9: Advanced C++ Features

Programming Technique II

(SECJ1023)

Adapted from Tony Gaddis and Barret Krupnow (2016), Starting out with C++: From Control Structures through Objects

Content

- Exceptions
- Templates
  - Function templates
  - Class templates
- Containers
  - Vectors
  - Maps
- Iterators

Exceptions

Introduction to Exceptions

- 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.

Terminology

- **Exception**: object or value that signals an error ⇨ exceptional circumstance ⇨ run-time errors.
- **Throw an exception**: send a signal that an error has occurred.
- **Catch/ Handle an exception**: process the exception; interpret the signal.

Keywords

- **`throw`**: send a signal that an error has occurred.

- **`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.

Flow of Control

- A function that throws an exception is called from within a `try` block.

- 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.

- 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.

Example 1a: Using throw

```
//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 a literal c-string
    else
        return (7 * weeks + days);
}
```

Example 1b: Using try…catch

```
int main(){
      int totDays,days, weeks;

      cout << "Enter no. of. days and no. of. weeks =>";
      cin >> days >> weeks;

      try{
            totDays = totalDays(days, weeks);
            cout << "Total days: " << totDays;
      }
      catch (const char *msg){
            cout << "Error: " << msg;
      }

      return 0;
}
//code in the try-block is called protected code
//code in the catch-block is called exception handler
```

Correction:

catch (**const** char *msg)

Example 1: What Happens?

- `try` block is entered. `totalDays` function is called to.

- If 1st parameter is between 0 and 7, total number of days is returned and `catch` block is skipped over (**no exception thrown**).

- If **exception is thrown**, function and `try` block are exited, `catch` blocks are scanned for 1st one that matches the data type of the thrown exception. `catch` block executes.

Additional Notes:
Dealing with string exceptions

- You can throw different types of string exceptions.
  - A literal string
  - C string
  - C++ string

```
 7       int choice;
 8       cout << "Enter choice => ";
 9       cin >> choice;
10
11       try {
12               if (choice == 1)
13                   throw "This is a const c-string";
14
15               if (choice == 2){
16                   char cStr[] = "This is a c-string";
17                   throw cStr;
18               }
19
20               if (choice==3){
21                   string cppStr="This is a cpp-string";
22                   throw cppStr;
23               }
24       }
```

Throwing a literal c-string

Throwing a c-string variable

Throwing a string object (c++ string)

Additional Notes:
Dealing with string exceptions

- Then, catch the exceptions accordingly

```
25      catch (const char *msg){
26          cout << "Caught msg: " << msg;
27      }
28      catch (char *msg){
29          cout << "Caught msg: " << msg;
30      }
31      catch (string msg){
32          cout << "Caught msg: " << msg;
33      }
34
```

Catching a literal c-string exception

Catching a c-string exception

Catching a c++ string exception

Example 2a: Using try...catch

```
 8  int main()
 9  {
10     int num1, num2;  // To hold two numbers
11     double quotient; // To hold the quotient of the numbers
12
13     // Get two numbers.
14     cout << "Enter two numbers: ";
15     cin >> num1 >> num2;
16
17     // Divide num1 by num2 and catch any
18     // potential exceptions.
19     try
20     {
21        quotient = divide(num1, num2);
22        cout << "The quotient is " << quotient << endl;
23     }
24     catch (char *exceptionString)
25     {
26        cout << exceptionString;
27     }
28
29     cout << "End of the program.\n";
30     return 0;
31  }
```

Correction:

catch (**const** char *exceptionString)

Example 2b: Using `throw`

```
33  //*********************************************
34  // The divide function divides numerator by  *
35  // denominator. If denominator is zero, the  *
36  // function throws an exception.             *
37  //*********************************************
38
39  double divide(int numerator, int denominator)
40  {
41     if (denominator == 0)
42        throw "ERROR: Cannot divide by zero.\n";
43
44     return static_cast<double>(numerator) / denominator;
45  }
```

Program Output with Example Input Shown in Bold

```
Enter two numbers: 12 2 [Enter]
The quotient is 6
End of the program.
```

Program Output with Example Input Shown in Bold

```
Enter two numbers: 12 0 [Enter]
ERROR: Cannot divide by zero.
End of the program.
```

Example 2: What Happens in the `try…catch` Construct?

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.

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

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

Correction:

catch (**const** char *exceptionString)

Example 2: What Happens 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;
```

Correction:

catch (**const** char *exceptionString)

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.

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.

Example 3a

Contents of Rectangle.h (Version 1)

```
1  // Specification file for the Rectangle class
2  #ifndef RECTANGLE_H
3  #define RECTANGLE_H
4
5  class Rectangle
6  {
7     private:
8        double width;      // The rectangle's width
9        double length;     // The rectangle's length
10    public:
11       // Exception class
12       class NegativeSize
13          { };            // Empty class declaration
14
15       // Default constructor
16       Rectangle()
17          { width = 0.0; length = 0.0; }
18
19       // Mutator functions, defined in Rectangle.cpp
20       void setWidth(double);
21       void setLength(double);
22
23       // Accessor functions
24       double getWidth() const
25          { return width; }
26
27       double getLength() const
28          { return length; }
29
30       double getArea() const
31          { return width * length; }
32 };
33 #endif
```

Example 3b

Contents of `Rectangle.cpp` (Version 1)

```
1  // Implementation file for the Rectangle class.
2  #include "Rectangle.h"
3
4  //*************************************************************
5  // setWidth sets the value of the member variable width.    *
6  //*************************************************************
7
8  void Rectangle::setWidth(double w)
9  {
10    if (w >= 0)
11       width = w;
12    else
13       throw NegativeSize();
14 }
15
16 //*************************************************************
17 // setLength sets the value of the member variable length.  *
18 //*************************************************************
19
20 void Rectangle::setLength(double len)
21 {
22    if (len >= 0)
23       length = len;
24    else
25       throw NegativeSize();
26 }
```

Example 3c

Program 16-2

```
1  // This program demonstrates Rectangle class exceptions.
2  #include <iostream>
3  #include "Rectangle.h"
4  using namespace std;
5
6  int main()
7  {
8     int width;
9     int length;
10
11    // Create a Rectangle object.
12    Rectangle myRectangle;
13
14    // Get the width and length.
15    cout << "Enter the rectangle's width: ";
16    cin >> width;
17    cout << "Enter the rectangle's length: ";
18    cin >> length;
19
20    // Store these values in the Rectangle object.
21    try
22    {
23       myRectangle.setWidth(width);
24       myRectangle.setLength(length);
25       cout << "The area of the rectangle is "
26            << myRectangle.getArea() << endl;
27    }
28    catch (Rectangle::NegativeSize)
29    {
30       cout << "Error: A negative value was entered.\n";
31    }
32    cout << "End of the program.\n";
33
34    return 0;
35 }
```

Example 3d: Sample Output

Program Output with Example Input Shown in Bold

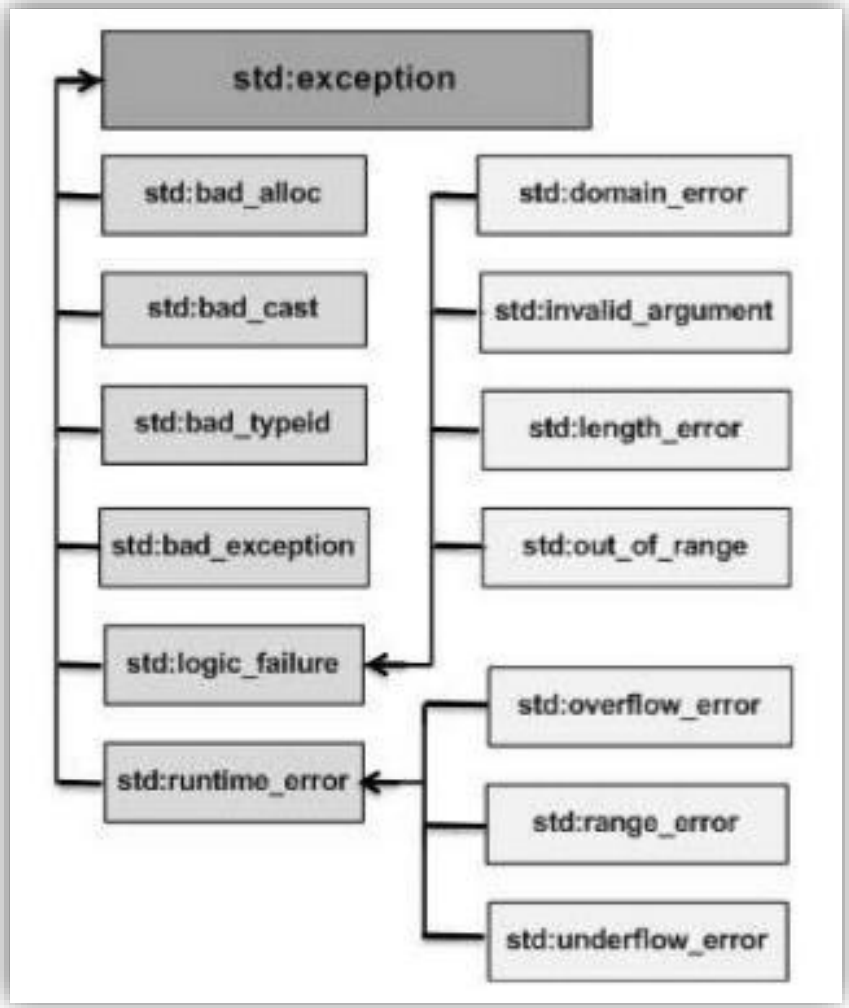
```
Enter the rectangle's width: 10 [Enter]
Enter the rectangle's length: 20 [Enter]
The area of the rectangle is 200
End of the program.
```

Program Output with Example Input Shown in Bold

```
Enter the rectangle's width: 5 [Enter]
Enter the rectangle's length: -5 [Enter]
Error: A negative value was entered.
End of the program.
```

Additional Notes:

C++ Built-in Exception Classes



Source:
https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm

Example

```
7   int main ()
8   {
9       double *p;
10      int count = 0;
11      int size = pow(2,20);   // 1MB
12
13      try{
14          while (true){
15              p = new double[size];
16              count++;
17              cout << "Count=" << count << endl;
18          }
19      }
20      catch (exception& e){
21          cout << "Something bad happened!!!" << e.what() << endl;
22      }
23
24      return 0;
25  }
26
```

An exception of `bad_alloc` will be thrown by the command `new` when there is not enough memory left.

Example (continued)

```
Count=1741
Count=1742
Count=1743
Count=1744
Count=1745
Count=1746
Count=1747
Count=1748
Count=1749
Count=1750
Count=1751
Count=1752
Count=1753
Count=1754
Count=1755
Something bad happened!!!std::bad_alloc
```

An exception of `bad_alloc` was caught

Additional Notes:
Creating a New Exception Class by Extending the class `exception`

- An exception class can also be defined outside of a class by **extending** the built-in classes e.g., the **class** `exception`

Members of the class exception

```
1 class exception {
2 public:
3   exception () throw();
4   exception (const exception&) throw();
5   exception& operator= (const exception&) throw();
6   virtual ~exception() throw();
7   virtual const char* what() const throw();
8 }
```

Source:
http://www.cplusplus.com/doc/tutorial/exceptions/

Example

Creating a new exception class by extending the class `exception`

```
1   #include <iostream>
2   #include <exception>
3   using namespace std;
4
5
6   class DivideByZero:public exception{
7       public:
8           const char* what() const throw() {
9                   return "division by zero";
10          }
11  };
12
13  double divide(double a, double b){
14      DivideByZero e;
15      double c;
16      if (b == 0)
17          throw e;  // or simply call directly to the constructor, //throw DivideByZero()
18
19      return a/b;
20  }
```

Then , catching exceptions is done as usual

```
21  int main ()
22  {
23      double a, b, c;
24
25
26      cout << "Enter two numbers => ";
27      cin >> a >> b;
28
29      try{
30          c = divide(a,b);
31          cout << "Divide " << a << " by " << b << " is " << c << endl;
32      }
33      catch (exception& e){  // make sure to use &, to ensure polymorphism functions
34          cout << "Something bad happened!!!" << e.what() << endl;
35      }
36
37      return 0;
38  }
```

Function Templates

Introduction

- **Function template**: a pattern for a function that can work with many data types.
- When written, parameters are left for the data types.
- When called, compiler generates code for specific data types in function call.

Example 4a

```
template <class T>
T times10(T num)
{
    return 10 * num;
}
```

Template prefix

Generic type

Type parameter

What gets generated when **times10** is called with an **int**:	What gets generated when **times10** is called with a **double**:

`int times10(int num)`
`{`
`  return 10 * num;`
`}`
	`double times10(double num)`
	`{`
	`  return 10 * num;`
	`}`

Example 4b

```
template <class T>
T times10(T num)
{
   return 10 * num;
}
```

Call a template function in the usual manner:

```
int ival = 3;
double dval = 2.55;
cout << times10(ival); // displays 30
cout << times10(dval); // displays 25.5
```

Notes 1

- Function templates can be overloaded.

- Each template must have a unique parameter list.

```
template <class T>
T sumAll(T num) ...
template <class T1, class T2>
T1 sumAll(T1 num1, T2 num2) ...
```

Notes 2

- 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**.
- 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.

Class Templates

Introduction

- Classes can also be represented by templates.
- When a class object is created, type information is supplied to define the type of data members of the class.
- Unlike functions, classes are instantiated by supplying the type name (**`int`**, **`double`** , **`string`** , etc.) at object definition.

Example 5a

```
template <class T>
class Grade
{
    private:
        T score;
    public:
        Grade(T);
        void setGrade(T);
        T getGrade()
};
```

Example 5b

- Pass type information to class template when defining objects:

```
Grade<int> testList[20];
Grade<double> quizList[20];
```

- Use as ordinary objects once defined

Standard Template Library (STL)

Introduction to the Standard Template Library

- Standard Template Library (STL): a library containing templates for frequently used data structures and algorithms
- Not supported by many older compilers

Standard Template Library

- Two important types of data structures in the STL:
  - **containers**: classes that stores data and imposes some organization on it
  - **iterators**: like pointers; mechanisms for accessing elements in a container

Containers

- Two types of container classes in STL:
  - **sequence containers**: organize and access data sequentially, as in an array. These include `array`, `vector`, `deque`, and `list`
  - **associative containers**: use keys to allow data elements to be quickly accessed. These include `set`, `multiset`, `map`, and `multimap`.

Sequence Containers

Container Name	Description
`array`	A fixed-sized sequence container. Allow random access
`vector`	A resizable array. Values may be added to or removed from the end or middle of a `vector` . Allow random access
`deque`	Like a `vector` , but allows values to be added to or removed from the front. Allow random access
`list`	A doubly linked list of data elements. Values may be inserted to or removed from any position. No random access

Container's Methods

Methods	Description
`capacity()`	Returns the maximum number of elements the container can hold
`size()`	Returns the size or the number of elements
`=`	To assign new contents to a container
`[]`	To random access of an element from a container
`front(), back()`	Returns the first and the last element respectively
`at()`	Returns the element at the specified position

Container's Methods

Methods	Description
`empty()`	Returns a boolean indicating whether a container is empty or not.
`begin()`, `end()`,	Returns an iterator pointing to the first and last element, respectively
`rbegin()`, `rend()`	Similar to previous iterators, but work in reversed order.

Container's Methods

Methods	Description
`clear()`	Removes all elements
`insert()`	Insert an element to a specified position
`erase()`	Removes an element (or elements) at specified position.

Container's Methods

Methods	Description
`push_back()`, `push_front()`	Append an element before the first or after the last element, respectively.
`pop_back()`, `pop_front()`	Remove the first or last element, respectively.

list Methods

Source: cplusplus.com

splice	Transfer elements from list to list (public member function)
remove	Remove elements with specific value (public member function)
remove_if	Remove elements fulfilling condition (public member function template)
unique	Remove duplicate values (public member function)
merge	Merge sorted lists (public member function)
sort	Sort elements in container (public member function)
reverse	Reverse the order of elements (public member function)

Examples

Sequence Container	See source code
array	stl_array_examples.cpp
vector	stl_vector_examples.cpp
deque	stl_deque_examples.cpp
list	stl_list_examples.cpp

Performance Differences Between vectors, deques, and lists

- A **vector** is capable of quickly adding values to its end. Insertions at other points are not as efficient.
- A **deque** is capable of quickly adding values to its front and its end. Deques are not efficient at inserting values at other positions, however.
- A **list** is capable of quickly inserting values anywhere in its sequence. Lists do not, however, provide random access.

Associative Containers

Container Name	Description
`set`	Stores a set of keys. No duplicate values are allowed.
`multiset`	Stores a set of keys. Duplicates are allowed.
`map`	Maps a set of keys to data elements. Only one key per data element is allowed. Duplicates are not allowed.
`multimap`	Maps a set of keys to data elements. Many keys per data element is allowed. Duplicates are allowed.

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)

Iterator Type

Type	Description
Forward	Can only move forward in a container (uses the ++ operator).
Bidirectional	Can move forward or backward in a container (uses the ++ and -- operators).
Random-access	Can move forward and backward, and can jump to a specific data element in a container.
Input	Can be used with an input stream to read data from an input device or a file.
Output	Can be used with an output stream to write data to an output device or a file.

STL vector

Introduction to the STL `vector`

- A data type defined in the Standard Template Library (STL).
- Can hold values of any type:

```
vector<int> scores;
```

- Automatically adds space as more is needed – no need to determine size at definition
- Can use `[]` to access elements

Declaring Vectors

- You must **`#include<vector>`**
- Declare a vector to hold `int` element:

```
vector<int> scores;
```

- Declare a vector with initial size 30:

```
vector<int> scores(30);
```

- Declare a vector and initialize all elements to 0:

```
vector<int> scores(30, 0);
```

- Declare a vector initialized to size and contents of another vector:

```
vector<int> finals(scores);
```

Adding Elements to a Vector

- Use **`push_back`** member function to add element to a full array or to an array that had no defined size:

```
scores.push_back(75);
```

- Use **`size`** member function to determine size of a vector:

```
howbig = scores.size();
```

Removing Vector Elements

- Use **`pop_back`** member function to remove last element from vector:

```
scores.pop_back();
```

- To remove all contents of vector, use **`clear`** member function:

```
scores.clear();
```

- To determine if vector is empty, use `empty` member function:

```
while (!scores.empty()) ...
```

Other Useful Member Functions

Member Function	Description	Example
`at(elt)`	Returns the value of the element at position `elt` in the vector	`cout << vec1.at(i);`
`capacity()`	Returns the maximum number of elements a vector can store without allocating more memory	`maxelts = vec1.capacity();`
`reverse()`	Reverse the order of the elements in a vector	`vec1.reverse();`
`resize (elts,val)`	Add elements to a vector, optionally initializes them	`vec1.resize(5,0);`
`swap(vec2)`	Exchange the contents of two vectors	`vec1.swap(vec2);`

Other Useful Member Functions

Member Function	Description	Example
`back()`	Returns a reference to the last element in the vector.	`cout << vect.back() << endl;`
`begin()`	Returns an iterator pointing to the vector's first element.	`iter = vect.begin();`
`end()`	Returns an iterator pointing to the vector's last element.	`iter = vect.end();`
`front()`	Returns a reference to the vector's first element.	`cout << vector.front() << endl;`
`erase(iter)`	Causes the vector element pointed to by the iterator `iter` to be removed.	`vect.erase(iter);`