C++ Object-Oriented Prog.
Unit 5: Object-Oriented Design

CHAPTER 18: EXCEPTION HANDLING AND FILE PROCESSING

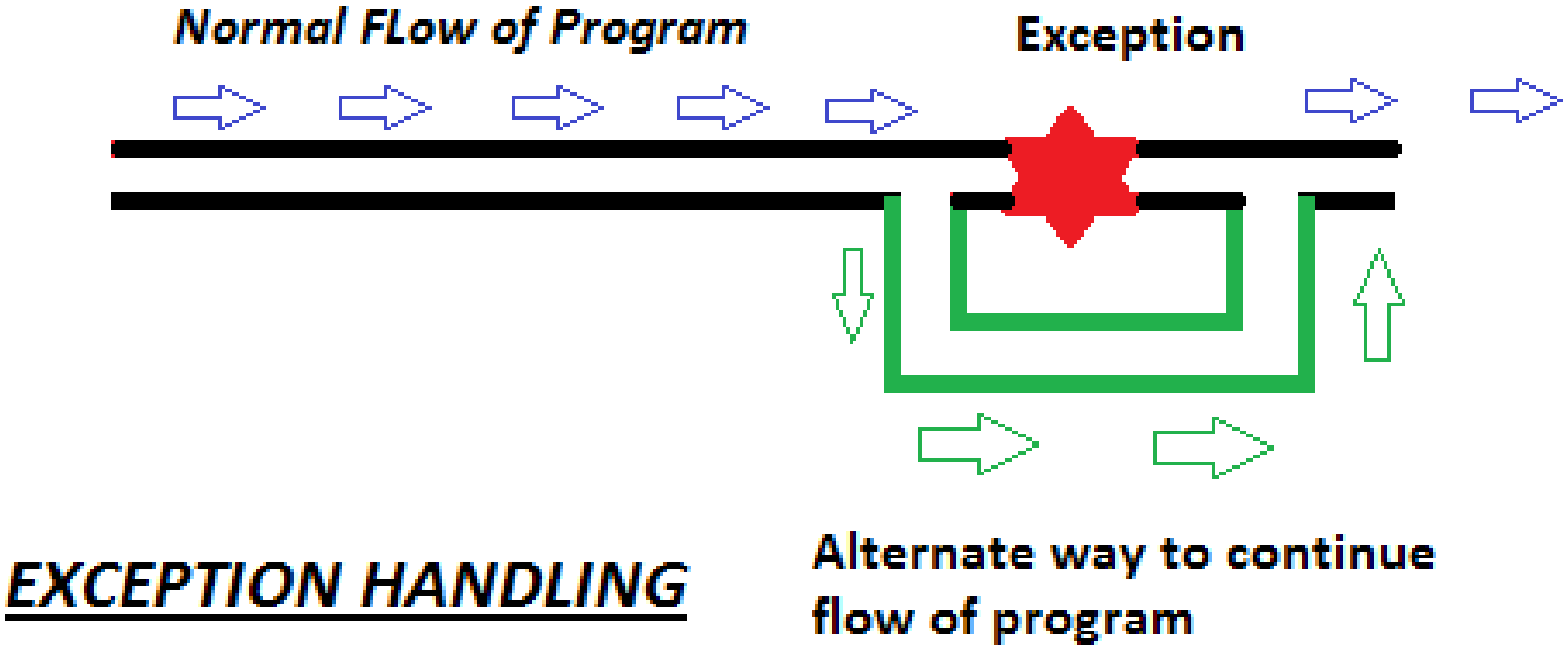
DR. ERIC CHOU IEEE SENIOR MEMBER

Topics

1. Overview of Exception Handling
2. Basic Exception Type: Data, Message, Exception Object
3. Exception Handling Flow
4. Re-throwing Exception
5. Processing Un-expected Exceptions
6. Standard Exception Classes

LECTURE 1

Overview of Exception Handling



Murphy’s Law

ANYTHING THAT CAN GO WRONG WILL GO WRONG

Exceptions

- Allow you to deal with the things that go wrong:
- Indicate that something unexpected has occurred or been detected
- Allow program to deal with the problem in a controlled manner
- Can be as simple or complex as program design requires

Exceptions -- Terminology

Exception: object or value that signals an error

Throw an exception: send a signal that an error has occurred

Catch/Handle an exception: process the exception; interpret the signal

Exceptions – Keywords

`throw` – followed by an argument, is used to throw an exception

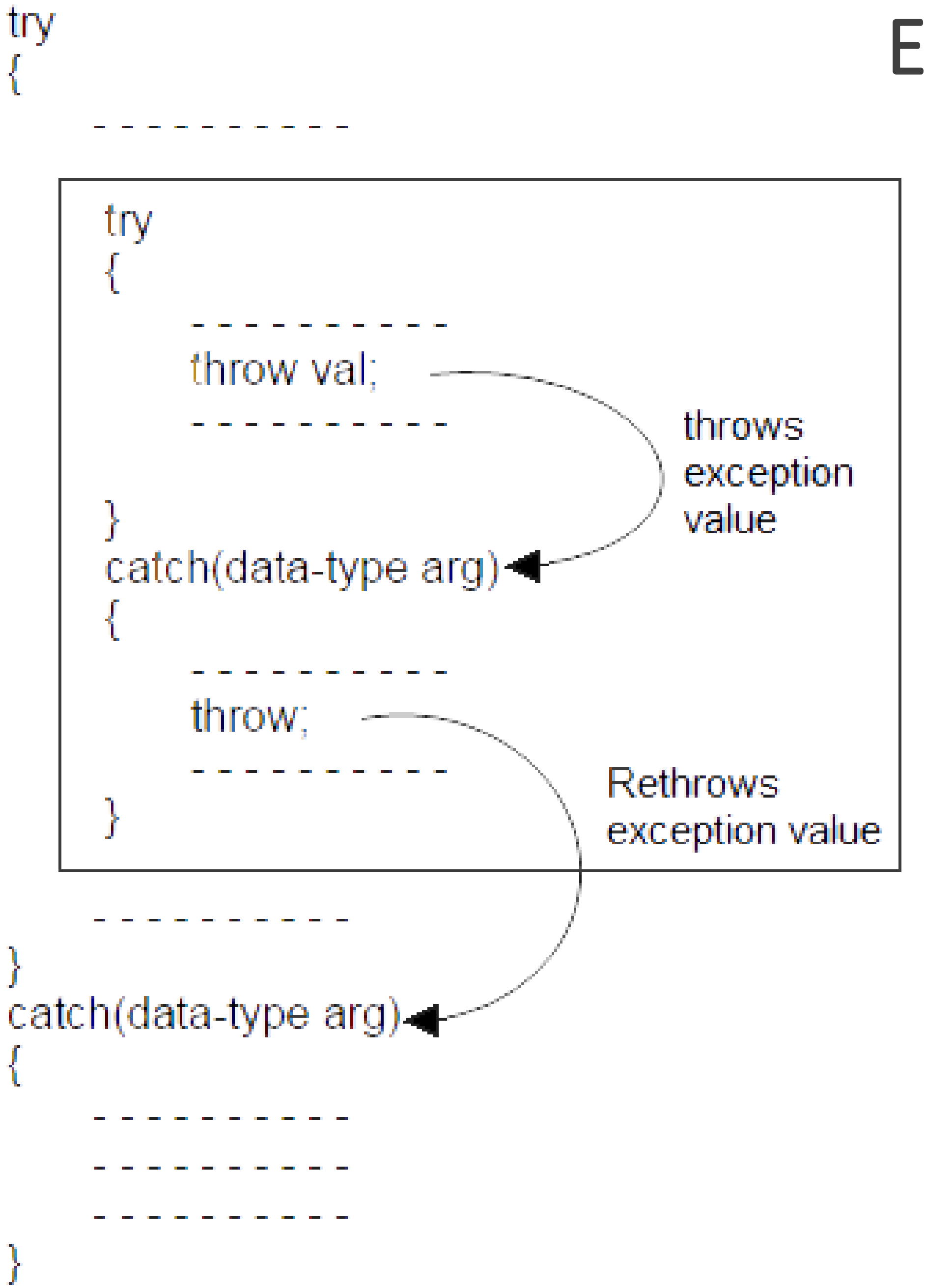
`try` – followed by a block `{ }`, is used to invoke code that throws an exception

`catch` – followed by a block `{ }`, is used to detect and process exceptions thrown in preceding `try` block. Takes a parameter that matches the type thrown.

Exceptions – Flow of Control

1. A function that throws an **exception** is called from within a try block
2. If the function throws an exception, the function terminates and the try block is immediately exited. A catch block to process the exception is searched for in the source code immediately following the try block.
3. If a catch block is found that matches the exception thrown, it is executed. If no catch block that matches the exception is found, the program terminates.

Exceptions – Flow of Control



```
try
{
      - - - - - - - - - - -

   try
   {
         - - - - - - - - - - -
         throw val;
         - - - - - - - - - - -

   }
   catch(data-type arg)
   {
         - - - - - - - - - - -
         throw;
         - - - - - - - - - - -
   }

      - - - - - - - - - - -
}
catch(data-type arg)
{
      - - - - - - - - - - -
      - - - - - - - - - - -
      - - - - - - - - - - -
}
```

Exceptions – Example(1)

```
// function that throws an exception
int totalDays(int days, int weeks)
{
     if ((days < 0) || (days > 7))
       throw "invalid number of days";
// the argument to throw is the
// character string
   else
       return (7 * weeks + days);
}
```

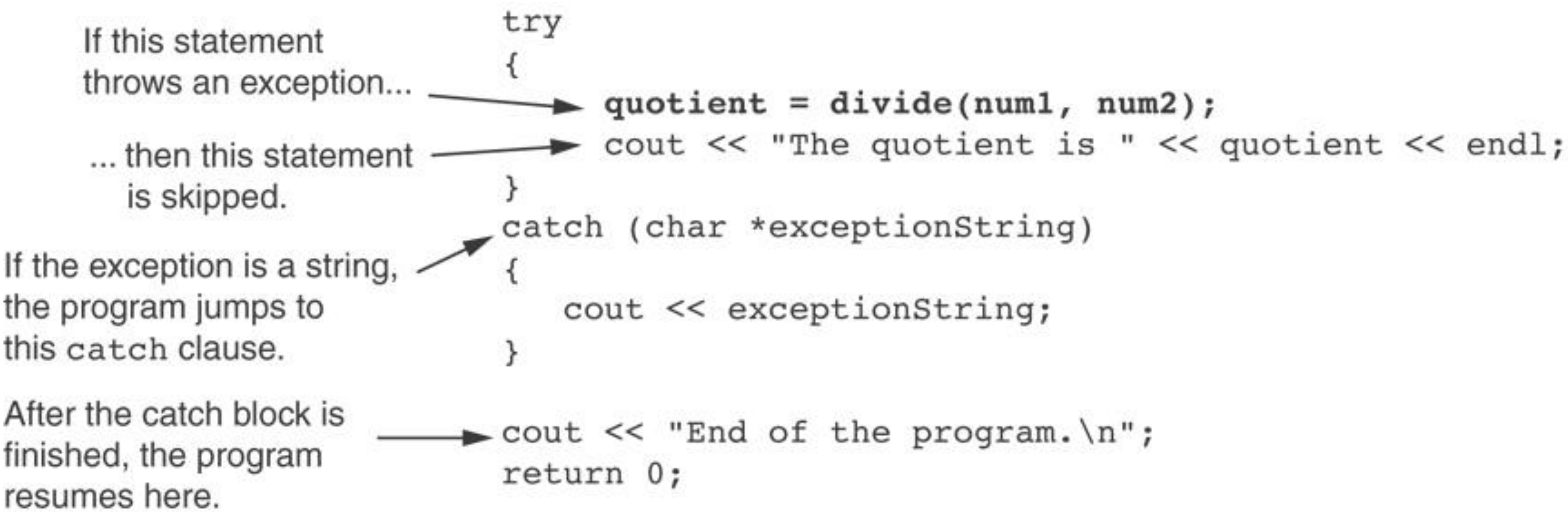
Exceptions – Example (2)

```
try // block that calls function
 {
       totDays = totalDays(days, weeks);
     cout << "Total days: " << days;
  }
  catch (char *msg) // interpret
                    //  exception
  {
     cout << "Error: " << msg;
  }
```

Exceptions – How It Works

1. `try` block is entered. `totalDays` function is called
2. If first parameter is between 0 and 7, total number of days is returned and `catch` block is skipped over (no exception thrown)
3. If exception is thrown, function and `try` block are exited, `catch` blocks are scanned for the first one that matches the data type of the thrown exception. `catch` block executes

Exceptions – How It Works



```
try
{
    quotient = divide(num1, num2);
    cout << "The quotient is " << quotient << endl;
}
catch (char *exceptionString)
{
   cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

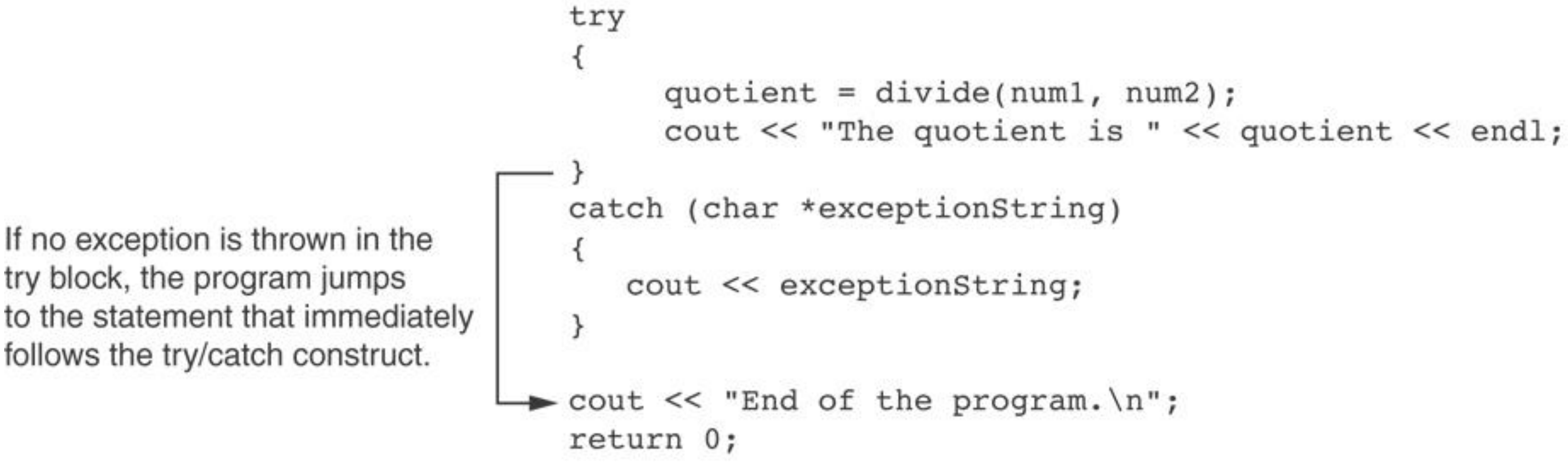
If this statement throws an exception...

... then this statement is skipped.

If the exception is a string, the program jumps to this catch clause.

After the catch block is finished, the program resumes here.

What if no Exception is Thrown?



If no exception is thrown in the try block, the program jumps to the statement that immediately follows the try/catch construct.

```
try
{
     quotient = divide(num1, num2);
     cout << "The quotient is " << quotient << endl;
}
catch (char *exceptionString)
{
   cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

Exceptions -- Notes

Predefined functions such as `new` may throw exceptions

The value that is thrown does not need to be used in `catch` block.

- in this case, no name is needed in catch parameter definition
- `catch` block parameter definition *does* need the type of exception being caught

Exception Not Caught?

An exception will not be caught if

- it is thrown from outside of a `try` block
- there is no `catch` block that matches the data type of the thrown exception

If an exception is not caught, the program will terminate

LECTURE 2

Basic exception types

Exception Type

- Number: int, double, and etc.
- Text Message: string, char *
- Exception Object: exception class
- All other types: …

Exceptions and Objects

An **exception class** can be defined in a class and thrown as an exception by a member function

An exception class may have:

- no members: used only to signal an error
- members: pass error data to `catch` block

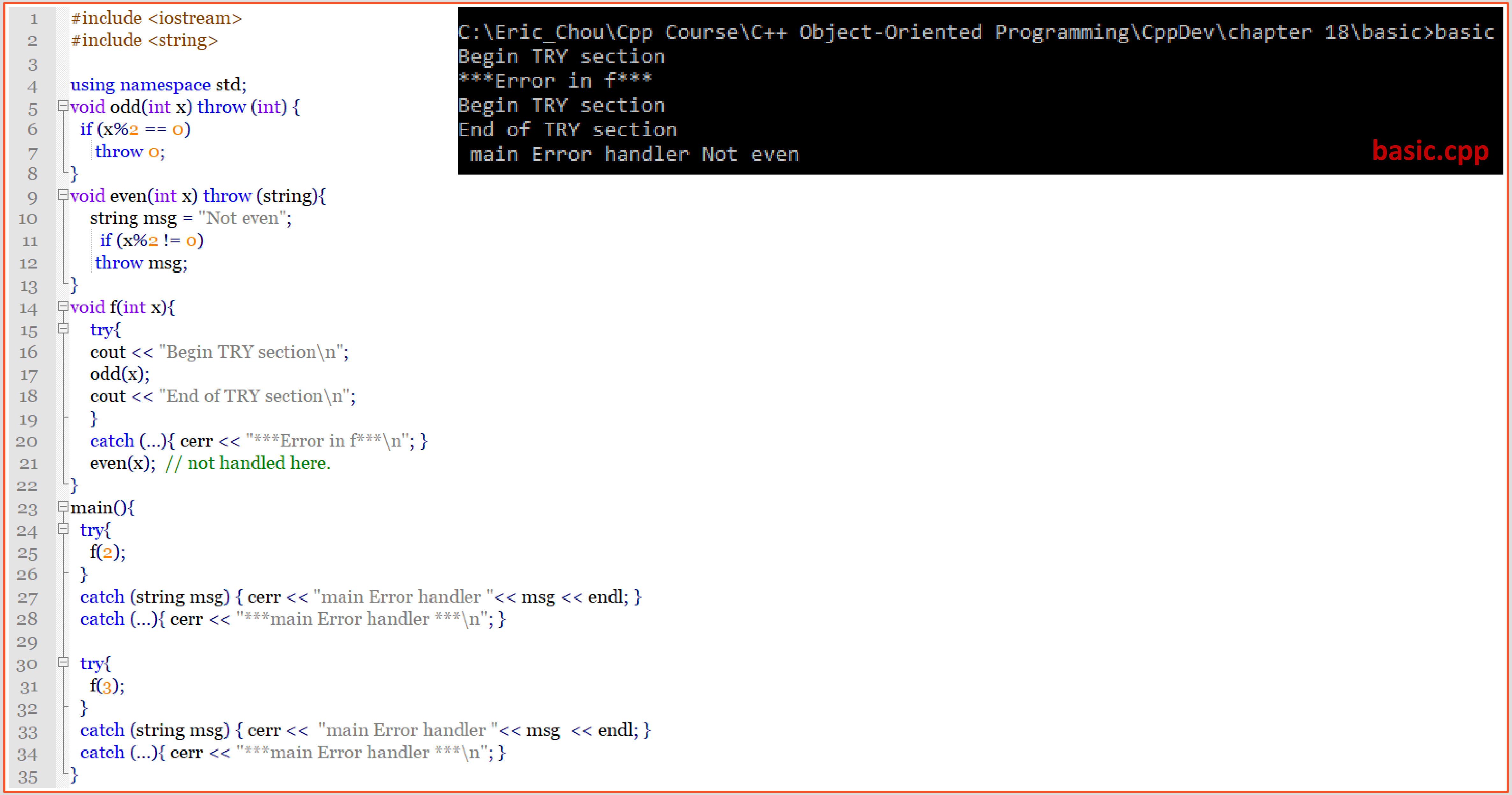
A class can have more than one exception class

What Happens After `catch` Block?

- Once an exception is thrown, the program cannot return to throw point. The function executing `throw` terminates (does not return), other calling functions in `try` block terminate, resulting in **unwinding the stack**
- If objects were created in the `try` block and an exception is thrown, they are destroyed.

Demo Program: basic.cpp

Go Notepad++!!!



```cpp
#include <iostream>
#include <string>

using namespace std;
void odd(int x) throw (int) {
  if (x%2 == 0)
     throw 0;
}
void even(int x) throw (string){
    string msg = "Not even";
      if (x%2 != 0)
     throw msg;
}
void f(int x){
    try{
    cout << "Begin TRY section\n";
    odd(x);
    cout << "End of TRY section\n";
    }
    catch (...){ cerr << "***Error in f***\n"; }
    even(x);  // not handled here.
}
main(){
  try{
    f(2);
  }
  catch (string msg) { cerr << "main Error handler "<< msg << endl; }
  catch (...){ cerr << "***main Error handler ***\n"; }

  try{
    f(3);
  }
  catch (string msg) { cerr <<  "main Error handler "<< msg  << endl; }
  catch (...){ cerr << "***main Error handler ***\n"; }
}
```

```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 18\basic>basic
Begin TRY section
***Error in f***
Begin TRY section
End of TRY section
 main Error handler Not even
```

basic.cpp

Nested `try` Blocks

`try/catch` blocks can occur within an enclosing `try` block

Exceptions caught at an inner level can be passed up to a `catch` block at an outer level:

```
catch ( )
{
    ...
      throw;  // pass exception up
}                // to next level
```

LECTURE 3

Introduction to Exception Handling

Introduction

Exceptions

- Indicates problem occurred in program
- Not common
  - An "exception" to a program that usually works

Exception Handling

- Resolve exceptions
- Program may be able to continue
  - Controlled termination
- Write fault-tolerant programs

Exception-Handling Overview

Consider pseudocode

```
Perform a task
If the preceding task did not execute correctly
    Perform error processing
Perform next task
If the preceding task did not execute correctly
    Perform error processing
```

Mixing logic and error handling

- Can make program difficult to read/debug
- Exception handling removes error correction from "main line" of program

Exception-Handling Overview

Exception handling

- For synchronous errors (divide by zero, null pointer)
  - Cannot handle asynchronous errors (independent of program)
  - Disk I/O, mouse, keyboard, network messages
- Easy to handle errors

Terminology

- Function that has error *throws an exception*
- *Exception handler* (if it exists) can deal with problem
  - *Catches* and *handles* exception
- If no exception handler, *uncaught* exception
  - Could terminate program

Exception-Handling Overview

C++ code

```
try {
   code that may raise exception
}
catch (exceptionType){
   code to handle exception
}
```

- `try` block encloses code that may raise exception
- One or more `catch` blocks follow
  - Catch and handle exception, if appropriate
  - Take parameter; if named, can access exception object

Exception-Handling Overview

Throw point

- Location in **`try`** block where exception occurred
- If exception handled
  - Program skips remainder of **`try`** block
  - Resumes after **`catch`** blocks
- If not handled
  - Function terminates
  - Looks for enclosing **`catch`** block (stack unwinding, 13.8)

If no exception

- Program skips **`catch`** blocks

Other Error-Handling Techniques

Ignore exception

- Typical for personal (not commercial) software
- Program may fail

Abort program

- Usually appropriate
- Not appropriate for mission-critical software

Set error indicators

- Unfortunately, may not test for these when necessary

Test for error condition

- Call exit (`<cstdlib>`) and pass error code

Other Error-Handling Techniques

`setjump` and `longjump`

- **`<csetjmp>`**
- Jump from deeply nested function to call error handler
- Can be dangerous

Dedicated error handling

- **`new`** can have a special handler

Simple Exception-Handling Example: Divide by Zero

Keyword `throw`

- Throws an exception
  - Use when error occurs
- Can throw almost anything (exception object, integer, etc.)
  - `throw myObject;`
  - `throw 5;`

Exception objects

- Base class `runtime_error` (`<stdexcept>`)
- Constructor can take a string (to describe exception)
- Member function `what()` returns that string

Simple Exception-Handling Example: Divide by Zero

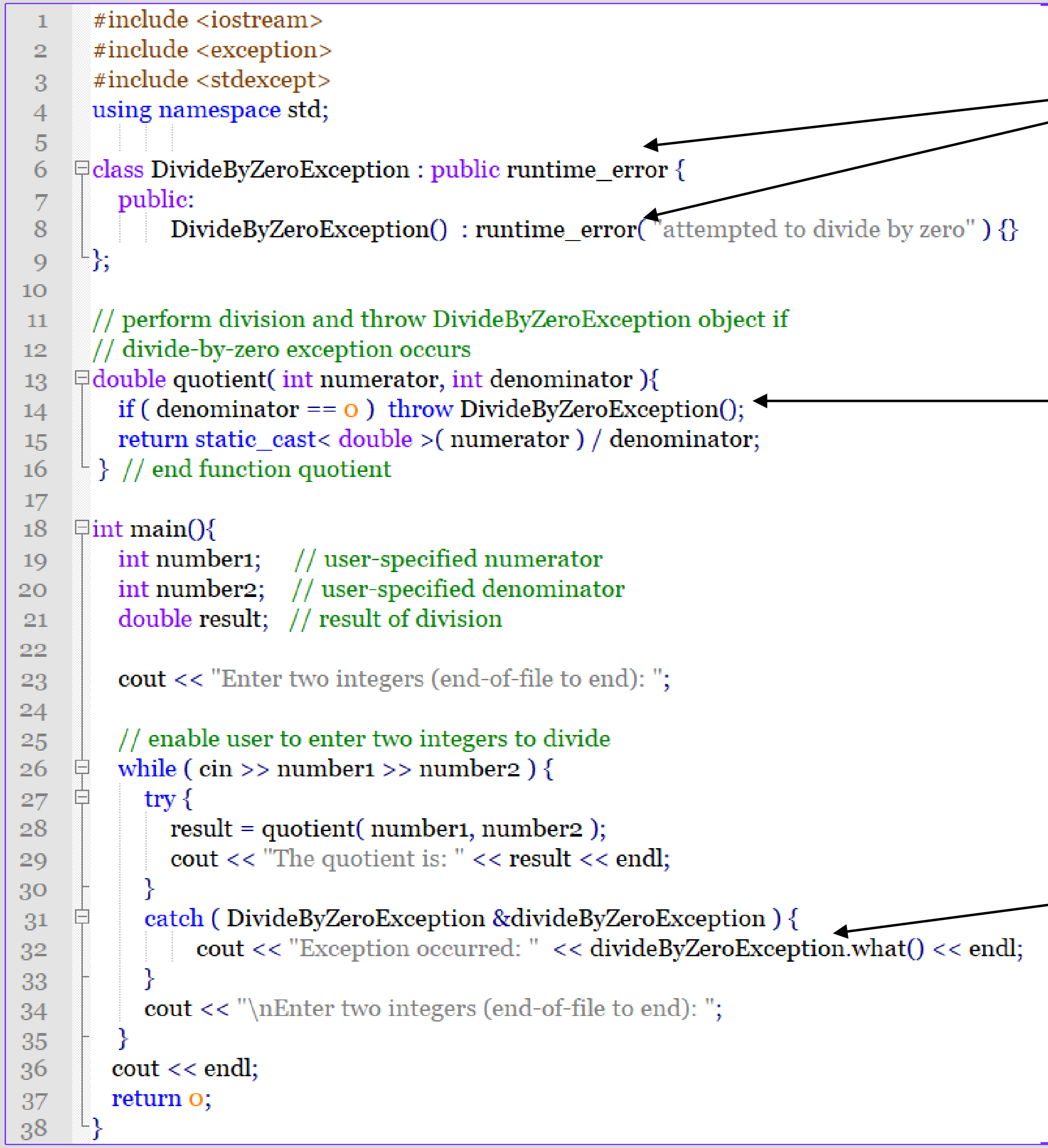
Upcoming example

- Handle divide-by-zero errors
- Define new exception class
  - **`DivideByZeroException`**
  - Inherit from **`runtime_error`**
- In division function
  - Test denominator
  - If zero, throw exception (**`throw object`**)
- In **`try`** block
  - Attempt to divide
  - Have enclosing **`catch`** block
    - Catch **`DivideByZeroException`** objects

Demo Program: custom_exception.cpp

Go Notepad++!!!

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z
```



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

class DivideByZeroException : public runtime_error {
   public:
        DivideByZeroException()  : runtime_error( "attempted to divide by zero" ) {}
};

// perform division and throw DivideByZeroException object if
// divide-by-zero exception occurs
double quotient( int numerator, int denominator ){
   if ( denominator == 0 )  throw DivideByZeroException();
   return static_cast< double >( numerator ) / denominator;
 }  // end function quotient

int main(){
   int number1;    // user-specified numerator
   int number2;   // user-specified denominator
   double result;   // result of division

   cout << "Enter two integers (end-of-file to end): ";

   // enable user to enter two integers to divide
   while ( cin >> number1 >> number2 ) {
      try {
         result = quotient( number1, number2 );
         cout << "The quotient is: " << result << endl;
      }
      catch ( DivideByZeroException &divideByZeroException ) {
         cout << "Exception occurred: "  << divideByZeroException.what() << endl;
      }
      cout << "\nEnter two integers (end-of-file to end): ";
   }
  cout << endl;
  return 0;
}
```

Define new exception class (inherit from **runtime_error**).
Pass a descriptive message to constructor

If the denominator is zero, **`throw`** a **`DivideByZeroException`** object.

Notice the structure of the **`try`** and **`catch`** blocks. The **`catch`** block can catch **`DivideByZeroException`** objects, and print an error message. If no exception occurs, the **`catch`** block is skipped.

Member function **`what`** returns the string describing the exception.

LECTURE 4

Re-throwing an Exception

Rethrowing an Exception

Rethrowing exceptions

- Use when exception handler cannot process exception
  - Can still rethrow if handler did some processing
- Can rethrow exception to another handler
  - Goes to next enclosing **`try`** block
  - Corresponding **`catch`** blocks try to handle

To rethrow

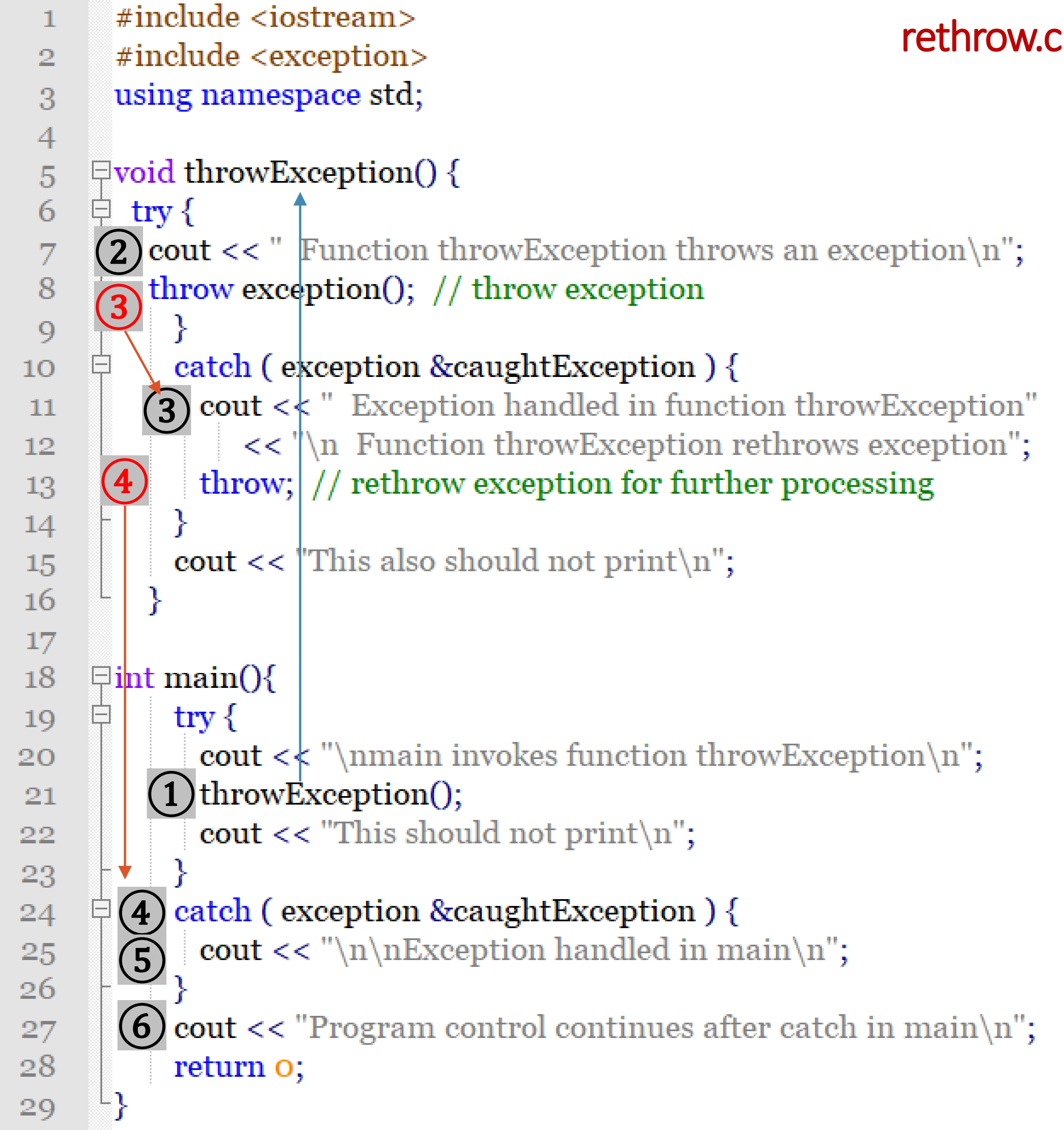
- Use statement "**`throw;`**"
  - No arguments
  - Terminates function

Demo Program: rethrow.cpp

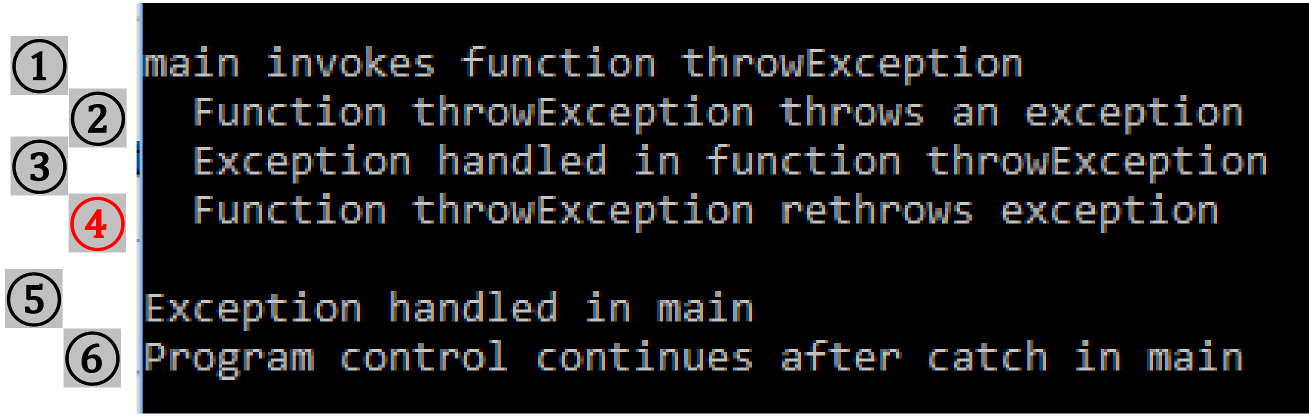
Go Notepad++!!!

rethrow.cpp

Re-throwing:
A the exception handling section, you throw the same exception again.



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

void throwException() {
 try {
  cout << "  Function throwException throws an exception\n";
  throw exception();  // throw exception
  }
   catch ( exception &caughtException ) {
    cout << "  Exception handled in function throwException"
       << "\n  Function throwException rethrows exception";
    throw;  // rethrow exception for further processing
   }
   cout << "This also should not print\n";
 }

int main(){
   try {
    cout << "\nmain invokes function throwException\n";
    throwException();
    cout << "This should not print\n";
   }
   catch ( exception &caughtException ) {
    cout << "\n\nException handled in main\n";
   }
   cout << "Program control continues after catch in main\n";
   return 0;
}
```



```
main invokes function throwException
  Function throwException throws an exception
  Exception handled in function throwException
  Function throwException rethrows exception

Exception handled in main
Program control continues after catch in main
```

Exception Specifications

List of exceptions function can throw

- Also called throw list

```
int someFunction( double value )
  throw ( ExceptionA, ExceptionB, ExceptionC )
{
 // function body
}
```

- Can only throw **`ExceptionA`**, **`ExceptionB`**, and **`ExceptionC`** (and derived classes)
  - If throws other type, function **`unexpected`** called
  - By default, terminates program (more 13.7)
- If no throw list, can throw any exception
- If empty throw list, cannot throw any exceptions

LECTURE 5

Process Un-expected exceptions

Processing Unexpected Exceptions

Function **`unexpected`**

- Calls function registered with **`set_unexpected`**
  - **`<exception>`**
  - Calls **`terminate`** by default
- **`set_terminate`**
  - Sets what function **`terminate`** calls
  - By default, calls **`abort`**
    - If redefined, still calls **`abort`** after new function finishes

Arguments for set functions

- Pass pointer to function
  - Function must take no arguments
  - Returns **`void`**

Stack Unwinding

If exception thrown but not caught

- Goes to enclosing **`try`** block
- Terminates current function
  - Unwinds function call stack
- Looks for **`try`**/**`catch`** that can handle exception
  - If none found, unwinds again

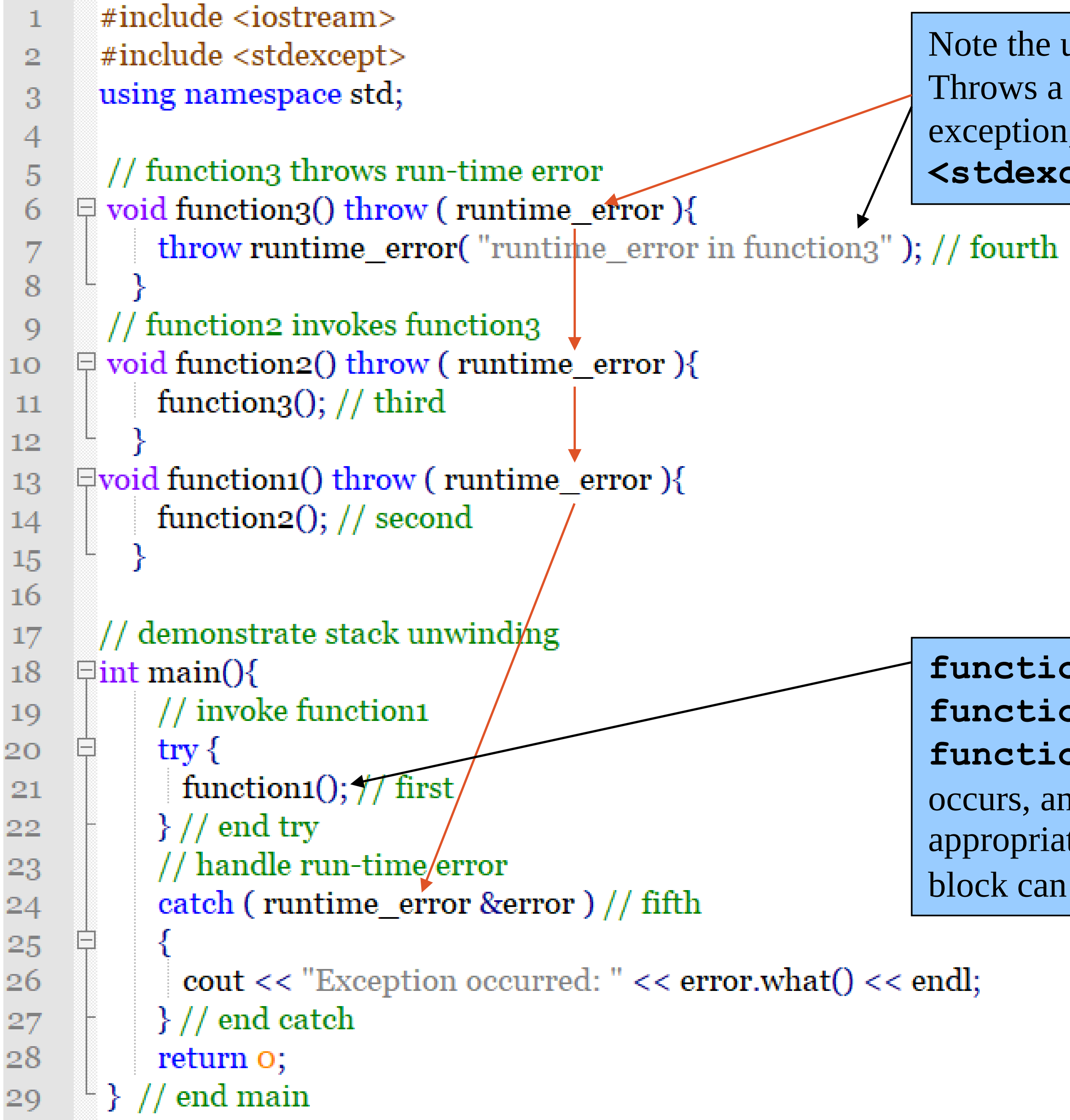
If exception never caught

- Calls **`terminate`**

Demo Program: stack_unwinding.cpp

Go Notepad!!!

stack_unwindind.cpp



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

 // function3 throws run-time error
 void function3() throw ( runtime_error ){
     throw runtime_error( "runtime_error in function3" ); // fourth
  }
 // function2 invokes function3
 void function2() throw ( runtime_error ){
     function3(); // third
  }
void function1() throw ( runtime_error ){
     function2(); // second
  }

// demonstrate stack unwinding
int main(){
     // invoke function1
     try {
        function1(); // first
     } // end try
     // handle run-time error
     catch ( runtime_error &error ) // fifth
     {
        cout << "Exception occurred: " << error.what() << endl;
     } // end catch
     return 0;
 } // end main
```

Note the use of the throw list. Throws a runtime error exception, defined in **`<stdexcept>`**.

Stack Unwinding:
The exception object is treated like a out-going variable of a function.

`function1` calls **`function2`** which calls **`function3`**. The exception occurs, and unwinds until an appropriate **`try`**/**`catch`** block can be found.

LECTURE 1

Exception Class and Objects

Constructors, Destructors and Exception Handling

Error in constructor

- **`new`** fails; cannot allocate memory
- Cannot return a value - how to inform user?
  - Hope user examines object, notices errors
  - Set some global variable
- Good alternative: throw an exception
  - Destructors automatically called for member objects
  - Called for automatic variables in **`try`** block

Can catch exceptions in destructor

Exceptions and Inheritance

Exception classes

- Can be derived from base classes
  - I.e., **`runtime_error`**; **`exception`**
- If **`catch`** can handle base class, can handle derived classes
  - Polymorphic programming

LECTURE 6

New Operator Error

Processing new Failures

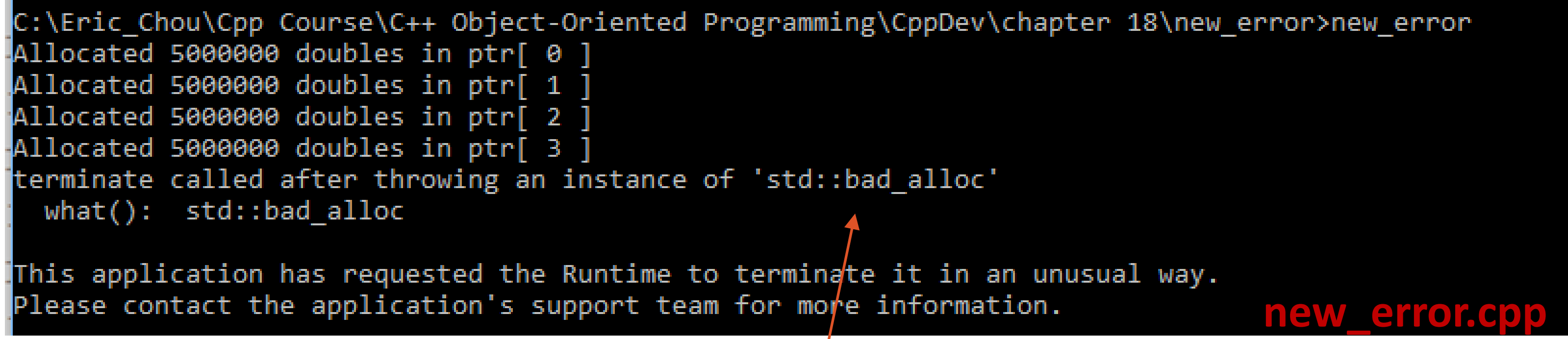
When **`new`** fails to get memory

- Should **`throw`** **`bad_alloc`** exception
  - Defined in **`<new>`**
- Some compilers have **`new`** return 0
- Result depends on compiler

Demo Program: new_error.cpp

Go Notepad++!!!

```cpp
#define LEN 50
#include <iostream>
using namespace std;

int main(){
  double *ptr[LEN];
  // allocate memory for ptr
  for ( int i = 0; i < LEN; i++ ) {
     ptr[ i ] = new double[50000000];
       // new returns 0 on failure to allocate memory
       if ( ptr[i] == 0 ) {
         cout << "Memory allocation failed for ptr[ "
              << i << " ]\n";
         break;
       } // end if
       // successful memory allocation
       else
         cout << "Allocated 5000000 doubles in ptr[ "
              << i << " ]\n";
     } // end for
     return 0;
} // end main
```

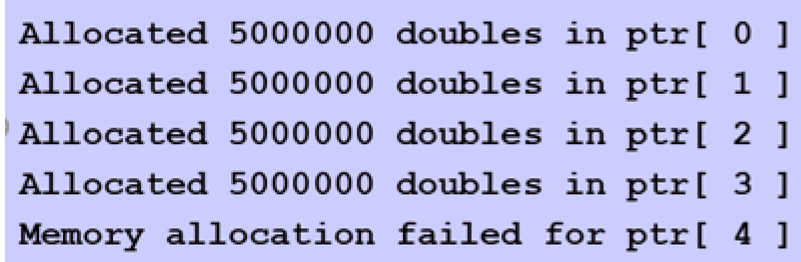


```
C:\Eric_Chou\Cpp Course\C++ Object-Oriented Programming\CppDev\chapter 18\new_error>new_error
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

new_error.cpp

bad_alloc exception object is thrown

In this program, the new operation fails to allocate memory. **bad_alloc** is issued. The program breaks. For some compiler a 0 will be returned to the ptr[i].



```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
Memory allocation failed for ptr[ 4 ]
```

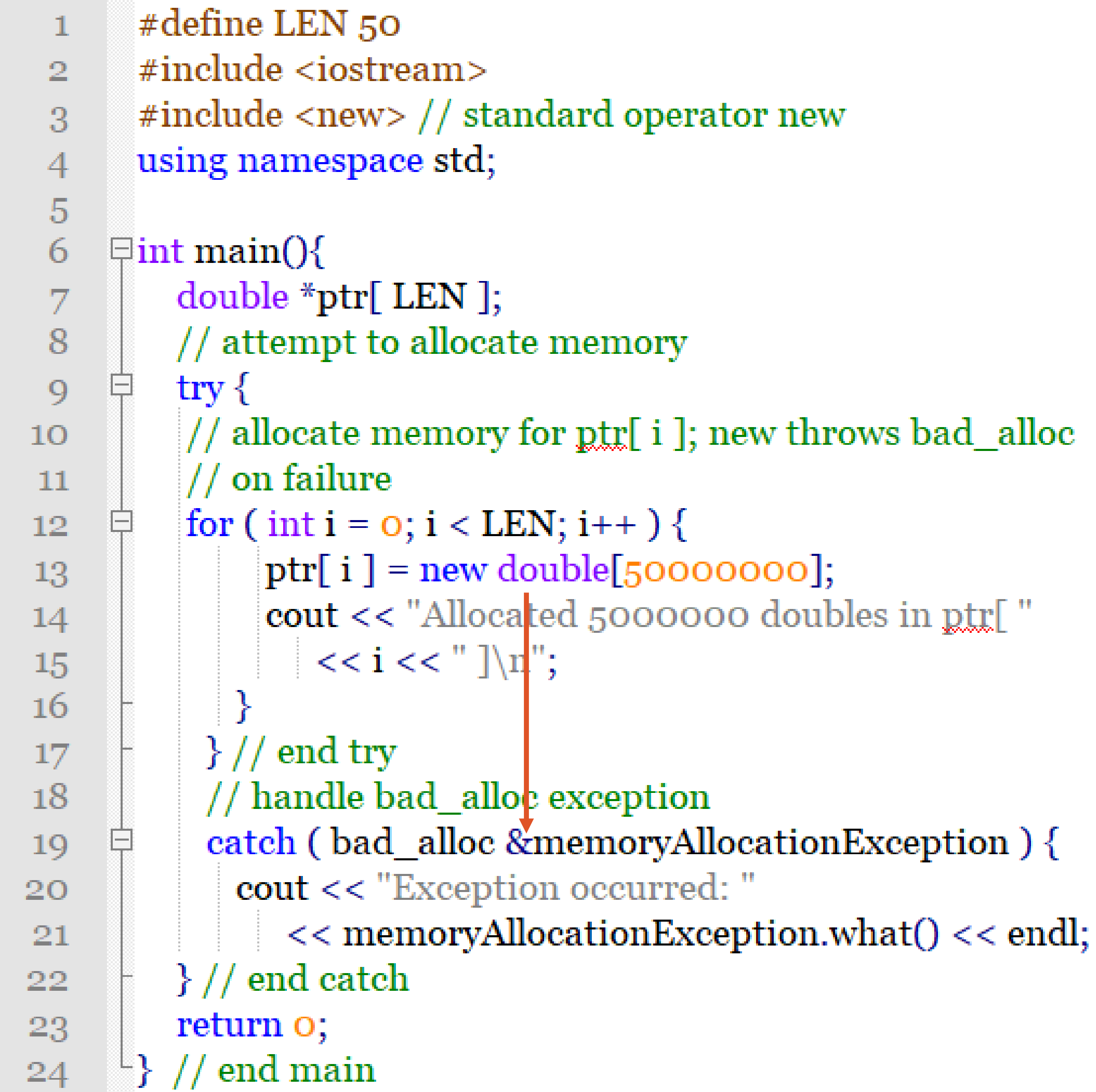
Demo Example with Exception Handling

Demo Program: new_error_handled.cpp

1. catch the bad_alloc exception.

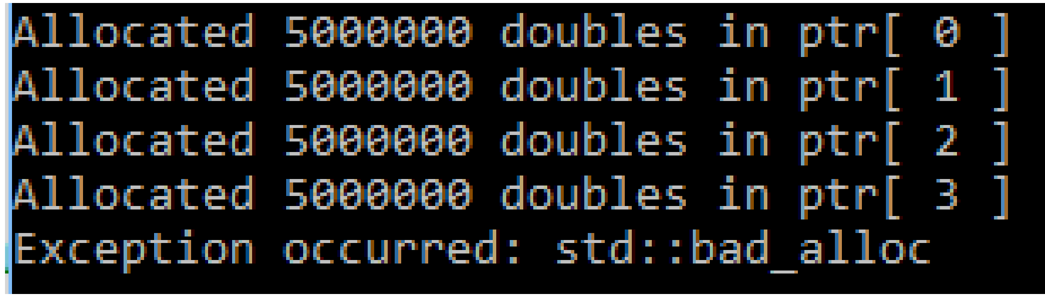
2. The program won’t terminate improperly.

new_error2.cpp



```
#define LEN 50
#include <iostream>
#include <new> // standard operator new
using namespace std;

int main(){
   double *ptr[ LEN ];
   // attempt to allocate memory
   try {
    // allocate memory for ptr[ i ]; new throws bad_alloc
    // on failure
    for ( int i = 0; i < LEN; i++ ) {
          ptr[ i ] = new double[50000000];
          cout << "Allocated 5000000 doubles in ptr[ "
              << i << " ]\n";
     }
   } // end try
   // handle bad_alloc exception
   catch ( bad_alloc &memoryAllocationException ) {
     cout << "Exception occurred: "
         << memoryAllocationException.what() << endl;
   } // end catch
   return 0;
} // end main
```

In this program, the new operation fails to allocate memory. **bad_alloc** is issued. It is handled.



```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
Exception occurred: std::bad_alloc
```

Processing new Failures

`set_new_handler`

- Header **`<new>`**
- Register function to call when **`new`** fails
- Takes function pointer to function that
  - Takes no arguments
  - Returns **`void`**
- Once registered, function called instead of throwing exception

New Exception Handler for Exception

1. Non-Try-Catch structure for exception handling.

2. Using Event-Handler type mechanism to handle exception.

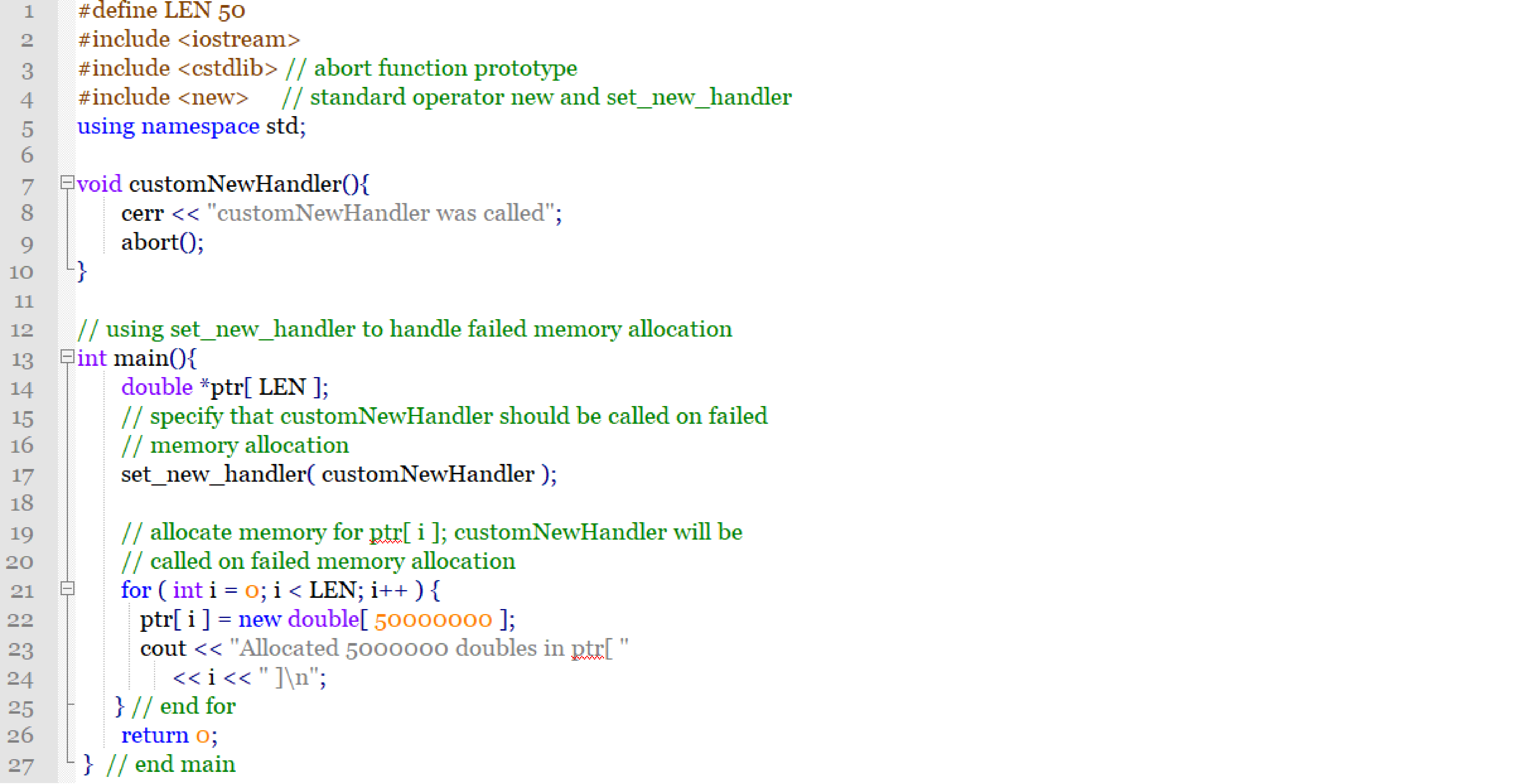
3. Using abort();

void abort (void);

- Aborts the current process, producing an abnormal program termination.
- The function raises the SIGABRT signal (as if raise(SIGABRT) was called). This, if uncaught, causes the program to terminate returning a platform-dependent unsuccessful termination error code to the host environment.
- The program is terminated without destroying any object and without calling any of the functions passed to **atexit** or **at_quick_exit**.

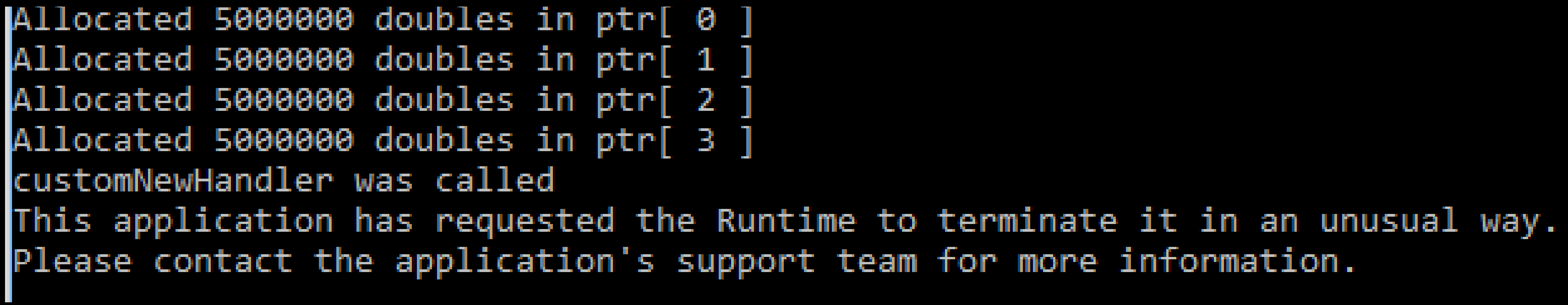
Demo Program: new_error3.cpp

Go Notepad++!!!



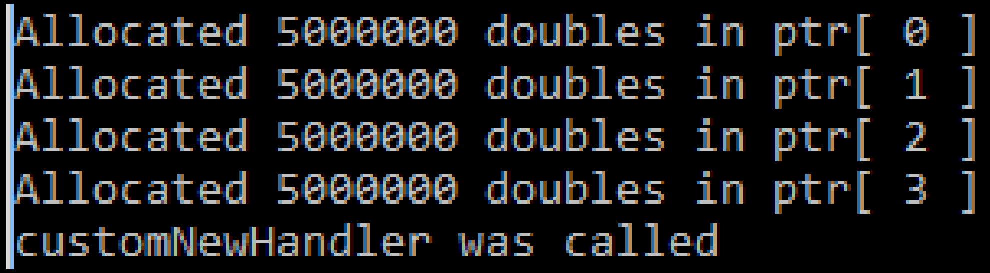
```cpp
#define LEN 50
#include <iostream>
#include <cstdlib> // abort function prototype
#include <new>     // standard operator new and set_new_handler
using namespace std;

void customNewHandler(){
    cerr << "customNewHandler was called";
    abort();
}

// using set_new_handler to handle failed memory allocation
int main(){
    double *ptr[ LEN ];
    // specify that customNewHandler should be called on failed
    // memory allocation
    set_new_handler( customNewHandler );

    // allocate memory for ptr[ i ]; customNewHandler will be
    // called on failed memory allocation
    for ( int i = 0; i < LEN; i++ ) {
      ptr[ i ] = new double[ 50000000 ];
      cout << "Allocated 5000000 doubles in ptr[ "
         << i << " ]\n";
   } // end for
    return 0;
 } // end main
```

Using abort();



```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
customNewHandler was called
This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

Using exit(1);



```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
customNewHandler was called
```

LECTURE 7

Memory Allocation Error

Class auto_ptr and Dynamic Memory Allocation

Declare pointer, allocate memory with `new`

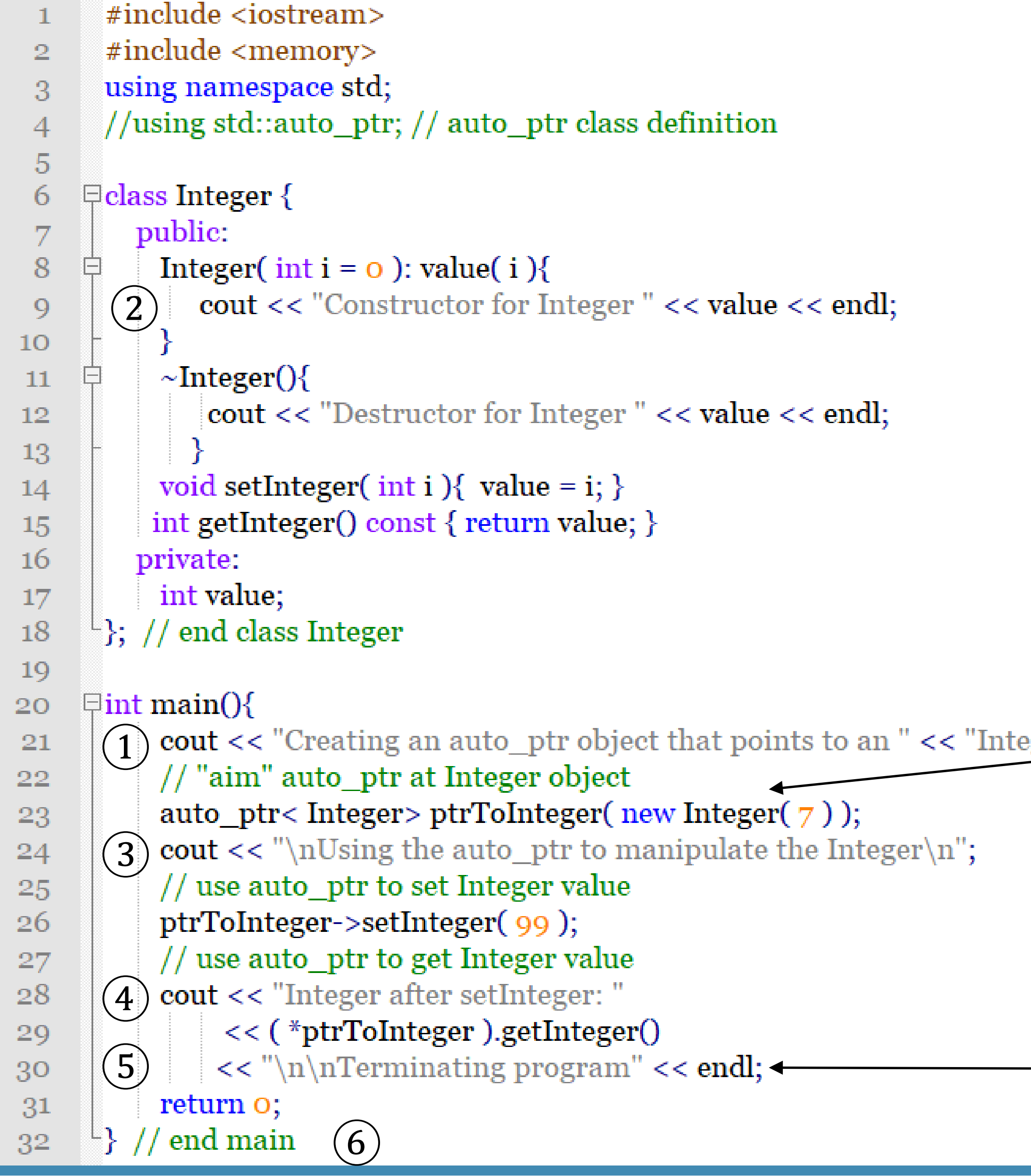
- What if exception occurs before you can **`delete`** it?
- Memory leak

Template class `auto_ptr`

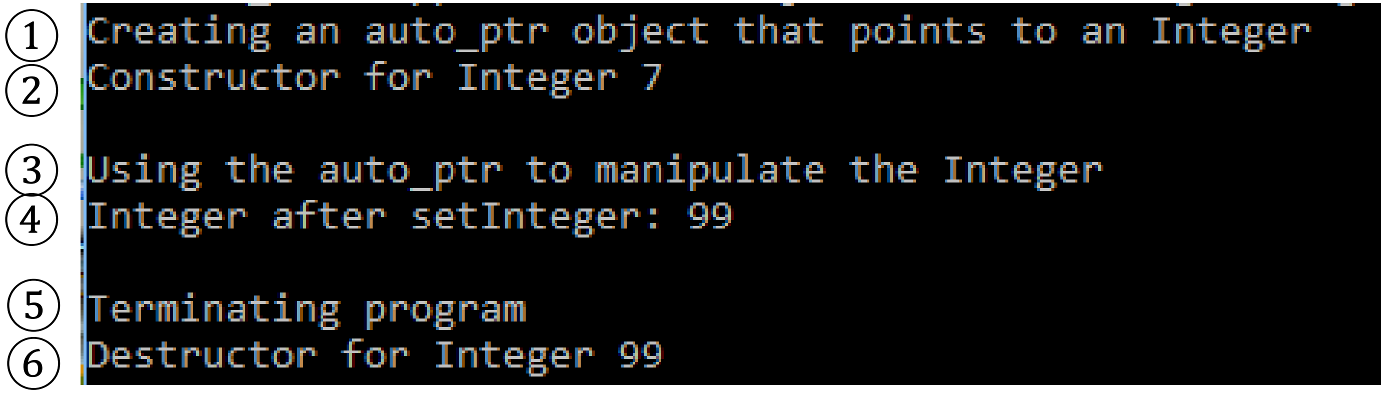
- Header **`<memory>`**
- Like regular pointers (has **`*`** and **`->`**)
- When pointer goes **out of scope**, calls **`delete`**
- Prevents memory leaks
- Usage
  - **`auto_ptr<MyClass> newPointer( new MyClass() );`**
  - **`newPointer`** points to dynamically allocated object

Demo Program: memory_error.cpp

Go Notepad++!!!



```cpp
#include <iostream>
#include <memory>
using namespace std;
//using std::auto_ptr; // auto_ptr class definition

class Integer {
   public:
     Integer( int i = 0 ): value( i ){
         cout << "Constructor for Integer " << value << endl;
     }
     ~Integer(){
          cout << "Destructor for Integer " << value << endl;
        }
     void setInteger( int i ){  value = i; }
    int getInteger() const { return value; }
   private:
     int value;
};  // end class Integer

int main(){
   cout << "Creating an auto_ptr object that points to an " << "Integer\n";
   // "aim" auto_ptr at Integer object
   auto_ptr< Integer> ptrToInteger( new Integer( 7 ) );
   cout << "\nUsing the auto_ptr to manipulate the Integer\n";
   // use auto_ptr to set Integer value
   ptrToInteger->setInteger( 99 );
   // use auto_ptr to get Integer value
   cout << "Integer after setInteger: "
         << ( *ptrToInteger ).getInteger()
        << "\n\nTerminating program" << endl;
   return 0;
} // end main
```



```
Creating an auto_ptr object that points to an Integer
Constructor for Integer 7

Using the auto_ptr to manipulate the Integer
Integer after setInteger: 99

Terminating program
Destructor for Integer 99
```

Create an **auto_ptr**. It can be manipulated like a regular pointer.

delete not explicitly called, but the **auto_ptr** will be destroyed once it leaves scope. Thus, the destructor for class **Integer** will be called.

LECTURE 8

Standard Exception Library

Standard Library Exception Hierarchy

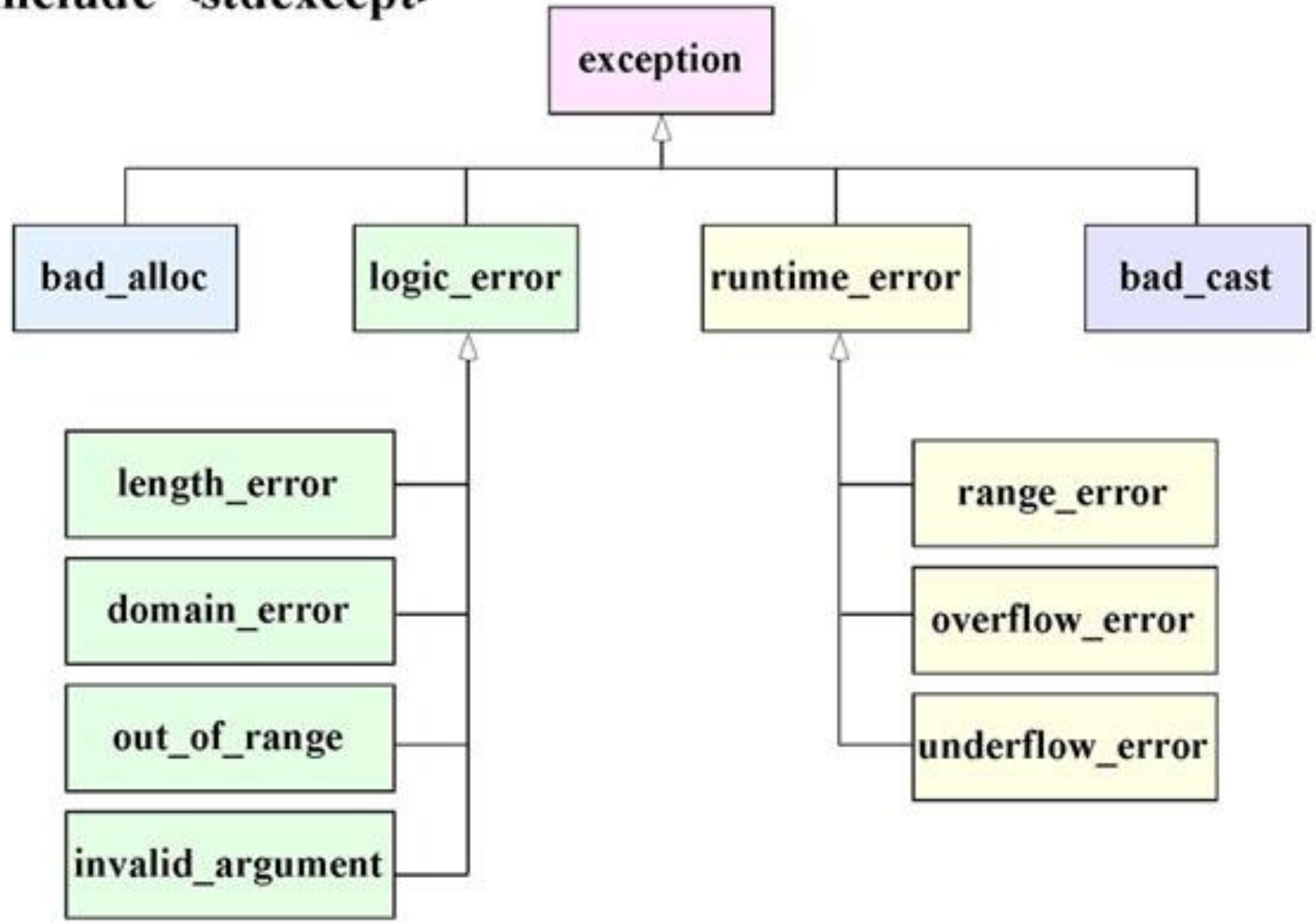
Exception hierarchy

- Base class **exception** (**<exception>**)
  - Virtual function **what**, overridden to provide error messages
- Sample derived classes
  - **runtime_error**, **logic_error**
  - **bad_alloc**, **bad_cast**, **bad_typeid**
    - Thrown by **new**, **dynamic_cast** and **typeid**

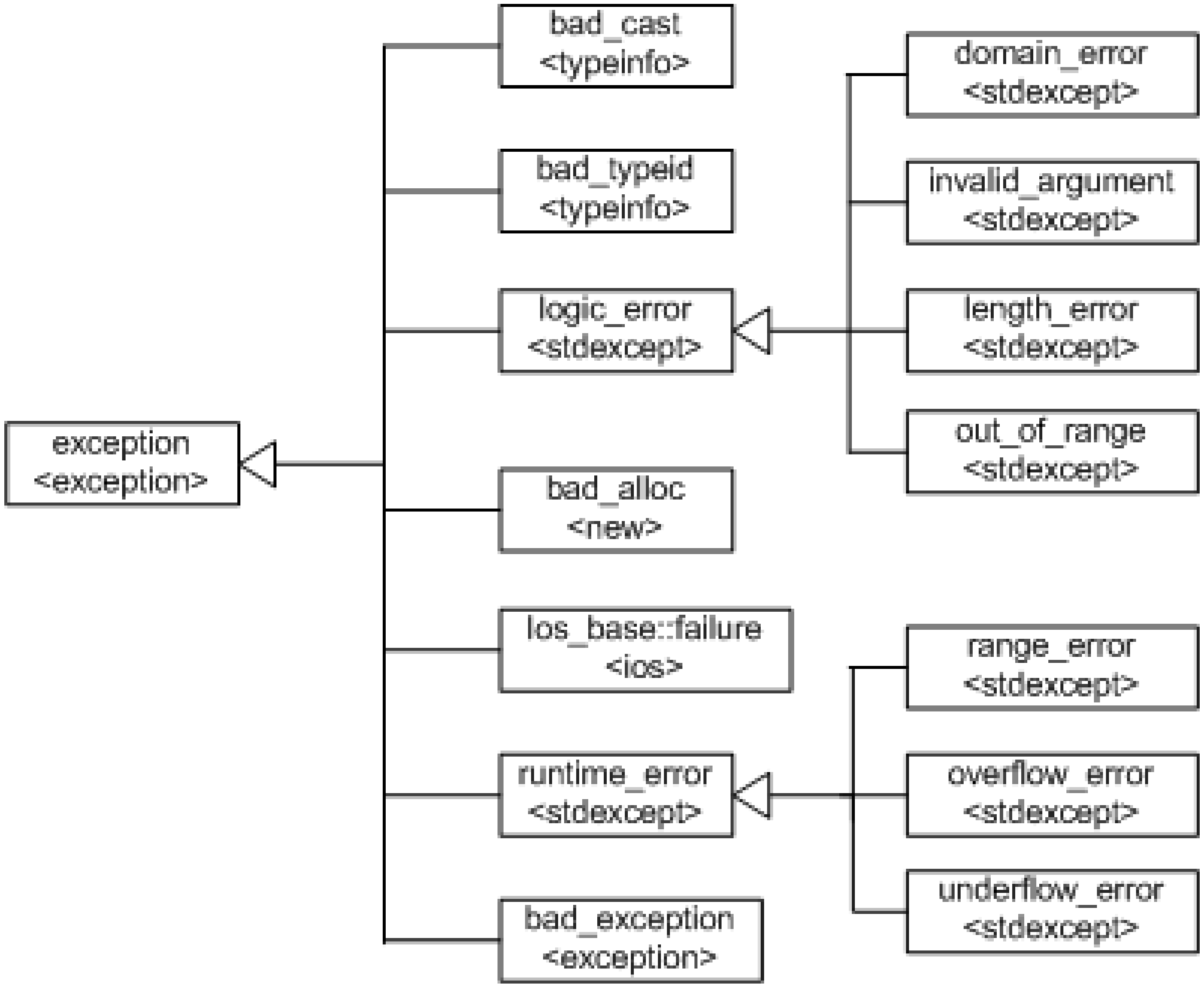
Exceptions

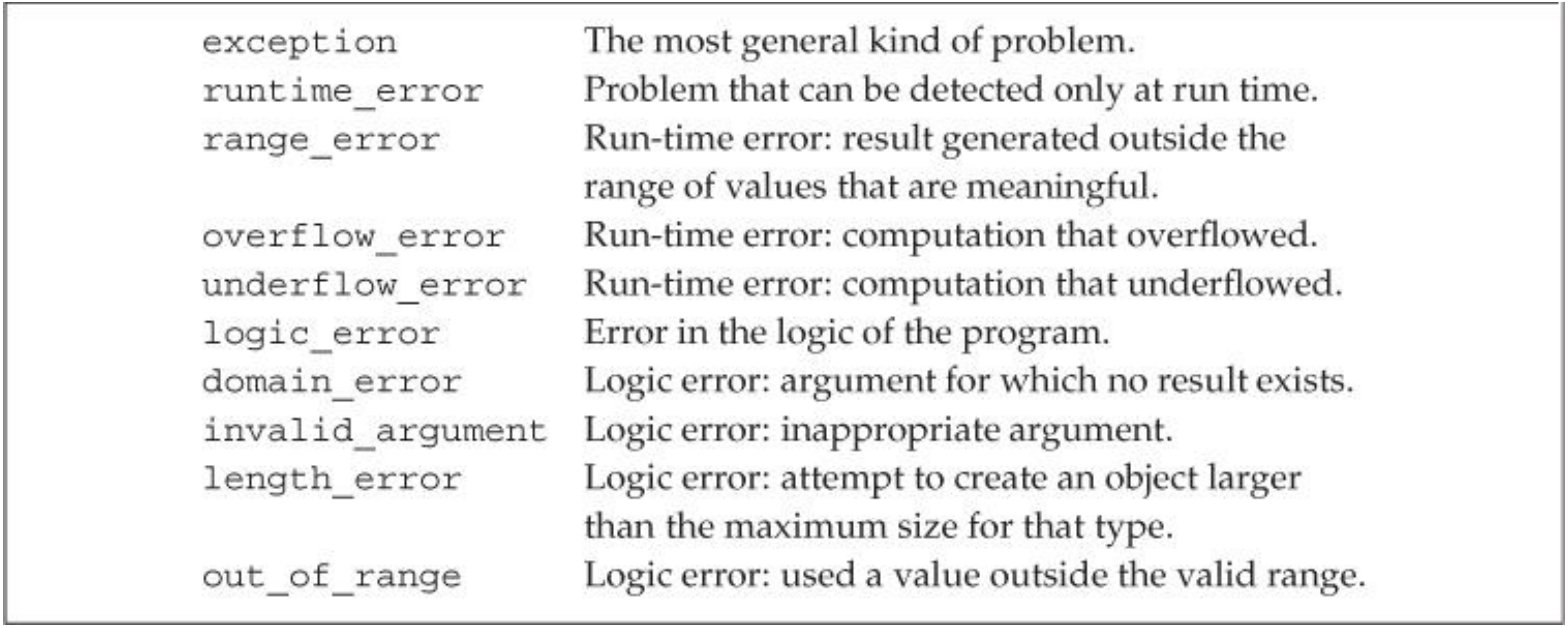
C++ Exception Classes

#include <stdexcept>



The C++ Exception Hierarchy





exception	The most general kind of problem.
runtime_error	Problem that can be detected only at run time.
range_error	Run-time error: result generated outside the range of values that are meaningful.
overflow_error	Run-time error: computation that overflowed.
underflow_error	Run-time error: computation that underflowed.
logic_error	Error in the logic of the program.
domain_error	Logic error: argument for which no result exists.
invalid_argument	Logic error: inappropriate argument.
length_error	Logic error: attempt to create an object larger than the maximum size for that type.
out_of_range	Logic error: used a value outside the valid range.

Standard Library Exception Hierarchy

To catch all exceptions

- `catch(...)`
- `catch( exception AnyException)`
  - Will not catch user-defined exceptions