Exception Handling

Chapter 18

Learning Objectives

- Exception Handling Basics
  - Defining exception classes
  - Multiple throws and catches
  - Exception specifications
- Programming Techniques for Exception Handling
  - When to throw exceptions
  - Exception class hierarchies

Introduction

- Typical approach to development:
  - Write programs assuming things go as planned
  - Get "core" working
  - Then take care of "exceptional" cases
- C++ exception-handling facilities
  - Handle "exceptional" situations
  - Mechanism "signals" unusual happening
  - Another place in code "deals" with exception

```
1)  int main( )
2)  {
3)     int donuts, milk;
4)     double dpg;
5)     cout << "Enter number of donuts:\n";    cin >> donuts;
6)     cout << "Enter number of glasses of milk:\n";    cin >> milk;

7)  if (milk <= 0)
8)     {
9)        cout << donuts << " donuts, and No Milk!\n"<< "Go buy some milk.\n";
10)    }
11)    else
12)    {
13) dpg = donuts/static_cast<double>(milk);
14)       cout << donuts << " donuts.\n" << milk << " glasses of milk.\n"
15)           << "You have " << dpg     << " donuts for each glass of milk.\n";
16)    }
17)    cout << "End of program.\n";
18)    return 0;
19) }
```

```
1)    int main( )
2)    {
3)       int donuts, milk;
4)       double dpg;

5)       try
6)       {
7)          cout << "Enter number of donuts:\n";
8)          cin >> donuts;
9)          cout << "Enter number of glasses of milk:\n";
10)         cin >> milk;
11)         if (milk <= 0) throw donuts;
12)         dpg = donuts/static_cast<double>(milk);
13)         cout << donuts << " donuts.\n"
14)            << milk << " glasses of milk.\n"
15)            << "You have " << dpg
16)            << " donuts for each glass of milk.\n";
17)      }
18)   catch(int e)
19)      {
20)         cout << e << " donuts, and No Milk!\n"
21)            << "Go buy some milk.\n";
22)      }

23)      cout << "End of program.\n";
24)      return 0;
25)   }
```

```
1)    int main( )
2)    {
3)       int donuts, milk;
4)       double dpg;
5)       cout << "Enter number of donuts:\n";
6)       cin >> donuts;
7)       cout << "Enter ... asses of milk:\n";
8)       cin >> milk;

9)       if (milk <= 0)
10)      {
11)         cout << donuts << " donuts.. No Milk!\n"
12)            << "Go buy some milk.\n";
13)      }
14)      else
15)      {
16)         dpg = donuts/static_cast<double>(milk);
17)         cout << donuts << " donuts.\n"
18)            << milk << " glasses of milk.\n"
19)            << "You have " << dpg
20)            << " donuts for each glass of milk.\n";
21)      }

22)      cout << "End of program.\n";
23)      return 0;
24)   }
```

try block

- Basic method of exception-handling is

try-throw-catch

- Try block:

```
try
{
    Some_Code;
}
```

  - Contains code for basic algorithm when all goes smoothly

throw

- Inside try-block, when something unusual happens:

```
try
{
        Code_To_Try
        if (exceptional_happened)
                throw donuts;
        More_Code
}
```

  - Keyword **throw** followed by **exception type**
  - Called "throwing an exception"

catch-block

- When something **thrown** → goes somewhere
  - In C++, **flow of control** goes from try-block to **catch-block**
    - try-block is "exited" and control passes to catch-block
  - Executing catch block called "catching the exception"
- **Exceptions** must be "**handled**" in some **catch block**

catch-block More

- Recall:

```
catch(int e)
{
        cout << e << " donuts, and no milk!\n";
                << " Go buy some milk.\n";
}
```

- Looks like function definition with `int` parameter!
  - Not a function, but works similarly
  - **Throw** like "function call"

catch-block Parameter

- Recall: catch(**int** e)
- "e" called catch-block parameter
  - Each catch block can have at most ONE catch-block parameter
- Does two things:
  1. **type** name specifies what kind of thrown value the catch-block can catch
  2. Provides name for thrown value caught; can "do things" with value

try-block

Associates one or more exception handlers (catch-clauses) with a compound statement.

Syntax

try *compound-statement handler-sequence*

where *handler-sequence* is a sequence of one or more *handler*s, which have the following syntax:

catch (*attr*(optional) *type-specifier-seq declarator* **)** *compound-statement*	(1)
catch (*attr*(optional) *type-specifier-seq abstract-declarator*(optional) **)** *compound-statement*	(2)
catch (...) *compound-statement*	(3)

compound-statement - brace-enclosed sequence of statements

***attr*(C++11)** - optional list of attributes, applies to the formal parameter

type-specifier-seq - part of a formal parameter declaration, same as in a function parameter list

declarator - part of a formal parameter declaration, same as in a function parameter list

abstract-declarator - part of an unnamed formal parameter declaration, same as in function parameter list

1) Catch-clause that declares a named formal parameter

```
try { /* */ } catch(const std::exception& e) { /* */ }
```

2) Catch-clause that declares an unnamed parameter

```
try { /* */ } catch(const std::exception&) { /* */ }
```

3) Catch-all handler, which is activated for any exception

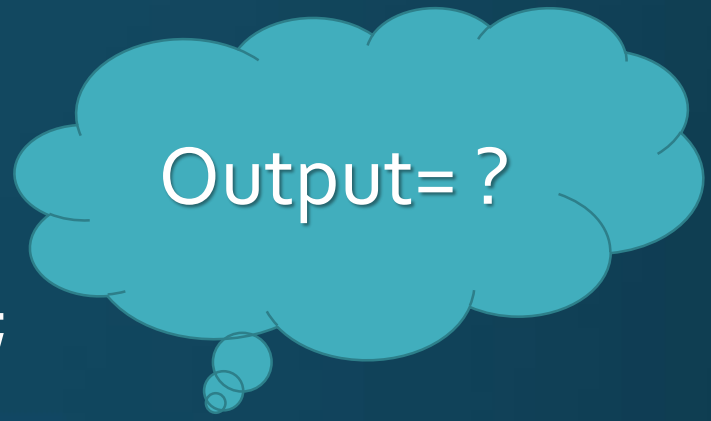
```
try { /* */ } catch(...) { /* */ }
```

catch(...)

- Matched exceptions of **any type**
- If present, it has to be the **last catch** clause
- Note
  - If no matches are found after all catch-clauses were examined, std::terminate is executed.
  - **So, use catch(...) as the last catch clause.**

```
1 try {
2   // code here
3 }
4 catch (int param) { cout << "int exception"; }
5 catch (char param) { cout << "char exception"; }
6 catch (...) { cout << "default exception"; }
```

```
1)  // …
2)  try
3)  {
4)      …
5)  }
6)  catch( … )
7)  {
8)      cout<< “Handle exception here.”;
9)  }

10) catch( const char * str )
11) {
12)     cout << "Caught exception: " << str << endl;
13) }

14) catch( CExcptClass E )
15) {
16)     cout<< “Handle CExcptClass exception here.”;
17) }
```



the ellipsis catch handler is the only handler that is examined.

throw expression

Signals an erroneous condition and executes an error handler.

Syntax

throw *expression*	(1)
throw	(2)

Explanation

See try-catch block for more information about *try* and *catch* (exception handler) blocks

1) First, copy-initializes the *exception object* from *expression* (this may call the move constructor for rvalue expression, and the copy/move may be subject to copy elision), then transfers control to the exception handler with the matching type whose compound statement or member initializer list was most recently entered and not exited by this thread of execution.

2) Rethrows the currently handled exception. Abandons the execution of the current catch block and passes control to the next matching exception handler (but not to another catch clause after the same try block: its compound-statement is considered to have been 'exited'), reusing the existing exception object: no new objects are made. This form is only allowed when an exception is presently being handled (it calls `std::terminate` if used otherwise). The catch clause associated with a function-try-block must exit via rethrowing if used on a constructor.

See `std::terminate` and `std::unexpected` for the handling of errors that arise during exception handling.

Throw expression Example:

```
1)  try {
2)    try {
3)       // code here
4)    }
5)    catch (int n) {
6)       throw;
7)    }
8)  }
9)  catch (...) {
10)   cout << "Exception occurred";
11) }
```

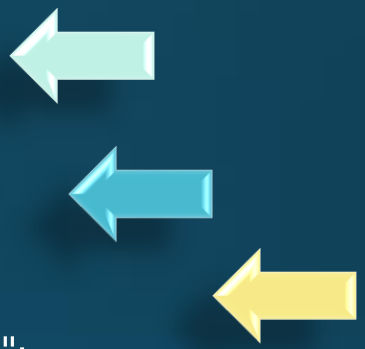
nest try-catch blocks

We have the possibility that an internal catch block forwards the exception to its external level

```
1)  int main( )
2)  {
3)     int donuts, milk;
4)     double dpg;
5)     try
6)     {
7)        cout << "Enter number of donuts:\n";
8)        cin >> donuts;
9)        cout << "Enter number of glasses of milk:\n";
10)       cin >> milk;

11)       if (milk <= 0)     throw NoMilk(donuts);
12)       dpg = donuts/static_cast<double>(milk);
13)       cout << donuts << " donuts.\n"
14)          << milk << " glasses of milk.\n"
15)          << "You have " << dpg
16)          << " donuts for each glass of milk.\n";
17)    }
18)    catch(NoMilk e)
19)    {
20)       cout << e.getCount( ) << " donuts, and No
    Milk!\n"
21)          << "Go buy some milk.\n";
22)    }
23)    cout << "End of program.\n";
24)    return 0;
25) }
```

```
1) class NoMilk
2) {
3) public:
4)    NoMilk( ) {}
5)    NoMilk(int howMany) : count(howMany) {}
6)    int getCount( ) const { return count; }
7) private:
8)    int count;
9)};
```

```
1)  int main( ) {
2)     int pencils, erasers;
3)     double ppe; //pencils per eraser
4)     try
5)     {
6)        cout << "How many pencils do you have?\n";
7)        cin >> pencils;
8)        if (pencils < 0)          throw NegativeNumber("pencils");
9)        cout << "How many erasers do you have?\n";
10)       cin >> erasers;
11)       if (erasers < 0)       throw NegativeNumber("erasers");
12)       if (erasers != 0)     ppe = pencils/static_cast<double>(erasers);
13)       else          throw DivideByZero( );
14)       cout << "Each eraser must last through "  << ppe << " pencils.\n";
15)    }
16)    catch(NegativeNumber e)
17)    {
18)      cout << "Cannot have a negative number of "
19)           << e.getMessage( ) << endl;
20)    }
21)    catch(DivideByZero)
22)    {
23)      cout << "Do not make any mistakes.\n";
24)    }
25)    cout << "End of program.\n";
26)    return 0;
27)}
```



```
1)  class NegativeNumber
2)  {
3)  public:
4)     NegativeNumber( ){}
5)     NegativeNumber(string theMessage):
                              message(theMessage) {}
6)     string getMessage( ) const { return message; }
7)  private:
8)     string message;
9)  };

10) class DivideByZero
11) {};
```

Throwing Exception in Function

Throwing Exception in Function

- **Function** might **throw** exception
- Callers might have different "reactions"
  - Some might desire to "end program"
  - Some might continue, or do something else
- Makes sense to "**catch**" exception **in** calling function's try-catch-block
  - Place call inside try-block
  - Handle in catch-block after try-block

Throwing Exception in Function Example

- Consider:

```
try
{
        quotient = safeDivide(num, den);
}
catch (DivideByZero)
{ … }
```

- safeDivide() function throws DividebyZero exception
  - Handled back **in caller's** catch-block

```
1)  int main( ) {
2)     int numerator, denominator;
3)     double quotient;
4)     cout << "Enter numerator:\n";
5)     cin >> numerator;
6)     cout << "Enter denominator:\n";
7)     cin >> denominator;
8)     try   {     quotient = safeDivide(numerator, denominator);   }
9)     catch(DivideByZero)   {
10)       cout << "Error: Division by zero!\n"    << "Program aborting.\n";
11)       exit(0);
12)    }
13)    cout << numerator << "/" << denominator  << " = " << quotient
             << endl << "End of program.\n";
14)    return 0;
15) }
16) double safeDivide(int top, int bottom) throw (DivideByZero)
17) {
18)    if (bottom == 0)       throw DivideByZero( );
19)    return top/static_cast<double>(bottom);
20) }
```

```
1)  class DivideByZero
2)  {};

3)  double safeDivide(int top, int bottom)
                          throw (DivideByZero);
```

double safeDivide(int top, int bottom) throw (DivideByZero)

Exception specification

Exception Specification

- Functions that don't catch exceptions
  - Should "warn" users that it could throw
  - But it won't catch!
- Should list such exceptions:

```
double safeDivide(int top, int bottom) throw (DividebyZero);
```

  - Called "**exception specification**" or "**throw list**"
  - Should be in declaration and definition
  - All types listed handled "normally"
  - If no throw list → all types considered there

Exception specification

- Older code ！ But still supported

```
double safeDivide(int top, int bottom) throw (DivideByZero);
```

- If **throws** an exception other than **DivideByZero**,
  - The function calls **std::unexpected** instead of looking for a handler or calling **std::terminate**
- If this **throw** specifier is left empty with no type

```
int myfunction (int param) throw(); // all exceptions call
                                    unexpected
```

  - This means that **std::unexpected** is called for any exception.
- No throw specifier (regular functions)

```
int myfunction (int param);        // normal exception handling
```

  - never call **std::unexpected**, but follow the normal path of looking for their **exception handler**.

Throw List Summary

- void someFunction() throw(DividebyZero, OtherException);
  //Exception types DividebyZero or OtherException treated normally.
  // All others invoke unexpected()

- void someFunction() throw ();
  //Empty exception list, all exceptions invoke unexpected()

- void someFunction();
  //All exceptions of all types treated normally

When to Throw Exceptions

- Typical to separate throws and catches
  - In separate functions
- Throwing function:
  - Include throw statements in definition
  - List exceptions in throw list
    - In both declaration and definition
- Catching function:
  - Different function, perhaps even in different file

Preferred throw-catch Triad: throw

- void functionA() **throw** (MyException)

```
{
    ...
    throw MyException(arg);
    ...
}
```

- Function throws exception as needed

Preferred throw-catch Triad: catch

- Then some other function:

```
void functionB()
{
        …
        try
        {
                …
                functionA();
                …
        }
        catch (MyException e)
        { // Handle exception
        }
        …
}
```

Uncaught Exceptions

- Should catch every exception thrown
- If not → program terminates
  - **terminate**() is called
- Recall for functions
  - If exception not in throw list: **unexpected**() is called
    - It in turn calls **terminate**()
- So same result

unexpected()

- Default action: terminates program
  - No special includes or using directives
- Normally no need to redefine
- But you can:
  - Use **set_unexpected**
  - Consult compiler manual or advanced text for details

Explicitly Call Terminate In Any Handlers

- The default action of **terminate** is to call **abort**
- Like "terminate" to call some other function before exiting the application

```
1)  void term_func()
2)  {
3)    cout << "term_func was called by terminate." << endl;
4)    exit( -1 );
5)  }
6)  int main()
7)  {
8)    try
9)    {
10)     set_terminate( term_func );   // instructs terminate to call term_func
11)     throw "Out of memory!"; // No catch handler for this exception
12)   }
13)   catch( int )
14)   {
15)     cout << "Integer exception raised." << endl;
16)   }
17)   return 0;
18) }
```

Standard Exception

Standard Exception

- std::exception
  - A base class specifically designed to declare objects to be thrown as exceptions
  - Has a virtual member function, can be overwritten
    - called **what**() returns a null-terminated char*
  - Defined in the <exception> header
- All exceptions thrown by C++ Standard library

=> throw exceptions **derived** from this exception class.

```
1)  #include <iostream>
2)  #include <exception>
3)  using namespace std;

4)  class myexception: public exception
5)  {
6)    virtual const char* what() const throw()
7)    {
8)      return "My exception happened";
9)    }
10) } myex;

11) int main () {
12)   try
13)   {
14)     throw myex;
15)   }
16)   catch (exception& e)
17)   {
18)     cout << e.what() << '\n';  // My exception happened.

19)   }
20)   return 0;
21) }
```

Example of std::exception

```
try {
  f();
} catch(const std::overflow_error& e) {
    // this executes if f() throws std::overflow_error (same type rule)
} catch(const std::runtime_error& e) {
    // this executes if f() throws std::underflow_error (base class rule)
} catch(const std::exception& e) {
    // this executes if f() throws std::logic_error (base class rule)
} catch(...) {
    // this executes if f() throws std::string or int or any other unrelated type
}
```

http://en.cppreference.com/w/cpp/error/exception

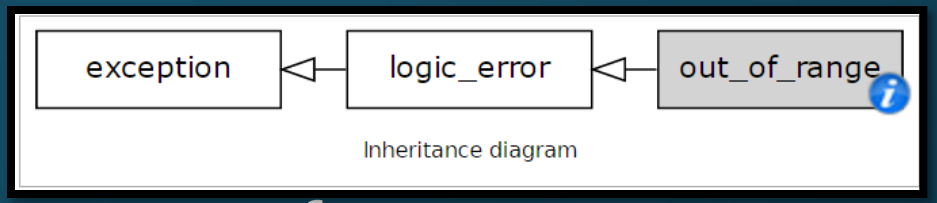
Member functions

(constructor)	constructs the exception object (public member function)
(destructor) [virtual]	destructs the exception object (virtual public member function)
operator=	copies exception object (public member function)
what [virtual]	returns an explanatory string (virtual public member function)

All exceptions generated by the standard library inherit from **std::exception**

- **logic_error**
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error(C++11)
  - bad_optional_access(C++17)
- **runtime_error**
  - range_error
  - overflow_error
  - underflow_error
  - regex_error(C++11)
  - tx_exception(TM TS)
  - system_error(C++11)
    - ios_base::failure(C++11)
    - filesystem::filesystem_error(C++17)
- bad_typeid
- bad_cast
  - bad_any_cast(C++17)
- bad_weak_ptr(C++11)
- bad_function_call(C++11)
- bad_alloc
  - bad_array_new_length(C++11)
- bad_exception
- ios_base::failure(until C++11)
- exception_list(C++17)

std::out_of_range



- Reports errors that are consequence of attempt to access elements out of defined range
- vector, deque, string and bitset also throw exceptions of this type to signal arguments out of range

std::out_of_range::out_of_range

```
explicit out_of_range( const std::string& what_arg );     (1)
explicit out_of_range( const char* what_arg );            (2)  (since C++11)
```

Constructs the exception object with what_arg as explanatory string that can be accessed through what().

Parameters

what_arg - explanatory string

Exceptions

(none)

Inherited from std::exception

Member functions

(destructor) [virtual]	destructs the exception object (virtual public member function of std::exception)
what [virtual]	returns an explanatory string (virtual public member function of std::exception)

Example of std::out_of_range

```
1)  #include <iostream>      // std::cerr
2)  #include <stdexcept>     // std::out_of_range
3)  #include <vector>        // std::vector

4)  int main (void)
    {
5)   std::vector<int> myvector(10);
6)   try
     {
7)     myvector.at(20)=100;     // vector::at throws an out-of-range
8)   }
9)   catch (const std::out_of_range& oor)
     {
10)    std::cerr << "Out of Range error: " << oor.what() << '\n';
11)  }
12)  return 0;
13) }
```

std::overflow_error

✞report arithmetic overflow errors

```
1)   #include <bitset>
2)   #include <iostream>

3)   using namespace std;

4)   int main( )
5)   {
6)     try
7)     {
8)       bitset< 33 > bitset;
9)       bitset[32] = 1;
10)      bitset[0] = 1;
11)      unsigned long x = bitset.to_ulong( ); // sizeof(unsigned long) =4
12)    }
13)    catch ( exception &e )
14)    {
15)      cerr << "Caught " << e.what( ) << endl;
16)      cerr << "Type " << typeid( e ).name( ) << endl;
17)    };
18)  }
```

std::bitset::to_ulong()

- Converts the contents of the bitset to an unsigned long integer.
- Exceptions

throws **std::overflow_error** if the value can not be represented in unsigned long.

std::bad_function_call

- thrown by std::function::operator() if the function wrapper has no target
  - Empty function objects: are function objects with no target callable object

```
1)  #include <iostream>    // std::cout
2)  #include <functional>  // std::function, std::plus, std::bad_function_call

3)  int main ()
4)  {
5)    std::function<int(int,int)> foo = std::plus<int>();
6)    std::function<int(int,int)> bar;

7)    try {
8)      std::cout << foo(10,20) << '\n';
9)      std::cout << bar(10,20) << '\n';
10)   }
11)   catch (std::bad_function_call& e)
12)   {
13)     std::cout << "ERROR: Bad function call\n";
14)   }

15)   return 0;
16) }
```

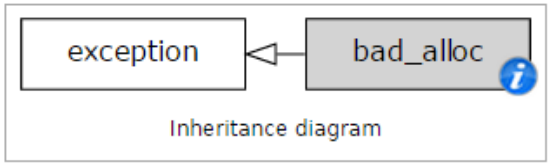
30
ERROR: Bad function call

std::bad_alloc

Defined in header <new>

```
class bad_alloc : public std::exception;
```

`std::bad_alloc` is the type of the object thrown as exceptions by the allocation functions to report failure to allocate storage.



Inheritance diagram

Member functions

(constructor)	constructs the bad_alloc objec (public member function)
operator=	replaces a bad_alloc object (public member function)
what	returns explanatory string (public member function)

```
#include <iostream>
#include <new>

int main()
{
    try {
        while (true) {
            new int[100000000ul];
        }
    } catch (const std::bad_alloc& e) {
        std::cout << "Allocation failed: " << e.what() << '\n';
    }
}
```

Possible output:

```
Allocation failed: std::bad_alloc
```

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

int main () {
  try
  {
    int* myarray= new int[1000];
  }
  catch (exception& e)
  {
    cout << "Standard exception: " << e.what() << endl;
  }
  return 0;
}
```

Testing Available Memory

- new operator throws **bad_alloc** exception if insufficient memory:

```
try
{
        NodePtr pointer = new Node;
}
catch (bad_alloc)
{
        cout << "Ran out of memory!";
        // Can do other things here as well…
}
```

- In library <new>, std namespace

How u know what exception thrown?

```
try {
   std::string("abc").substr(10); // throws std::length_error
// } catch( std::exception e ) { // copy-initialization from the std::exception base
//     std::cout << e.what(); // information from length_error is lost
// }
} catch( const std::exception& e ) { // reference to the base of a polymorphic object
     std::cout << e.what(); // information from length_error printed
}
```

Look at: string.substr()

std::basic_string::substr

```
basic_string substr( size_type pos = 0,
                     size_type count = npos ) const;
```

Returns a substring [pos, pos+count). If the requested substring extends past the end of the string, or if `count == npos`, the returned substring is [pos, size()).

Parameters

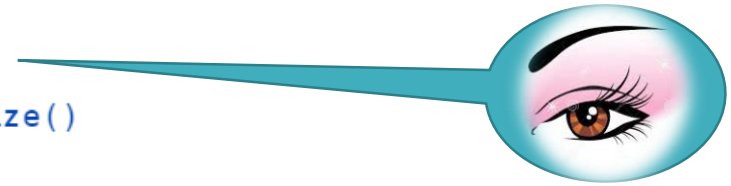
pos	-	position of the first character to include
count	-	length of the substring

Return value

1. An empty string if pos == size().
2. String containing the substring [pos, pos+count).

Exceptions

std::out_of_range if `pos > size()`



Complexity

Linear in count

Throwing Exception Objects

Throwing Exception Objects

- Ways to throw an exception object

1. Throw a pointer to the object
2. Throw an object by value
3. Thrown by value, and caught by (usually const) reference

Throw a pointer to the object

```
1)  void foo()
2)  {
3)     throw new MyApplicationException();
4)  }

5)  void bar()
6)  {
7)     try
8)     {
9)        foo();
10)    }
11)    catch(MyApplicationException* e)
12)    {
13)       // Handle exception
14)    }
15) }
```

```
Class MyApplicationException
{
...
};
```

Who is responsible to delete the exception?
The handler?
=> This makes code uglier.

Throw an object by value

```
1)  void bar(){
2)     try
3)     {
4)        foo();
5)     }   catch(MyApplicationException e)
6)     {
7)        // Handle exception
8)     }
9)  }
```

```
1)  void foo()
2)  {
3)     throw MyApplicationException();
4)  }
```

Catches the exception by value

⇒A copy constructor is called

⇒Cause to crash if the exception caught was a bad_alloc caused by insufficient memory. i.e, to handle memory allocation

⇒May cause the copy to have different behavior because of object slicing

Thrown by value, and caught by reference

```
1)  void bar() {
2)      try    {
3)          foo();
4)      }
5)      catch(MyApplicationException const& e)   {
6)          // Handle exception
7)      }
8)  }
```

```
1)  void foo()
2)  {
3)      throw MyApplicationException();
4)  }
```

the **compiler** is responsible for **destroying** the object, and no **copying** is done at catch time!

~exceptions should be thrown by value, and caught by reference~

Stack Unwinding

Example of Stack unwinding

```
1)   void g()
2)   {
3)      throw std::exception();
4)   }
5)
6)   void f()
7)   {
8)      std::string str = "Hello";
        // This string is newly allocated
9)      g();
10)  }
11)
12)  int main()
13)  {
14)     try
15)     {
16)        f();
17)     }
18)     catch(...)
19)     {}
20)  }
```

1. main() calls f()
2. f() creates a local variable str
3. str constructor allocates a memory chunk to hold the string "Hello"
4. f() calls g()
5. g()throws an exception
6. f() does not catch the exception.
7. Because the exception was not caught, we now need to exit f() in a clean fashion.
8. At this point, all the destructors of local variables previous to the throw are called ('stack unwinding')
9. The destructor of str is called.
10. main() catches the exception
11. The program continues.

'stack unwinding'

- prevent resource leaks
- applies to destructors for objects

Recap

Multiple Throws and Catches

- try-block typically throws any number of exception values, of differing types
- Of course only one exception thrown
  - Since throw statement ends try-block
- But different types can be thrown
  - Each catch block only catches "one type"
  - Typical to place many catch-blocks after each try-block
    - To catch "all-possible" exceptions to be thrown

Catching

- Order of catch blocks important
- **Catch-blocks** tried "**in order**" after try-block
  - First match handles it!
- Consider:

```
catch (...) { }
```

  - Called "catch-all", "default" exception handler
  - Catches any exception
  - Ensure catch-all placed AFTER more specific exceptions!
    - Or others will never be caught!

```
1)  void __declspec(nothrow) f2(void)
2)  {
3)    try
4)    {
5)      f1();
6)    }  catch(int)  {
7)      handler();
8)    }
9)  }

10) void f4(void) {  f1(); }

11) int main() {
12)   f2();
13)   try {    f4();  }
14)   catch(...)
15)   {
16)     printf_s("Caught exception from f4\n");
17)   }
18)   f5();
19) }
```

```
1)  void handler()
2)  {
3)    printf_s("in handler\n");
4)  }
5)  void f1(void) throw(int)
6)  {
7)    printf_s("About to throw 1\n");
8)    if (1)
9)      throw 1;
10) }
11) void f5(void) throw()
12) {
13)   try {
14)     f1();
15)   }  catch(...)  {
16)     handler();
17)   }
18) }
```

```
About to throw 1
in handler
About to throw 1
Caught exception from f4
About to throw 1
in handler
```

Overuse of Exceptions

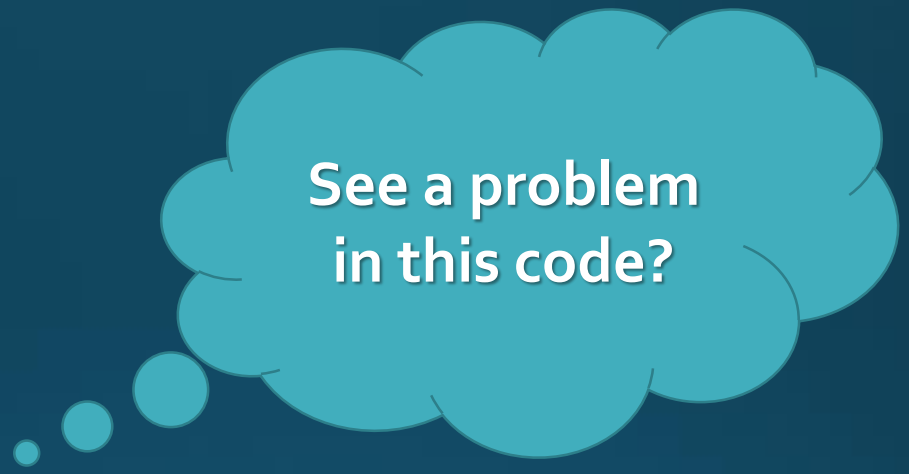
- Exceptions alter flow of control
  - Similar to old "goto" construct
  - "Unrestricted" flow of control
- Should be used sparingly
- **Good rule:**
  - If desire a "throw": consider how to write program without throw
  - If alternative reasonable → do it

Partial handling

Writing exception safe code

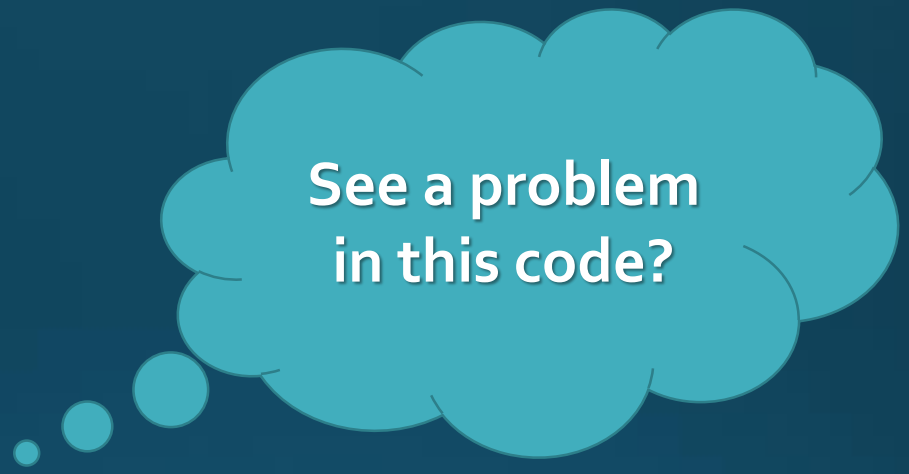
Partial handling

```
1)  void g()
2)  {
3)      throw "Exception";
4)  }
5)
6)  void f()
7)  {
8)      int* pI = new int(0);
9)      g();
10)     delete pI;
11) }

12) int main()
13) {
14)     f();
15)     return 0;
16) }
```



Partial handling

```
1)  void g()
2)  {
3)      throw "Exception";
4)  }
5)
6)  void f()
7)  {
8)      int* pI = new int(0);
9)      g();
10)     delete pI;
11) }

12) int main()
13) {
14)     f();
15)     return 0;
16) }
```



If g() throws an exception, the variable pI is never deleted and we have a memory leak.

Partial handling

```
1)  void g()
2)  {
3)      throw "Exception";
4)  }
5)
6)  void f()
7)  {
8)      int* pI = new int(0);
9)      g();
10)     delete pI;
11) }

12) int main()
13) {
14)     f();
15)     return 0;
16) }
```

One solution!

```
1)   void g() {   throw "Exception"; }
2)
3)   void f()
4)   {
5)      int* pI = new int(0);

6)      try   {      g();   }
7)      catch (...)
8)      {
9)         delete pI;
10)        throw; // This empty throw re-throws the exception we caught
11)                  // An empty throw can only exist in a catch block
12)     }

13)     delete pI;
14)  }

15)  int main()
16)  {
17)     f();
18)     return 0;
19)  }
```

There's a better way though;
using **RAII** classes to avoid the need to use exception handling.

A neater solution using the 'stack unwinding'

```
1)  class IntDeleter {
2)  public:
3)    IntDeleter(int* piValue)  {    m_piValue = piValue;  }
4)    ~IntDeleter()  {    delete m_piValue;  }
5)     // operator *, enables us to dereference the object and use it like a regular pointer.
6)    int&  operator *()  {    return *m_piValue;  }
7)  private:
8)     int* m_piValue;
9)  };

10) int main()
11) {
12)   try { f(); }
13)   catch (...)   { }
14)   return 0;
15) }
```

```
void g(){  throw std::exception(); }
void f()
{
  IntDeleter pI(new int(2));

  *pI = 3;
   g();   // No need to delete pI, this will be done in destruction.
  // This code is also exception safe.
}
```

The pattern presented here is called a guard (STL has unique_ptr)

Exception Class Hierarchies

- Useful to have; consider:
  DivideByZero class derives from:
  ArithmeticError exception class
  - All catch-blocks for ArithmeticError also catch DivideByZero
  - If ArithmeticError in throw list, then DividebyZero also considered there

Example of Exception hierarchy

✞ create a class hierarchy of exception classes:

1) class **MyApplicationException** {};
2) class MathematicalException : public **MyApplicationException** {};
3) class DivisionByZeroException : public MathematicalException {};
4) class InvalidArgumentException : public **MyApplicationException** {};

```
1)   float divide(float fNumerator, float fDenominator)
2)   {
3)      if (fDenominator == 0.0)   {      throw DivisionByZeroException();   }
4)      return fNumerator/fDenominator;
5)   }

6)   enum MathOperators {DIVISION, PRODUCT};

7)   float operate(int iAction, float fArgLeft, float fArgRight)
8)   {
9)      if (iAction == DIVISION)   {      return divide(fArgLeft, fArgRight);   }
10)     else if (iAction == PRODUCT))   {      // call the product function      // ...   }
11)     // No match for the action! iAction is an invalid agument
12)     throw InvalidArgumentException();
13)  }
14)
15)  int main(int iArgc, char* a_pchArgv[])
16)  {
17)     try   {      operate(atoi(a_pchArgv[0]), atof(a_pchArgv[1]), atof(a_pchArgv[2]));   }
18)     catch(MathematicalException& )   {      // Handle Error   }
19)     catch(MyApplicationException& )
20)     {
21)        // This will catch in InvalidArgumentException too.
22)        // Display help to the user, and explain about the arguments.
23)     }

24)     return 0;
25)  }
```

Summary 1

- Exception handling allows separation of "normal" cases and "exceptional" cases
- Exceptions thrown in try-block
  - Or within a function whose call is in try-block
- Exceptions caught in catch-block
- try-blocks typically followed by more than one catch-block
  - List more specific exceptions first

Summary 2

- Best used with separate functions
  - Especially considering callers might handle differently
- Exceptions thrown in but not caught in function, should be listed in throw list
- Exceptions thrown but never caught → program terminates
- Resist overuse of exceptions
  - Unrestricted flow of control

RAII

- Resource Acquisition Is Initialization means「資源獲得即初始化」
  - 意即：一旦在程式中有「資源配置」的行為，也就是一旦有「配置、釋放」的動作，就讓「配置」成為一個初始化的動作

  => 如此，釋放動作就變成自動的了（依物件的 scope 決定）。

- RAII exploits C++ destructor to manage resources automatically

Resource management automatically

一、使用 auto_ptr 避免手動 delete

```
1) void f()
2) {
3)   ...
4)   std::auto_ptr<TMemoryStream> p(new TMemoryStream);
5)   ...
6)   if (...) { throw 1; }
7)   ...
8) } // OK, 沒問題，一旦碰到右大括號，即使發生異常，
     p 也會正確被釋放。
```

二、使用 vector 取代手動配置 array

```
1) void g()
2) {
3)   int N;
4)   std::cin >> N;
5)   std::vector<int> v(N);
6)   …
7)   if (…) { throw 1; }
8)   …
9) } // OK, 沒問題，即使發生異常，也不必操心 v 內部的記憶體管理
```

三、以回傳物件的方式，取代回傳函式內部 new 的物件的指標

```
std::string g2()
{
  std::string s;
  ...
  return s;
} // OK, 外部模組不必擔心忘記釋放記憶體的問題。
```

應用 RAII 的精神

- The following code snippets seem no problem!

```
1)  void f() // 未考慮「異常安全」的版本
2)  {
3)    ...
4)    ThreadLock(...);
5)    ... // 在此區域內，不允許兩個執行緒同時進入。
6)    ThreadUnlock(...);
7)    ...
8)  };
```

- What if exceptions happened between Lock() and Unlock()（而且被上層處理掉了）
  - =>Unlock 就沒被執行到，就出大事了

應用 RAII 的精神 (Contd.)

```
1)  struct Lock
2)  {
3)    Lock(...) {ThreadLock(...); }
4)   ~Lock() {ThreadUnlock(...); }
5)    ...
6)  };

7)  void f2() // RAII 版本
8)  {
9)    ...
10)   {
11)    Lock lock(...);
12)    ...
13)   } // OK, 不管是否發生異常，lock 物件一旦至此就會被解構。
14)   ...
15) }
```