Chapter 16: Exceptions, Templates, and the Standard Template Library (STL)

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Topics

16.1 Exceptions

16.2 Function Templates

16.3 Class Templates

16.4 Class Templates and Inheritance

16.5 Introduction to the Standard Template Library

16.1 Exceptions

- An exception is a condition that occurs at execution time and makes normal continuation of the program impossible
- When an exception occurs, the program must either terminate or jump to special code for handling the exception.
- The special code for handling the exception is called an exception handler

Exceptions – Key Words

- **`throw`** – followed by an argument, is used to signal an exception
- **`try`** – followed by a block **`{ }`**, is used to invoke code that throws an exception
- **`catch`** – followed by a block **`{ }`**, is used to process exceptions thrown in preceding **`try`** block. Takes a parameter that matches the type thrown.

Throwing an Exception

- Code that detects the exception must pass information to the exception handler. This is done using a **`throw`** statement:

```
throw "Emergency!"
throw 12;
```

- In C++, information thrown by the **`throw`** statement may be a value of any type

Catching an Exception

- Block of code that handles the exception is said to catch the exception and is called an exception handler
- An exception handler is written to catch exceptions of a given type: For example, the code

```
catch(char *str)
{
   cout << str;
}
```

can only catch exceptions of C-string type

Catching an Exception

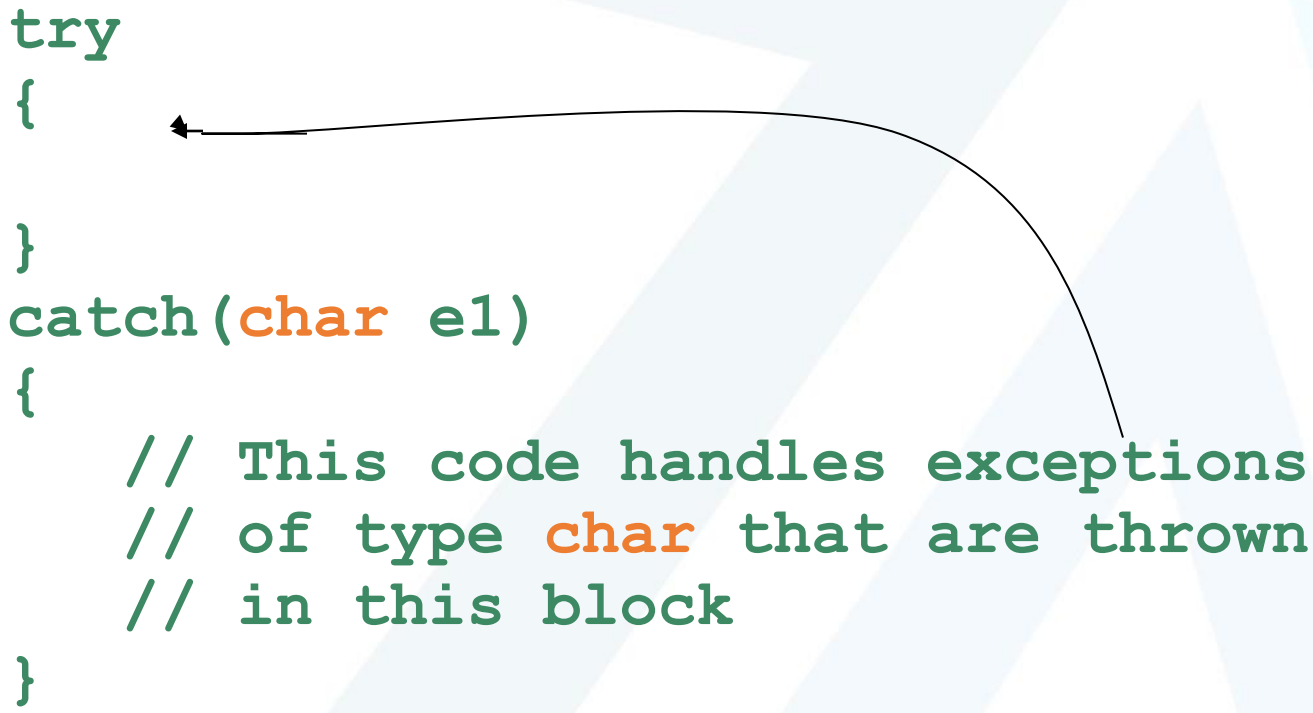
Another example of a handler:

```
catch(int x)
{
  cerr << "Error: " << x;
}
```

This can catch exceptions of type **`int`**

Connecting to the Handler

Every catch block is attached to a try block of code and is responsible for handling exceptions thrown from that block



```
try
{

}
catch(char e1)
{
    // This code handles exceptions
    // of type char that are thrown
    // in this block
}
```

Execution of Catch Blocks

- The catch block syntax is similar to a that of a function
- Catch block has a formal parameter that is initialized to the value of the thrown exception before the block is executed

Exception Example

- An example of exception handling is code that computes the square root of a number.
- It throws an exception in the form of a C-string if the user enters a negative number

Example

```
int main( )
{
  try
  {
    double x;
    cout << "Enter a number: ";
    cin >> x;
    if (x < 0) throw "Bad argument!";
    cout<<"Square root of "<< x << " is "<< sqrt(x);
  }
  catch(char *str)
  {
       cout << str;
  }
  return 0;
}
```

Flow of Control

1. Computer encounters a **`throw`** statement in a **`try`** block
2. The computer evaluates the **`throw`** expression, and immediately exits the **`try`** block
3. The computer selects an attached **`catch`** block that matches the type of the thrown value, places the value in the catch block's formal parameter, and executes the catch block

Uncaught Exception

- An exception may be uncaught if
  - there is no **`catch`** block with a data type that matches the exception that was thrown, or
  - it was not thrown from within a **`try`** block
- The program will terminate in either case

Handling Multiple Exceptions

Multiple catch blocks can be attached to the same block of code. The catch blocks should handle exceptions of different types

```
try{...}
catch(int iEx){ }
catch(char *strEx){ }
catch(double dEx){ }
```

Throwing an Exception Class

- An exception class can be defined and thrown
- Catch block must be designed to catch an object of the exception class
- Exception class object can pass data to exception handler via data members

Exception When Calling **new**

- If **new** cannot allocate memory, it throws an exception of type **bad_alloc**
- Must **#include <new>** to use **bad_alloc**
- Can invoke **new** from within a **try** block, use a **catch** block to detect that memory was not allocated.

Nested Exception Handling

`try` blocks can be nested in other **`try`** blocks and even in catch blocks

```
try
{
   try{ } catch(int i){ }
}
catch(char *s)
{ }
```

Where to Find an Exception Handler?

- The compiler looks for a suitable handler attached to an enclosing **`try`** block in the same function
- If there is no matching handler in the function, it terminates execution of the function, and continues the search for a handler at the point of the call in the calling function.

Unwinding the Stack

- An unhandled exception propagates backwards into the calling function and appears to be thrown at the point of the call
- The computer will keep terminating function calls and tracing backwards along the call chain until it finds an enclosing **`try`** block with a matching handler, or until the exception propagates out of **`main`** (terminating the program).
- This process is called unwinding the call stack

Rethrowing an Exception

- Sometimes an exception handler may need to do some tasks, then pass the exception to a handler in the calling environment.
- The statement

```
throw;
```

with no parameters can be used within a **`catch`** block to pass the exception to a handler in the outer block

16.2 Function Templates

- Function template: A pattern for creating definitions of functions that differ only in the type of data they manipulate

- Better than overloaded functions, since the code defining the algorithm of the function is only written once

Example

Two functions that differ only in the type of the data they manipulate

```
void swap(int &x, int &y)
 { int temp = x; x = y;
   y = temp;
 }

 void swap(char &x, char &y)
 { char temp = x; x = y;
   y = temp;
 }
```

A **swap** Template

The logic of both functions can be captured with one template function definition

```
template<class T>
void swap(T &x, T &y)
{ T temp = x; x = y;
  y = temp;
}
```

Using a Template Function

- When a function defined by a template is called, the compiler creates the actual definition from the template by inferring the type of the type parameters from the arguments in the call:

```
int i = 1, j = 2;
swap(i,j);
```

- This code makes the compiler instantiate the template with type `int` in place of the type parameter `T`

Function Template Notes

- A function template is a pattern
- No actual code is generated until the function named in the template is called
- A function template uses no memory
- When passing a class object to a function template, ensure that all operators referred to in the template are defined or overloaded in the class definition

Function Template Notes

- All data types specified in template prefix must be used in template definition
- Function calls must pass parameters for all data types specified in the template prefix
- Function templates can be overloaded – need different parameter lists
- Like regular functions, function templates must be defined before being called

Where to Start When Defining Templates

- Templates are often appropriate for multiple functions that perform the same task with different parameter data types
- Develop function using usual data types first, then convert to a template:
  - add template prefix
  - convert data type names in the function to a type parameter (*i.e.*, a T type) in the template

16.3 Class Templates

- It is possible to define templates for classes.
- Unlike functions, a class template is instantiated by supplying the type name (**`int`**, **`float`**, **`string`**, etc.) at object definition

Class Template

Consider the following classes

1. Class used to join two integers by adding them:

```
class Joiner
{ public:
     int combine(int x, int y)
     {return x + y;}
};
```

2. Class used to join two strings by concatenating them:

```
class Joiner
{ public:
     string combine(string x, string y)
     {return x + y;}
};
```

Example class Template

A single class template can capture the logic of both classes: it is written with a template prefix that specifies the data type parameters:

```
template <class T>
class Joiner
{
public:
    T combine(T x, T y)
        {return x + y;}
};
```

Using Class Templates

To create an object of a class defined by a template, specify the actual parameters for the formal data types

```
Joiner<double> jd;
Joiner<string> sd;
cout << jd.combine(3.0, 5.0);
cout << sd.combine("Hi ", "Ho");
```

Prints **`8.0`** and **`Hi Ho`**

16.5 Introduction to the Standard Template Library

- Standard Template Library (STL): a library containing templates for frequently used data structures and algorithms
- Programs can be developed faster and are more portable if they use templates from the STL

Standard Template Library

Two important types of data structures in the STL:

- containers: classes that store data and impose some organization on it
- iterators: like pointers; provides mechanisms for accessing elements in a container

Containers

Two types of container classes in STL:

- sequential containers: organize and access data sequentially, as in an array. These include **`vector`**, **`dequeue`**, and **`list`** containers.
- associative containers: use keys to allow data elements to be quickly accessed. These include **`set`**, **`multiset`**, **`map`**, and **`multimap`** containers.

Creating Container Objects

- To create a list of **`int`**, write

```
list<int> mylist;
```

- To create a vector of **`string`** objects, write

```
vector<string> myvector;
```

- Requires the **`vector`** header file

Iterators

- Generalization of pointers, used to access information in containers
- Four types:
  - forward (uses **++**)
  - bidirectional (uses **++** and **--**)
  - random-access
  - input (can be used with **`cin`** and **`istream`** objects)
  - output (can be used with **`cout`** and **`ostream`** objects)

Containers and Iterators

- Each container class defines an iterator type, used to access its contents
- The type of an iterator is determined by the type of the container:

```
list<int>::iterator x;
list<string>::iterator y;
```

x is an iterator for a container of type `list<int>`

Containers and Iterators

Each container class defines functions that return iterators:

`begin()`: returns iterator to item at start

`end()`: returns iterator denoting end of container

Containers and Iterators

- Iterators support pointer-like operations: if **`iter`** is an iterator:
  - **`*iter`** is the item it points to: this dereferences the iterator
  - **`iter++`** advances to the next item in the container
  - **`iter--`** backs up in the container
- The **`end()`** iterator points to past the end: it should never be dereferenced

Traversing a Container

Given a vector:

```
vector<int> v;
for  (int k=1; k<= 5; k++)
  v.push_back(k*k);
```

Traverse it using iterators:

```
vector<int>::iterator iter = v.begin();
while (iter != v.end())
  { cout << *iter << " "; iter++;}
```

Prints **1 4 9 16 25**

Algorithms

- STL contains algorithms implemented as function templates to perform operations on containers.
- Requires **`algorithm`** header file
- Collection of algorithms includes

`binary_search`	**`count`**
`for_each`	**`find`**
`max_element`	**`min_element`**
`random_shuffle`	**`sort`**

and others

Using STL algorithms

- Many STL algorithms manipulate portions of STL containers specified by a begin and end iterator
- **`max_element(iter1, iter2)`** finds max element in the portion of a container delimited by **`iter1`**, **`iter2`**
- **`min_element(iter1, iter2)`** is similar to above

More STL algorithms

- **`random_shuffle(iter1, iter2)`** randomly reorders the portion of the container in the given range
- **`sort(iter1, iter2)`** sorts the portion of the container specified by the given range

`random-shuffle` Example

The following example

- stores the squares 1, 4, 9, 16, 25 in a vector
- shuffles the vector
- then prints it out

random_shuffle example

```
int main()
{
    vector<int> vec;
    for (int k = 1; k <= 5; k++)
      vec.push_back(k*k);
    random_shuffle(vec.begin(),vec.end());
    vector<int>::iterator p = vec.begin();
    while (p != vec.end())
    { cout << *p << "  "; p++;
    }
    return 0;
}
```

Chapter 16: Exceptions, Templates, and the Standard Template Library (STL)

Starting Out with C++
Early Objects
Seventh Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda