which is ascending order by default. A predicate may be used to define the order.
- The STL provides several associative containers, including: `set`, `multiset`, `map`, and `multimap`.

15.3.9.2. **Characteristics of associative containers:**

- The different types of associative containers are organized as follows:
  - the `set` and `multiset` containers store keys only
  - the `map` and `multimap` containers store pairs of keys and values

  - `set` and `map` do not allow duplicates
  - `multiset` and `multimap` do allow duplicates
- Each associative container implements a number of useful member functions, including `insert`, `find`, `lower_bound`, `upper_bound`, and many more.
- Associative containers support bidirectional iterators.

15.3.10. Container adapters

15.3.10.1. What is a *container adapter*:

- A *container adapter* is a higher-level STL container that strictly regulates the access to its elements, which are stored in a programmer-selected underlying container.
- The STL provides several container adapters, including: `stack`, `queue`, and `priority_queue`.

15.3.10.2. Characteristics of container adapters:

- A container adapter can use different types of underlying containers to store its elements:
  - a `stack` can be implemented using any sequence container
  - a `queue` can be implemented using an STL `deque` or `list`
  - a `priority_queue` can be implemented using an STL `vector` or `deque`
- Container adapters do *not* support iterators.

15.4. STL Algorithms

We discuss the STL algorithms and their characteristics.

15.4.1. What is an *STL algorithm*:

- An *STL algorithm* is a global function template that performs useful operations on containers.
- STL algorithms use iterators instead of accessing the containers and their elements directly. This allows for the development of more generic algorithms that can work with different types of containers.
- Some algorithms also work on non-STL containers, such as primitive arrays.

15.4.2. Characteristics of STL algorithms:

- STL algorithms often operate on containers using a pair of iterators, for example to indicate the range (the beginning and the end) of an operation.
- The return value of an STL algorithm is often an iterator.
- Each algorithm requires a specific category of iterators in order to work. As a result, each STL algorithm only works with certain types of containers.
- The STL provides several useful algorithms, including: `copy()`, `sort()`, `remove()`, and `fill()`, and many more.

Chapter 16

Streams and Files

The standard C++ library provides several useful classes and predefined objects to assist with input/output (I/O), including console I/O and files.

In this chapter, we introduce the basics of I/O in C++, including streams and files.

16.1. Streams

We discuss the concept of streams in Unix programming, and how they are implemented in C++.

16.1.1. Concepts

16.1.1.1. **What is a *stream*:**

- A *stream* is a linear sequence of bytes that flows from a *data source* into program memory, and from program memory into a *data sink*.
- Examples of a data source include a keyboard, an input file, and a network adapter.
- Examples of a data sink include a screen, an output file, a printer, and a network adapter.

16.1.1.2. **The `iostream` classes:**

- The `iostream` library is part of the C++ standard library, and it contains many I/O template specializations.
- The `istream` class provides functionality for input streams. For example, the standard input object, `cin`, is an instance of the `istream` class.
- The `ostream` class provides functionality for output streams. For example, the standard output object, `cout`, is an instance of the `ostream` class, as are the standard error and standard log objects, `cerr` and `clog`, respectively.

16.1.1.3. **Characteristics of stream classes:**

- Every stream object maintains a set of error flags that indicate its current state.
- The error flags include: `goodbit`, `failbit`, and `badbit`.
- The `ostream` and `istream` classes provide member functions that test the error flags, as we discuss in section 16.3.
- They also provide several overloaded operators, as discussed shortly.

16.1.1.4. **Characteristics of input streams:**

- There are two ways to read data from an input stream:
  - to read *formatted* data, we use the stream extraction operator (`>>`)
  - to read *unformatted* data, we use `istream` member functions like `get()` or `getline()`
- Formatted data is only read up to a white-space delimiter, like a space, a tab, or a newline. If we need to input multiple items on the same line, we need to use the unformatted data approach.
- Unformatted data is read as one long string, up to a delimiter, with a newline character as the default delimiter. If the program needs this long string to be separated into different parts, then it must parse the string explicitly.
- Every input stream has an end-of-file flag, called `eofbit`, to indicate that the end of the input has been reached. When this happens, the stream's `eofbit` automatically gets set, and it can be tested using the `eof()` member function.

16.1.2. Overloaded stream operators

16.1.2.1. **The overloaded logical NOT operator (`!`):**

- When used on stream object operand, the logical NOT operator (`!`) returns true if one of the following flags is set: `failbit`, or `badbit`, or `eofbit`.
- This allows a program to test if any stream has encountered an error, or if an input stream has reached the end of the input.

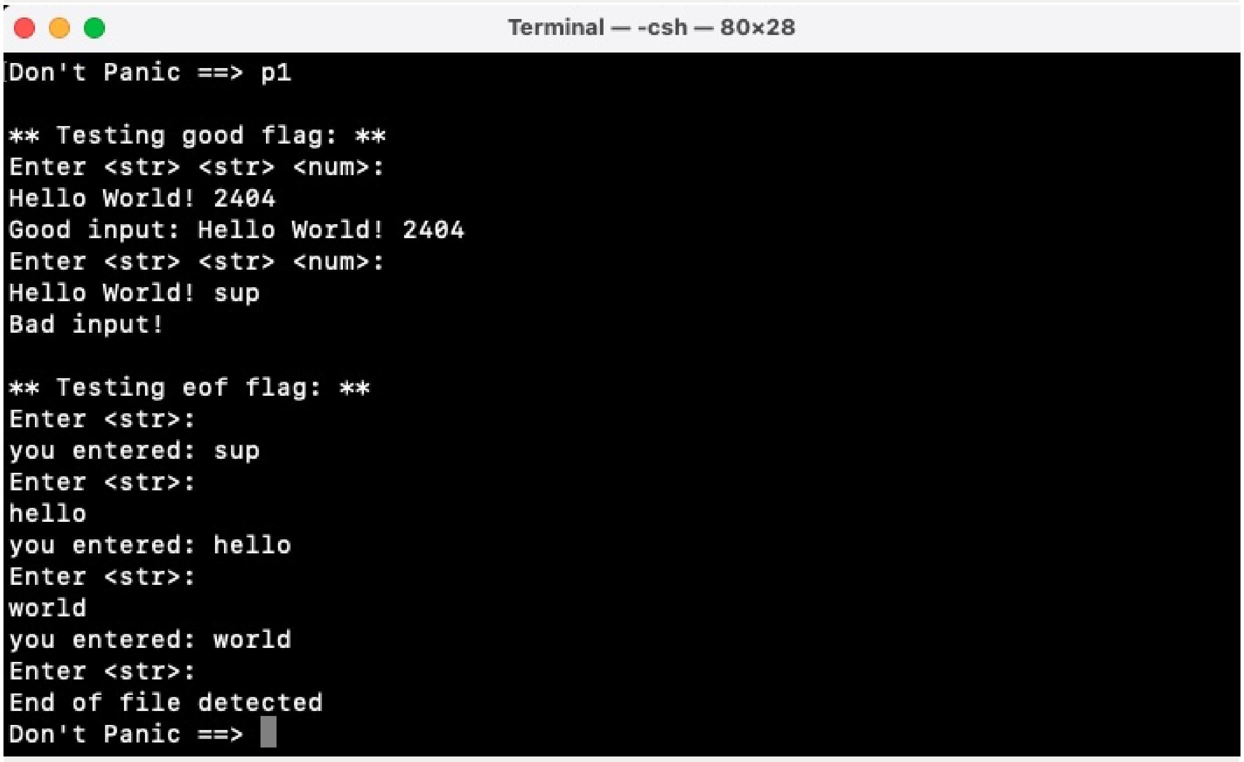
16.1.2.2. **The overloaded conversion (typecast) to `void*` operator**:

- The overloaded typecast operator is called implicitly when a stream object is tested as a condition, for example in an `if`-statement condition or a loop header.
- When used on stream object operand, the operator first converts a stream to a `void` pointer:
  - the conversion returns a null pointer if one of the `failbit`, `badbit`, or `eofbit` flags is true
  - otherwise, it returns a non-null pointer
- Then the operator tests the `void` pointer:
  - the test returns true if the typecast operator returns a non-null pointer
  - it returns false if the typecast operator returns a null pointer
- This allows our programs to easily check if a stream is in an error state or not, or if the end of an input stream or file has been reached.

16.1.3. Coding example: Streams

```
int main()
{
  cout << endl << "** Testing good flag: **" << endl;
  testGoodFlag();
  cin.clear();
  testGoodFlag();
  cin.clear();

  cout << endl << "** Testing eof flag: **" << endl;
  testEOF();
  cin.clear();

  return 0;
}
```

```
void testGoodFlag()
{
  string s1, s2;
  int    num;

  cout << "Enter <str> <str> <num>:" << endl;
  cin >> s1 >> s2 >> num;
  if (cin.good()) {
    cout << "Good input: " << s1 << " " << s2 << " " << num << endl;
  }
  else {
    cout << "Bad input!" << endl;
  }
}

void testEOF()
{
  string str;

  cout << "Enter <str>:" << endl;
  cin >> str;
  while (!cin.eof()) {
    cout << "you entered: " << str << endl;
    cout << "Enter <str>:" << endl;
    cin  >> str;
  }
  cout << "End of file detected" << endl;
}
```



```
Don't Panic ==> p1

** Testing good flag: **
Enter <str> <str> <num>:
Hello World! 2404
Good input: Hello World! 2404
Enter <str> <str> <num>:
Hello World! sup
Bad input!

** Testing eof flag: **
Enter <str>:
you entered: sup
Enter <str>:
hello
you entered: hello
Enter <str>:
world
you entered: world
Enter <str>:
End of file detected
Don't Panic ==>
```

Program-16.1: Streams

Program purpose:

- Program-16.1 demonstrates the `istream` class's `good()` and `eof()` member functions on the standard input stream.

Lines 1-14:

- These lines show the implementation of the `main()` function.
- Lines 3-7 call the `testGoodFlag()` function twice to test both correct and incorrect input.
- After each function call, the `istream` class's `clear()` member function is used to unset the error flags on the `cin` object on lines 5 and 7.
- Line 10 calls the `testEOF()` function to demonstrate the use of the end-of-file marker on an input stream.

Lines 15-28:

- These lines show the implementation of the `testGoodFlag()` global function.
- Line 21 uses the stream extraction operator to read from the end-user three different values: two strings and an integer.
- Line 22 calls the `istream` class's `good()` member function to see if the `cin` stream is fine, or if it's in an error state.
- From the program output, we see that, during the first call to `testGoodFlag()`, the user enters a value of `2404` for the integer. This is a valid value, so the `good()` member function returns true, and the program prints out the user-entered values on line 23.
- However, during the second call to `testGoodFlag()`, the user enters a string instead of an integer. Because this is an invalid value, the `good()` member function returns false, and an error message is printed out on line 26.

Lines 30-42:

- These lines show the implementation of the `testEOF()` global function.
- Lines 36-40 implement a loop that repeatedly reads a string from the end-user until the end-of-file value is entered. In Unix-type systems, the end-of-file value is `Ctrl+D`.
- From the program output, we see that the string entered instead of an integer value during the second call to the `testGoodFlag()` function on line 6 is still in the input stream. This value is read as the first user-entered string. The user enters two more strings, then presses `Ctrl+D`, which sets the `eofbit` flag. The condition on line 36 is now false, and the loop terminates.

16.2. Files

We discuss how file I/O operates in C++.

16.2.1. Concepts

16.2.1.1. **What is a *file*:**

- A *file* is a stream that is stored in *persistent storage*, also called *non-volatile storage*.
- Persistent storage is memory that is *not* deallocated when the OS shuts down, for example a hard disk or a solid-state drive (SSD).
- Volatile storage is memory that is deallocated and does not persist beyond a system shut down, for example random access memory (RAM), also called *main memory*.

16.2.1.2. **Characteristics of files:**

- A file is a linear sequence of bytes, terminated by an end-of-file marker. This is a special character that is OS-dependent and indicates that the end of the file has been reached.
- In C++, a file is represented virtually as an object.

16.2.1.3. **The `iostream` file I/O classes:**

- The `iostream` library contains file-related I/O template specializations.
- The `ifstream` class is derived from the `istream` class, and instances of `ifstream` are used to represent input files.
- The `ofstream` class is derived from the `ostream` class, and instances of `ofstream` are used to represent output files.

16.2.2. Files as objects

16.2.2.1. **Characteristics of input and output files:**

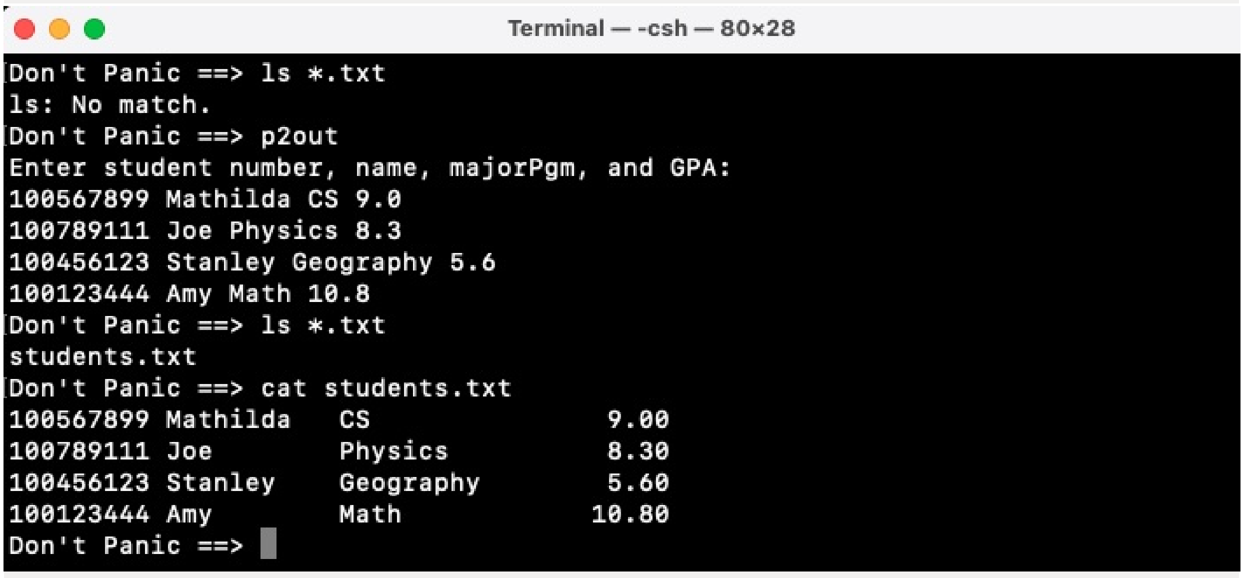
- Every instance of `ifstream` or `ofstream` maintains an internal file buffer object.
- The destructor for this file buffer closes the corresponding file.
- Every file maintains a set of error flags to indicate its current state, and an input file also stores an `eofbit` flag.
- The overloaded operators to test for errors and the end of the file can be used with files, as with any stream.

16.2.2.2. **Useful member functions for input and output files:**

- The `ifstream` and `ofstream` classes provide many useful member functions to work with input and output files.
- The constructor can optionally open the file. A second constructor parameter is specified to indicate the mode of file opening, for example input, output, append, and so on.
- The `ifstream` and `ofstream` classes also provide member functions for file management, for example opening or closing the file.
- Some useful member functions for input files:
  - the stream extraction operator (`>>`)
  - `get()` or `getline()`
- Some useful member functions for output files:
  - the stream insertion operator (`<<`)
  - `put()` or `write()`
  - `flush()`

16.2.3. Coding example: File output

```
 1 int main()
 2 {
 3   string num, name, majorPgm;
 4   float  gpa;

 6   ofstream outFile("students.txt", ios::out);
 7   if (!outFile) {
 8     cout<<"Could not open file"<<endl;
 9     exit(1);
10   }

12   cout<<"Enter student number, name, majorPgm, and GPA: " << endl;
13   while ( cin >> num >> name >> majorPgm >> gpa ) {
14     Student stu(num, name, majorPgm, gpa);
15     outFile << stu;
16   }

18   return 0;
19 }
```



```
Don't Panic ==> ls *.txt
ls: No match.
Don't Panic ==> p2out
Enter student number, name, majorPgm, and GPA:
100567899 Mathilda CS 9.0
100789111 Joe Physics 8.3
100456123 Stanley Geography 5.6
100123444 Amy Math 10.8
Don't Panic ==> ls *.txt
students.txt
Don't Panic ==> cat students.txt
100567899 Mathilda   CS                 9.00
100789111 Joe        Physics            8.30
100456123 Stanley    Geography          5.60
100123444 Amy        Math              10.80
Don't Panic ==> 
```

Program-16.2: File output

Program purpose:

- Program-16.2 demonstrates the use of an output file to store user-entered data.
- The program repeatedly prompts the end-user to enter student data, creates a new `Student` object for each one, and calls the `Student` class's overloaded stream insertion operator to output each `Student` object to a file.
- The `Student` class used in this program is identical to Program-13.5.

Lines 6-10:

- These lines open a file for output and check that the operation succeeded.
- Line 6 declares an instance of the `ofstream` class, called `outFile`. The first parameter to the `ofstream` constructor indicates that the file name is `"students.txt"`, and the second parameter specifies that the file must be opened for output.
- Line 7 uses the overloaded logical NOT operator to test if the file was opened successfully. If not, lines 8-9 print out an error message and terminate the program.

- There are several reasons why opening a file for output might fail. For example, the end-user may not have write permission in the directory where the file is created.

Lines 13-16:

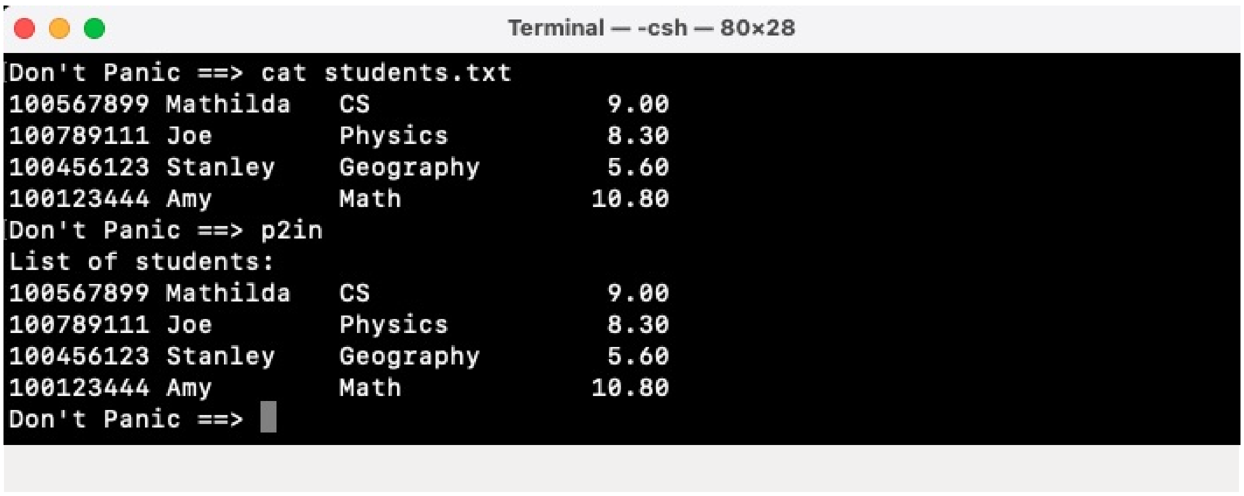
- These lines contain a loop that reads data from the end-user until the end-of-file value is entered.
- Line 13 shows the `while`-loop header. It uses the stream extraction operator to read four values from the end-user. When the user enters the end-of-file value, the `eofbit` flag for `cin` gets set. Testing the `cin` object returns true as long as no error occurs, and false when it reads in the end-of-file value. At that point, the loop terminates.
- At every iteration of the loop, line 14 creates a new `Student` object with the entered values. Then line 15 uses the stream insertion operator to output that object to the `outFile` file.
- When the program ends, the `outFile` object's destructor closes the `"students.txt"` file.

Program output:

- The program output shows that the `"students.txt"` file doesn't exist beforehand.
- After we run the program and enter the data, the file has been created and contains the `Student` data that we entered.

16.2.4. Coding example: File input

```
int main()
{
  string num, name, majorPgm;
  float  gpa;
  vector<Student> stuVect;
  ostream_iterator<Student> outItr(cout);

  ifstream inFile("students.txt", ios::in);
  if (!inFile) {
    cout<<"Could not open file"<<endl;
    exit(1);
  }

  while ( inFile >> num >> name >> majorPgm >> gpa ) {
    Student stu(num, name, majorPgm, gpa);
    stuVect.push_back(stu);
  }

  cout << "List of students:" << endl;
  copy(stuVect.begin(), stuVect.end(), outItr);

  return 0;
}
```



```
Don't Panic ==> cat students.txt
100567899 Mathilda   CS              9.00
100789111 Joe        Physics         8.30
100456123 Stanley    Geography       5.60
100123444 Amy        Math           10.80
Don't Panic ==> p2in
List of students:
100567899 Mathilda   CS              9.00
100789111 Joe        Physics         8.30
100456123 Stanley    Geography       5.60
100123444 Amy        Math           10.80
Don't Panic ==> 
```

Program-16.3: File input

Program purpose:

- Program-16.3 demonstrates the use of an input file to read in student data.
- The program reads in the student data from a file, creates a new `Student` object for each one, and adds it to an STL `vector`. The `vector` is then printed to the screen.
- The `Student` class used in this program is identical to Program-13.5.

Lines 8-12:

- These lines open a file for input and check that the operation succeeded.
- Line 8 declares an instance of the `ifstream` class, called `inFile`. The first parameter to the `ifstream` constructor indicates that the file name is `"students.txt"`, and the second parameter specifies that the file must be opened for input.
- Line 9 uses the overloaded logical NOT operator to test if the file was opened successfully. If not, lines 10-11 print out an error message and terminate the program.
- There are several reasons why opening a file for input might fail. For example, the file may be missing, or the end-user may not have read permission in the directory where the file is located.

Lines 14-17:

- These lines contain a loop that reads data from the input file until the end-of-file marker is reached.
- Line 14 shows the `while`-loop header. It uses the stream extraction operator to read four values from the input file. When the end-of-file value is reached, the `eofbit` flag for the input file gets set. Testing the `inFile` object returns true as long as no error occurs, and false when it reads in the end-of-file value. At that point, the loop terminates.
- At every iteration of the loop, line 15 creates a new `Student` object with the values read from the input file. Then line 16 adds that object to an STL `vector` called `stuVect`.

Lines 19-20:

- Line 20 uses the STL `copy()` algorithm, with an output stream iterator, to print out the entire contents of `stuVect` to the screen.
- When the program ends, the `inFile` object's destructor closes the `"students.txt"` file.

Program output:

- The program output shows that the data stored in the `"students.txt"` file in Program-16.2 is read from the input file into the STL `vector`, and the same data is printed to the screen.

16.3. Error state flags

We discuss the error state flags associated with every stream in C++.

16.3.1. Concepts

16.3.1.1. What are *error state flags*:

- Every stream object maintains a set of error flags that indicate its current state. These are called the stream's *error state flags*.
- The flags include:
  - the `goodbit` flag: if set, it indicates that none of the other error flags are currently set
  - the `failbit` flag: if set, it indicates that a formatting error has occurred
  - the `badbit` flag: if set, it indicates an unrecoverable error
- In addition, instances of the `istream` class, including its `ifstream` derived objects, maintain an `eofbit` flag. If set, it indicates that the end of the input has been reached.

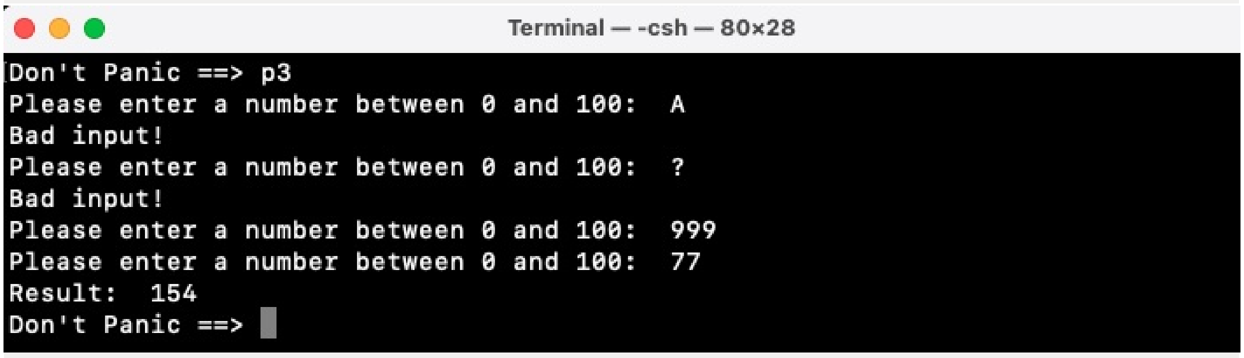
16.3.1.2. Stream member functions for testing error state flags:

- The following member functions are used to test the individual flags:
  - `fail()` returns true if `failbit` is set
  - `bad()` returns true if `badbit` is set
  - `eof()` returns true if `eofbit` is set
  - `good()` returns true if `goodbit` is set
- We can unset a stream's error state flags with the `clear()` member function. By default, this member function clears all the errors and sets the `goodbit` flag.

16.3.2. Coding example: Error state flags

```
int main()
{
  bool inputOk = false;
  int  num, result;

  while (!inputOk) {
    cout << "Please enter a number between 0 and 100:  ";
    cin >> num;
    inputOk = checkNum(num);
  }

  doubleNum(num, result);
  cout << "Result:  " << result << endl;

  return 0;
}

void doubleNum(int n, int& res)
{
  res = n * 2;
}

```

```
23 bool checkNum(int n)
24 {
25   if (!cin.good()) {
26     cout << "Bad input!" << endl;
27     cin.clear();
28     cin.ignore();
29     return false;
30   }

32   return ( n>=0 && n<=100);
33 }
```



```
Don't Panic ==> p3
Please enter a number between 0 and 100:  A
Bad input!
Please enter a number between 0 and 100:  ?
Bad input!
Please enter a number between 0 and 100:  999
Please enter a number between 0 and 100:  77
Result:  154
Don't Panic ==>
```

Program-16.4: Error state flags

Program purpose:

- Program-16.4 demonstrates how the standard input stream is tested for errors.
- The program is a variation of Program-2.6, which prompts the end-user for a numeric value and doubles it.

Lines 1-21:

- Lines 1-16 show the implementation of the `main()` function.
- Lines 6-10 contain a loop that prompts the end-user until a valid value is entered, and line 9 calls the `checkNum()` function to verify the value.
- Line 12 calls the `doubleNum()` function to double the value, and line 13 prints out the result.
- Lines 18-21 show the implementation of the `doubleNum()` function.

Lines 23-33:

- These lines show the implementation of the `checkNum()` function.
- Line 25 calls the `istream` class's `good()` member function to test the `cin` object for errors.
- If line 25 detects an error, line 26 prints out an error message. Then line 27 clears the standard input stream of errors, line 28 forces the standard input stream to skip over the invalid value, and line 29 returns false to indicate that the entered value is invalid.
- If no input error is detected, we know that a numeric value was entered. Line 32 tests the entered value for the correct range and returns the result of that test.
- From the program output, we see that a letter and punctuation character are entered instead of a numeric value in the first two iterations of the loop on lines 6-10. An error message is printed out, and the user is prompted again for a valid value.

Bibliography

[1] Free Software Foundation, Inc., “GCC, the GNU compiler collection,” https://gcc.gnu.org/, 2025.

[2] S. I. Feldman, “Make - a program for maintaining computer programs,” *Software: Practice and Experience*, vol. 9, no. 4, pp. 255–265, 1979. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380090402

[3] “Standard C++ library reference,” https://cplusplus.com/reference/, 2025.

[4] “C++ reference,” https://en.cppreference.com/w/, 2025.

[5] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Addison-Wesley, 1997.

[6] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Addison-Wesley, 1999.

[7] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[8] B. H. Liskov, “Keynote address - data abstraction and hierarchy,” in *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1988. [Online]. Available: https://api.semanticscholar.org/CorpusID:14219043

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov, *C++ Standard Template Library*. Prentice Hall, 2000.