



Exception Handling

(sample code based on Deitel&Deitel)

C++ Exception Handling

Topics

- **Exception Handling**
- **C++ Exception Handling Basics**
- **Throwing and Catching Exceptions**
- **Unexpected Exceptions**
- **Constructors, Destructors and Exceptions**
- **Exceptions and Inheritance**

Handling Errors: Approach 1

Put error checking code among application logic

Pro:

- **Error code is placed where error occurs**
- **Makes it obvious how errors are dealt with**

Con:

- **Application logic becomes littered with error handling logic**
- **Makes it difficult to follow application logic**
- **Code (e.g. function returns) must anticipate the kind of error.**

T & pop() //what if stack is empty

Handling Errors: Approach 2

Remove Error Handling code from “Main code”

Pro:

- **Application logic is easier to follow and maintain**
- **Errors are handled consistently**
- **Increases the kinds of error one can catch**

Con:

- **Hides error handling when reading application logic -but that's the point**

Exceptions Handling is appropriate when...

- **The system can recover from the error using a recovery procedure (exception handler)**
- **Errors will be dealt with by a different part of the program than that which detected the error.**
- **A program that queries the user for input should probably not use exception handling to process user input errors**
- **Exception handling should probably not be used for conventional program flow of control because compilers may treat exceptions as rare and not produce optimized code**

C++ Exception Handling

Topics

- Exception Handling
- **C++ Exception Handling Basics**
- Throwing and Catching Exceptions
- Unexpected Exceptions
- Constructors, Destructors and Exceptions
- Exceptions and Inheritance

Exception Handling Basics

- **Exception handling is for situations where the function that detects the error cannot deal with it**
- **Such a function will throw an exception, and “hope” there is an exception handler to catch it**
- **There is no guarantee that the exception will be caught**
- **If it’s not caught the program terminates**

...Exception Handling Basics

- **Exceptions are thrown and caught using `try` and `catch` blocks**

```
try {  
    ...  
}  
catch {  
    ...  
}
```

- **Code in the `try` block can throw an exception**
- **The `catch` blocks contains handlers for the various exceptions**
- **When an exception is thrown in the `try` block, the `catch` blocks are searched for an appropriate exception handler**

Creating an error object to throw

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

class DivideByZeroError {
public:
    DivideByZeroError() : message("Divide by Zero") {}
    void printMessage() const {cout << message;}
private:
    const char* message;
};

int main(){
    DivideByZeroError err;
    err.printMessage();
    return 0;
}
```



Divide by Zero

Divide Function that throws an exception

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

float quotient(int numerator, int denominator) {
    if (denominator == 0)
        throw DivideByZeroError();
    return (float) numerator / denominator;
}

int main(){
    int i = 10, j = 3;
    cout << quotient(i,j) << "\n";
    return 0;
}
```




3.33333

Execution Generating an Error (without try block)

```
#include <iostream.h>

float quotient(int numerator, int denominator) {
    if (denominator == 0)
        throw DivideByZeroError();
    return (float) numerator / denominator;
}

main(){
    int i = 10, j = 0;
    cout << quotient(i,j);
    return 0;
}
```



```
TESTPROG.EXE
PROGRAM ABORTED
OK
```

Executing Code in a try block

```
int main(){
    cout << "Enter numerator and denominator: ";
    int numerator, denominator;
    cin >> numerator >> denominator;
    try {
        float answer = quotient(numerator, denominator);
        cout << "\n" << numerator << "/"
            << denominator << " = " << answer;
    }
    catch (DivideByZeroError error) { //error handler
        cout << "ERROR: ";
        error.printMessage();
        cout << endl;
        return 1; //terminate because of error
    }
    return 0; //terminate normally
}
```

```
Enter numerator and denominator: 3 4
```

```
3/4 = 0.75
```

...Executing Code in a try block

```
main() {
    cout << "Enter numerator and denominator: ";
    int numerator, denominator;
    cin >> numerator >> denominator;
    try {
        float answer = quotient(numerator, denominator);
        cout << "\n" << numerator << "/"
            << denominator << " = " << answer;
    }
    catch (DivideByZeroError error) { //error handler
        cout << "ERROR: ";
        error.printMessage();
        cout << endl;
        return 1; //terminate because of error
    }
    return 0; //terminate normally
}
```

```
Enter numerator and denominator: 3 0
ERROR: Divide by Zero
```

C++ Exception Handling

Topics


- Exception Handling
- C++ Exception Handling Basics
- Throwing and Catching Exceptions
- Unexpected Exceptions
- Constructors, Destructors and Exceptions
- Exceptions and Inheritance

Throwing an Exception

- When a program encounters an error it throws an exception using `throw anObject`
- `anObject` can be any type of object and is called the “exception object”
- Exception will be caught by the closest exception handler (for the `try` block from which the exception was thrown) which matches the exception object type
- A temporary copy of the exception object is created and initialized, which in turn initializes the parameter in the exception handler
- The temporary object is destroyed when the exception handler completes execution

...Throwing an Exception

```
class DivideByZeroError {
public:
    DivideByZeroError() : message("Divide by Zero")
        {cout << "construct: DivByZeroObject\n"; }
    ~DivideByZeroError()
        {cout << "delete: DivByZeroObject\n"; }
    void printMessage() const {cout << message;}
private:
    const char* message;
};
```



```
Enter numerator and denominator: 3 0
construct: DivByZeroObject
delete: DivByZeroObject
ERROR: Divide by Zero
delete: DivByZeroObject
delete: DivByZeroObject
```

Executing Code in a try block

```
main() {
    cout << "Enter numerator and denominator: ";
    int numerator, denominator;
    cin >> numerator >> denominator;
    try {
        float answer = quotient(numerator, denominator);
        cout << "\n" << numerator << "/"
            << denominator << " = " << answer;
    }
    catch (DivideByZeroError error) { //error handler
        cout << "ERROR: ";
        error.printMessage();
        cout << endl;
        return 1; //terminate because of error
    }
    return 0;
}
```

```
Enter numerator and denominator: 3 0
construct: DivByZeroObject
delete: DivByZeroObject
ERROR: Divide by Zero
delete: DivByZeroObject
delete: DivByZeroObject
```

...Throwing an Exception

- Control exits the current `try` block and goes to appropriate catch handler
- The `try` blocks or functions which throw the exception can be deeply nested
- An exception can be thrown by code indirectly referenced from within a `try` block
- An exception terminates the block in which the exception occurred, and may cause program termination

Catching an Exception

- **Exception handlers are contained within catch blocks**

```
catch(exception_object)
{ //exception handler code }
```

- **Catch handler specifies the type of object that it catches**
- **Exceptions which are not caught will cause program termination, by invoking function `terminate`, which in turn invokes `abort`**

...Catching an Exception

- **Specifying only the exception type is allowed, no object is passed, the type is simply used to select the appropriate handler**


```
catch(exception_type)
    { //exception handler code }
```

- **Specifying ellipses catches all exceptions**

```
catch(...)
    { //exception handler code }
```

Unnamed Exception Handlers -only type specified

```
try {  
    float answer = quotient(numerator, denominator);  
    cout << "\n";  
    cout << numerator << "/"  
        << denominator << " = " << answer;  
}  
catch (DivideByZeroError) { //type only error handler  
    cout << "ERROR: DIVIDE_BY_ZERO\n";  
    return 1; //terminate because of error  
}
```



```
Enter numerator and denominator: 3 0  
construct: DivByZeroObject  
delete: DivByZeroObject  
ERROR: DIVIDE_BY_ZERO  
delete: DivByZeroObject  
delete: DivByZeroObject
```

Catching all Exceptions

```
try {
    float answer = quotient( numerator, denominator );
    cout << "\n";
    cout << numerator << "/"
         << denominator << " = " << answer;
}
catch (...) { //error handler
    cout << "ERROR: ERROR_IN_PROGRAM\n";
    return 1; //terminate because of error
}
```



One fewer
destructors
ran



```
Enter numerator and denominator: 3 0
construct: DivByZeroObject
delete: DivByZeroObject
ERROR: ERROR_IN_PROGRAM
delete: DivByZeroObject
```

...Catching an Exception

- **Exception handlers are searched in order, the first one that matches is executed**
- **Exception object can match different handlers**
- **Handler type matches thrown object type if**
 - the are the same type
 - the handle type is a public base class of the thrown object type
 - handler type is a pointer and the thrown object is a pointer type convertible to the handler type by allowable pointer conversion (derived class pointer is convertible to base class pointer)
 - catch handler is `catch(...)`

Notes

- Unlike a `switch` statement you don't need a `break` in the catch block to “jump over” remaining handlers (each handler has its own scope)
- By the time in the handler runs, the stack associated with the try block has already been unwound (popped)
- Implication: catch handlers cannot access object defined in the try block
- Implication: execution does not continue from where the exception was thrown

Exceptions Handlers that Cause Exceptions

- **There is a greater than zero probability that exception handlers can cause exceptions - because they consist of have greater than zero lines of code**


Exception Handler Exception

```
class ExceptionHandlerError {
public:
    ExceptionHandlerError() :
        message("Exception Handler Error\n")
        {cout << "construct: ExceptHandErr\n"; }
    ~ExceptionHandlerError()
        {cout << "delete: ExceptHandErr\n"; }
    void printMessage() const {cout << message;}
private:
    const char* message;
};


void causeExceptionHandlerError() {
    throw ExceptionHandlerError();
    return;
}
```



Exception Handler Exception

```
try {
    float answer = quotient(numerator, denominator);
    cout << "\n";
    cout << numerator << "/"
         << denominator << " = " << answer;
}
catch (DivideByZeroError error) { //error handler
    cout << "ERROR: ";
    error.printMessage();
    causeExceptionError(); 
    cout << "\n";
    return 1; //terminate because of error
}
catch (ExceptionHandlerError error) { //error handler
    cout << "ERROR: ";
    error.printMessage();
    cout << "\n";
    return 1; //terminate because of error
}
```

Exception Handler Exception

```
try {
    float answer = quotient(numerator, denominator);
    cout << "\n";
    cout << numerator << "/"
         << denominator << " = " << answer;
}
catch (DivideByZeroError error) { //error handler
    cout << "ERROR: ";
    error.printMessage();
    causeExceptionError(); 
    cout << "\n";
    return 1; //terminate because of error
}
catch (ExceptionHandlerError error) { //error handler
    cout << "ERROR: ";
    error.printMessage();
    cout << "\n";
    return 1; //terminate because of error
}
```

...Exception Handler Exception

```
try {
    float answer;
    cout << "\nEnter numerator and denominator: ";
    cout << num;
    <<
}
catch (DivideByZeroObject error) {
    cout << "ERROR: Divide by Zero\n";
    error.printStackTrace();
    causeExceptionHandlerError();
    cout << "\n";
    return 1; //terminate because of error
}
catch (ExceptionHandlerError error) {
    cout << "ERROR: ";
    error.printStackTrace();
    cout << "\n";
    return 1; //terminate because of error
}
```

```
Enter numerator and denominator: 3 0
construct: DivByZeroObject
delete: DivByZeroObject
ERROR: Divide by Zero
construct: ExceptHandErr
delete: ExceptHandErr
delete: DivByZeroObject
delete: DivByZeroObject
```



...Exception Handler Exception

```
try {
    try {
        float answer = quotient(numerator, denominator);
        cout << "\n";
        cout << numerator << "/"
            << denominator << " = " << answer;
    }
    catch (DivideByZeroError error) { //error handler
        cout << "ERROR: ";
        error.printStackTrace();
        causeExceptionError(); //cause exception error
        cout << "\n";
        return 1; //terminate because of error
    }
}
catch (ExceptionHandlerError error) { //error handler
    cout << "ERROR: ";
    error.printStackTrace();
    cout << "\n";
    return 1; //terminate because of error
}
}
```

...Exception Handler Exception

```
try {
    try {
        float an
        cout <<
        cout <<
            <<
        }
        catch (Div
            cout <<
            error.p
            causeEx
            cout <<
            return
        }
    }
    catch (ExceptionHandlerError error) { //error handler
        cout << "ERROR: ";
        error.printStackTrace();
        cout << "\n";
        return 1; //terminate because of error
    }
}
```

```
Enter numerator and denominator: 3 0
construct: DivByZeroObject
delete: DivByZeroObject
ERROR: Divide by Zero
construct: ExceptHandErr
delete: ExceptHandErr
delete: DivByZeroObject
delete: DivByZeroObject
ERROR: Exception Handler Error
delete: ExceptHandErr
delete: ExceptHandErr
```

...Exception Handler Exception

- An exception caused by an exception handler cannot be caught by another handler associated with the same `try` block
- It must be caught by a handler of an more outer `try` block

Exception Handlers Can ...

- **Examine the error and call terminate**
- **Re-throw the error for another handler**
- **Convert the error and throw a different exception**
- **Perform recovery operations and continue executing statements after the catch blocks**
- **Remove the cause of error and retry by invoking the same function that caused the error (this will not cause an infinite loop)**
- **Can simply return a status indicator to their invoking environment**

Exception Handlers Continuation

```
try {  
    ...  
    throw exception  
    ...  
}  
catch {  
    ...  
}  
catch {  
    ...  
}  
statement;
```

...

- If the try block does not throw an exception execution continues after the last catch block of the try block (at `statement ;`)
- An exception handler cannot return control back to the `try` block that threw the exception

Re-throwing Exceptions

```
try {
    try {
        float answer = quotient(numerator, denominator);
        cout << "\n";
        cout << numerator << "/"
             << denominator << " = " << answer;
    }
    catch (DivideByZeroError error) { //error handler
        //clean up local resources
        cout << "ERROR: ";
        error.printStackTrace();
        cout << "\n...Clean up local resources\n";
        return 1; //terminate because of error
    }
}
catch (DivideByZeroError error) { //error handler
    //clean up more globally
    cout << "ERROR: ";
    error.printStackTrace();
    cout << "\n...General Cleanup after math error\n";
    return 1; //terminate because of error
}
```



... Re-throwing Exceptions

```
try {
    try {
        float answer = quotient(numerator, denominator);
        cout << "\n";
        cout << numerator << "/"
             << denominator << " = " << answer;
    }
    catch (DivideByZeroError error) { //error handler
        //clean up local resources
        cout << "ERROR: ";
        error.printMessage();
        cout << "\n...Clean up local resources\n";
        return 1; //terminate because of error
    }
}
catch (DivideByZeroError error) { //error handler
    //clean up more globally
    cout << "ERROR: ";
    error.printMessage();
    cout << "\n...Clean up local resources\n";
    return 1; //terminate because of error
}
```

```
Enter numerator and denominator: 3 0
ERROR: Divide by Zero
...Clean up local resources
```


Re-throwing Exceptions with throw keyword

```
try {
    try {
        float answer = quotient(numerator, denominator);
        cout << "\n";
        cout << numerator << "/"
             << denominator << " = " << answer;
    }
    catch (DivideByZeroError error) { //error handler
        //clean up local resources
        cout << "ERROR: ";
        error.printMessage();
        cout << "\n...Clean up local resources\n";
        throw;
        return 1; //terminate because of error
    }
}
catch (DivideByZeroError error) { //error handler
    //clean up more globally
    cout << "ERROR: ";
    error.printMessage();
    cout << "\n...General Cleanup after math error\n";
    return 1; //terminate because of error
}
```



... Re-throwing Exceptions

```
try {
    try {
        float answer = quotient(numerator, denominator);
        cout << "\n";
        cout << numerator << "/"
             << denominator << " = " << answer;
    }
    catch (DivideByZeroError error) { //error handler
        //clean up local resources
        cout << "ERROR: ";
        error.printMessage();
        cout << "\n...Clean up local resources\n";
        throw;
        return 1; //terminate because of error
    }
}
catch (DivideByZeroError error) {
    //clean up m
    cout << "ERR
    error.printM
    cout << "\n.
    return 1; //
}
```



```
Enter numerator and denominator: 3 0
ERROR: Divide by Zero
...Clean up local resources
ERROR: Divide by Zero
...General Cleanup after math error
```

Throwing ints, doubles, etc

```
int factorial(int n) {  
    if (n < 0) throw int(n);  
    if (n == 0) return 1;  
    return n*factorial(n-1);  
}  
int factorial(double n) {  
    throw double(n);  
    return 1;  
}
```



... Throwing ints, doubles, etc

```
main(){
  int n = 5;
  double x = 5.0;
  try {
    cout << n << " factorial is " << factorial(n) << "\n";
  }
  catch (int error) { //error handler
    cout << "\nERROR INTEGER = " << error << "\n";
    if (error < 0)
      cout << "...INT MUST BE GREATER THAN ZERO\n";
    return 1; //terminate because of error
  }
  catch (double error) { //error handler
    cout << "\nERROR Double = " << error << "\n";
    cout << "\n...ARGUMENT CANNOT BE TYPE double\n";
    return 1; //terminate because of error
  }

  return 0;
}
```

5 factorial is 120

... Throwing ints, doubles, etc

```
main(){
  int n = -5;
  double x = 5.0;
  try {
    cout << n << " factorial is " << factorial(n) << "\n";
  }
  catch (int error) { //error handler
    cout << "\nERROR INTEGER = " << error << "\n";
    if (error < 0)
      cout << "...INT MUST BE GREATER THAN ZERO\n";
    return 1; //terminate because of error
  }
  catch (double error) { //error handler
    cout << "\nERROR Double = " << error << "\n";
    cout << "\n...ARGUMENT CANNOT BE TYPE double\n";
    return 1; //terminate because of error
  }

  return 0;
}
```

```
ERROR INTEGER = -5
...INT MUST BE GREATER THAN ZERO
```

... Throwing ints, doubles, etc

```
main(){
    int n = -5;
    double x = 5.0;
    try {
        cout << x << " factorial is " << factorial(x) << "\n";
    }
    catch (int error) { //error handler
        cout << "\nERROR INTEGER = " << error << "\n";
        if (error < 0)
            cout << "...INT MUST BE GREATER THAN ZERO\n";
        return 1; //terminate because of error
    }
    catch (double error) { //error handler
        cout << "\nERROR Double = " << error << "\n";
        cout << "\n...ARGUMENT CANNOT BE TYPE double\n";
        return 1; //terminate because of error
    }

    return 0;
}
```



```
ERROR Double = 5
```

```
...ARGUMENT CANNOT BE TYPE double
```

Throwing from Conditional Expressions

```
main(){
    int n = 5;
    double x = 5.0;
    try {
        n < 0 ? throw int(n) : cout << factorial(n);
    }
    catch (int error) { //error handler
        cout << "\nERROR INTEGER = " << error << "\n";
        if (error < 0)
            cout << "...INT MUST BE GREATER THAN ZERO\n";
        return 1; //terminate because of error
    }
    catch (double error) { //error handler
        cout << "\nERROR Double = " << error << "\n";
        cout << "\n...ARGUMENT CANNOT BE TYPE double\n";
        return 1; //terminate because of error
    }

    return 0; //terminate normally
}
```



120

...Throwing from Conditional Expressions

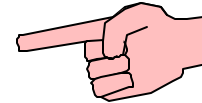
```
main(){
  int n = -5;
  double x = 5.0;
  try {
    n < 0 ? throw int(n) : cout << factorial(n);
  }
  catch (int error) { //error handler
    cout << "\nERROR INTEGER = " << error << "\n";
    if (error < 0)
      cout << "...INT MUST BE GREATER THAN ZERO\n";
    return 1; //terminate because of error
  }
  catch (double error) { //error handler
    cout << "\nERROR Double = " << error << "\n";
    cout << "\n...ARGUMENT CANNOT BE TYPE double\n";
    return 1; //terminate because of error
  }

  return 0;
}
```

```
ERROR INTEGER = -5
...INT MUST BE GREATER THAN ZERO
```

...Throwing from Conditional Expressions

```
main(){  
    int n = -5;  
    double x = 5.0;  
    try {  
        n < 0 ? throw int(n) : throw double(x);  
    }  
    catch ...
```



```
ERROR INTEGER = -5  
...INT MUST BE GREATER THAN ZERO
```

```
main(){  
    int n = 5;  
    double x = 5.0;  
    try {  
        n < 0 ? throw int(n) : throw double(x);  
    }  
    catch ...
```





```
ERROR Double = 5  
...ARGUMENT CANNOT BE TYPE double
```

Throwing a Conditional Expression

It is possible to throw a conditional expression. But be careful because promotion rules may cause the value returned by the conditional expression to be of a different type than you may expect.

For example, when throwing an `int` or `double` from the same conditional expression, the conditional expression will convert the `int` to a `double`

...Continuing After an Exception

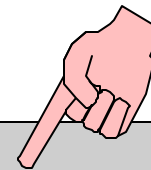
```
main(){
    int n = -5;
    double x = 5.0;
    try {
        cout << n << " factorial is " << factorial(n) << "\n";
    }
    catch (int error) { //error handler
        cout << "\nERROR INTEGER = " << error << "\n";
        if (error < 0)
            cout << "...INT MUST BE GREATER THAN ZERO\n";
        return 1; //terminate because of an error 
    }
    catch (double error) { //error handler
        cout << "\nERROR Double = " << error << "\n";
        cout << "\n...ARGUMENT CANNOT BE TYPE double\n";
        return 1; //terminate because of error
    }
    cout << "\n...processing continues...\n"; 
    return 0; //terminate normally
}
```


...Continuing After an Exception

```
main(){
    int n = -5;
    double x = 5.0;
    try {
        cout << n << " factorial is " << factorial(n) << "\n";
    }
    catch (int error) { //error handler
        cout << "\nERROR INTEGER = " << error << "\n";
        if (error < 0)
            cout << "...INT MUST BE GREATER THAN ZERO\n";
        return 0; //return indicating normal termination
    }
    catch (double error) { //error handler
        cout << "\nERROR Double = " << error << "\n";
        cout << "\n...ARGUMENT CANNOT BE TYPE double\n";
        return 1; //terminate because of error
    }
    cout << "\n...processing continues...\n";
    return 0; //terminate normally
}
```



Same Result



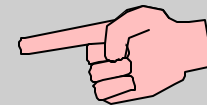
```
ERROR INTEGER = -5
...INT MUST BE GREATER THAN ZERO
```

...Continuing After an Exception (No Return Specified)

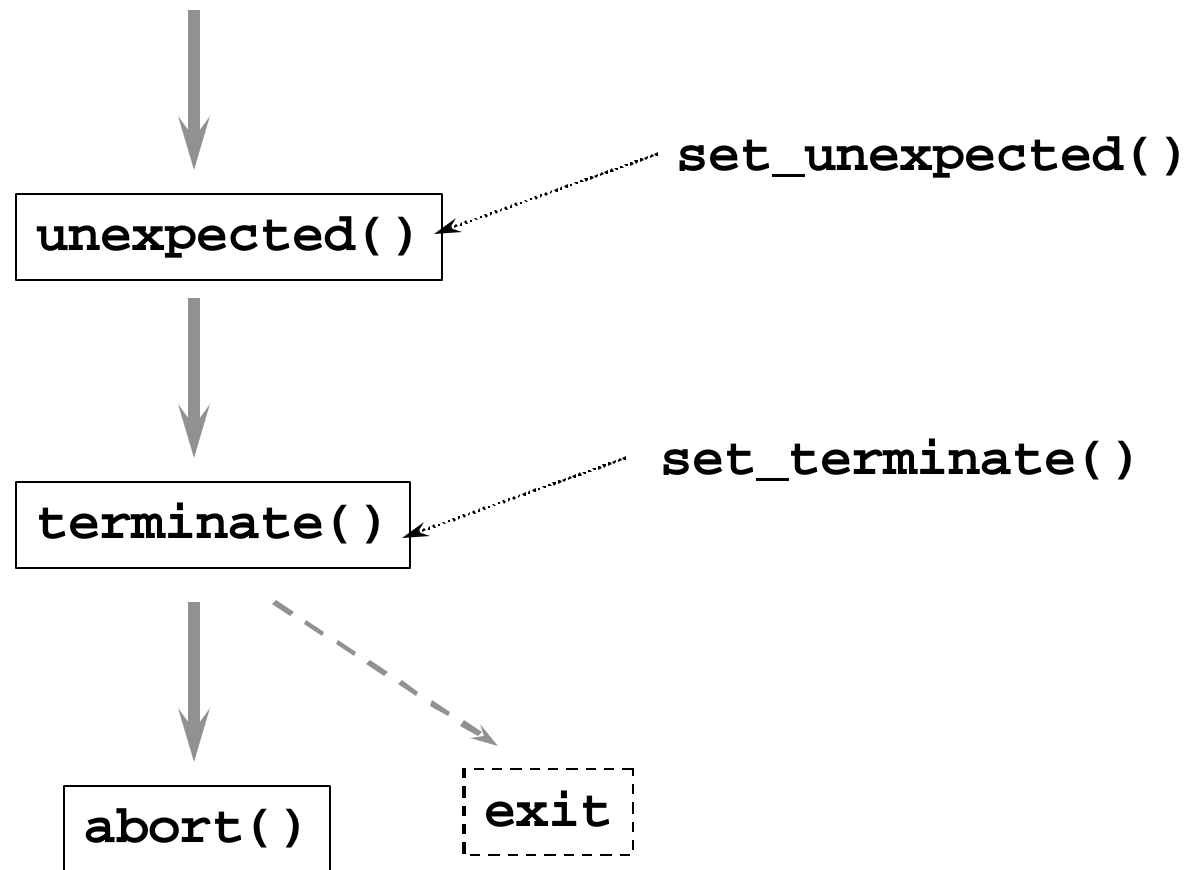
```
main(){
    int n = -5;
    double x = 5.0;
    try {
        cout << n << " factorial is " << factorial(n) << "\n";
    }
    catch (int error) { //error handler
        cout << "\nERROR INTEGER = " << error << "\n";
        if (error < 0)
            cout << "...INT MUST BE GREATER THAN ZERO\n";
    }
    catch (double error) { //error handler
        cout << "\nERROR Double = " << error << "\n";
        cout << "\n...ARGUMENT CANNOT BE TYPE double\n";
        return 1; //terminate because of error
    }
    cout << "\n...processing continues...";
    return 0; //terminate
}
```



```
ERROR INTEGER = -5
...INT MUST BE GREATER THAN ZERO
...processing continues...
```



Processing Unexpected Exceptions



Exception Specification

- The definition of a function can specify which exceptions the function throws -sort of
- ```
int g(float h) throw(a, b, c) {
 //...
}
```
- Function `g( )` throws only exception objects of type `a`, `b`, or `c` -or a type derived from them
- If the function throws any other type of exception, it will be processed as an unexpected exception
- So it actually can throw any exception?

## Exception Specification

- The definition of a function can specify that it throws no exceptions -sort of
- ```
int g(float h) throw() {  
    //...  
}
```
- Function `g()` does not throw any exceptions
- If the function throws any exception, it will be processed as an unexpected exception
- So it actually can throw any exception?

Exception Specification

- The definition of a function can specify that it throws any exception
- ```
int g(float h) {
 //...
}
```
- Function `g()` can throw any exception
- **NOTE:** The compiler won't care what type of exceptions are thrown, this is handled at run-time

## Exceptions and Constructors

- **What happens when a constructor encounters an error?**
- **Example: How should a constructor respond if in an attempt to allocate memory for a member object, `new` returns 0 indicating no memory was available**
- **Problem: Constructors don't have return values**
- **Solutions:**
  - return the damaged object anyway
  - set some variable outside the constructor
  - throw an exception (and exception object)

## ...Exceptions and Constructors

- **Throwing an exception passes to the “outside” world information about what went wrong in a constructor**
- **But to catch an exception, the exception handler must have access to a copy constructor -to copy the exception object (default member-wise copy is OK)**
- **The are potentially problems throwing the ill-constructed object**

## ...Exceptions and Constructors

### Notes

- **Exceptions thrown in constructors cause destructors to be invoked for any object built prior to throwing the exception**
- **Destructors are called for any stack objects built in the try block before the exception was thrown**
- **What happens if the destructor throws an exception?**

**“If the constructor of an object has run completely its destructor will be run”**

## ...Exceptions and Constructors

### Further Claims

- **If an object has member objects, then destructors will be executed for the member objects that have been completely constructed prior to the occurrence of the exception**
- **If an array of objects has been partially constructed, only destructors of the constructed array elements will be called**

## ...Exceptions and Constructors

### **WARNING**

- **The interactions between constructors, allocated resources, and exception handling can be tricky**