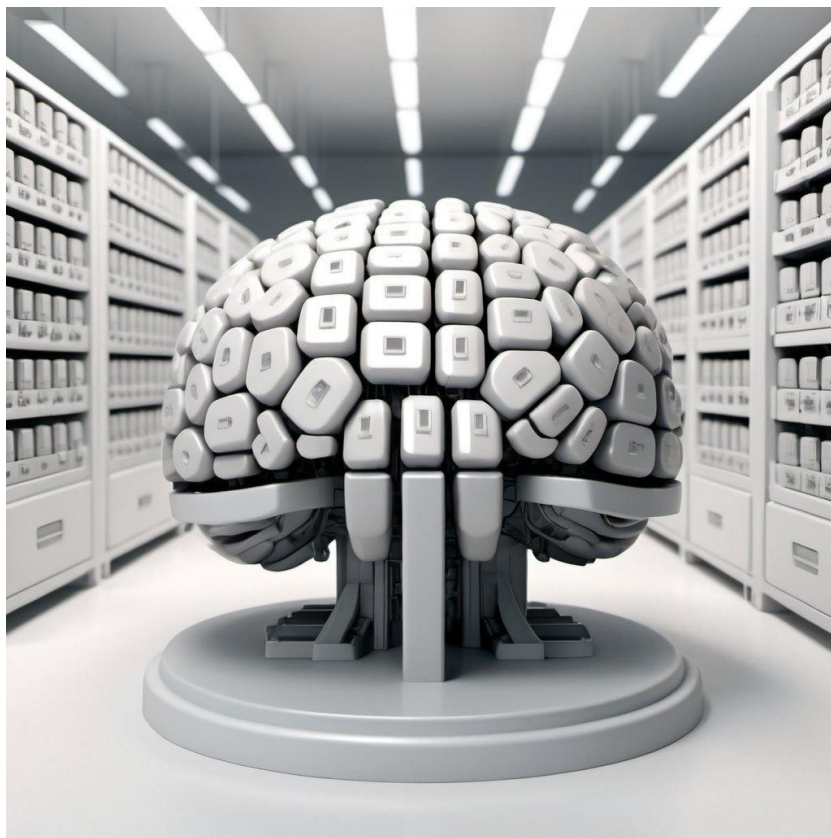

Chapter 3

Pointers and Memory Management

What is in This Chapter?

This chapter presents the fundamental programming concept of **pointers**. Pointers are the basis for efficient storage and reference of data. If you want to be a decent C programmer, it is vital that you fully understand how pointers are used. **Command-line arguments** are then explained, as they allow you to run your program with different parameters without having to re-compile. There is a section on **Memory Management** that will help you understand the memory model being used in C. It explains how and where everything is stored so that you properly understand how the memory is being used by your program. **Dynamic Memory Allocation** is then discussed, which allows you to write flexible code that can handle changes in data size. The final section discusses a couple of programming examples that make use of Dynamic Memory Allocation. The examples are the construction and usage of **Singly-Linked Lists** and **Doubly-Linked Lists**. The concept of flexible-storage data structures, such as these linked lists, will be important for you to understand in your life as a C programmer.



3.1 Pointers

Students learning C programming often find it difficult to work with pointers. Pointers, however, are fairly simple conceptually. In fact, we have already been using them in some of our code. It will be important for you to understand the fundamental concept of a pointer and get lots of practice using them. So, what is a pointer?

*A **pointer** is a reference to a memory location.
(the term is also used for a variable that stores a memory address).*



So, a pointer refers to (i.e., is a reference to) a place in memory where some data is stored.

Perhaps the simplest analogy may be to compare pointers to indices in an array. Each array item has its own **location** (or **address**) within the array at which it is stored. The **index** of the item in the array is like a **pointer** to that item:

245	83	5	232	171	83	220	44
0	1	2	3	4	5	6	7

An array index is like an “address” in memory.
Index **2** is like a “pointer” to the value **5** stored there.



In a sense, all of the computer’s memory is like an array of consecutive bytes in memory. Therefore, an **address** in memory, can be viewed as an **index** somewhere within the memory’s large array of bytes. So, to keep things simple, imagine a pointer to simply be an index into an array.

In memory-managed languages, such as JAVA, we don’t really have to be concerned about where things are stored in memory. All of that is hidden from the programmer. It makes a programmer’s life much more pleasurable, allowing him/her to focus on higher level tasks at hand. For example, in JAVA, each time we create an object using a constructor, we actually get back a reference to (i.e., a pointer to) the object’s virtual memory location. We did something simple like this:

```
Person    p = new Person();
```

Here, **p** actually stores a pointer to the memory location at which the **Person** object is stored.



In C, however, nobody manages memory for us ... so we often need to be aware of where (and how) our data is stored in memory. We will talk more about how to allocate and deallocate memory later. For now, we need to understand just the basics of simple pointers.

A pointer can store, as its value, the memory address of either:

- a variable, or
- a block of memory that you reserved (a.k.a. allocated) yourself.

Since pointers are somewhat confusing to people, why bother using them? There are a few reasons:

- ✓ pointers can be stored in a “fairly small” fixed-size variable (**8 bytes**)
- ✓ pointers allow you to change memory that is out of scope (i.e., outside the function)
 - e.g., you can modify a variable that is passed in as a parameter
- ✓ pointers allow you to have more than one variable pointing to (i.e., sharing) the same data which can save us from having to copy the same data multiple times.

In C, a pointer is identified by a ***** character in front of the variable name:

```
int *salary;
```

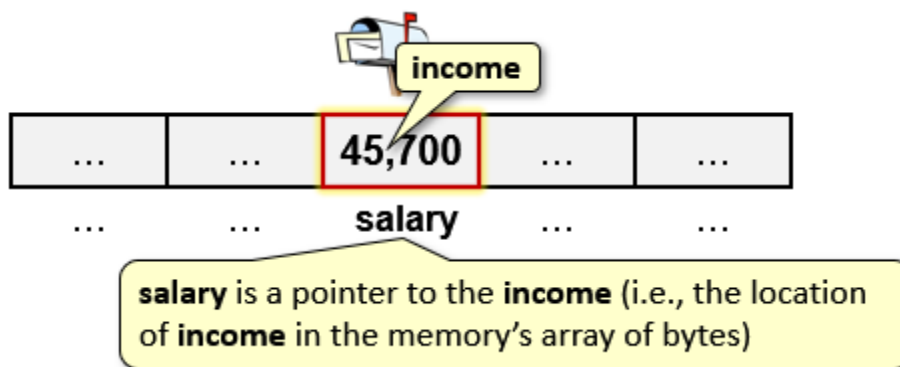
This means, for example, that `salary` is NOT an `int` ... but instead it points to the memory location that contains an `int`. Consider this coding example which shows the difference between an `int` and an `int *`:

```
int income;
int *salary;

income = 45700;
salary = &income; // salary is a pointer to the income variable
```

The **&** operator returns the memory **address** of a variable.

Notice that the `salary` variable is pointing to the `income` variable's memory location (i.e., the address (or **&**) of `income`). Visually, imagine the pointer as follows:



When printing an address, you should typecast to a **(void *)** and use **%p** to display it in hexadecimal.

```
printf("income = %u\n", income);
printf("salary = %p\n\n", (void *)salary);
printf("address of income = %p\n", (void *)&income);
printf("address of salary = %p\n", (void *)&salary);
```

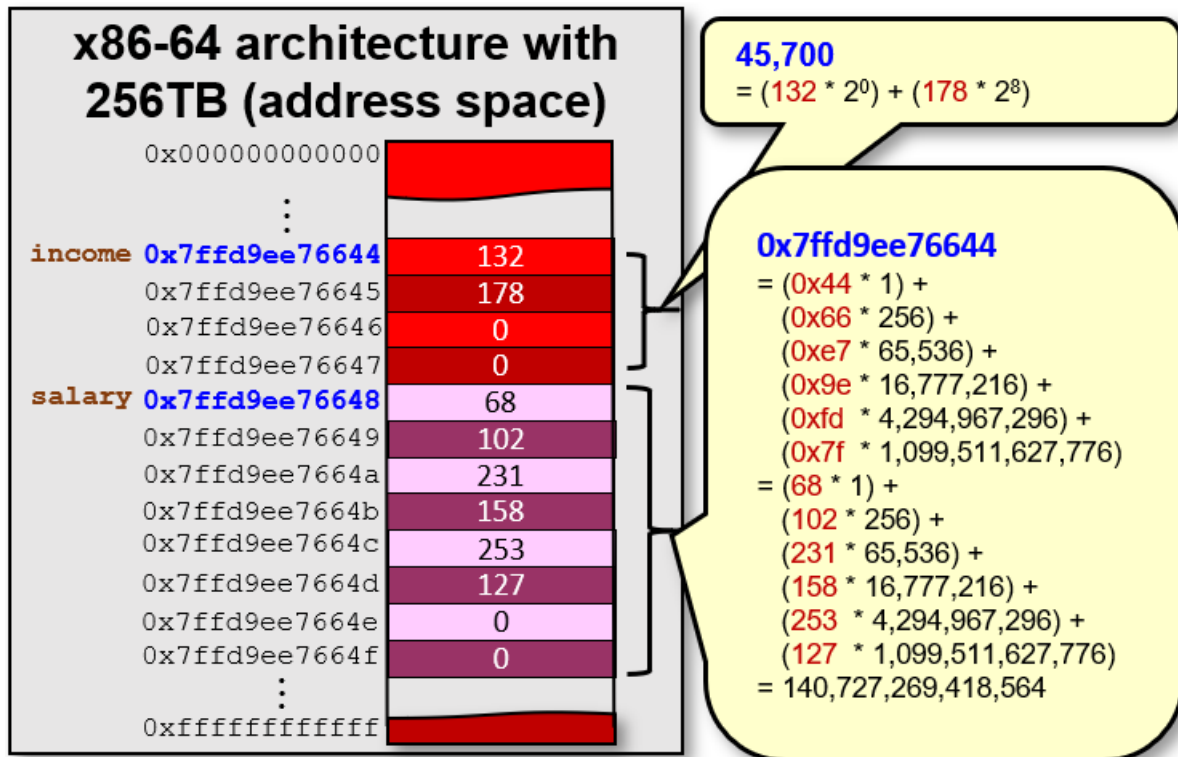
Use **%p** and **(void *)** to print out a pointer.

Here is the output, although you should realize that the memory locations will change each time that you run the code:

```
income = 45700
salary = 0x7ffd9ee76644
```

Memory **addresses** are values up to 256TB!!

```
address of income = 0x7ffd9ee76644
address of salary = 0x7ffd9ee76648
```



As you can see, the bytes stored in the `salary` variable represent the *Little Endian* byte order for the number 140,727,269,418,564... which is the memory address of the `income` variable. Changing the value of the `income` variable will not alter the value of the `salary` variable since the `income` variable stays in the same location regardless of what value it has.

```
income = 52300;
printf("income = %u\n", income);           // prints 52300 now
printf("salary = %p\n\n", (void *)salary); // still 0x7ffd9ee76644
```

You will notice that the pointer addresses in the image above are **48-bit** addresses. This gives a range of $2^{48} = 281,474,976,710,656$ (i.e., 256TB) unique addresses! That is a lot of address space. Under the x86-64 architecture, even though it is **64-bit**, only **48** bits are used. The topmost **16** bits are zeroed.



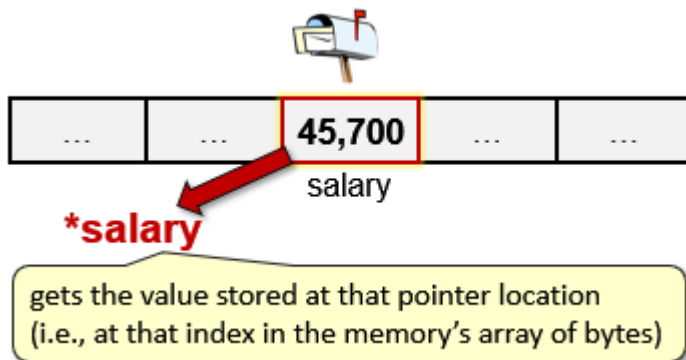
The `*` symbol is also used to **dereference** the value of a pointer.

Dereferencing a pointer means getting the value that is stored in the memory location pointed to by the pointer.

Continuing our previous example, we can ask for the “value being pointed to by” the `salary` variable:

```
income = 45700;
printf(" salary = %p\n", (void *)salary);           // still 0x7ffd9ee76644
printf("*salary = %u\n", (unsigned int)*salary); // prints 45700
```

So, `*salary` gives us the value at the memory address that `salary` “is pointing to” ... which is the value of the `income` variable, since `salary` points to the `income` variable’s address.

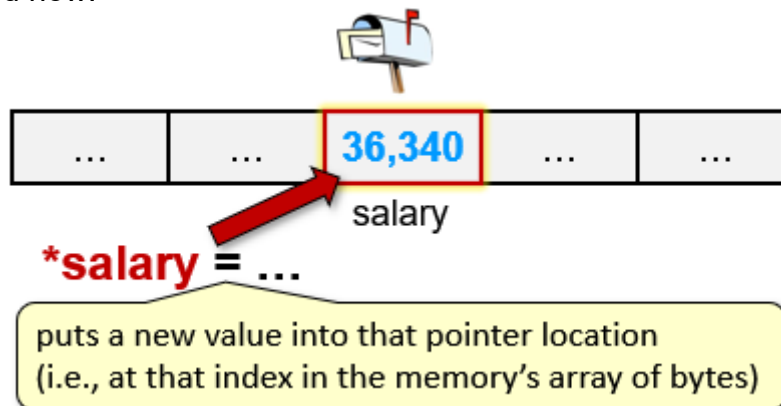


That is not so difficult to understand. But here is where it gets tricky. We are able to assign a value to `*salary`. That is, when we use `*salary` to the left of the assignment operator, we are *changing the value that is stored at the “address that salary is pointing to”*:



```
*salary = 36340;
printf("income = %u\n", income);           // prints 36340 now!
printf(" salary = %p\n", (void *)salary); // still 0x7ffd9ee76644
printf("*salary = %u\n", (unsigned int)*salary); // prints 36340
```

Notice that since we changed the value at the “location pointed to by” `salary`, the `income` variable’s value has also changed now.



The value of a pointer can change at any time throughout our program. There are two situations that can cause problems ... when a pointer is ...

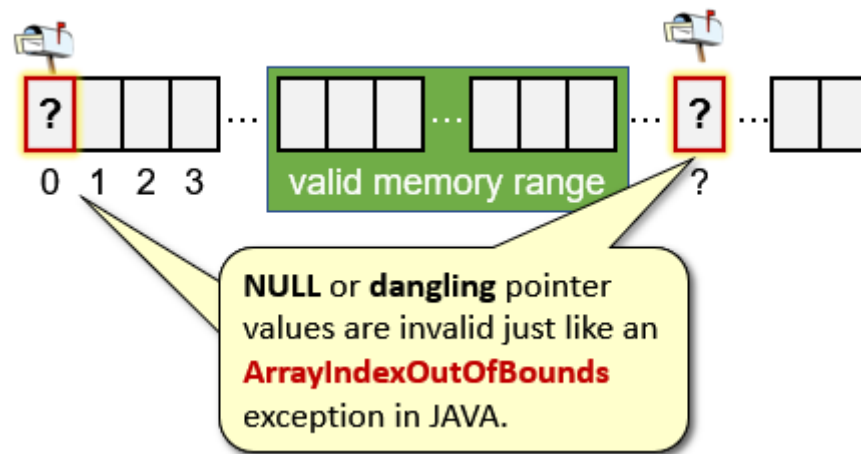
- **NULL** (i.e., uninitialized)
- **dangling** (i.e., pointing to an invalid/corrupt location)

Ideally, whenever we have a pointer variable that has not been assigned a valid memory address yet, we should initialize it with **NULL** ... which is easily distinguishable from a dangling or invalid pointer. But we need to be careful not to dereference **NULL** or dangling pointers:

```
salary = NULL;
printf(" salary = %p\n", (void *)salary);
printf("*salary = %u\n", (unsigned int)*salary); // BAD IDEA!
*salary = 200; // BAD IDEA!
```



In either case of dereferencing `salary` in the above code, the **NULL** pointer is not a valid memory location. Therefore, the program will stop with a **segmentation fault** ... which is like a **NullPointerException** in Java.



We can always check for this first though:

```
if (salary != NULL)
    printf("*salary = %u\n", (unsigned int)*salary);

if (salary != NULL)
    *salary = 200;
```

However, if `salary` was a dangling pointer (i.e., pointing to some invalid address) then things are much worse. Why? Because you would be accessing and/or modifying memory locations containing other parts of your program!

The result is that your program may crash right away, or you may be overwriting some important data ... or your code may crash at some other point in your code ... making it very difficult to debug.

Keep in mind that with pointers, we can point to any type of variable:

```
float      *age;
char       *name;
AddressType *home;
PersonType *friend;
```

Whenever we use pointers on arrays, the pointer typically points to the address of the first element of the array. Consider this array:

intArr	23	54	67	88	43	12	83	46
	0	1	2	3	4	5	6	7

With code:

```
int intArr[8] = {23, 54, 67, 88, 43, 12, 83, 46};
```

For this array, if we just use the variable `intArr` (i.e., without brackets) in our code, that is equivalent to using `&(intArr[0])`. That is, when we just use the name of the array, without an index, it really means that we have a “pointer to the first element of” the array. So, the following will print out the same address value:

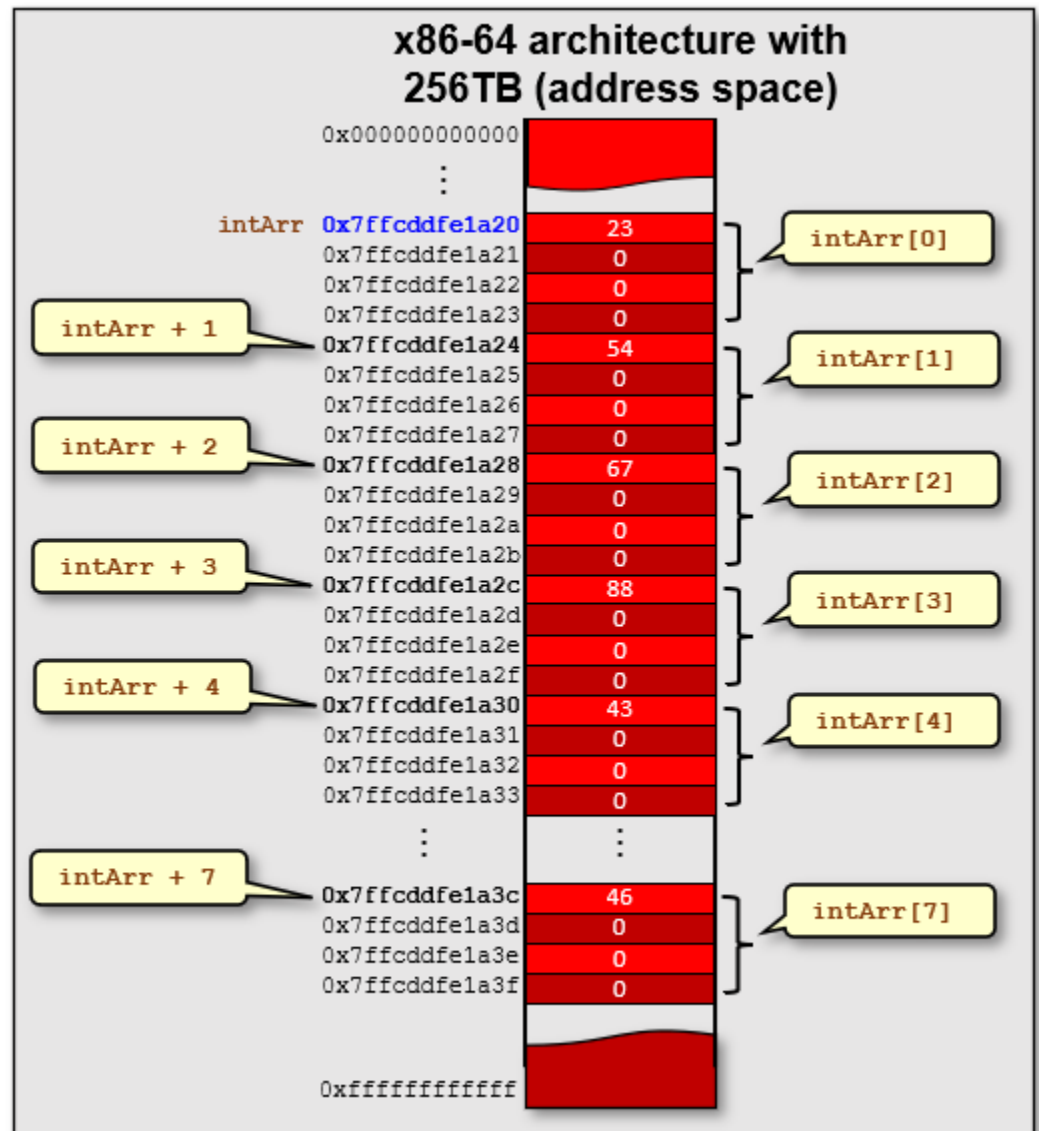
```
printf("%p\n", (void *)intArr);
printf("%p\n", (void *)&(intArr[0]));
```

Interestingly, since memory addresses are just numbers, we can add to and subtract from them to get memory addresses *before* and *after* a pointer address. This adding and subtracting, however, is with respect to the size of the elements in the array.

So, for example, if we use `intArr + 3` then we get the address of the fourth element in the array (since arrays start at 0).

Note that it is NOT the memory location that is 3 bytes after the array's memory location. Rather, it is 12 bytes after (since an `int` requires 4 bytes).

Therefore, given a pointer `ptr` to an array, the value of `ptr + n` is the address of the array plus `n * sizeof(ptr[0])`.



Make sure that you understand the following example:

Code from `arrayPointers.c`

```
#include <stdio.h>

int main() {
    int intArr[8] = {23, 54, 67, 88, 43, 12, 83, 46};

    printf("int array addr: %p \n", (void *)intArr);
    printf("First item addr: %p \n", (void *)&intArr[0]);
    printf("Last item addr: %p \n\n", (void *)&intArr[7]);

    printf("First int:          %d \n", intArr[0]);
    printf("First int again:    %d \n", *intArr);
    printf("First int plus 3: %d \n", *intArr + 3);
    printf("Fourth int:         %d \n", *(intArr + 3));
    printf("\n");

    int *ptr;

    ptr = &(intArr[6]);
    ptr = intArr + 6; // does same as above

    printf("Seventh int: %d \n", *ptr);
    printf("Eighth int:  %d \n", ptr[1]);
    printf("Fifth int:   %d \n", *(ptr - 2));

    char charArr[32] = "SAM PULL";

    printf("\n");
    printf("char array addr: %p \n", (void *)charArr);
    printf("First item addr: %p \n", (void *)&charArr[0]);
    printf("Last item addr: %p \n\n", (void *)&charArr[7]);

    printf("First char:          %c \n", charArr[0]);
    printf("First char again:    %c \n", *charArr);
    printf("First char plus 4: %c \n", *charArr + 4);
    printf("Fifth char:         %c \n", *(charArr + 4));
    printf("\n");

    char *cptr;

    cptr = &(charArr[4]);
    cptr = charArr + 4; // does same as above

    printf("Fifth char: %c \n", *cptr);
    printf("Sixth char: %c \n", cptr[1]);
    printf("Third char: %c \n", *(cptr - 2));
    printf("\n");

    return 0;
}
```

Here is the output (keep in mind that memory locations will differ each time):

```
int array addr: 0x7ffcddfe1a20
First item addr: 0x7ffcddfe1a20
Last item addr: 0x7ffcddfe1a3c
```

```
First int:      23
First int again: 23
First int plus 3: 26
Fourth int:     88
```

```
Seventh int: 83
Eighth int:  46
Fifth int:   43
```

```
char array addr: 0x7ffcddfe1a40
First item addr: 0x7ffcddfe1a40
Last item addr:  0x7ffcddfe1a47
```

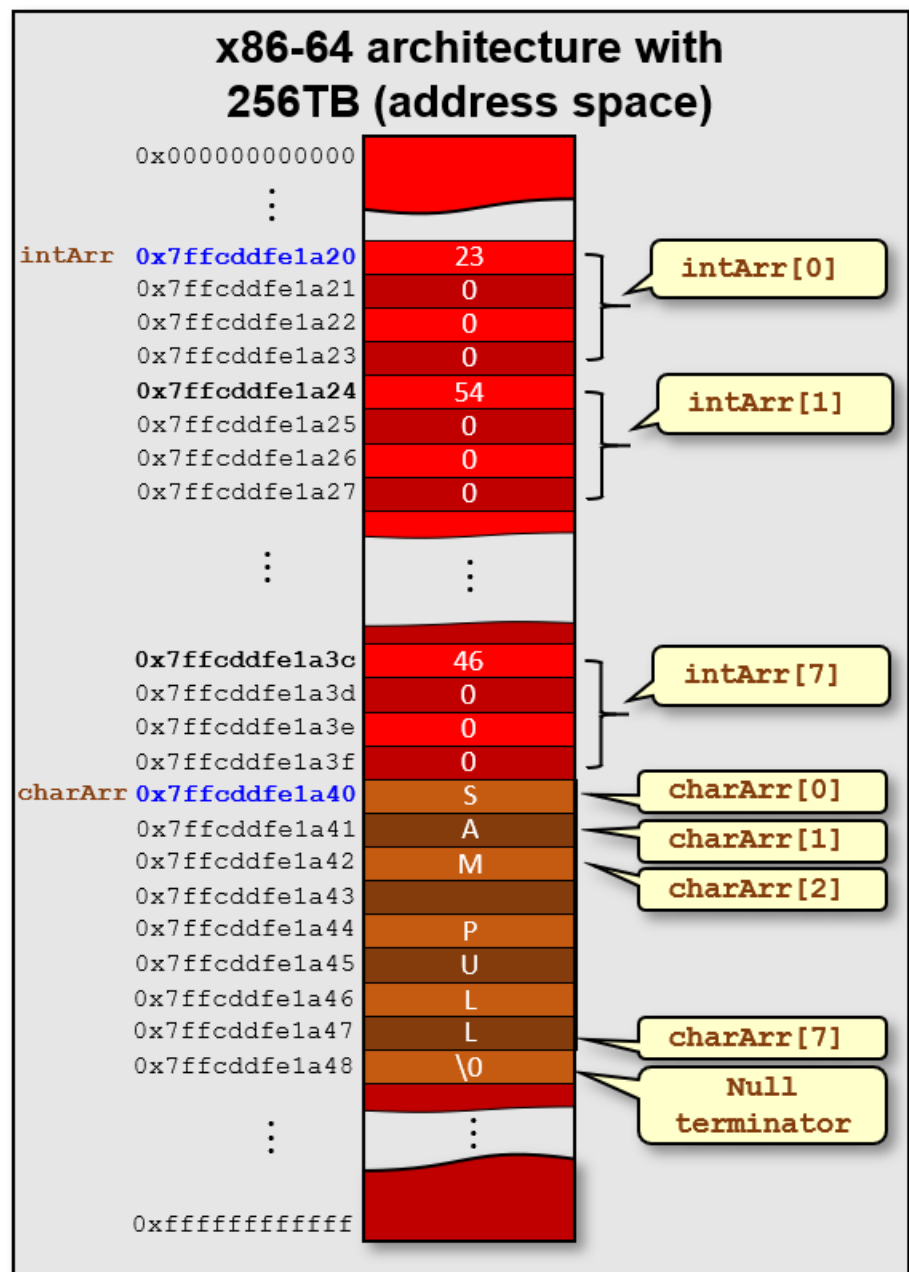
```
First char:      S
First char again: S
First char plus 4: W
Fifth char:      P
```

```
Fifth char: P
Sixth char: U
Third char: M
```

Make sure that you understand how the values are stored in memory ... as this is a key to understanding how to program well in C.



What about pointers to structures ... do they work the same way? ... **Yes!**



However, there is a different syntax that we generally use to dereference.

Consider this bank account type:

```
typedef struct {
    char *owner;
    int   accNumber;
    float balance;
} BankAccountType;
```

Recall that we can set the values of a variable of this type as follows:

```

BankAccountType  account;

account.owner = "Rob Banks";
account.accNumber = 190219;
account.balance = 2573.81;

```

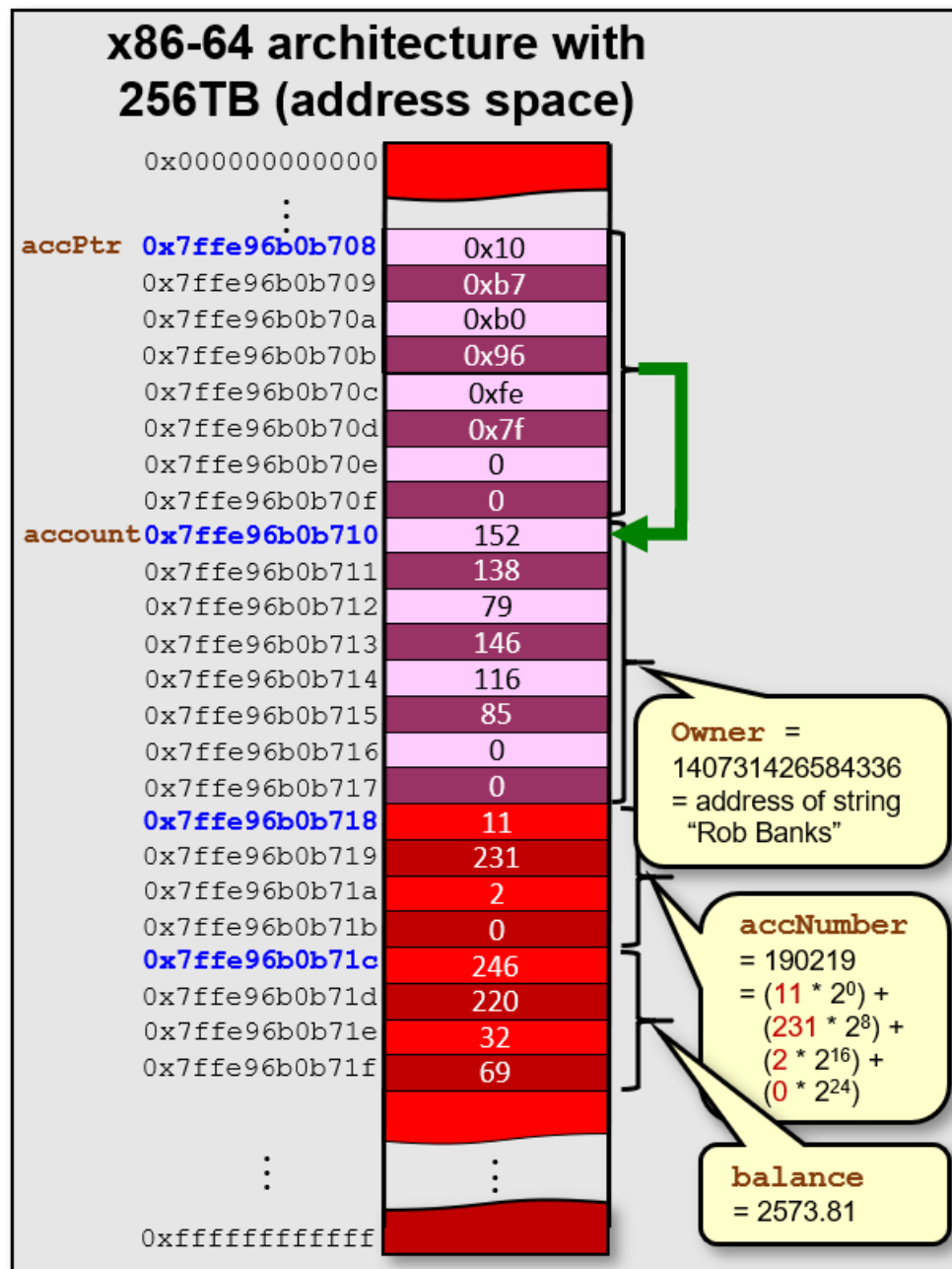
Now consider a pointer to the account:

```

BankAccountType  *accPtr = &account;

```

Here **accPtr** points to the same data that is stored in the **account** variable:



We can dereference the pointer and then access its internals by using the dot operator:

```
(*accPtr).owner = "Robin Banks";
(*accPtr).accNumber = 193248;
(*accPtr).balance = (*accPtr).balance - 573.00;
```



This will alter the contents of the structure's attributes to be the new values. However, it is a bit cumbersome to put the brackets, * and . characters to do this. The C language developers wanted to simplify things, so they came up with another syntax for dereferencing struct pointer attributes. The -> characters can also be used, which is simpler:

```
accPtr->owner = "Robin Hood";
accPtr->accNumber = 193249;
accPtr->balance = accPtr->balance - 200.00;
```



Here is a full program to test this:

Code from **structPointers.c**

```
#include <stdio.h>

// Structure that represents a simple bank account
typedef struct {
    char *owner;
    int accNumber;
    float balance;
} BankAccountType;

int main() {
    BankAccountType account;

    account.owner = "Rob Banks";
    account.accNumber = 190219;
    account.balance = 2573.81;

    printf("%s' account (#", account.owner);
    printf("%d) with ", account.accNumber);
    printf("$%0.2f\n", account.balance);

    BankAccountType *accPtr = &account;
    printf("account = %p\n", (void *)accPtr);
    printf("accPtr = %p\n", (void *)&accPtr);

    (*accPtr).owner = "Robin Banks";
    (*accPtr).accNumber = 193248;
    (*accPtr).balance = (*accPtr).balance - 573.00;
    printf("%s' account (#", (*accPtr).owner);
    printf("%d) with ", (*accPtr).accNumber);
    printf("$%0.2f\n", (*accPtr).balance);

    accPtr->owner = "Robin Hood";
    accPtr->accNumber = 193249;
    accPtr->balance = accPtr->balance - 200.00;
    printf("%s's account (#", accPtr->owner);
    printf("%d) with ", accPtr->accNumber);
    printf("$%0.2f\n", accPtr->balance);
}
```

Here is the output ... it is fairly straight forward:

```
Rob Banks' account (#190219) with $2573.81
account = 0x7ffe96b0b710
accPtr = 0x7ffe96b0b708
Robin Banks' account (#193248) with $2000.81
Robin Hood's account (#193249) with $1800.81
```



The use of pointers can speed up our program when it comes to calling functions. It allows us to pass a reference to some data rather than passing the entire set of data. For example, consider the following **typedefs** which define a person and a student:

<pre>typedef struct { char *first; char *last; int age; } PersonType;</pre>	$(8 + 8 + 4 + 4 \text{ padding})$ $= 24 \text{ bytes}$ $(24 + 8 + 4 + 4 \text{ padding})$ $= 40 \text{ bytes}$	<pre>typedef struct { PersonType personalInfo; char *stuNumber; float gpa; } StudentType;</pre>
---	---	---

We can verify the sizes with these lines of code:

```
printf("PersonType requires %zu bytes\n", sizeof(PersonType)); // = 24
printf("StudentType requires %zu bytes\n", sizeof(StudentType)); // = 40
```

In addition, extra storage would be required to store the characters of the three strings.

Consider creating a variable to hold one of these students and filling it up:

```
StudentType aStudent;

aStudent.personalInfo.first = "April";
aStudent.personalInfo.last = "Rain";
aStudent.personalInfo.age = 22;
aStudent.stuNumber = "100444555";
aStudent.gpa = 9.0;
```

Now consider a simple function which is supposed to increase the age of a student:

```
void increaseAge(StudentType stu) {
    stu.personalInfo.age++;
}
```

If we were to call this function with the student that we just created, what would happen?

```
increaseAge(aStudent);
```

Well, looking inside the function, it goes into the **student** struct and gets the age and then increases it. However, why does the following code print out the same number twice?

```
printf("%d\n", aStudent.personalInfo.age); // displays 22
increaseAge(aStudent);
printf("%d\n", aStudent.personalInfo.age); // displays 22
```

The problem lies in the way in which the student is passed to the function. Recall our discussion about **Pass-by-value** and **Pass-by-reference** from chapter 1. In our code, which one are we doing? Are we passing in a value or are we passing in a reference?

We are in fact, passing in a value, not a reference. Notice the difference:

Pass-by-value

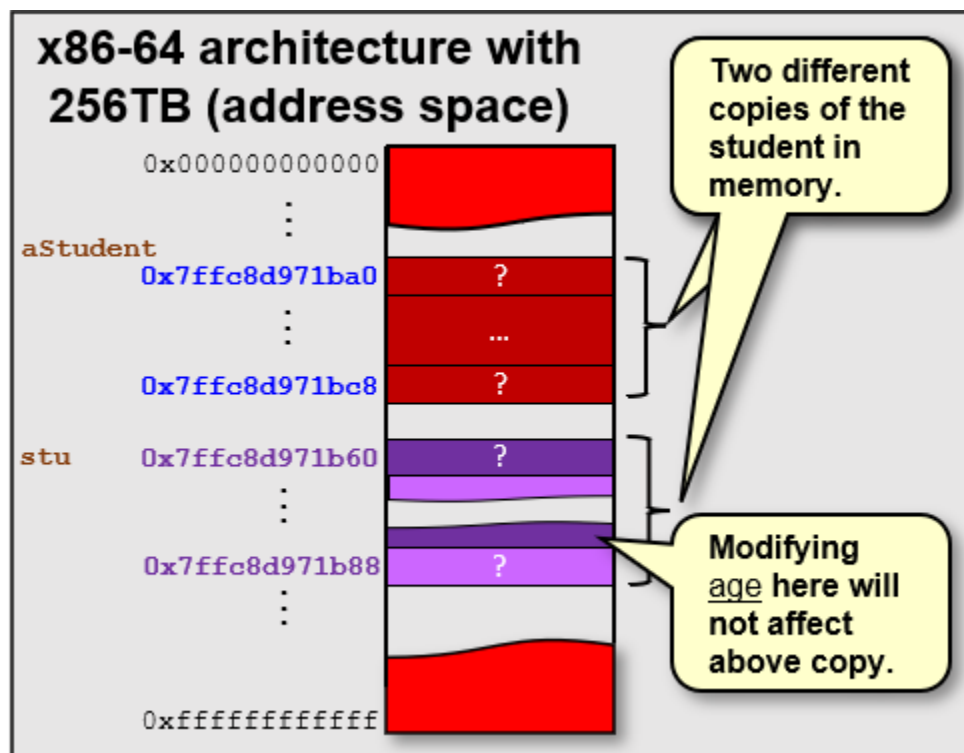
- value is copied into function
- function works on the local copy
- copy is lost when function returns
- value in calling function cannot be changed



Pass-by-reference

- address of value is passed into function
- value's data in the calling function can be changed

So, when we pass in `aStudent` to the function, the parameter `stu`, actually gets a copy of the student data. When we increase the age, we are increasing the copy's age, not the original's.



What actually happens to the `stu` variable once the function completes? It is discarded. The memory is freed up again once the function returns. Therefore, it is useless if we try to modify the value of an incoming pass-by-value parameter within the function.

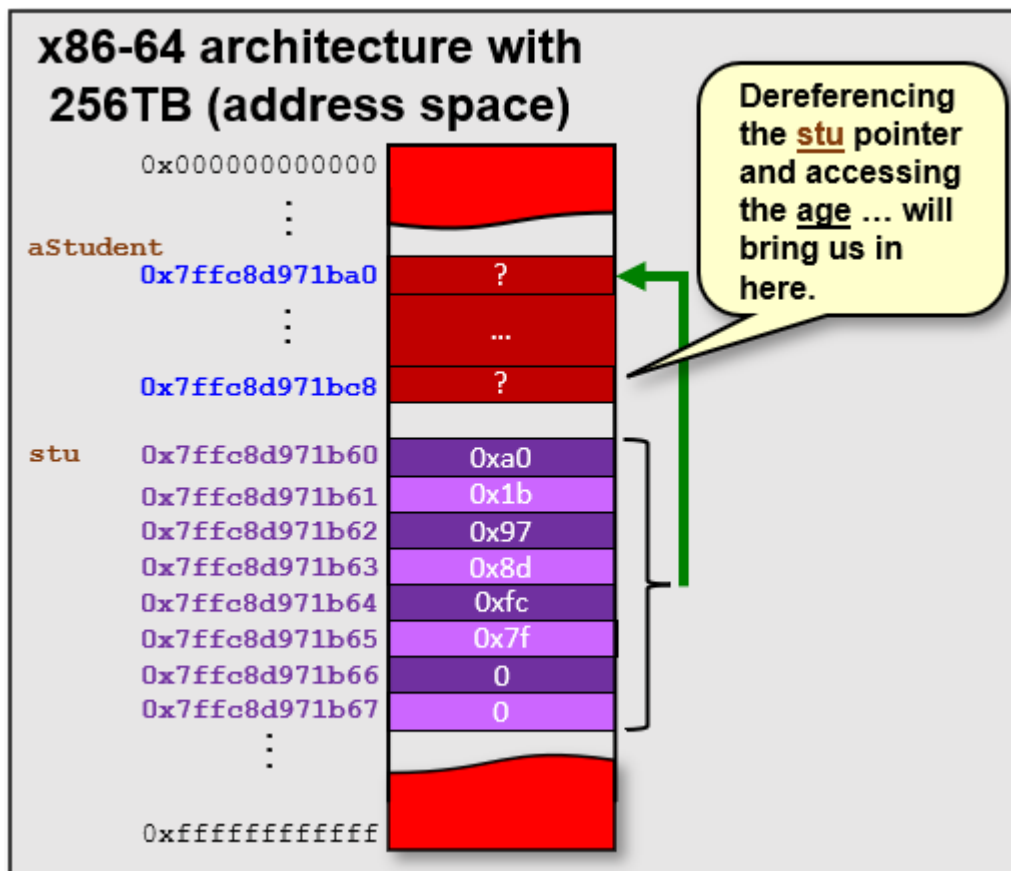
Now consider re-writing the function to take a **StudentType *** as follows:

```
void increaseAge(StudentType *stu) {
    stu->personalInfo.age++;
}
```

To call the function, we will pass in the address of the student as follows:

```
increaseAge(&aStudent);
```

Now what will happen? Well ... we are passing in a pointer to the memory location of the **struct** that contains the student's data ... so we are passing in a reference to the **struct**. Now, when we increase the **age**, we are increasing the **age** of the original student being passed in:



Hence, the following code prints out the correct results:

```
printf("%d\n", aStudent.personalInfo.age); // displays 22
increaseAge(&aStudent);
printf("%d\n", aStudent.personalInfo.age); // displays 23
```

Now consider an array of such students:

```
StudentType students[250];
int numStudents = 3;

students[0].personalInfo.first = "April";
students[0].personalInfo.last = "Rain";
students[0].personalInfo.age = 22;
```



```
students[0].stuNumber = "100444555";
students[0].gpa = 9.0;

students[1].personalInfo.first = "May";
students[1].personalInfo.last = "Flowers";
students[1].personalInfo.age = 24;
students[1].stuNumber = "100222333";
students[1].gpa = 8.7;

students[2].personalInfo.first = "June";
students[2].personalInfo.last = "Bugs";
students[2].personalInfo.age = 99;
students[2].stuNumber = "100777888";
students[2].gpa = 11.5;
```

Interestingly, we can set up a pointer to the beginning of the array like this:

```
StudentType *studentPtr = students; // &(students[0])
```

We can then iterate through the array without indices by increasing the pointer value:

```
for (int i=0; i<numStudents; i++) {
    increaseAge(studentPtr);
    ++studentPtr; // Go to the next student (or studentPtr++; )
}
```

Note that we simply increase the pointer with the `++` operator. This does not increase the pointer by **1**, but actually increases by **sizeof(StudentType)** ... which is **40**. Adding this line in the loop will verify this:

```
printf("studentPtr = %p\n", (void *)studentPtr);
```

You should see something like this (although numbers will vary each time you run):

```
studentPtr = 0x7ffc887bfa10
studentPtr = 0x7ffc887bfa38
studentPtr = 0x7ffc887bfa60
```

Here is a complete program for you to try:

Code from **moreStructPointers.c**

```
#include <stdio.h>
#include <string.h>

#define MAX_STUDENTS 250

typedef struct {
    char *first;
    char *last;
    int age;
} PersonType;
```

```
typedef struct {
    PersonType    personalInfo;
    char          *stuNumber;
    float         gpa;
} StudentType;

// Functions/Procedures used in this program
void increaseAge(StudentType *);
void printStudent(StudentType *);

int main() {
    StudentType students[MAX_STUDENTS];
    int         numStudents = 3;

    printf("StudentType requires %zu bytes\n", sizeof(StudentType));

    students[0].personalInfo.first = "April";
    students[0].personalInfo.last  = "Rain";
    students[0].personalInfo.age   = 22;
    students[0].stuNumber = "100444555";
    students[0].gpa = 9.0;
    students[1].personalInfo.first = "May";
    students[1].personalInfo.last  = "Flowers";
    students[1].personalInfo.age   = 24;
    students[1].stuNumber = "100222333";
    students[1].gpa = 8.7;
    students[2].personalInfo.first = "June";
    students[2].personalInfo.last  = "Bugs";
    students[2].personalInfo.age   = 99;
    students[2].stuNumber = "100777888";
    students[2].gpa = 11.5;

    printf("Age before increasing: %d\n", students[0].personalInfo.age);
    increaseAge(&students[0]);
    printf("Age after increasing: %d\n\n", students[0].personalInfo.age);

    StudentType *studentPtr = students;
    for (int i=0; i<numStudents; i++) {
        printf("studentPtr = %p\n", (void *)studentPtr);
        increaseAge(studentPtr);
        printStudent(studentPtr);
        ++studentPtr; // Go to the next student
    }
    printf("\n");
    return 0;
}

// Increases the student's age
void increaseAge (StudentType *s) {
    s->personalInfo.age++;
}

// Displays student to the console showing name, age and GPA.
void printStudent (StudentType *s) {
    printf("%d year old %s %s has a GPA of %.1f \n",
        s->personalInfo.age,
        s->personalInfo.first,
        s->personalInfo.last,
        s->gpa);
}
```

3.2 Command-Line Arguments

Up until now, we have written programs with a **main()** function that has no parameters. We will now consider what are called Command-Line Arguments.

Command-Line Arguments are parameters/values that can be passed into your program when it starts. These parameters/values are supplied in the command line when the program is run.



Command-line arguments in C are represented as an array of strings. If you want your program to read in these values, you need to supply parameters to the **main()** function.

Here are the options for the **main()** function signature:

```
int main() { ... } // no parameters
int main(int argc, char *argv[]) { ... } // we will use this
int main(int argc, char **argv) { ... } // this will make sense later
```

The **argc** parameter tells you how many command-line arguments there are while the **argv** array contains the strings that represent the arguments.

Here is a program that shows all the command-line arguments:

Code from **cmdLineArgs.c**

```
#include <stdio.h>

int main(int argc, char *argv[]) {

    printf("There are %d arguments\n", argc);

    for (int i=0; i<argc; ++i)
        printf("Argument %d is %s \n", i, argv[i]);

    return 0;
}
```

What would be the result when running this code? Well, it really depends on what you type in on the command-line when you run the program. Here are some examples:

```
student@COMPBase:~$ ./cmdLineArgs
There are 1 arguments
Argument 0 is ./cmdLineArgs
student@COMPBase:~$
```

```
student@COMPBase:~$ ./cmdLineArgs 24
There are 2 arguments
Argument 0 is ./cmdLineArgs
Argument 1 is 24
student@COMPBase:~$
```

```
student@COMPBase:~$ ./cmdLineArgs 24 67.934 false Mark
There are 5 arguments
Argument 0 is ./cmdLineArgs
Argument 1 is 24
Argument 2 is 67.934
Argument 3 is false
Argument 4 is Mark
student@COMPBase:~$
```

As you can see, the arguments are separated by space characters as their delimiter. Beware though, the arguments are all strings ... so if you want to enter numbers and use them in your program, you will have to perform a conversion. In the `<stdlib.h>` package, there are some functions for converting strings to other types:

```
char    *str = "...";

int      iVal;
double   dVal;
long int lVal;

iVal = atoi(str);
dVal = atof(str);
lVal = atol(str);
```



So, we could for example, write a program that reads in numbers from the command-line, converts them to **ints** and then performs some calculation (e.g., average) on them as follows:

Code from **average.c**

```
#include <stdio.h>
#include <stdlib.h> // needed for conversion function atoi

int main(int argc, char *argv[]) {
    double total = 0;

    for (int i=1; i<argc; ++i)
        total += atoi(argv[i]);

    printf("The average of those %d numbers is %0.1f\n", argc-1, total/(argc-1));

    return 0;
}
```

Notice that we subtract 1 from **argc** to get the actual number of numbers, since the program name is the first argument in the array. Here is the output after running a few times:

```
student@COMPBase:~$ ./average 12 64 55
The average of those 3 numbers is 43.7
student@COMPBase:~$
```

```
student@COMPBase:~$ ./average 1 2 3 4 5 6 7 8 9 10
The average of those 10 numbers is 5.5
student@COMPBase:~$
```

```
student@COMPBase:~$ ./average
The average of those 0 numbers is -nan
student@COMPBase:~$
```



The big advantage of using command-line arguments is that we can run our program many times with different values and we won't need to compile.

Often, command-line arguments are used for setting parameters to the program, as opposed to passing in data to be processed. For example, arguments are often:

- flags to enable/disable certain parts of your program
- file names
- number of items to be processed
- iterations to perform (e.g., simulation programs)
- etc..

3.3 Memory Management

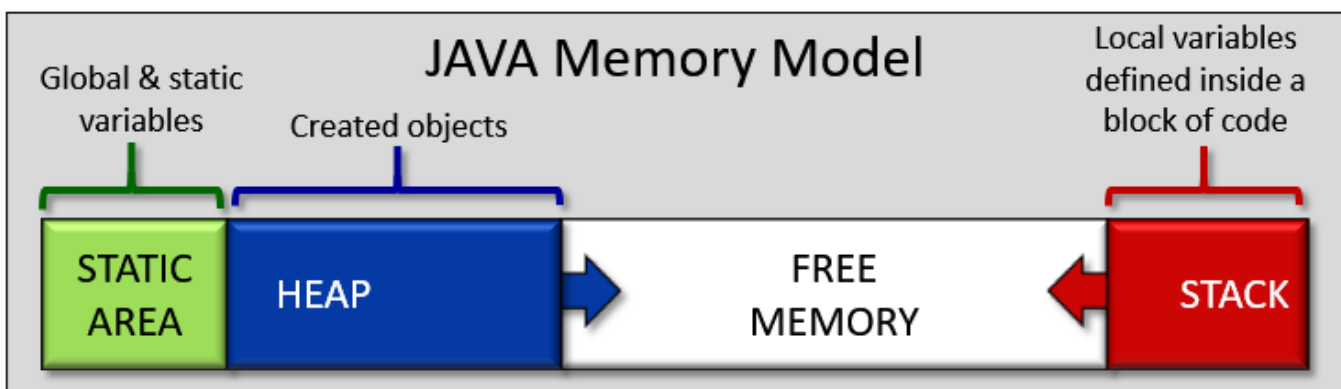
Some languages, such as JAVA, are memory-managed. That means that programmers do not need to concern themselves with allocating and deallocating chunks of memory to store data (e.g., objects). In JAVA, for example, we simply write code like this:

```
Car      myCar    = new Car("Red", "Porsche", "911");
Car      yourCar  = new Car("Green", "Ford", "Escort");
Person[] people   = new Person[200];
```

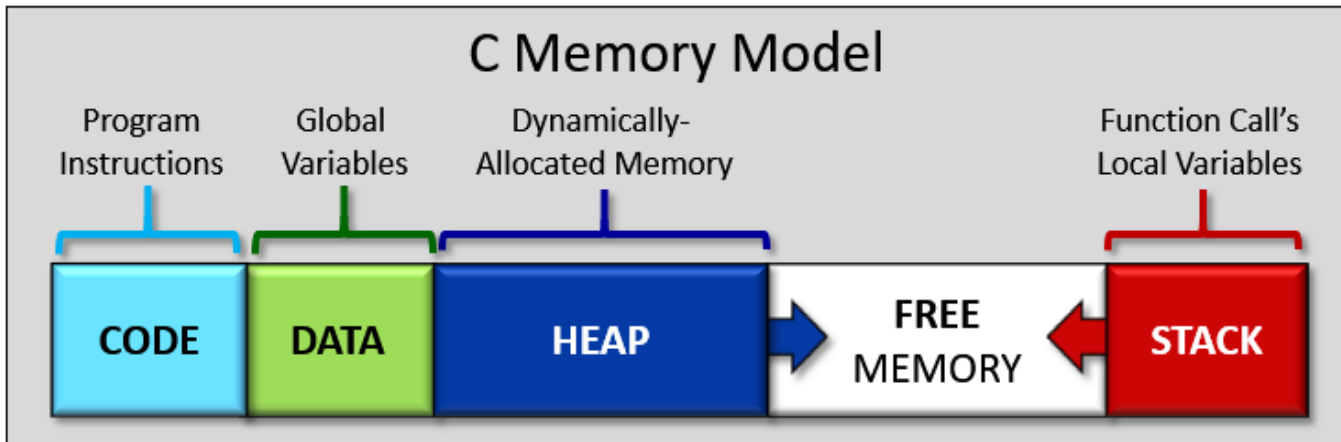
Then we use these objects in our program and when we are finished using them, we don't really do anything, we just leave them as they are. Eventually, a "garbage collector" process comes along and cleans things up by releasing (i.e., recycling) the memory that is being taken up by these objects that are no longer being used. All is hidden "behind the scenes" to make our life easier as programmers, allowing us to concentrate on the higher-level logic of our code without having to worry about these tedious aspects of memory management.



Here is the JAVA memory model:



In this model, the HEAP memory grows and shrinks as objects are created and destroyed, respectively. The STACK memory grows when a method is called and its parameters/variables are declared, and then it shrinks when the method is done. As you will see, there are some similarities in C. The C memory model has 4 main segments:



Notice a couple of similarities to the JAVA model. Both have **HEAP** and **STACK** space as well as a *static* area. In C, the *static* area is separated in two: the **CODE** and **DATA** segments. The **CODE** and **DATA** segments do not change in size as the program runs, but the **HEAP** and **STACK** segments grow and shrink. Here is what each segment of the memory stores:

- **CODE** segment
 - program instructions
 - addresses of functions
 - sometimes string literals
- **DATA** segment
 - global variables
 - static variables and constants
 - literals (e.g., fixed strings)
- **STACK** segment
 - manages order of function calls
 - local variables
- **HEAP** segment
 - manages dynamically-allocated data

Fun Fact: The **segmentation fault** error that we sometimes get, means that we are accessing memory locations outside of our allowed *segment* in memory.

First, consider the **CODE** segment, which is also known as the **Text** segment. This contains all your program's executable instructions ... all the instructions that are sent to the CPU to do things and make stuff happen. Interestingly, this **does not include any constant values, variables or allocated memory**. It is just the instructions produced after you compile and link your code to get your executable. It is machine-dependent code. It represents a static area in memory that will not need to change ... in fact ... it is often read-only so that the program does not overwrite this area of memory. This data is not meant to be displayed as text, as you can see here →

[illegible]

The **DATA** segment is also static/unchanging in that it is determined at compile time. It is actually broken into two chunks ... uninitialized and initialized data. The compiler will go through your program to identify any **static/global variables** and **constants** as well as **string literals** and will then allocate enough space for the **DATA** segment to store all that information. Keep in mind, however, that it does NOT store any local variables ... only static ones whose value will not change. Local variables are stored in the **STACK** segment.

One way to get a bit of a feel for this is to use the **size** command in Linux. This will give you an idea as to how the static memory has been allocated in your executable program.

For example, consider this simple “empty” program, stored in a file called **memory.c**:

```
int main() {
}
```

Assuming that we compiled the program to produce the executable called **memory**, we can use the **size** command to see how the static memory is allocated:

```
student@COMPBase:~$ size memory
text      data      bss      dec      hex      filename
1228      544         8      1780     6f4      memory
student@COMPBase:~$
```

This tells us that the **CODE** (i.e., text) segment takes up **1228** bytes (that is a lot of overhead for an empty program isn't it?). The **DATA** segment is comprised of the **data** and **bss** (from the words “block started by symbol”) portions ... corresponding to initialized and uninitialized static data, respectively. So, there are $544 + 8 = 552$ bytes of static/global variables/constants for a blank program. In total, this blank **memory.c** program takes up **1780** bytes.

What if we add a variable to the main function? How will the **CODE** & **DATA** space change?

```
int main() {
    int x;
}
```

It won't ! Why not? Well, we have not really added any instructions to the program ... we just created a variable. Moreover, this variable is not **static** ... it is a regular variable, so it will be allocated and stored in the **STACK** space. What if we put the variable outside the **main()** function ... making it a global variable?

```
int x;
int main() {
}
```

There will be no change! It seems that there must be some padding going on. If we add a second variable, we get a change:

```
int x, y;
int main() {
}
```



```
student@COMPBase:~$ size memory
text      data      bss      dec      hex      filename
1228      544        16     1788     6fc      memory
student@COMPBase:~$
```

With the two variables, the **bss** jumps to **16**. It must be padding to multiples of 8 byte-chunks. We can verify this by trying: `int x,y,z;` which still stays at **16** bytes, but `int w,x,y,z;` jumps to **24** bytes. If we were to give values to **x** and **y**, it should allocate those extra bytes under **data** instead of **bss**:

```
int x = 890, y = 75;
int main() {
}
```

```
student@COMPBase:~$ size memory
text      data      bss      dec      hex      filename
1228      552         8     1788     6fc      memory
student@COMPBase:~$
```

Consider adding another initialized variable and one more uninitialized one:

```
int x = 890, y = 75;
char *c;

int main() {
    static float y = 200.67;
}
```

```
student@COMPBase:~$ size memory
text      data      bss      dec      hex      filename
1228      556        16     1800     708      memory
student@COMPBase:~$
```

Finally, we will change ***c** to be a string literal:

```
int x = 890, y = 75;
char *c = "HELLO";

int main() {
    static float y = 200.67;
}
```

```
student@COMPBase:~$ size memory
text      data      bss      dec      hex      filename
1258      568         8     1834     72a      memory
student@COMPBase:~$
```

Notice that ***c** is now initialized, so **8** bytes less in **bss** and **12** bytes more in **data** (i.e., **8** from the pointer moving over plus 4 bytes padding). Also, take note that the **CODE** segment (i.e., text) now increased by **30** ... apparently **24** bytes of overhead plus **1** byte for each of the **6** characters (including **null** terminator) that make up the string literal. Some compilers will store string literals in the **DATA** segment.

Hopefully, you have a rough idea now as to what is stored in these static areas at compile time. We will now look at the **STACK** and **HEAP** segments which will grow and shrink over time as the program is running.

The **STACK** segment is also called the **Function Call Stack**. You may not have thought about the lower-level details before, but when multiple functions are called in sequence, the program needs to remember the order in which the functions are called as well as the location in the program to return to when the function call returns. In addition, each time a function with parameters is called, the program needs to store those parameters for use in the function as well as any local variables declared in that function. It then needs to release them afterwards since they won't be needed anymore once the function completes.

Consider the following code. We will examine exactly what happens with the **STACK** when each function is called. In the code, the **main()** function calls a **stat()** function, which calls the **avg()** function which calls the **add()** function. The program, therefore, has nested function calls. Each time a function is called, you will notice that the *parameters*, *return address* and *local variables* are all added to the **STACK** memory. These are all shown as single items, but you should keep in mind that they represent int & float types as well as memory addresses ...each taking up 4 to 8 bytes of memory. Try to follow along with the explanation given.



Code from **stackExample.c**

```
#include <stdio.h>

1.  int add(int n1, int n2) {
2.      int sum = n1 + n2;
3.      return sum;
4.  }

5.  float avg(int m1, int m2, int m3) {
6.      int ttl = add(m1, m2) + m3;
7.      return ttl/3.0;
8.  }

9.  void stat(int i1, int i2, int i3) {
10.     float r = avg(i1, i2, i3);
11.     printf("%0.2f\n", r);
12. }

13. int main() {
14.     int t = 30;
15.     stat(t, 25, 55);
16. }
```

When the program begins with the **main** function, the local variable **t** is placed onto the **STACK**, using up 4 bytes of memory. Then, the program continues until the **stat()** function is called. At this point, the **STACK** memory increases...

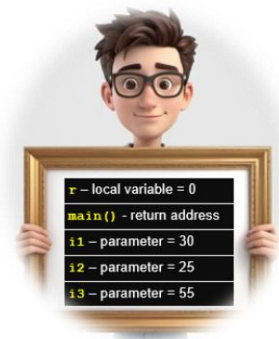
t – local variable = 30

When the **stat()** function is called, the parameters to the function (i.e., **i1**, **i2**, **i3**) are pushed onto the **STACK** in reverse order. Then the **return address** of the calling function (i.e., the main function) is placed onto the **STACK**. The program lies in the **CODE** segment of memory. This **return address** represents the next instruction that will be executed upon return of the **stat()** function. In order to keep things simple, we will assume that the program returns to line **16** of the code. Lastly, all local variables of the **stat()** function (i.e., just **r** in this case) are pushed onto the **STACK**.

All items just pushed onto the **STACK** are implicitly grouped into what is called a **Stack Frame**. The Stack Frame contains all “dynamic” data required for the function (i.e., excludes global variables). When the function returns, the Stack Frame is removed from the **STACK** and discarded.

Now ... the program continues until the **avg()** function is called. At this point, the **STACK** memory increases again.

r – local variable = 0
return address (line 15)
i1 – parameter = 30
i2 – parameter = 25
i3 – parameter = 55
t – local variable = 30



In a similar manner, when the **avg()** function is called, the parameters **m1**, **m2** and **m3** are pushed onto the **STACK** and the **return address** of the **stat()** function is placed onto the **STACK**. This would correspond to the instruction (at around line **10** of the code) that would assign **r** to the value returned from the function. Lastly, local variable **ttl** is pushed onto the **STACK**.

Now ... the program continues until the **add()** function is called. At this point, the **STACK** memory increases again.

ttl – local variable = 0
return address (line 10)
m1 – parameter = 30
m2 – parameter = 25
m3 – parameter = 55
r – local variable = 0
return address (line 15)
i1 – parameter = 30
i2 – parameter = 25
i3 – parameter = 55
t – local variable = 30

When the **avg()** function is called, the parameters **n1** and **n2** are pushed onto the **STACK** followed by the **return address** of **avg()** (at around line **6** of the code where **m3** is added to the return value from **add()**). Lastly, local variable **sum** is pushed onto the **STACK**.

Finally, the program continues until the **add()** function has completed. At this point, the program will be returning from the function.

How many Stack Frames are there? Well, we made 3 successive function calls and we started with the **main()** function... so there are **4** Stack Frames. The memory being used is **(12 x 4)** bytes for variables + **(3 x 8)** bytes for return addresses = **72** bytes.

Do you now understand why we sometimes get a **Stack Overflow Error** when we write recursive functions? If we write a function that keeps calling itself (perhaps taking up **16** to **24** bytes each time), then you can easily see that the **STACK** will just keep growing, taking up more and more space until there is no space left.



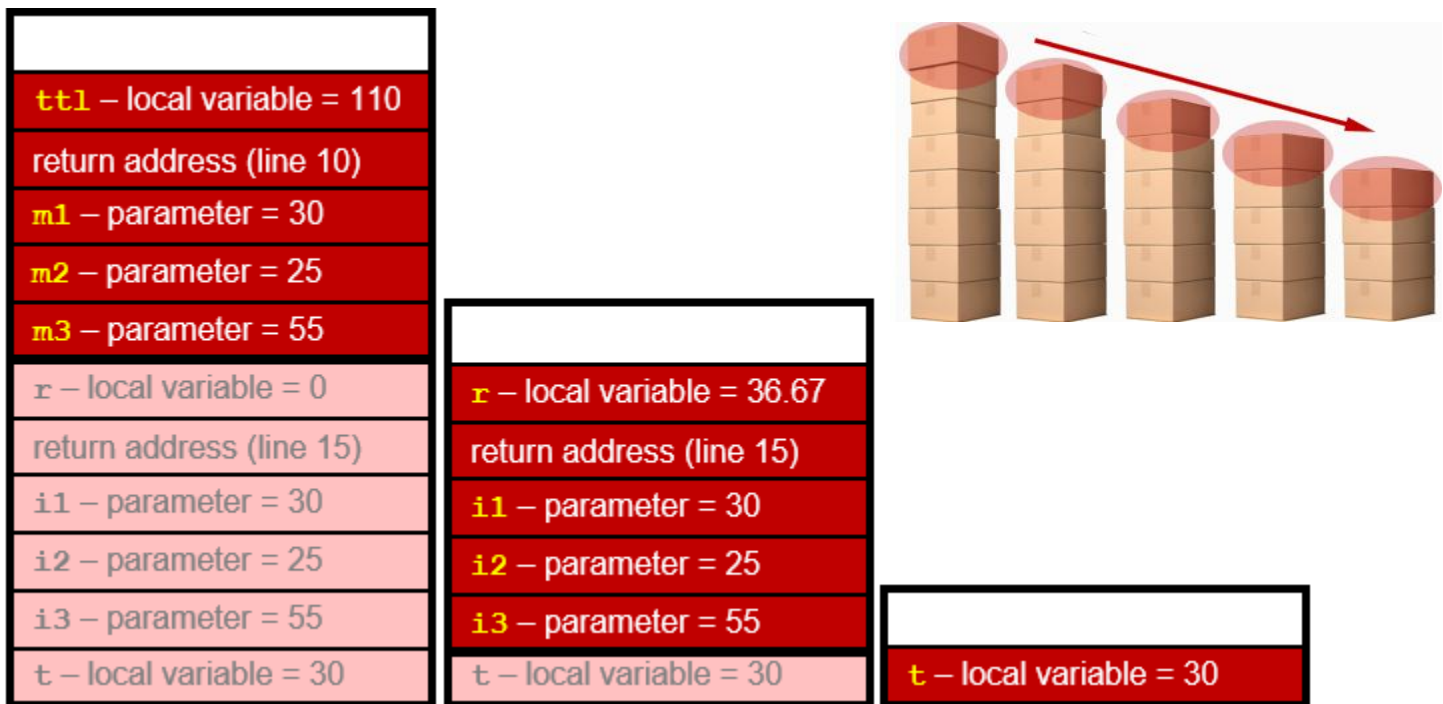
sum – local variable = 0
return address (line 6)
n1 – parameter = 30
n2 – parameter = 25
ttl – local variable = 0
return address (line 10)
m1 – parameter = 30
m2 – parameter = 25
m3 – parameter = 55
r – local variable = 0
return address (line 15)
i1 – parameter = 30
i2 – parameter = 25
i3 – parameter = 55
t – local variable = 30

Now what happens as the functions start to return?

A single stack frame is removed as follows:

- All local variables are removed from the **STACK** and discarded.
- The program returns control to the memory address corresponding to the return address which is popped off the **STACK**.
- All parameters are removed from the **STACK**.

So, you can see that when the **add()** function returns, the **STACK** shrinks back to the way it was before the **add()** function was called (left picture on next page). Then the result is added to **m3** and that **ttl** is divided by **3** and this value is returned from the **avg()** function. Then the Stack Frame corresponding to the **avg()** function is removed from the **STACK** in the same way. As a result, we end up with the shrunken **STACK** shown in the middle picture on the next page. Finally, the returned average is stored in variable **r** and then printed by the **stat()** function, at which point the Stack Frame for the **stat()** function is removed from the **STACK** and control returns to the **main()** function (see rightmost picture on next page). Once the program ends with the completion of the **main()** function, the **STACK** is empty.



What about the **HEAP** space?

The **HEAP** segment is used whenever we want to store chunks of memory that we reserve on our own. To use this space, we need to understand how to allocate and de-allocate memory.

Similar to the **STACK** segment, when we allocate memory the **HEAP** space grows and the **FREE** space shrinks. One difference is that the **STACK** space is deallocated in the *reverse* order in which it was allocated (i.e., order of function calls). Whereas, with the **HEAP** space, we can allocate and de-allocate memory chunks at any time, which may be in any unspecified/random order.

***Allocating** means reserving (i.e., using up) a sequential chunk of memory.*

***De-allocating** means releasing (i.e., freeing up) an allocated chunk of memory.*

Dynamic memory allocation and de-allocation implies that it all happens while our program is running (i.e., at run time). This is different from the memory allocation that the compiler did in our **DATA** and **CODE** segments (i.e., at compile time).

Since the compiler will automatically allocate memory for us at compile time to store our variables ... why would we want to allocate memory on our own?

The main advantage is that we can make more efficient use of the computer's memory. For example, suppose that you want a program to store bank accounts. You can allocate an array to do this, but you need to know the maximum size for that array. If you choose too small of a number (e.g., 500) then you cannot store more accounts past that number. If you choose a big number (e.g., 10,000,000), then you may not need all that space and may be wasting memory by reserving it. Dynamic memory allocation allows you to reserve the exact space that you need without waste. The next section goes into much more detail about this.

3.4 Dynamic Memory Allocation

Allocating and de-allocating memory is as simple as calling predefined C functions. It is not hard. However, it can become difficult to *manage* all the allocated memory. That is, you **MUST ALWAYS** carefully keep track of the memory that you allocated so that you use it properly and so that you free it properly.

All too often, when C programming, programs will crash because the programmer did not properly keep track of allocated memory. It is important that you keep organized while programming and that you have some fixed ways of remembering what has been allocated and when it should be freed.

If memory is continually allocated and never freed ... the program will eventually run out of memory and crash. Sometimes you may think that you have freed up all the memory that you allocated but there may be some lingering chunks of memory that never get freed. These are known as **memory leaks**:

*A **Memory Leak** is a chunk of allocated memory that is never freed.*

It is called a “leak” because your program may slowly lose available memory. This would be like a leaking tire on your car that is losing air slowly ... eventually going flat. Unfortunately, memory leaks often happen at inconvenient times (like when you are under pressure to meet a software deadline at work).



The most annoying thing about memory leaks is that they can be difficult to locate in your program. It can be a difficult task to sift through thousands of lines of C code looking for a memory leak. So, do your best to stay organized and write your code neatly, to minimize the likelihood of getting leaks.

Let's get started with a simple function. To allocate a chunk of memory in the **HEAP** space we use the **malloc()** function. The **malloc()** function takes a single parameter that indicates the number of bytes that you want to reserve for yourself. It returns a pointer to the memory location representing the start of the reserved chunk of bytes in memory.

This is similar to the idea in JAVA when we call a constructor by using **new**. When we say **new Person()** for example, we get back the pointer (or reference) to the object in memory ... which is really just the starting address of a sequence of bytes that store the object's attribute values.

Assume that we want to store some integers. We already know that we can do this:

```
int grades[500];
```

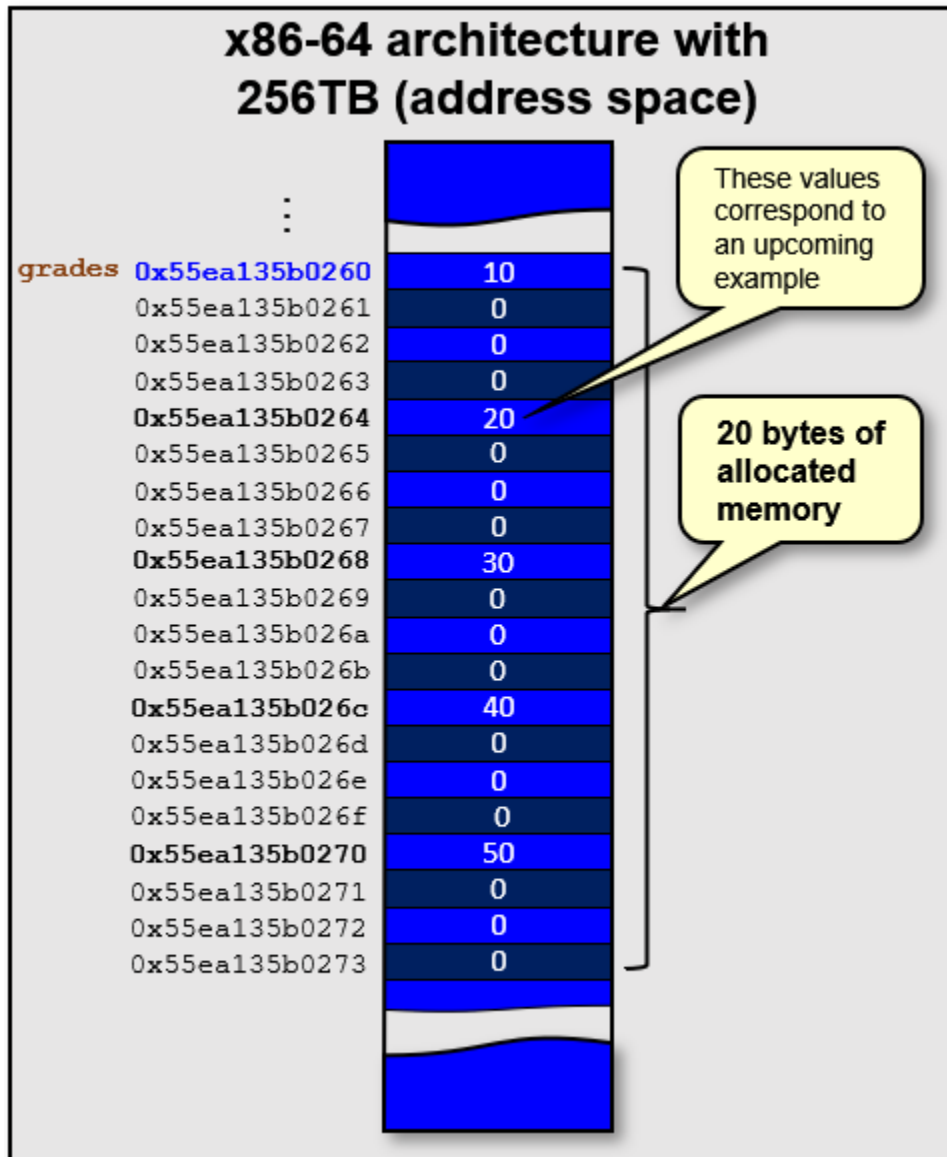
This will allow us to store up to **500** integers. But remember the advantage of dynamic allocation ... we may only want **5** integers ... or maybe even up to **50,000** integers. An array size of **500** can be either wasteful or not enough.

Using dynamic memory allocation, we can do this instead:

```
int    numGrades = 5;
int    *grades; // This will point to some allocated memory

grades = (int *) malloc(numGrades * sizeof(int));
```

This code allocates $5 * 4 = 20$ bytes that will allow us to store 5 integers.



In order to use the **malloc()** function, we'll need to include the **<stdlib.h>** header file.

You will notice that we typecasted the result of **malloc** to **int ***. The **malloc()** function returns a type of **void ***. Although a type-cast is not required, it is proper programming style to typecast the result of **malloc()** to the type of the variable that you are storing it in. This allows for more robust error-checking by the compiler.

In addition, it is possible that the memory cannot be allocated (i.e., if the system is out of memory). In such a situation, a value of **NULL** (i.e., a **NULL** pointer) will be returned. If you tried to use the pointer, you would then get an error and the program would crash. Therefore, it is proper to check for **NULL** each time that you call **malloc()** with some kind of error message and perhaps exiting the program as follows:

```
if (grades == NULL) {  
    printf("Memory allocation error\n");  
    exit(-1);  
}
```

Once this memory is allocated, we can do what we want with it. For example, we can treat it as an array of **5** integers and we can iterate through the **grades** data using indices. Or we can just treat the returned reference address as a pointer and work with it that way.

Here is a sample program that does this:

Code from **malloc.c**

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main() {  
    int    numGrades = 5;  
    int    *grades;  
  
    grades = (int *) malloc(numGrades * sizeof(int));  
    if (grades == NULL) {  
        printf("Memory allocation error\n");  
        exit(-1);  
    }  
  
    grades[0]    = 10;  
    grades[1]    = 20;  
    grades[2]    = 30;  
    *(grades+3) = 40;  
    *(grades+4) = 50;  
  
    for (int i=0; i<numGrades; i++)  
        printf("%d ", grades[i]); // use it like an array  
    printf("\n");  
  
    for (int i=0; i<numGrades; i++)  
        printf("%d ", *(grades++)); // use it via pointers  
    printf("\n");  
}
```

The output is the same for both loops:

```
10 20 30 40 50  
10 20 30 40 50
```

In the above code, **grades** is a pointer to the allocated memory. You must be **VERY careful in your code not to lose this pointer!** If you lose it, or erase it somehow, then you will NEVER be able to free up the reserved memory.



In our code above, we are never freeing up the memory. It will not matter in this case, however, because the program is small and we know that it won't run out of memory. But this is poor coding style. You should always free up allocated memory that you will no longer be using.

To free allocated memory in C, you use the **free(*ptr)** function. This function takes the pointer (e.g., **grades**) that you got back from the **malloc()** function call. It has no return value.

What would happen if we put **free(grades);** as the last line of code in the above program?

```
...
for (int i=0; i<numGrades; i++)
    printf("%d ", *(grades++));
printf("\n");

free(grades);
}
```



You might think that all is ok ... but the program will crash! Why? Well, within the loop we are increasing the **grades** pointer by 4 (i.e., size of **int**) each time by using **++**. Hence, we are actually losing the original pointer location!

A solution to this would be NOT to alter the **grades** pointer at any time, or so store a pointer to the original start location:

```
...
int *grades, *gradesStart;

grades = gradesStart = (int *) malloc(numGrades * sizeof(int));

...
for (int i=0; i<numGrades; i++)
    printf("%d ", *(grades++));
printf("\n");

free(gradesStart);
}
```

As you might start to see ... it is easier than you think to lose track of pointers. Sometimes another part of your code can *clobber* (i.e., overwrite) other parts of the code, including pointers. From my personal experience, this often happened when dealing with **char *** types. It can also happen that you allocate a pointer within a function and store it in a local variable but upon returning from the function you no longer have access to that variable.



Remember ... once a pointer has been lost ... it is lost forever. If this happens too often, your program will run out of **HEAP** space.

Sometimes a memory leak will occur and can be hard to find. There is a Linux tool called **valgrind** which you can use to check for a memory leak. You use it on your compiled executable file. It will tell you whether or not you have memory leaks.

For example, consider these two simplified programs:

Code from leakTest1.c

```
#include <stdlib.h>

int main() {
    int    numGrades = 5;
    int    *grades;

    grades = (int *) malloc(numGrades * sizeof(int));
    free(grades);
}
```

Code from leakTest2.c

```
#include <stdlib.h>

int main() {
    int    numGrades = 5;
    int    *grades;

    grades = (int *) malloc(numGrades * sizeof(int));
}
```

One has the memory allocated and freed ... the other allocates without freeing. Assume that both programs have been compiled. We can then run **valgrind** on them as shown here. Notice the difference in output as highlighted:

```
student@COMPBase:~$ valgrind --leak-check=yes ./leakTest1
==3088== Memcheck, a memory error detector
==3088== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3088== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3088== Command: ./leakTest1
==3088==
==3088==
==3088== HEAP SUMMARY:
==3088==   in use at exit: 0 bytes in 0 blocks
==3088==   total heap usage: 1 allocs, 1 frees, 20 bytes allocated
==3088==
==3088== All heap blocks were freed -- no leaks are possible
==3088==
==3088== For lists of detected and suppressed errors, rerun with: -s
==3088== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
student@COMPBase:~$
```

```
student@COMPBase:~$ valgrind --leak-check=yes ./leakTest2
==3109== Memcheck, a memory error detector
==3109== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3109== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3109== Command: ./leakTest2
==3109==
==3109==
==3109== HEAP SUMMARY:
==3109==   in use at exit: 20 bytes in 1 blocks
==3109==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
==3109==
==3109== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3109==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-
```



```

linux.so)
==3109== by 0x10916C: main (in /home/student/code/ch3/leakTest2)
==3109==
==3109== LEAK SUMMARY:
==3109== definitely lost: 20 bytes in 1 blocks
==3109== indirectly lost: 0 bytes in 0 blocks
==3109== possibly lost: 0 bytes in 0 blocks
==3109== still reachable: 0 bytes in 0 blocks
==3109== suppressed: 0 bytes in 0 blocks
==3109==
==3109== For lists of detected and suppressed errors, rerun with: -s
==3109== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from
0) 0) student@COMPBase:~$

```

So ... you can see that if we forget to free some allocated memory, then it can be detected. It even mentions the function (in this case **main**) that the unfreed **malloc** was made within.

valgrind can also detect when you are reading or writing to invalid locations in memory. This can be very useful when debugging. Here is an example:

Code from **badReadWrite.c**

```

#include <stdio.h>
#include <stdlib.h>

int main() {
    int numGrades = 5;
    int *grades;

    grades = (int *) malloc(numGrades * sizeof(int));

    grades[0] = 10;
    grades[1] = 20;
    grades[2] = 30;
    grades[67] = 4544;           // this is an invalid write

    printf("%d\n", grades[99]); // this is an invalid read

    free(grades);
}

```

As you can see, we are attempting to read from an unallocated memory location as well as write to an unallocated memory location. The program actually runs without any noticeable error ! But here is a **valgrind** test on the program:

```

student@COMPBase:~$ valgrind --leak-check=yes ./badReadWrite
==3531== Memcheck, a memory error detector
==3531== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3531== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3531== Command: ./badReadWrite
==3531==
==3531== Invalid write of size 4
==3531== at 0x1091E1: main (in /home/student/code/ch3/badReadWrite)
==3531== Address 0x4a9114c is 172 bytes inside an unallocated block of size 4,194,112
in arena "client"
==3531==

```



```

==3531== Invalid read of size 4
==3531==    at 0x1091F1: main (in /home/student/code/ch3/badReadWrite)
==3531== Address 0x4a911cc is 300 bytes inside an unallocated block of size 4,194,112
in arena "client"
==3531==
0
==3531==
==3531== HEAP SUMMARY:
==3531==    in use at exit: 0 bytes in 0 blocks
==3531==    total heap usage: 2 allocs, 2 frees, 1,044 bytes allocated
==3531==
==3531== All heap blocks were freed -- no leaks are possible
==3531==
==3531== For lists of detected and suppressed errors, rerun with: -s
==3531== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
student@COMPBase:~$

```

As you can see, **valgrind** can find memory read/write errors that we may not even be aware of when we run our programs. You should make good use of **valgrind** to ensure that your code is running cleanly and properly with respect to memory allocation, memory access and memory modification.

Sometimes, memory problems occur because we are misusing pointers. That is, sometimes we think that we are using pointers a certain way, but we get confused and end up writing code that does not work the way that we expected.



Consider this function that creates (and returns a pointer to) a random integer array with the specified **amount** of items in it:

```

int *getRandomArray(int amount) {
    int *memoryPointer = (int *) malloc(amount * sizeof(int));
    if (memoryPointer == NULL) {
        printf("Memory allocation error\n");
        exit(-1);
    }
    for (int i=0; i<amount; i++)
        memoryPointer[i] = rand();
    return memoryPointer;
}

```

The code works. We can test it as follows:

```

int main() {
    int *nums;

    nums = getRandomArray(5);

    for (int i=0; i<5; i++)
        printf("%d ", nums[i]);
    printf("\n");

    free(nums);
}

```

The code will produce what was expected ... 5 random integers:

1804289383 846930886 1681692777 1714636915 1957747793

Now consider altering the function to allocate two integer arrays. We would need to return two arrays from the function, so we will need to use parameters instead of the return value:

```
void getRandomArrays(int *a1, int *a2, int amount) {
    a1 = (int *) malloc(amount * sizeof(int));
    a2 = (int *) malloc(amount * sizeof(int));
    if ((a1 == NULL) || (a2 == NULL)) {
        printf("Memory allocation error\n");
        exit(-1);
    }
    for (int i=0; i<amount; i++) {
        a1[i] = rand();
        a2[i] = rand();
    }
}
```

The code creates the two arrays properly and fills them in with random values. How do we call this function now? Well, here is how we will try to do it:

```
int main() {
    int *array1 = NULL, *array2 = NULL;
    getRandomArrays(array1, array2, 5);
    free(array1);
    free(array2);
}
```

However, if we were to run this program, we would get an error when we try to free the arrays. Why? Well, we should examine the code carefully. We are defining two arrays whose values are uninitialized at first. Then we call the function, which should “hopefully” set the array pointers properly so that we can use them. But we can insert some print statements to check and see if these pointers are being set properly:

```
int main() {
    int *array1 = NULL, *array2 = NULL;

    printf("array1 = %p\n", (void *)array1);
    printf("array2 = %p\n", (void *)array2);

    getRandomArrays(array1, array2, 5);

    printf("array1 = %p\n", (void *)array1);
    printf("array2 = %p\n", (void *)array2);

    free(array1);
    free(array2);
}
```

If you were to run the above code, you would notice that the output would be as follows:

```
array1 = (nil)
array2 = (nil)
array1 = (nil)
array2 = (nil)
```

Clearly, the pointers are not being set. Why not? Well, when we call the function, we are actually passing in the *value* of the pointer ... which is a **NULL** address ... which is **0**. In the function, we take in these **NULL** pointers as parameters and then assign the result from the **malloc** calls to the *parameter*. Since we are only altering the *parameter*, we never alter the pointers out in the main function. This is a very common problem in C programming that we must be aware of. We are essentially passing-by-value instead of what we need to do ... pass-by-reference. So, we need to pass in the memory address of the pointers that we want to alter. Here is the changed code:



```
int main() {
    int *array1= NULL, *array2= NULL;

    printf("array1 = %p\n", (void *)array1);
    printf("array2 = %p\n", (void *)array2);

    getRandomArrays(&array1, &array2, 5);

    printf("array1 = %p\n", (void *)array1);
    printf("array2 = %p\n", (void *)array2);

    free(array1);
    free(array2);
}
```

Now, we will need to alter the function so that it knows that it is getting an address to a pointer each time, not the pointer itself:

```
void getRandomArrays(int **a1, int **a2, int amount) {
    *a1 = (int *) malloc(amount * sizeof(int));
    *a2 = (int *) malloc(amount * sizeof(int));
    if ((*a1 == NULL) || (*a2 == NULL)) {
        printf("Memory allocation error\n");
        exit(0);
    }
    for (int i=0; i<amount; i++) {
        (*a1)[i] = rand();
        (*a2)[i] = rand();
    }
}
```



Notice the use of the double ****** characters. This is called a **double pointer**. They are used often in C programming. They are essentially pointers to pointers.

Notice that we use ***a1** and ***a2** to the left of the assignment operators. That means, we are dereferencing the double pointer to get the actual pointer that was passed in. Then we can assign values to those pointers. Also, to use the pointers in the function, we must first dereference the double pointer to get the single pointer by using **(*a1)** and **(*a2)**.

Finally, the output of the main function will show proper addresses:

```
array1 = (nil)
array2 = (nil)
array1 = 0x5608f9213670
array2 = 0x5608f9213690
```

Here is the completed, working code:



Code from **doublePointer.c**

```
#include <stdio.h>
#include <stdlib.h>

void getRandomArrays(int **a1, int **a2, int amount) {
    *a1 = (int *) malloc(amount * sizeof(int));
    *a2 = (int *) malloc(amount * sizeof(int));
    if ((*a1 == NULL) || (*a2 == NULL)) {
        printf("Memory allocation error\n");
        exit(0);
    }
    for (int i=0; i<amount; i++) {
        (*a1)[i] = rand();
        (*a2)[i] = rand();
    }
}

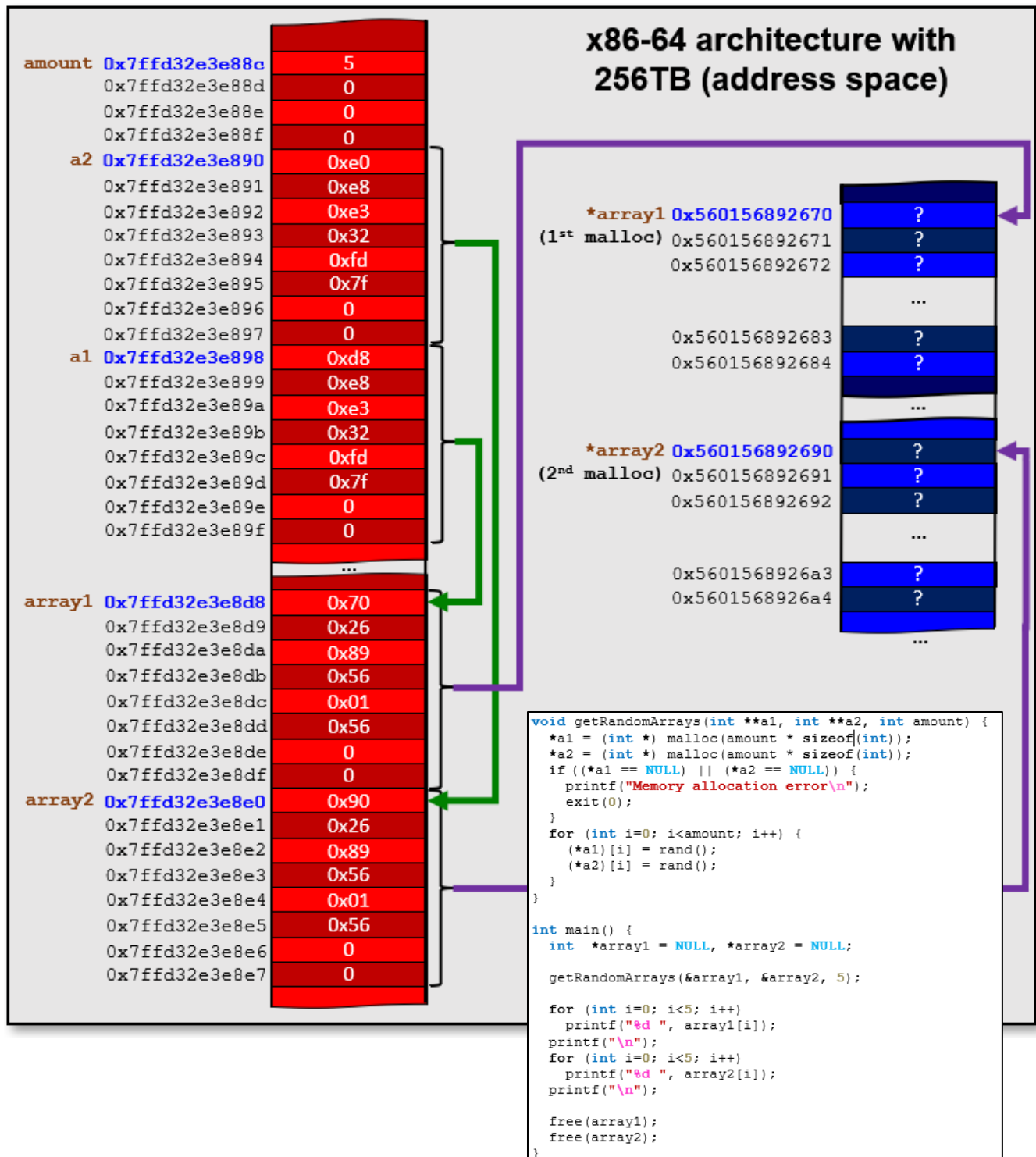
int main() {
    int *array1 = NULL, *array2 = NULL;

    getRandomArrays(&array1, &array2, 5);

    for (int i=0; i<5; i++)
        printf("%d ", array1[i]);
    printf("\n");
    for (int i=0; i<5; i++)
        printf("%d ", array2[i]);
    printf("\n");

    free(array1);
    free(array2);
}
```

Here is the memory map showing how the pointers are stored. Note that the parameters **a1**, **a2** and **amount** are also shown, which are valid only during the function call:



Why use double pointers? So that we can change the value of a parameter:

```

void function1(int p1) {
    p1 = 14;
}

void function2(int *p2) {
    *p2 = 14;
}

void function3(int **p3) {
    int *tmp = (int *) malloc(sizeof(int));

    *tmp = 86;

    *p3 = tmp;
}

void main() {
    int x = 37;
    int *y = &x;

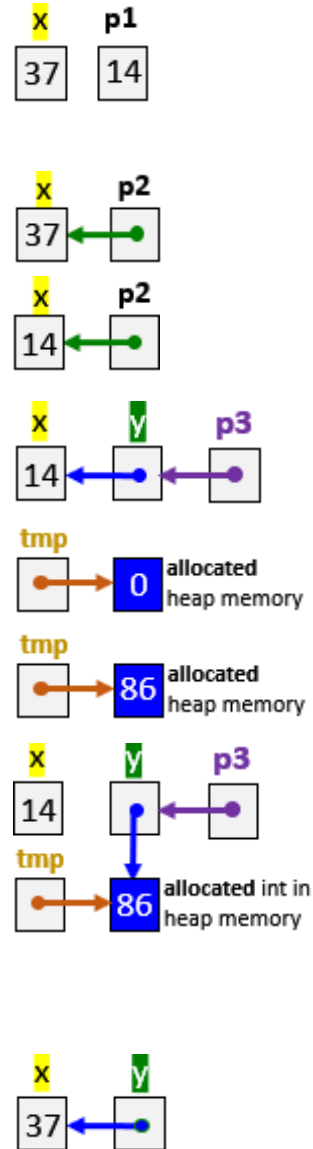
    function1(x);
    printf("%d\n", x); // prints 37

    function2(&x);
    printf("%d\n", x); // prints 14 now
    printf("%d\n", *y); // prints 14 as well

    function3(&y);
    printf("%d, %d\n", x, *y); // prints 14, 86

    free(y); // frees the allocated memory
}

```



We use a double pointer so that `y` can be changed from within the function.

When you use **malloc()**, you should remember that it will allocate memory which is not initialized. That is, there could be garbage data in the memory locations. This is not usually a problem since the programmer knows that the memory has not been filled in with valid data when it is first obtained. Normally the programmer will keep track of what data is valid. For example, when we allocate big arrays (e.g., size **10,000**) and then put a couple of hundred items into the array ... we also keep track of how many items we put in there so that we do not end up accessing invalid/garbage data.

If you want to ensure that the data is initialized (i.e., not garbage but zeroed), then there is a **calloc()** function that you can use. **calloc()** will allocate memory and also clear all the bytes to zero. It is used similarly to **malloc()** except that we don't need to multiply the size of the type by the number of elements we want, we keep the two parameters separate. Here is the difference:

```
pointer = malloc(numberOfArrayItems * sizeof(int));  
pointer = calloc(numberOfArrayItems, sizeof(int));
```

The advantage of using **calloc()** is that you are sure that there is no garbage data ... it will all be zeroed (which is easily identifiable as being uninitialized). The downside of using **calloc()** is that it is slower than **malloc()** since it must go through all the bytes and fill them with zero. It is up to the programmer as to whether or not it is worth initializing, at the expense of slower code.

There is one more memory allocation function to mention ... **realloc()**. The **realloc()** function is used to reallocate memory in a situation in which we want to "grow" an array, for example. Here is how to use it to grow a memory chunk that was used to store a string:

Code from **realloc.c**

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
int main() {  
    char *string;  
  
    string = (char *) malloc(10);  
    strcpy(string, "Small");  
    printf("Initial String = \"%s\" stored at address = %p\n",  
        string, (void *) string);  
  
    string = (char *) realloc(string, 40);  
    strcat(string, ", but now the string is bigger.");  
    printf("Bigger String = \"%s\" stored at address = %p\n",  
        string, (void *) string);  
  
    free(string);  
}
```

Here is the output:

```
Initial String = "Small" stored at = 0x560b7a4ca260  
Bigger String = "Small, but now the string is bigger." stored at = 0x560b7a4ca690
```

Notice a couple of things. First, the original pointer (i.e., **string** variable) is passed in as a parameter to **realloc()**. This must either be a valid memory location that was obtained from **malloc()**, **calloc()** or **realloc()** previously ... or **NULL**. If it is **NULL**, then the function behaves just like a regular call to **malloc()**.

In the output, you may have noticed that the address of the **string** changes. As it turns out, if the function is able to extend the current block of memory further, it will maintain the same address. However, if it is unable to allocate a bigger contiguous (i.e., all together) block of memory, it will find a different block in memory that is big enough and return a pointer to that location. Regardless, you will notice that the data values that are in the original memory block are copied over to the new block. You can see this in the example, since the “**Small**” part of the string was in the original allocated memory and it also appears in the newly-allocated memory block. You may also reallocate to a smaller memory chunk if you want.

Regardless of whether or not we use **malloc()**, **calloc()** or **realloc()**, it is possible that the function will not be able to allocate memory. If this is the case, the function will return **NULL**. Therefore, you should always check the return value from these memory allocation functions to ensure that the memory has been allocated:

```
string = (char *) realloc(string, 40);
if (string == NULL) {
    printf("It's all over man!   Error (re)allocating Memory!\n");
    exit(-1);
}
```

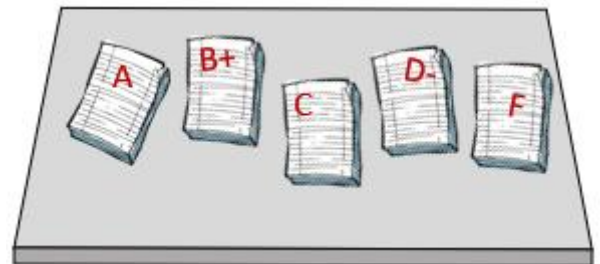
3.5 Linked Lists

As you may now realize, a lot of the time we spend programming has to do with writing data to memory, reading that data from memory later on and processing it ... and then re-writing the new data to memory. Since much of our program's time is spent on this memory reading/writing, we want to write code that allows the fastest possible access to memory and that also uses the least amount of memory.

There is always a tradeoff in computer science when it comes to speed versus memory. This is understandable. Imagine, for example, that you had to organize/sort **500** exam papers by putting them in order of grade from lowest to highest. Imagine having very little physical space to do this (e.g., on your lap).

It would take you a ridiculous amount of time to sort them because you don't have enough space to work on your lap. It would be much easier if you had a large desk on which to work so that you can make partially-sorted piles.

To get the best use of space and speed, it is important to use the right data structure. You have had ample opportunity to work with arrays. You should have also been introduced to linked-lists as well by now. Here are the tradeoffs between the two:



Arrays

Advantage:

- ✓ Faster access since elements are contiguous (one beside another in sequence).

Disadvantage:

- ✗ re-size limitations.
 - If a static array is made too big, you waste memory space. If it is made too small, you run the risk of running out of space to store your items.
 - You can grow or shrink (i.e., realloc) an array to match the amount of data that you currently have but there is a time cost to copy the elements over.

Linked Lists

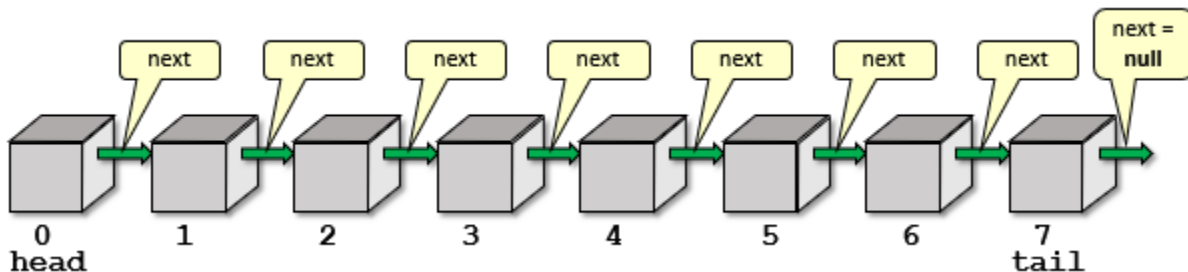
Advantage:

- ✓ no size-limitations.
 - can be resized any time and elements can be inserted, removed anywhere in the list.

Disadvantage:

- ✗ slower access since elements are not contiguous ... we need to follow the pointers.

The most basic linked list is a **singly-linked list**. It has a **head** which points to the first item in the list. Optionally, it may have a **tail**, which points to the last item in the list.



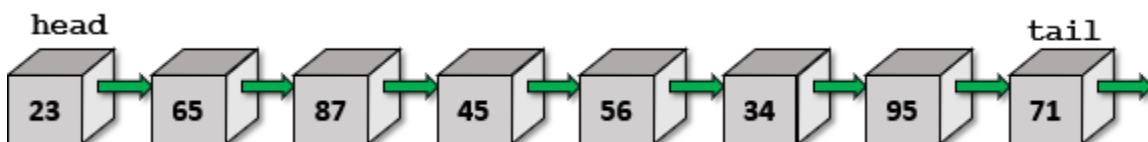
Each element of the list is actually a list in itself. That is, if we grab any item and “shake off” the items before it ... we actually have a sub-list. Here is a **struct** definition:

```

struct LinkedListItem {
    int data;
    struct LinkedListItem *next;
};
  
```

Notice that the **LinkedListItem** is just a piece of data (i.e., an **int**) and then a pointer to the next **LinkedListItem** in the list. It is a self-referencing data structure.

How do we create the following singly-linked list ?



We would need to allocate memory for each item:

```

struct LinkedListItem  *myList, *myList1, *myList2, *myList3,
                        *myList4, *myList5, *myList6, *myList7;

// Set up all the data
myList = (struct LinkedListItem *) malloc(sizeof(struct LinkedListItem));
myList->data = 23;
myList1 = (struct LinkedListItem *) malloc(sizeof(struct LinkedListItem));
myList1->data = 65;
myList2 = (struct LinkedListItem *) malloc(sizeof(struct LinkedListItem));
myList2->data = 87;
myList3 = (struct LinkedListItem *) malloc(sizeof(struct LinkedListItem));
myList3->data = 45;
myList4 = (struct LinkedListItem *) malloc(sizeof(struct LinkedListItem));
myList4->data = 56;
myList5 = (struct LinkedListItem *) malloc(sizeof(struct LinkedListItem));
myList5->data = 34;
myList6 = (struct LinkedListItem *) malloc(sizeof(struct LinkedListItem));
myList6->data = 95;
myList7 = (struct LinkedListItem *) malloc(sizeof(struct LinkedListItem));
myList7->data = 71;

// Connect them all together now
myList->next = myList1;
myList1->next = myList2;
myList2->next = myList3;
myList3->next = myList4;
myList4->next = myList5;
myList5->next = myList6;
myList6->next = myList7;
myList7->next = NULL;

```



This sure seems like a lot of work. Not only that, but we seem to be using a variable for each list item. This approach is not scalable. How is the list supposed to be able to grow without requiring more variables? Well, we are hardly done. Agreeably, the above code is long because we are manually making the list and connecting things together. It makes more sense, however, to write a function to do this.

We can write a function that takes the **tail** of the list and then simply adds an item to grow the list by connecting that **tail** to a new item which we will allocate. Here is a function to do this:

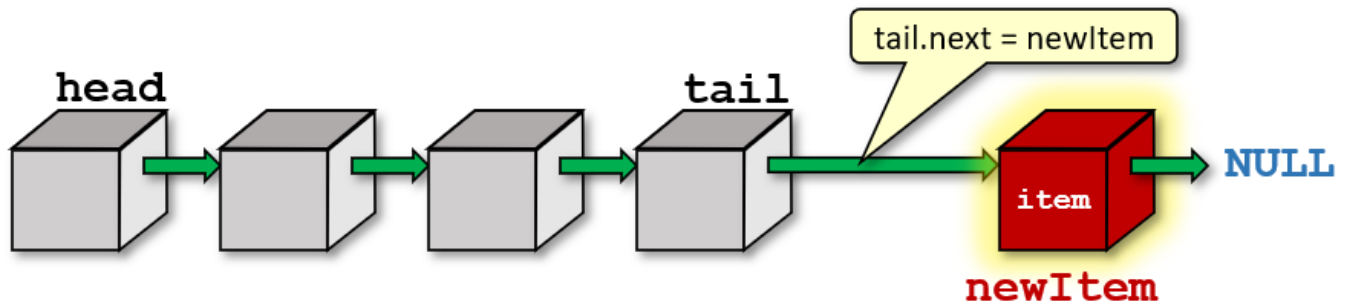
```

struct LinkedListItem *add(struct LinkedListItem *tail, int item) {
    struct LinkedListItem *newItem;
    newItem = (struct LinkedListItem *) malloc(sizeof(struct
                                                    LinkedListItem));

    if (newItem == NULL) {
        printf("Memory allocation error\n");
        exit(-1);
    }
    newItem->data = item;
    newItem->next = NULL;
    tail->next = newItem;
    return newItem;
}

```

As you can see, the function takes in a **LinkedListItem** called **tail** which must be the **tail** of the list, otherwise we will lose any data after this list item. It then creates a **newItem** to add to the list by allocating memory. Finally, it connects it to the **tail** via the **next** pointer. The **newItem** is returned from the function so that we can have access to this item as the list's new tail in order to add onto it the next time.



How will this simplify our list-building code? Look ...

```
struct LinkedListItem *myList;

myList = (struct LinkedListItem *) malloc(sizeof(struct
                                           LinkedListItem));

myList->data = 23;
add(add(add(add(add(add(add(myList, 65), 87), 45), 56), 34), 95), 71));
```



As you can see, since the call to the **add()** function returns a **LinkedListItem** structure which is the new tail of the list, we just use that as the parameter for the next **add()** function call. So, they are all chained together.

We can even add some code in the **add()** function to handle a new (i.e., **NULL**) list so that we don't need to do the **malloc()** outside the function to start things off:

```
struct LinkedListItem *add(struct LinkedListItem *tail, int item) {
    struct LinkedListItem *newItem;
    newItem = (struct LinkedListItem *) malloc(sizeof(struct
                                                  LinkedListItem));

    if (newItem == NULL) {
        printf("Memory allocation error\n");
        exit(-1);
    }
    newItem->data = item;
    newItem->next = NULL;
    if (tail != NULL)
        tail->next = newItem;
    return newItem;
}
```



Then the creation of the list is a bit simpler ... although we have to make sure to hang on to the head of the list, by storing it into the **myList** variable:

```
struct LinkedListItem *myList;
add(add(add(add(add(add(add(myList = add(NULL, 23), 65), 87), 45), 56), 34),
95), 71));
```

How can we display the list? We can write a function that takes the **head** of the list as a parameter, and then repeatedly iterates through the list items one-by-one by following the **next** pointers until **NULL** is reached:

```
void printList(struct LinkedListItem *listItem) {
    while(listItem != NULL) {
        printf("%d", listItem->data);
        if (listItem->next != NULL)
            printf(" ---> ");
        else
            printf("\n");
        listItem = listItem->next;
    }
}
```

Calling the function is as easy as this: `printList(myList);` Here is the output:

```
23 ---> 65 ---> 87 ---> 45 ---> 56 ---> 34 ---> 95 ---> 71
```

There are many ways to play around with the code to allow different ways of creating the list. For example, what if we wanted to create a linked-list from this array:

```
int initData[] = {23, 65, 87, 45, 56, 34, 95, 71};
```

We can make a function called **addAll()** and perhaps pass a pointer to this list as well as the array and the size of the array:

```
void addAll(struct LinkedListItem **initTail, int items[], int size) {
    struct LinkedListItem *newItem;
    struct LinkedListItem *tail = *initTail;

    for (int i=0; i<size; i++) {
        newItem = (struct LinkedListItem *) malloc(sizeof(struct
                                                    LinkedListItem));

        if (newItem == NULL) {
            printf("Memory allocation error\n");
            exit(-1);
        }
        newItem->data = items[i];
        newItem->next = NULL;
        if (tail != NULL)
            tail->next = newItem;
        else
            *initTail = newItem; // newItem becomes the head of the list
        tail = newItem;
    }
}
```

Notice that the **initTail** parameter is actually a “pointer to the **tail**” of the list, not the **tail** itself. This allows us to set it from within the function. In our example, we will create a new list, so the **head** will also be the **tail** ... which will be **NULL** when we call the function. Near the end of the function there is a check to see if the **tail** is **NULL**. If it is, it sets the first created **newItem** to be the first **tail** of the list.

This first **tail** will also be the **head** of the list (i.e., it is a list with one item in it). Therefore, we return this (through the **initTail** parameter) as the **head** of the newly-created list. Here is how we should call the function:

```
struct LinkedListItem *yourList = NULL;

int initData[] = {23, 65, 87, 45, 56, 34, 95, 71};
addAll(&yourList, initData, sizeof(initData)/sizeof(int));
```

Notice that **yourList** is initialized to **NULL** when the variable is declared. This is IMPORTANT! If we do not do this, the **yourList** variable may point to a garbage (i.e., invalid) memory address. In our **addAll()** function, we are explicitly checking for **NULL** as the incoming value. The function is relying on a **NULL** value for new lists. If we did not initialize to **NULL**, we'd likely get a **Segmentation Fault** in our code.



The last thing that we need to do is free the list. Consider the two lists that we made. We could do this:

```
free(myList);
free(yourList);
```

This code will compile and run fine. However, if we were to do a **valgrind** on the code, we would get this result:

```
...
==3329== LEAK SUMMARY:
==3329==      definitely lost: 16 bytes in 2 blocks
==3329==      indirectly lost: 96 bytes in 12 blocks
...
```



There is a memory leak! What's wrong? Didn't we free the two lists?

Think for a moment. Each time we do a **malloc()** call, we reserved a chunk of memory. There should be a **free()** call for each **malloc()** that we did. When creating the two lists ... we did **16 malloc()** calls in total in order to create the **16** list items. But we only did **2** calls to **free()**. Sadly, a common problem in C-programming is forgetting to free the *pieces* of our linked-lists.

To free the memory properly, we would need to iterate through the lists and free the items one-by-one. We will have to write a function:

```
void freeList(struct LinkedListItem *listItem) {
    struct LinkedListItem *nextItem;

    while(listItem != NULL) {
        nextItem = listItem->next;
        free(listItem);
        listItem = nextItem;
    }
}
```

This should free up all items. The completed programming example is shown here:

Code from **singlyLinkedList.c**

```
#include <stdio.h>
#include <stdlib.h>

// Structure that represents a singly-linked-list of integers
struct LinkedListItem {
    int data;
    struct LinkedListItem *next;
};

// Takes a list tail and adds the given item to it, returning the new item added
struct LinkedListItem *add(struct LinkedListItem *tail, int item) {
    struct LinkedListItem *newItem;
    newItem = (struct LinkedListItem *) malloc(sizeof(struct LinkedListItem));
    if (newItem == NULL) {
        printf("Memory allocation error\n");
        exit(-1);
    }
    newItem->data = item;
    newItem->next = NULL;
    if (tail != NULL)
        tail->next = newItem;
    return newItem;
}

// Add all elements from items to the given Singly-Linked List and
// set the list to point to the head of the resulting list
void addAll(struct LinkedListItem **initTail, int items[], int size) {
    struct LinkedListItem *newItem;
    struct LinkedListItem *tail = *initTail;
    for (int i=0; i<size; i++) {
        newItem = (struct LinkedListItem *) malloc(sizeof(struct LinkedListItem));
        if (newItem == NULL) {
            printf("Memory allocation error\n");
            exit(-1);
        }
        newItem->data = items[i];
        newItem->next = NULL;
        if (tail != NULL)
            tail->next = newItem;
        else
            *initTail = newItem; // newItem becomes the head of the list
        tail = newItem;
    }
}

// Print the contents of a Singly-Linked List
void printList(struct LinkedListItem *listItem) {
    while(listItem != NULL) {
        printf("%d", listItem->data);
        if (listItem->next != NULL)
            printf(" ---> ");
        else
            printf("\n");
        listItem = listItem->next;
    }
}
```

```
// Free all items in a Singly-Linked List
void freeList(struct LinkedListItem *listItem) {
    struct LinkedListItem *nextItem;

    while(listItem != NULL) {
        nextItem = listItem->next;
        free(listItem);
        listItem = nextItem;
    }
}

int main() {
    struct LinkedListItem *myList = NULL, *yourList = NULL;

    add(add(add(add(add(add(add(myList = add(NULL, 23), 65), 87), 45), 56), 34), 95), 71));

    int initData[] = {23, 65, 87, 45, 56, 34, 95, 71};
    addAll(&yourList, initData, sizeof(initData)/sizeof(int));

    printList(myList);
    printf("\n");
    printList(yourList);

    freeList(myList);
    freeList(yourList);
}
```

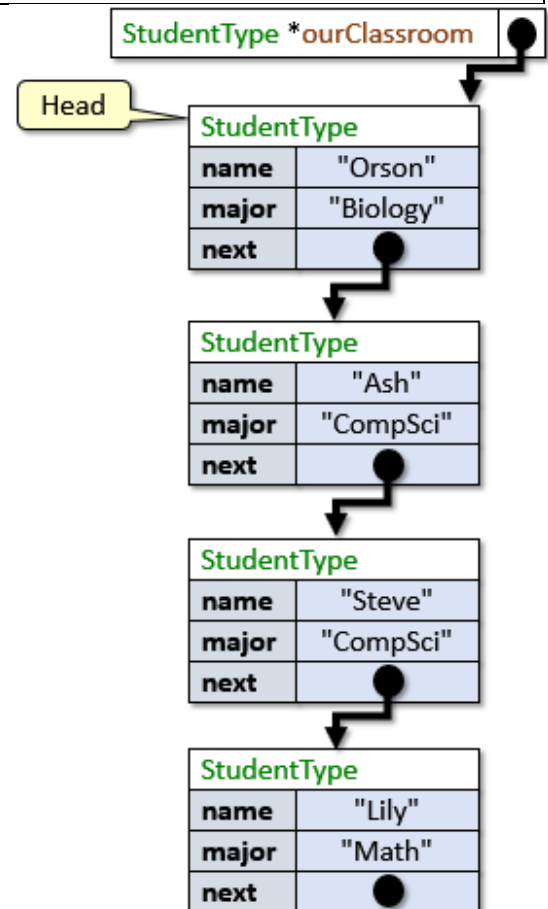
Now consider writing a program that creates a list of students and their majors. Assume that we do not know how many students there will be. So, we will need to create a list of students, allocating memory for each student as that student is entered into the system.

Here, to the right, is an example of the list that we will create, using the **struct** defined below:

```
typedef struct Student {
    char name[MAX_STR];
    char major[MAX_STR];
    struct Student *next;
} StudentType;
```

Notice that the struct is a **Student** struct and the overall type is defined as **StudentType**. The program is on the next page. It follows from the previous example that we just completed.

Interestingly, notice how the code produces the list in the reverse order that the items are entered, with the most recent one being the head. As an exercise, see if you can alter the code to reverse the order.



Code from **basicStudentList.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_STR 32

typedef struct Student {
    char    name[MAX_STR];
    char    major[MAX_STR];
    struct Student *next;
} StudentType;

// These are the functions used in the main function.
// They are defined here since the main function appears
// before them.
void createStudent(char*, char*, StudentType**);
void printStudent(StudentType*);
void freeList(StudentType*);

int main() {
    StudentType *ourClassroom = NULL;
    StudentType *currStudent;
    char    str1[MAX_STR];
    char    str2[MAX_STR];

    printf("\nEnter student names and their majors (use -1 when done): ");
    while(1) {
        printf("\nEnter name: ");
        scanf("%s", str1);
        if (strcmp(str1, "-1") == 0)
            break;
        printf("Enter major: ");
        scanf("%s", str2);

        createStudent(str1, str2, &currStudent);

        currStudent->next = ourClassroom;
        ourClassroom = currStudent;
    }

    printf("\nHere is the list:\n");
    printf("%-15s %-15s\n", "NAME", "MAJOR");
    printf("-----\n");

    currStudent = ourClassroom;
    while(currStudent != NULL) {
        printStudent(currStudent);
        currStudent = currStudent->next;
    }
    freeList(ourClassroom);
}
```

```
// Allocates memory for a new student and initializes it with the given data
void createStudent(char *name, char *major, StudentType **student) {
    *student = (StudentType *) malloc(sizeof(StudentType));
    if (*student == NULL) {
        printf("Memory allocation error\n");
        exit(-1);
    }
    strcpy((*student)->name, name);
    strcpy((*student)->major, major);
    (*student)->next = NULL;
}

// Prints a single student's information
void printStudent(StudentType *sPtr) {
    printf("%-15s %-15s\n", sPtr->name, sPtr->major);
}

// Free all items in a Singly-Linked List
void freeList(StudentType *listItem) {
    StudentType *nextItem;

    while(listItem != NULL) {
        nextItem = listItem->next;
        free(listItem);
        listItem = nextItem;
    }
}
```

Here is the output for our particular example shown earlier:

```
Enter student names and their majors (use -1 when done):
Enter name: Orson
Enter major: Biology

Enter name: Ash
Enter major: CompSci

Enter name: Steve
Enter major: CompSci

Enter name: Lily
Enter major: Math

Enter name: -1

Here is the list:
NAME                MAJOR
-----
Lily                Math
Steve              CompSci
Ash                CompSci
Orson              Biology
```

In this example, we are actually doing some very bad software engineering 😞. Why? You may have noticed that the structure mixed the data of the list item with the list mechanics. That is, we have name, major and next as all seemingly equal parts of the structure.

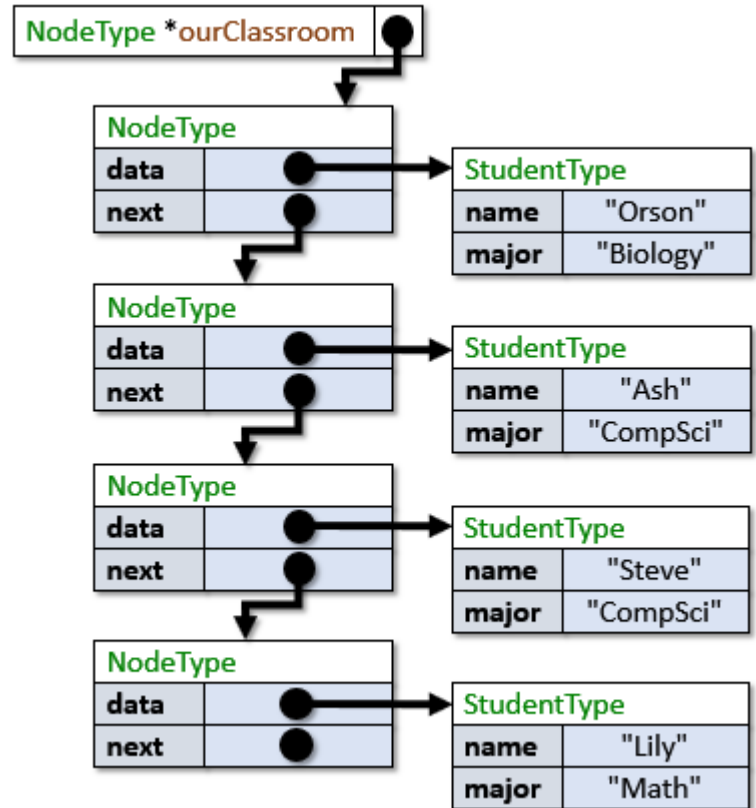
This is not proper *encapsulation*. When dealing with lists of items, the mechanics of *how* the list is defined (i.e., next pointer ... and previous pointers for doubly-linked lists ... we'll talk about that later) is not clearly identifiable. Each item is hard-coded to point to a specific other item.

What if we wanted the same item to appear in multiple lists in order to share data? We cannot do it with this current structure.

A solution to this poor design is to apply encapsulation. We will keep the data-related stuff together by making a separate structure for the data itself. Then we can point to that data and even re-use it in other lists whenever we need to. So, we should define two separate structs as follows:

```
typedef struct {
    char    name[MAX_STR];
    char    major[MAX_STR];
} StudentType;
```

```
typedef struct Node {
    StudentType *data;
    struct Node *next;
} NodeType;
```



Of course, there is a bit more work now. We have to allocate memory for the student data and also allocate memory for the node itself. Here is the updated code:

Code from **advancedStudentList.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_STR 32

typedef struct {
    char    name[MAX_STR];
    char    major[MAX_STR];
} StudentType;

typedef struct Node {
    StudentType *data;
    struct Node *next;
} NodeType;

void createStudent(char*, char*, StudentType**);
```

```

void createNode(NodeType**, StudentType*);
void printStudent(StudentType*);
void freeList(NodeType*);

int main() {
    NodeType    *ourClassroom = NULL;
    NodeType    *currNode = NULL;
    StudentType *currStudent;
    char         str1[MAX_STR];
    char         str2[MAX_STR];

    printf("\nEnter student names and their majors (use -1 when done): ");
    while(1) {
        printf("\nEnter name: ");
        scanf("%s", str1);
        if (strcmp(str1, "-1") == 0)
            break;
        printf("Enter major: ");
        scanf("%s", str2);

        createStudent(str1, str2, &currStudent);
        createNode(&currNode, currStudent);

        currNode->next = ourClassroom;
        ourClassroom = currNode;
    }

    printf("\nHere is the list:\n");
    printf("%-15s %-15s\n", "NAME", "MAJOR");
    printf("-----\n");

    currNode = ourClassroom;
    while(currNode != NULL) {
        printStudent(currNode->data);
        currNode = currNode->next;
    }
    freeList(ourClassroom);
}

// Allocates memory for a new student and initializes it with the given data
void createStudent(char *name, char *major, StudentType **student) {
    *student = (StudentType *) malloc(sizeof(StudentType));
    if (*student == NULL) {
        printf("Memory allocation error\n");
        exit(-1);
    }
    strcpy((*student)->name, name);
    strcpy((*student)->major, major);
}

// Allocates memory for a new list Node
void createNode(NodeType **node, StudentType *data) {
    *node = (NodeType *) malloc(sizeof(NodeType));
    if (*node == NULL) {
        printf("Memory allocation error\n");
        exit(-1);
    }
    (*node)->data = data;
    (*node)->next = NULL;
}

```

```
// Prints a single student's information
void printStudent(StudentType *sPtr) {
    printf("%-15s %-15s\n", sPtr->name, sPtr->major);
}

// Free all items in a Singly-Linked List
void freeList(NodeType *aNode) {
    NodeType *nextItem;

    while(aNode != NULL) {
        nextItem = aNode->next;
        free(aNode->data);
        free(aNode);
        aNode = nextItem;
    }
}
```

The code produces the same output, but it adheres to proper software-engineering principles.

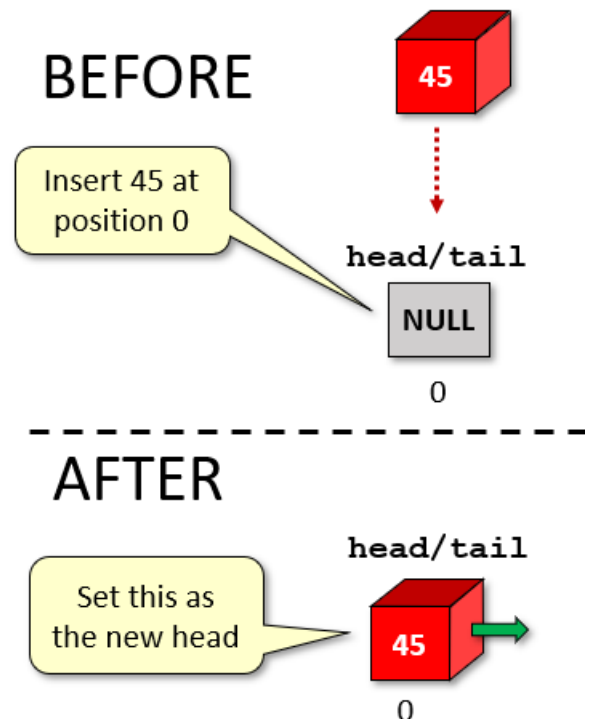
Let's now work on the functions of inserting and deleting elements in a list, since these are very common operations that we need to perform on dynamic lists.

When inserting ... there are four cases that we will need to consider:

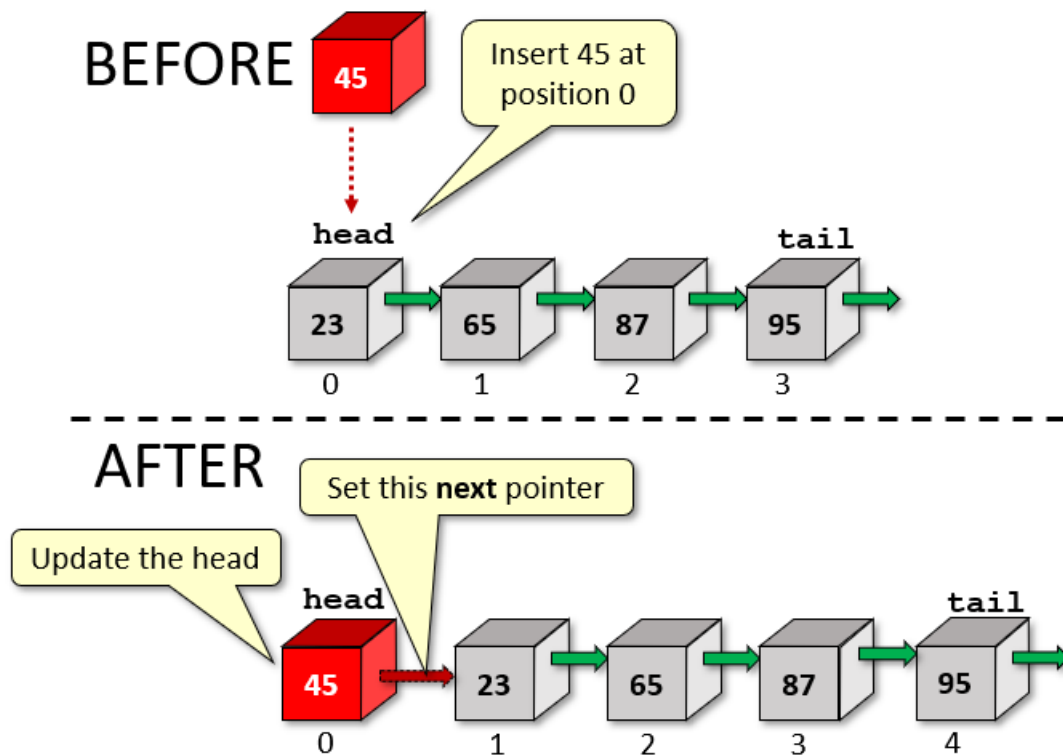
- inserting into an empty list
- inserting at the front of a list (i.e., a new head)
- inserting at the end of a list (i.e., a new tail)
- inserting in the middle of the list

Assume that we want to write our **insert** function so that it takes the head of the list, the data and the position in the list that we want to insert at (assuming 0 is the front of the list).

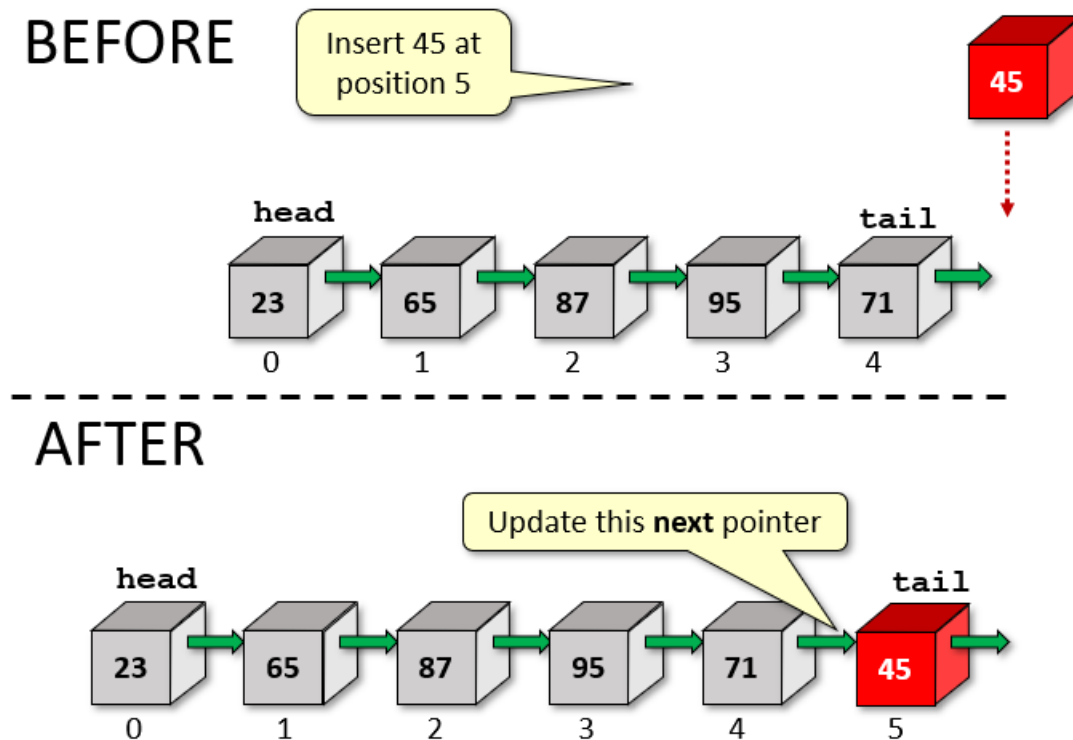
Here is the case for an empty list insertion:



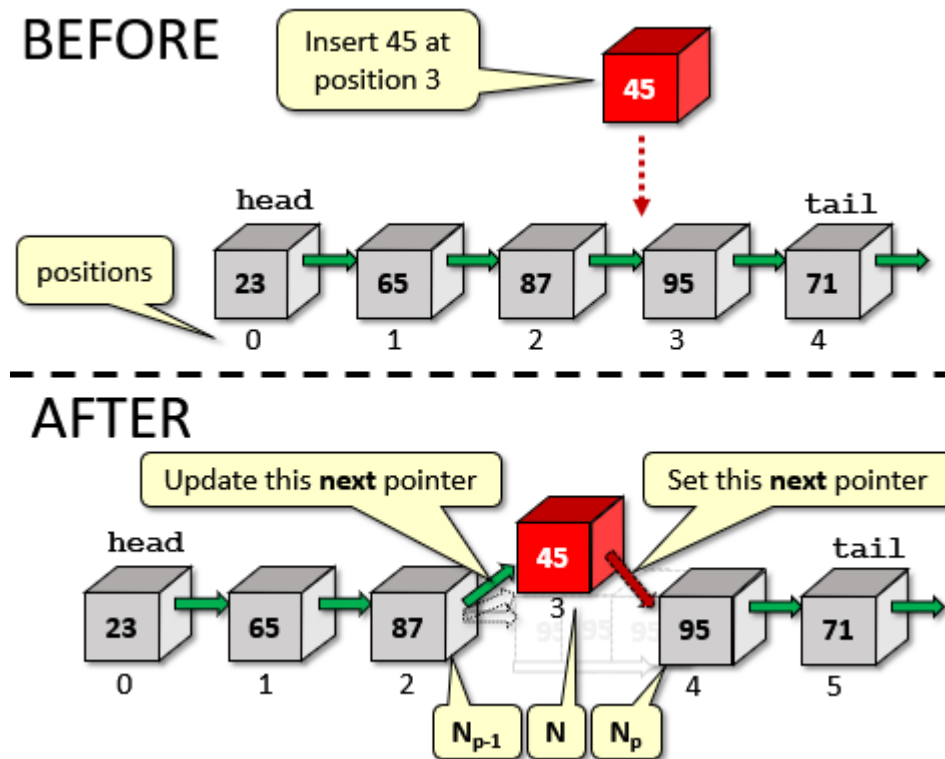
Here is the case for a new head insertion:



Here is the case for a new tail insertion:



Here is what we need to do for the general case of insertion into the middle:



For the special cases of inserting at the front of the list or for an empty list, we just have to make sure that we update the head. Here are the steps involved:

- Iterate through p nodes to get to node N_p at position p .
- If we run out of nodes before getting to p , then p is invalid.
- Allocate memory for new node N .
- If $p == 0$, then make node N the new head.
- Otherwise update the next pointer for node at position N_{p-1} to point to node N .
- Set the next pointer for node N to point to node N_p .

Here is the code:

```
void insertStudent(NodeType **head, StudentType *student, int pos){
    NodeType *newNode;
    NodeType *currNode, *prevNode;
    int currPos;

    // Iterate through the list up to the position to insert
    // at, keeping track of the previous node in the list so that
    // we can connect to it.
    prevNode = NULL;
    currNode = *head;
    currPos = 0;
```

```

while (currNode != NULL) {
    if (currPos == pos)
        break;
    currPos++;
    prevNode = currNode;
    currNode = currNode->next;
}

// If the position was invalid, then quit
if (currPos != pos) {
    printf("invalid position\n");
    free(student); // needed for our code, but not in general
    return;
}

// Create the new node
newNode = (NodeType *) malloc(sizeof(NodeType));
if (newNode == NULL) {
    printf("Memory allocation error\n");
    exit(-1);
}
newNode->data = student;
newNode->next = NULL;

// If prevNode is NULL, then this is the first position in
// the list, or the list was NULL to begin with. Otherwise
// we are inserting in the middle or at the end of the list.
if (prevNode == NULL)
    *head = newNode;
else
    prevNode->next = newNode;

newNode->next = currNode; // Connect new node to rest of the list
}

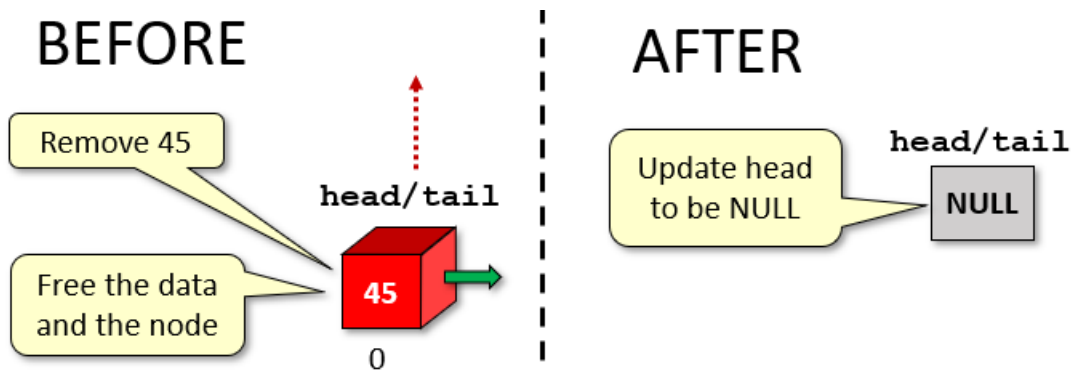
```

What about the removal of nodes? When deleting ... there are 5 cases to consider:

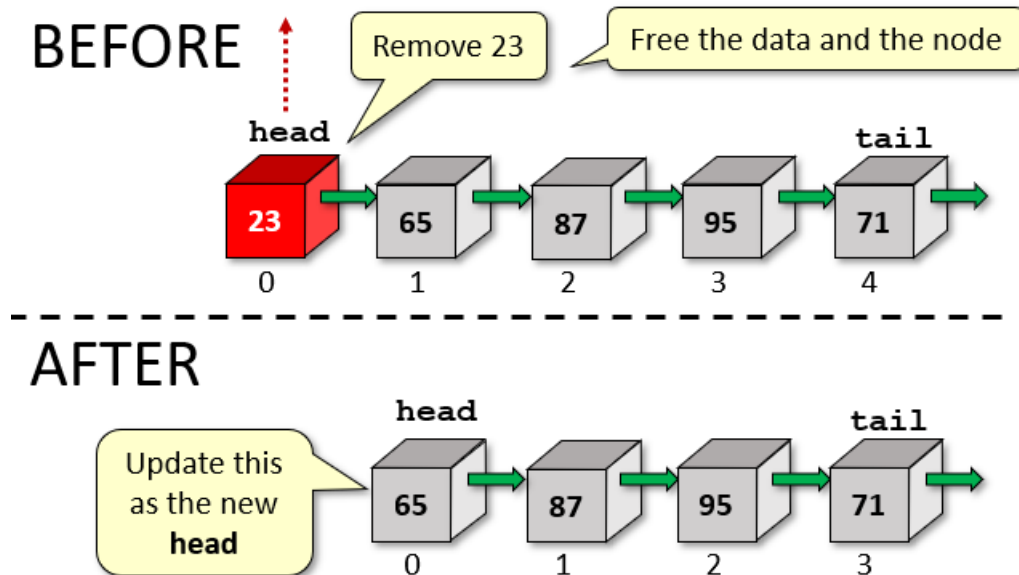
- removing from an empty list
- removing the only element in the list
- removing from the front of a list (i.e., we'll need to update the head)
- removing from the end of a list (i.e., there will be a new tail)
- removing from the middle of the list

Assume that we want to write our **remove** function so that it takes the head of the list and the data to be removed. Of course, we could have written a function that removed the element at a given position, but for variety, we'll remove based on finding a matching element.

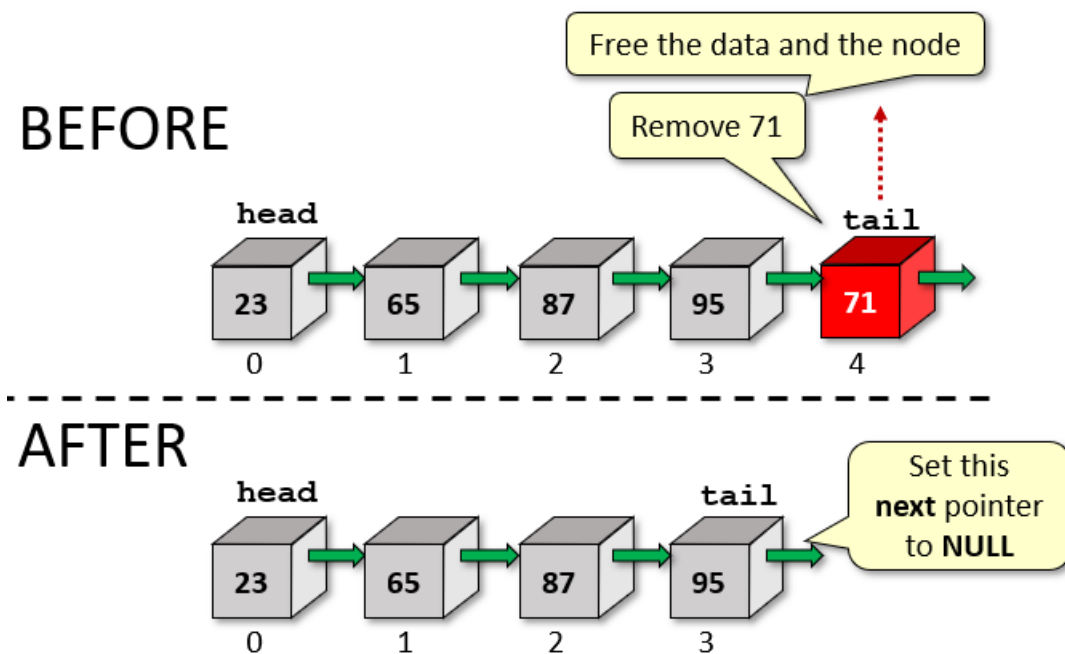
The case for removing from an empty list is simple ... if the list is **NULL**, then do nothing. Also, if it is the only element in the list, then we just need to update the head of the list when done.



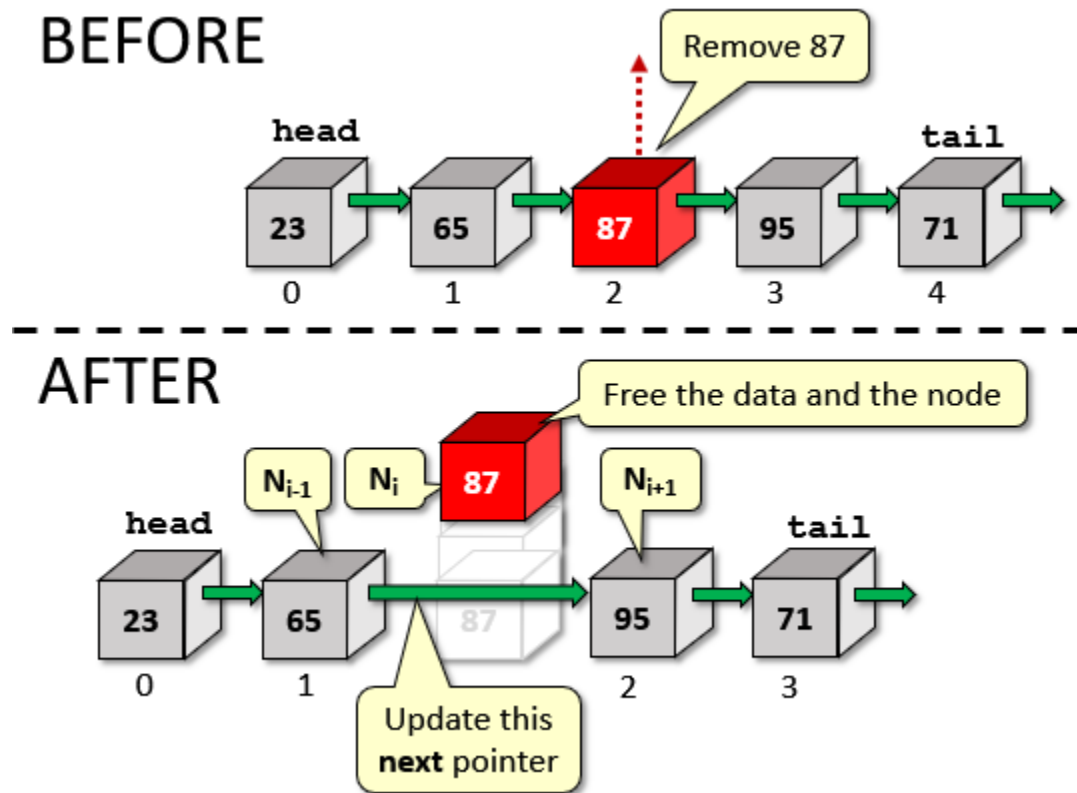
If it is the head of an existing list with multiple elements ... we just move the **head** over:



For removing the tail ... there is only one pointer to update:



Finally, here is what we need to do for the general case of removal from the middle:



Here are the steps involved:

- Iterate through the nodes to get to node N_i whose data matches the item to remove.
- If $i == 0$, then make N_{i+1} the new head.
- Otherwise update the next pointer for the node at position N_{i-1} to point to node N_{i+1} .
- Free the memory corresponding to the data of the removed node, if necessary.
- Free the memory corresponding to the removed node itself.

Here is the code:

```
int deleteStudent(NodeType **head, char *nameToDelete) {
    NodeType *currNode, *prevNode;

    // Iterate through the list to find the student with the given
    // name, keeping track of the previous node in the list so that
    // we can disconnect it.
    prevNode = NULL;
    currNode = *head;
    while (currNode != NULL) {
        if (strcmp(currNode->data->name, nameToDelete) == 0)
            break;
        prevNode = currNode;
        currNode = currNode->next;
    }
}
```

```

    // If the name was not found, then quit with a -1
    if (currNode == NULL)
        return -1;

    // If the removed node was the head, then update the head,
    // otherwise move the next pointer around this removed node.
    if (prevNode == NULL)
        *head = currNode->next;
    else
        prevNode->next = currNode->next;

    // Make sure to free up the node and the data!
    free(currNode->data); // does not necessarily need to be done here
    free(currNode);

    return 0;
}

```

Here is the completed code all together:

Code from **completeStudentList.c**

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_STR 32

typedef struct {
    char    name[MAX_STR];
    char    major[MAX_STR];
} StudentType;

typedef struct Node {
    StudentType *data;
    struct Node *next;
} NodeType;

void createStudent(char*, char*, StudentType**);
void createNode(NodeType**, StudentType*);
void printStudent(StudentType*);
void freeList(NodeType*);
void insertStudent(NodeType**, StudentType*, int);
int deleteStudent(NodeType**, char*);

int main() {
    NodeType    *ourClassroom = NULL;
    NodeType    *currNode = NULL;
    StudentType *currStudent;
    char        str1[MAX_STR];
    char        str2[MAX_STR];

    printf("\nEnter student names and their majors (use -1 when done): ");
    while(1) {
        printf("\nEnter name: ");
        scanf("%s", str1);

```

```

    if (strcmp(str1, "-1") == 0)
        break;
    printf("Enter major: ");
    scanf("%s", str2);

    createStudent(str1, str2, &currStudent);
    insertStudent(&ourClassroom, currStudent, 0);
}

printf("\nHere is the list:\n");
printf("%-15s %-15s\n", "NAME", "MAJOR");
printf("-----\n");

currNode = ourClassroom;
while(currNode != NULL) {
    printStudent(currNode->data);
    currNode = currNode->next;
}

printf("Who would you like to delete? ");
scanf("%s", str1);
printf("Deleting %s ...\n", str1);
if (deleteStudent(&ourClassroom, str1) == -1) {
    printf("Error deleting student %s ... continuing with program ...\n", str1);
}

printf("\nHere is the list:\n");
printf("%-15s %-15s\n", "NAME", "MAJOR");
printf("-----\n");

currNode = ourClassroom;
while(currNode != NULL) {
    printStudent(currNode->data);
    currNode = currNode->next;
}
freeList(ourClassroom);
}

// Allocates memory for a new student and initializes it with the given data
void createStudent(char *name, char *major, StudentType **student) {
    *student = (StudentType *) malloc(sizeof(StudentType));
    if (*student == NULL) {
        printf("Memory allocation error\n");
        exit(-1);
    }
    strcpy((*student)->name, name);
    strcpy((*student)->major, major);
}

// Allocates memory for a new list Node
void createNode(NodeType **node, StudentType *data) {
    *node = (NodeType *) malloc(sizeof(NodeType));
    if (*node == NULL) {
        printf("Memory allocation error\n");
        exit(-1);
    }
    (*node)->data = data;
    (*node)->next = NULL;
}

```

```

// Prints a single student's information
void printStudent(StudentType *sPtr) {
    printf("%-15s %-15s\n", sPtr->name, sPtr->major);
}

// Free all items in a Singly-Linked List
void freeList(NodeType *listItem) {
    NodeType *nextItem;

    while(listItem != NULL) {
        nextItem = listItem->next;
        free(listItem->data);
        free(listItem);
        listItem = nextItem;
    }
}

// Insert the given student into the given list at the specified position.
// If position is 0, then insert as new head of the list.
void insertStudent(NodeType **head, StudentType *student, int pos){
    NodeType *newNode;
    NodeType *currNode, *prevNode;
    int currPos;

    // Iterate through the list up to the position to insert
    // at, keeping track of the previous node in the list so that
    // we can connect to it.
    prevNode = NULL;
    currNode = *head;
    currPos = 0;
    while (currNode != NULL) {
        if (currPos == pos)
            break;
        currPos++;
        prevNode = currNode;
        currNode = currNode->next;
    }

    // If the position was invalid, then quit
    if (currPos != pos) {
        printf("invalid position\n");
        free(student); // needed for our code, but not in general
        return;
    }

    // Create the new node
    newNode = (NodeType *) malloc(sizeof(NodeType));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(-1);
    }
    newNode->data = student;
    newNode->next = currNode; // Connect rest of the list to the new node

    // If prevNode is NULL, then this is the first position in
    // the list, or the list was NULL to begin with. Otherwise
    // we are inserting in the middle or at the end of the list.
    if (prevNode == NULL)
        *head = newNode;

```

```

else
    prevNode->next = newNode;
}

// Delete the student with the given name from the given list.
// If position is 0, then change the head of the list. Return
// -1 if the name was not found in the list, else return 0.
int deleteStudent(NodeType **head, char *nameToDelete) {
    NodeType *currNode, *prevNode;

    // Iterate through the list to find the student with the given
    // name, keeping track of the previous node in the list so that
    // we can disconnect it.
    prevNode = NULL;
    currNode = *head;
    while (currNode != NULL) {
        if (strcmp(currNode->data->name, nameToDelete) == 0)
            break;
        prevNode = currNode;
        currNode = currNode->next;
    }

    // If the name was not found, then quit with a -1
    if (currNode == NULL)
        return -1;

    // If the removed node was the head, then update the head,
    // otherwise move the next pointer around this removed node.
    if (prevNode == NULL)
        *head = currNode->next;
    else
        prevNode->next = currNode->next;

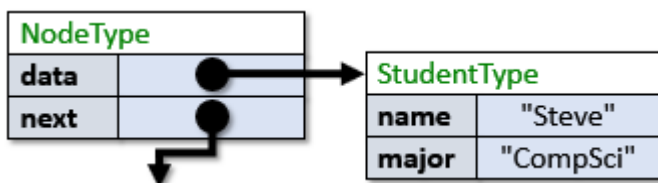
    // Make sure to free up the node and the data!
    free(currNode->data); // does not necessarily need to be done here
    free(currNode);

    return 0;
}

```

You may have noticed that the list functions that we wrote always required us to pass in the head of the list. Sometimes, however, it is more convenient and faster to pass in arbitrary elements from a list. For example, what if you iterate through a list and find a particular item that you were looking for and then you pass that item to a function to inspect or modify it. Perhaps you may end up wanting to delete it from the list.

Assume that we found **Steve** here from the middle of the **ourClassroom** list in our example:

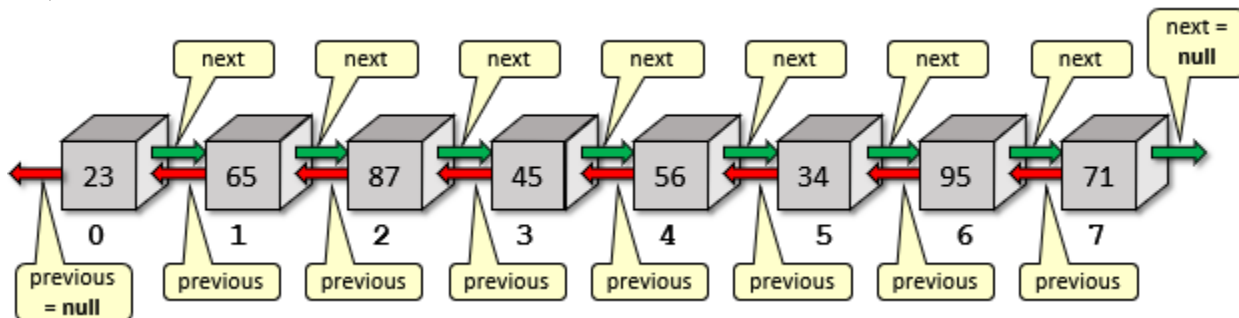


Assume that there is much more information than just the name and major ... such as GPA, CGPA, etc... If we are in a function that is examining Steve's data and we decide that Steve should be removed from the class list ... how do we do it? Since Steve is in the middle of some list somewhere ... we will need to know who was before Steve in the list so that we could update that node's **next** pointer to bypass Steve. We could always start at the front of the list again and iterate through the nodes until we find Steve ... keeping track of the previous item ... just as we did in the **deleteStudent** function that we wrote. But this is slow.

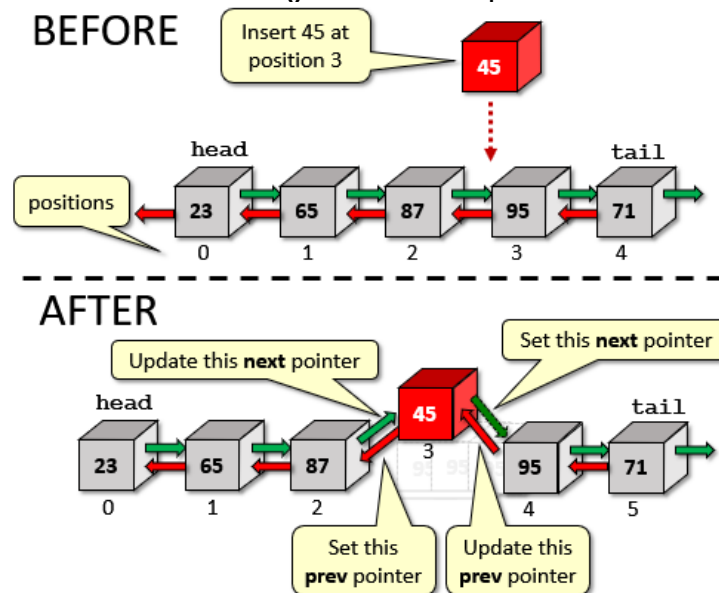
A quicker way to do this would be to have the nodes in the list keep track of the **previous** node along with the **next** node. Then we would know who comes before Steve in the list and could update that node's **next** pointer quickly to bypass Steve. This would make the deletion an **O(1)** operation instead of **O(n)**. To make this happen, we would have to redefine the **NodeType** data structure to be a **Doubly-Linked List**. That means, we would need to add a link (i.e., a pointer) to the previous node as follows:

```
typedef struct Node {
    StudentType *data;
    struct Node *next;
    struct Node *prev;
} NodeType;
```

So, each item in the list will know the item *before* it and the item *after* it:



We would need to update our **insertStudent()** function to update both **next** and **prev** pointers:



We just need to set the **prev** pointers for the **newNode** and **currNode**. But we can also simplify the code by not having to hold onto the previous node in a separate variable.

```
void insertStudent(NodeType **head, StudentType *student, int pos){
    NodeType    *newNode;
    NodeType    *currNode, *prevNode; // don't need prevNode variable now
    int         currPos;

    // Iterate through the list up to the position to insert at
    prevNode = NULL;
    currNode = *head;
    currPos  = 0;

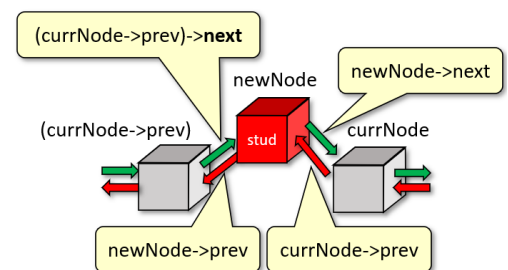
    while (currNode != NULL) {
        if (currPos == pos)
            break;
        currPos++;
        prevNode = currNode;
        currNode = currNode->next;
    }

    // If the position was invalid, then quit
    if (currPos != pos) {
        printf("invalid position\n");
        free(student); // needed for our example, but not in general
        return;
    }

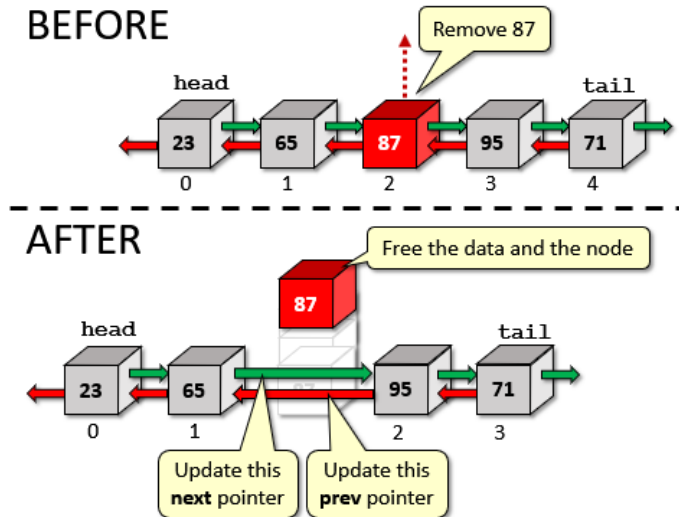
    // Create the new node
    newNode = (NodeType *) malloc(sizeof(NodeType));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(-1);
    }
    newNode->data = student;
    newNode->next = NULL;
    newNode->prev = NULL;

    // If the currentNode is NULL then this is an empty list
    if (currNode == NULL)
        *head = newNode;
    else {
        // If currentNode's previous is NULL then insert at front
        if (currNode->prev == NULL)
            *head = newNode;
        else // Otherwise we are inserting in the middle somewhere
            (currNode->prev)->next = newNode;

        newNode->prev = currNode->prev;
        currNode->prev = newNode;
        newNode->next = currNode;
    }
}
```



We would also need to update our **deleteStudent()** function, making sure that the **next** and **prev** pointers are set properly:



Here is the code:

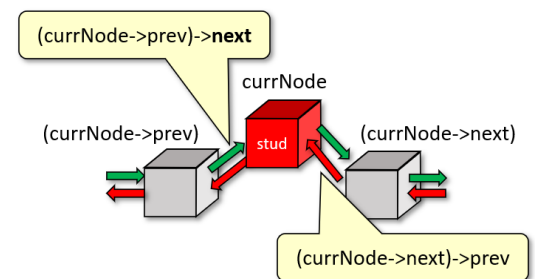
```
int deleteStudent(NodeType **head, char *nameToDelete) {
    NodeType *currNode;

    // Iterate through the list to find the student with the given name
    currNode = *head;
    while (currNode != NULL) {
        if (strcmp(currNode->data->name, nameToDelete) == 0)
            break;
        currNode = currNode->next;
    }
    // If the name was not found, then quit with a -1
    if (currNode == NULL)
        return -1;

    // If the removed node was the head, then update the head,
    // otherwise move the next pointer around this removed node.
    if (currNode->prev == NULL)
        *head = currNode->next;
    else
        currNode->prev->next = currNode->next;

    if (currNode->next != NULL)
        currNode->next->prev = currNode->prev;

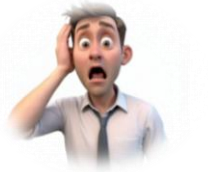
    // Make sure to free up the node and the data!
    free(currNode->data); // needed for our example
    free(currNode);
    return 0;
}
```



Linked-lists have an advantage over static arrays in that they can grow and shrink as needed to accommodate the current amount of data. In that way, they are space-efficient with respect to the data being stored.

However, we must keep in mind that there is a space overhead involved since we need to store the next and previous pointers for each item in the doubly-linked list.

This is an overhead of **16** bytes per item for a 64-bit system. If we are trying to store **1MB** of data, then this can be a **16MB** overhead! Singly-linked lists would have half the overhead (i.e., **8MB**). Nevertheless, this is wasteful.



Another option that can be used for the efficient storage of data that does not require this large pointer overhead is that of using **dynamically-allocated arrays**. As a bit of review, in order to make this clear, consider a **statically-allocated array** of **StudentType** data as follows:

```
#define MAX_STR      32
#define MAX_CAPACITY 375

typedef struct {
    char    name[MAX_STR];
    char    major[MAX_STR];
} StudentType;

StudentType myClassroom[MAX_CAPACITY];
```

Here we would be creating a fixed (i.e., static) array that could store exactly **375** Students, each requiring **64** bytes of storage to store their name and major. That means, we would be taking up **375 * (64) = 24,000** bytes upon initialization. This memory would be allocated permanently for the program regardless of whether or not we have less (or more) students. It is clearly inefficient.

Instead, we could create **myClassroom** as a dynamically-allocated array as follows:

```
StudentType *myClassroom[MAX_CAPACITY];
```

How much space does this require? Well ... it still stores **375** pointers ... but each pointer is just **8** bytes (assuming a 64-bit system) ... so it takes up just **3,000** bytes. This is much smaller upon initialization. But ... these are just pointers. We would still need to allocate space for each student that we added. With the array of pointers, there is an **8-byte** overhead in addition to allocating the space to store the name and major for a student. Therefore, a full array of **375** students would require **375 * (64 + 8) = 27,000** bytes. So, it takes up a little bit more storage. However, if we only needed to store **100** students, the first strategy takes up **24,000** bytes (since it is static ... fixed-size upon compiling) while the pointer version would only take up **7,200** bytes. In conclusion, it is more efficient to use the pointer version as long as there are less than **334** students (calculated as **24,000 / 27,000 * 375**).



In the case where we don't have any idea as to what capacity to use, we can make that flexible as well.

Consider the following **typedef**:

```
typedef struct {
    int    capacity; // Allocated array size
    int    size;     // Number of items currently in array
    StudentType **items; // Pointer to pointer to first item in array
} DynamicArrayType;
```

Now we can create the **myClassroom** array using a completely dynamically-allocated array as follows:

```
DynamicArrayType myClassroom;
```

Now this variable only takes up **16** bytes (assuming 64-bit system) upon initialization. It keeps track of its own capacity and number of elements currently in it. The items are then dynamically-allocated. If we want **100** students, we just do this:

```
myClassroom.capacity = 100; // Maximum items we can add
myClassroom.size = 0;      // Nothing in it yet
myClassroom.items = (StudentType **)malloc(100*sizeof(StudentType *));
```

This makes the array able to hold **100** pointers to students, although no students have been created as of yet. We would need to **malloc()** space for each student that we added and store the resulting pointer in the **myClassroom.items** array.

This is the most “size-flexible” way of allocating arrays if you are worried about storage space.

3.6 Function Pointers

You should now understand how to create and use pointers to *variables*. We will now look at pointers to *functions*.

*A **function pointer** is a variable that stores a pointer to a function's code in memory.*

We can call a function by using a function pointer. These function pointers can be passed in as parameters ... just like any other variable.

Why should we use them? They can be used a little bit like a selection dial to select a function/mode (e.g., function dial on a camera). They allow us substitute whatever function we want to call while our program is running. So, rather than requiring a set of **IF** statements to decide which function to call in a certain circumstance, we just pass in the function as needed.

In JAVA, we “sort of” used function pointers when we plugged in our event handlers. That allowed us to plug in our own function that was to be called whenever a button was clicked, for example, on the user interface.



Here is an example of how to declare a function pointer variable:

```
void (*fPtr) (int, float);
```

Here, **fPtr** is the variable name. It is being declared as a pointer to a function that takes two parameters (i.e., an **int** and a **float**) and returns **void**.

Consider a function that takes two such parameters:

```
void add(int x, float y) {
    printf("%f\n", x + y);
}
```

We can “plug” this function into the `fPtr` variable as follows:

```
fPtr = add;           or           fPtr = &add;
```

Notice that the `&` character is optional.

Once we have plugged the function in, we can then call the `add()` function either directly, or through the function pointer.

All three lines below will call the `add()` function resulting in the output of `7.000000`:

```
add(2, 5.0);           // call the function directly
fPtr(2, 5.0);          // call the function via the pointer
(*fPtr)(2, 5.0);       // call function via de-referenced pointer
```

The benefit of the pointer is not seen in this example. But consider now using different functions for different operations. Here is code that plugs **3** different functions into the pointer:

Code from `functionPointer.c`

```
#include <stdio.h>
#include <stdlib.h>

int add(int x, int y) {
    return (x + y);
}
int subtract(int x, int y) {
    return (x - y);
}
int multiply(int x, int y) {
    return (x * y);
}

int main() {
    int (*fPtr)(int, int);

    fPtr = add;
    printf("%d\n", fPtr(2, 5)); // Call the add function

    fPtr = subtract;
    printf("%d\n", fPtr(2, 5)); // Call the subtract function

    fPtr = multiply;
    printf("%d\n", fPtr(2, 5)); // Call the multiply function
}
```

Notice how the function is called the same way all three times. The result is **7**, **-3** and **10**.

The following example shows how flexible this pointer can be. It creates an array of **4** function pointers and uses a **FOR** loop to iterate through the 4 functions:

Code from **functionPointerArray.c**

```
#include <stdio.h>
#include <stdlib.h>

int add(int x, int y) { return (x + y); }
int subtract(int x, int y) { return (x - y); }
int multiply(int x, int y) { return (x * y); }
int divide(int x, int y) { return (x / y); }

int main() {
    int (*fPtr[4])(int, int) = {add, subtract, multiply, divide};

    for (int i=0; i<4; i++)
        printf("%d\n", fPtr[i](2,5));
}
```

This code produces the numbers **7, -3, 10, 0**.

Hopefully, you are beginning to see how flexible our code can become with function pointers. Here is one more example showing how we can plug a function in according to our needs. It processes an array of **int** types ... first to print out the odd numbers, then to print out the even:

Code from **processArray.c**

```
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE 10

int processArray(int *arr, void (*printFunction)(int *)) {
    for (int i=0; i<ARRAY_SIZE; i++, arr++)
        printFunction(arr);
}

void printOdd(int *num) {
    if (*num %2 == 1) printf("%d\n", *num);
}

void printEven(int *num) {
    if (*num %2 == 0) printf("%d\n", *num);
}

int main() {
    int arr[10] = {11, 14, 22, 34, 41, 53, 61, 76, 87, 98};

    printf("Odd: \n");
    processArray(arr, printOdd);
    printf("\nEven: \n");
    processArray(arr, printEven);
}
```

The output:

```
Odd:
11
41
53
61
87

Even:
14
22
34
76
98
```

Here is a variation of the code that uses the individual functions to decide if the number should be selected for printing by returning **1** or **0** as a **char** (since booleans are not available in C).

Code from <code>processArray2.c</code>	The output:
<pre> #include <stdio.h> #include <stdlib.h> int processArray(int *arr, char (*shouldPrint)(int *)) { for (int i=0; i<10; i++, arr++) if (shouldPrint(arr)) printf("%d\n", *arr); } char odd(int *num) { return (*num%2 == 1); } char even(int *num) { return (*num%2 == 0); } char all(int *num) { return 1; } int main() { int arr[10] = {11, 14, 22, 34, 41, 53, 61, 76, 87, 98}; printf("Odd: \n"); processArray(arr, odd); printf("\nEven: \n"); processArray(arr, even); printf("\nAll: \n"); processArray(arr, all); } </pre>	<pre> Odd: 11 41 53 61 87 Even: 14 22 34 76 98 All: 11 14 22 34 41 53 61 76 87 98 </pre>



As a final example, we will consider the quick-sort sorting algorithm. This algorithm sorts by comparing two values at a time through use of a **comparator** function (just as we used in JAVA). We will want to be able to plug in different comparator functions in order to sort in different ways, such as increasing or decreasing order.

C provides a built-in **qsort()** function that implements the quicksort algorithm. It operates on arrays and can sort any data type. It is available in `<stdlib.h>`. The function is defined as follows:

```

void qsort(void *buf, size_t numItems, size_t itemSize,
           int (*compare)(const void *, const void *));

```

Here, **buf** is the address of the array to be sorted, **numItems** is the number of items to be sorted and **itemSize** is the size (in bytes) of each item. **compare** is a comparison function that accepts two array items as parameters and returns an integer indicating the relationship between the items.

The order by which things are sorted is based on the comparison function.

Consider a simple comparison function that we might write it in JAVA which takes two integers n_1 and n_2 . The function should return a negative number (e.g., -1) if n_1 is supposed to come before n_2 in the sort order. It should return a positive non-zero number (e.g., 1) if n_1 is supposed to come after n_2 in the sort order. It should return 0 if n_1 equals n_2 . Here is how to write it:

```
int compare(int n1, int n2) {
    if (n1 < n2) return -1;
    if (n1 > n2) return 1;
    return 0;
}
```

In C, however, the parameters must be declared as **const void ***, not as **int**. So, we will need to do some typecasting and dereferencing:

```
int compare(const void *p1, const void *p2) {
    int n1 = *(int *)p1;
    int n2 = *(int *)p2;

    if (n1 < n2) return -1;
    if (n1 > n2) return 1;
    return 0;
}
```



Here is an example showing how to plug such a function into the **qsort()** function to sort a list of **10** randomly-created integers. We plug in three functions ... one to sort in increasing order, one to sort in decreasing order and one to sort in order of the number of digits in each number:

Code from qsort.c	The output:
<pre>#include <stdio.h> #include <stdlib.h> #include <string.h> #include <time.h> #define ARRAY_SIZE 10 void printArray(int *arr) { for(int i=0; i<ARRAY_SIZE; i++) printf("%d\n", arr[i]); } int compareIncreasing(const void *p1, const void *p2) { int n1 = *(int *)p1; int n2 = *(int *)p2; if (n1 < n2) return -1; if (n1 > n2) return 1; return 0; }</pre>	Original Order 18 255 195 72 9 14 190 18 0 192 Increasing Order 0 9 14 18 18 72 190 192 195 255

<pre> int compareDecreasing(const void *p1, const void *p2) { int n1 = *(int *)p1; int n2 = *(int *)p2; if (n1 < n2) return 1; if (n1 > n2) return -1; return 0; } int compareDigits(const void *p1, const void *p2) { int n1 = *(int *)p1; int n2 = *(int *)p2; char s1[10], s2[10]; sprintf(s1, "%d", n1); sprintf(s2, "%d", n2); if (strlen(s1) < strlen(s2)) return -1; if (strlen(s1) > strlen(s2)) return 1; return 0; } int main() { int arr[ARRAY_SIZE]; srand(time(NULL)); for(int i=0; i<ARRAY_SIZE; i++) arr[i] = rand()%20 * (rand()%20); printf("Before qsort() \n"); printArray(arr); qsort(arr, ARRAY_SIZE, sizeof(int), compareIncreasing); printf("\nAfter qsort() in Increasing Order\n"); printArray(arr); qsort(arr, ARRAY_SIZE, sizeof(int), compareDecreasing); printf("\nAfter qsort() in Decreasing Order\n"); printArray(arr); qsort(arr, ARRAY_SIZE, sizeof(int), compareDigits); printf("\nAfter qsort() in Digit Count Order\n"); printArray(arr); } </pre>	<pre> Decreasing Order 255 195 192 190 72 18 18 14 9 0 Digit Count Order 9 0 72 18 18 14 255 195 192 190 </pre>
---	--