C++ Advanced Features

Trenton Computer Festival
May 1st & 2nd, 2004

Michael P. Redlich
Senior Research Technician
ExxonMobil Research & Engineering
michael.p.redlich@exxonmobil.com

Table of Contents

1 Introduction

C++ offers all of the advantages of object-oriented programming (OOP) by allowing the developer to create user-defined data types for modeling real world situations. However, the real power within C++ is contained in its features. Four main topics will be covered in this document:

- Overloaded operators
- Templates
- Exception handling
- Namespaces

There will also be an introduction to the Standard Template Library.

An example C++ application was developed to demonstrate the content described in this document and the ***Introduction to C++*** document. The application encapsulates sports data such as team name, wins, losses, etc. The source code can be obtained from **`http://www.tcf-nj.org/`** or **`http://www.redlich.net/tcf/`**.

2 Overloaded Operators

Operator overloading allows the developer to define basic operations (such as +,- ,×,/) for objects of user-defined data types as if they were built-in data types. For example, consider a simple string class:

```
class string
    {
    private:
        char *str;

    public:
        string(char const *s)
            {
            str = new char[strlen(s) + 1];
            strcpy(str,s);
            }
        ~string(void)
            {
            delete[] str;
            }
        char *getStr(void) const
            {
            return str;
            }
    };
```

Two string objects are created within an application that uses the simple string class:

```
string s1("Hello, world!");
string s2("Hello, out there!");
```

These objects are tested for equality using the C library function **`strcmp()`** and the **`getStr()`** function defined in the string class:

```
if(strcmp(s1.getStr(),s2.getStr()) == 0)
    // do this
else
    // do that
```

The conditional expression **`if(strcmp(s1.getStr(),s2.getStr()) == 0)`** is a bit lengthy and not as easy to read at first glance. However, writing an overloaded equality operator (**`operator==`**) defined as:

```
bool operator==(string const &str)
    {
    return(strcmp(getStr(),str.getStr()) == 0);
    }
```

for the string class allows for a comparison of the two string objects as if the string objects were built-in data types using the normal equality operator (**`==`**):

```
if(s1 == s2)
    // do this
else
    // do that
```

This is obviously much easier to read. The compiler interprets the statement **`s1 == s2`** as:

```
s1.operator==(s2)
```

`s1` is the object in control and **`s2`** is the argument passed into the overloaded equality operator. Because of the simplified syntax when working with user-defined types, overloaded operators are essentially considered "syntactic sugar." They are very attractive, but can be dangerous, as described in the next section.

Deep vs. Shallow Copy

The compiler automatically generates certain constructors and overloaded operators if they are not explicitly defined in a user-defined type. In general, the compiler automatically creates:

- A default constructor.
- A destructor
- An assignment operator (**`operator=`**)

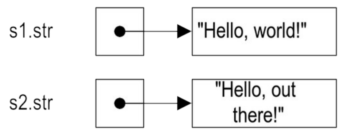
The compiler-generated assignment operator has the function signature:

```
T &T::operator=(T const &);
```

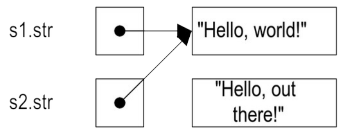
where **`T`** is a data type. For the string class above, the compiler will generate an assignment operator that might look like:

```
string &string::operator=(string const &s)
    {
    str = s.str;
    return *this;
    }
```

This function performs ***memberwise*** assignments. However, since **`str`** is of type **`char *`**, the memory for the character string must be released and reallocated. The compiler-generated assignment operator does **not** address the required memory handling and thus provides a ***shallow*** copy. When **`s1`** and **`s2`** were constructed, the pointers to their respective character string assignments were represented as:



With the shallow copy assignment operator, an assignment of **`s2 = s1`** would yield:

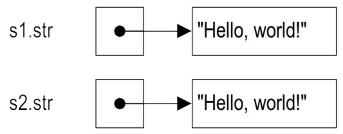


causing a memory leak.

A ***deep*** copy addresses the required memory handling. A user-defined assignment operator for the string class should look like:

```
string &string::operator=(string const &s)
    {
    delete[] str;
    str = new char[strlen(s) + 1];
    strcpy(str,s);
    return *this;
    }
```

Now an assignment of **`s2 = s1`** would yield:



correctly copying the string from **`s1`** to **`s2`**.

Operators That Can Be Overloaded

There are a variety of operators that can be overloaded. The table below contains the complete list:

`+`	`-`	`*`	`/`	`%`	`^`	`&`	`|`	`~`	`!`
`=`	`<`	`>`	`+=`	`-=`	`*=`	`/=`	`%=`	`^=`	`&=`
`|=`	`<<`	`>>`	`>>=`	`<<=`	`==`	`!=`	`<=`	`>=`	`&&`
`||`	`++`	`--`	`,`	`->*`	`->`	`()`	`[]`	`new`	`delete`

Operators That Cannot Be Overloaded

The following short list of operators **cannot** be overloaded:

`.`	`.*`	`::`	`?:`

These operators already have predefined meaning with regard to class objects. However, the standards committee has considered the possibility of overloading the conditional operator (**`?:`**).

Limitations

There are several restrictions for operator overloading:

- The meaning of an operator as it is applied to a built-in type cannot be changed.
- The number of operands for an operator as it is applied to a built-in type cannot be changed.
- Operator precedence and associativity cannot be changed.
- Personal operators cannot be defined.

3 Templates

One benefit of ***generic programming*** is that it eliminates code redundancy. Consider the following function:

```
void swap(int &first,int &second)
    {
    int temp = second;
    second = first;
    first = temp;
    }
```

This function is sufficient for swapping elements of type **`int`**. If it is necessary to swap two floating-point values, the same function must be rewritten using type **`float`** for every instance of type **`int`**:

```
void swap(float &first,float &second)
    {
    float temp = second;
    second = first;
    first = temp;
    }
```

The basic algorithm is the same. The only difference is the data type of the elements being swapped. Additional functions must be written in the same manner to swap elements of any other data type. This is, of course, very inefficient. The ***template*** mechanism in C++ was designed for generic programming.

There are two kinds of templates:

- Function templates
- Class templates

As their names imply class templates are applied to an entire class, and function templates are applied to individual functions.

To create a function or class template, the declaration and definition of that function or class must be preceded with the statement:

```
template <class T>
```

where **`<class T>`** is the ***template parameter list***. More than one parameter, separated by commas, may be specified. The **`T`** in **`<class T>`** can be any variable name; it just so happens that **`T`** is most commonly used. Also, the keyword **`class`** in **`<class T>`** implies that **`T`** must be of class type, however **`T`** can be any data type, built-in or user-defined. Defining a function template to swap elements of any data type will eliminate the multiple versions of **`swap`**:

```
template <class T>
void swap(T &,T &); // declaration

template <class T>
void swap(T &first,T &second) // definition
    {
    T temp = second;
    second = first;
    first = temp;
    }
```

A template ***specialization*** is the specific use of the function or class for a particular data type. For example:

```
swap<int>(1,2);
swap<float>(1.7,3.5);
swap<char *>("Mets","Jets");
swap<char>('a','b');
```

are four different specializations for **`swap`**. The compiler generates the necessary code for each specialization.

In most cases, the compiler can deduce the data types of the arguments being passed in a function template. In the case of:

```
swap<int>(1,2);
```

the compiler can deduce that arguments **`1`** and **`2`** are integers. This eliminates the need to write out the data type explicitly within angle brackets. Therefore, the above function call can be rewritten as:

```
swap(1,2);
```

Default Arguments

A template parameter list can contain default arguments. They are defined just like default arguments in class constructors. For example:

```
template < class Key,class T,class Compare = less<Key> >
class multimap
    {
    ...
    }
```

is the definition for the Standard Template Library container **`multimap`**. The third parameter, **`Compare = less<Key>`** specifies a default argument of **`less<Key>`** which is an ascending ordering of the data stored within **`multimap`**. Therefore, an instantiation of **`multimap`** declared as:

```
typedef multimap<int,string> myMap;
```

assumes that **`less<int>`** is the desired ordering of the data. If a different ordering is desired, e.g., **`greater<int>`**, it must be specified in the declaration:

```
typedef multimap< int,string,greater<int> > myMap;
```

Notice the space in between **`greater<int>`** and the closing angle bracket (**`>`**) at the end of the template parameter list. This is significant because the compiler will interpret the double angle brackets (**`>>`**) in **`<int>>`** (no space) as the right shift operator.

Default arguments in template parameter lists are very new (as opposed to default arguments for constructors) to the C++ standard. Some compilers do not support this feature. Therefore, if a template class has default parameters, the argument for that parameter must be supplied even if that default parameter is the desired choice.

4 Exception Handling

Detecting and handling errors within an application has traditionally been implemented using return codes. For example, a function may return zero on success and non-zero on failure. This is, of course, how most of the standard C library functions are defined. However, detecting and handling errors this way can become cumbersome and tedious especially in larger applications. The application's program logic can be obscured as well.

The ***exception handling*** mechanism in C++ is a more robust method for handling errors than fastidiously checking for error codes. It is a convenient means for returning from deeply nested function calls when an exception is encountered. Exception handling is implemented with the keywords **`try`**, **`throw`**, and **`catch`**. An exception is raised with a ***throw-expression*** at a point in the code where an error may occur. The throw-expression has the form:

```
throw T;
```

where **`T`** can be any data type for which there is an exception handler defined for that type. A ***try-block*** is a section of code containing a throw-expression or a function containing a throw-expression. A ***catch clause*** defined immediately after the try-block, handles exceptions. More than one catch clause can be defined. For example:

```
int f(void)
    {
    FILE *fp = fopen("filename.txt","rt");
    try
        {
        if(fp == NULL) // file could not be opened
            throw 1;
        g(-1);
        }
    catch(int e) // catches thrown integers
        {
        cout << "Could not open input file" << endl;
        }
    catch(string const str) // catches thrown strings
        {
        cout << str << endl;
        }
    ...
    }

int g(int n)
    {
    if(n < 0)
        throw "n is less than zero";
    ...
    }
```

The C++ library contains a defined class hierarchy for exceptions:

`exception`
- **`logic_error`** (client program errors)
  - **`domain_error`**
  - **`invalid_argument`**
  - **`length_error`**
  - **`out_of_range`**
- **`runtime_error`** (external errors)
- **`bad_alloc`** (memory allocation errors)
- **`bad_cast`** (**`dynamic_cast`** with reference errors)
- **`bad_exception`** (used with **`unexpected()`** function)
- **`bad_typeid`** (**`typeid`** with null errors)

where **exception** is the base class. The remaining classes inherit from **exception** as indicated by the indentation. These exceptions can be used for specific errors. The file open check in function **f()** above can be rewritten using the **runtime_error** exception:

```
int f(void)
    {
    FILE *fp = fopen("filename.txt","rt");
    try
        {
        if(fp == NULL) // file could not be opened
          throw runtime_error("Could not open input file");
        g(-1);
        }
    catch(runtime_error &re)
        {
        cout << re.what() << endl;
        }
    catch(string const str)
        {
        cout << str << endl;
        }
    ...
    }
```

The function **what()** as shown in the **re.what()** statement above is a virtual constant member function defined in **exception**. It returns type **char const *** which is the null-terminated constant character string that is stored in the current **runtime_error** object when it was constructed with the **throw** statement.

One of the main features of exception handling is that destructors are invoked for all live objects as the stack of function calls "unwinds" until an appropriate exception handler (i.e., catch clause) is found.

Exceptions should be thrown for things that are ***truly*** exceptional. They should not be thrown to indicate special return values or within copy constructors or assignment operators.

5 Namespaces

A ***namespace*** is primarily used to reduce potential global name conflicts that may arise when using various header files from different sources. For example, one header file named **baseball.h** may contain:

```
// baseball.h
...
int strike = 0;
...
```

while another header file named **bowling.h** may contain:

```
// bowling.h
...
bool strike = false;
...
```

If both of these header files are included in an application as in:

```
// sportsapp.cpp

#include baseball.h
#include bowling.h // error
...
```

the compiler will generate an error because **`strike`** was already declared in **`baseball.h`**.

One way of resolving this problem is to create a class in each header file. For example:

```
// baseball.h

class baseball
    {
    private:
         ...
         int strike = 0;
         ...
    public:
         ...
    };

// bowling.h

class bowling
    {
    private:
         ...
         bool strike = false;
         ...
    public:
         ...
    };
```

However, classes and objects instantiated from those classes are intended to be user-defined data types that model real-world situations. Simply storing a series of variable names into a class for avoiding a name conflict makes no sense and should be avoided.

Namespaces solve this problem. The two header files can be rewritten using namespaces:

```
// baseball.h

namespace baseball {
...
int strike = 0;
...
}

// bowling.h

namespace bowling {
...
bool strike = false;
...
}
```

Both **strike**s along with any other variables are now declared under their own namespaces named **baseball** and **bowling**. Outside of their namespace definitions, each **strike** must be referred by its ***fully qualified member names***. This is similar to C++ classes when referring to member names outside of a class definition. A fully qualified member name is comprised of:

- A namespace name
- A scope resolution operator (**::**)
- A member name (variable)

Therefore, **baseball::strike** and **bowling::strike** are the fully-qualified member names for each **strike**.

Some noteworthy differences between namespace and class definitions include:

- A semicolon after the ending curly brace is an error, however some compilers are forgiving about enforcing this syntax rule.
- Access specifiers (**public**, **protected**, and **private**) are not allowed in namespace definitions.
- Namespaces are "open," i.e., namespace definitions do not have to be contiguous. They can be in separate parts of the same file or located in different files as well. The standard namespace defined in the standard C++ library is implemented using this technique.

Aliases

Writing the fully qualified member name for a particular variable in an application can become extremely tedious if the variable is used numerous times. A ***namespace alias*** can be defined to eliminate some of the additional typing. For example:

```
namespace bb = baseball;
...
bb::strike = 3; // yer out!
...
```

refers to the fully qualified **baseball::strike**.

Using-Directives

A ***using-directive*** provides access to **all** members of a namespace without having to write the fully qualified member names. It has the form:

```
using namespace N;
```

where **N** is the name of a defined namespace. Therefore, stating:

```
using namespace baseball;
using namespace bowling;
```

will allow reference to all members of each namespace.

Using-Declarations

A ***using-declaration*** provides access to **individual** members of a namespace without having to write the fully-qualified member names. It has the form:

```
using N::m
```

where **N** is the name of a defined namespace and **m** is the member name. Therefore, stating:

```
using baseball::strike
using bowling::strike
```

will only allow reference to each **strike**.

The C++ Standard

The C++ standard now uses header file names without the familiar **.h** suffix. For example, what was once written:

```
#include <iostream.h>
```

is now:

```
#include <iostream>
```

These new headers include a common defined namespace named **std**. All C++ standard library components are declared within this namespace. These library components will be available if the statement:

```
using namespace std;
```

is included within an application.

The **std** namespace has been added to the C library headers as well. The format for those header filenames is slightly different. Along with the missing **.h** suffix, a **c** precedes the name of the header file. So:

```
#include <stdio.h>
```

becomes

```
#include <cstdio>
```

Not all compilers support this new format for the C library headers.

6 Introduction to the Standard Template Library (STL)

The Standard Template Library (STL) is as subset of the entire C++ standard. Hewlett-Packard (HP) first introduced it in 1994. Alex Stepanov (now at Silicon Graphics, Inc.), Meng Lee, and David R. Musser (from Rensselaer Polytechnic Institute) developed the STL at HP Labs.

There are three main parts to the STL:
- Containers
- Iterators
- Algorithms

Containers

A ***container*** is a data structure that contains a sequence of elements. There are a variety of containers available in the STL.

Sequential containers organize elements linearly. The sequential containers are:
- **vector** – an indexed data structure (like an array) that supports random access to the elements with efficient insertion only at the end.
- **deque** (pronounced "deck") – like **vector**, except efficient insertion/deletion from the beginning and the end.
- **list** – a double linked list that supports bi-directional access to the elements.
- **string** – a data structure that encapsulates and maintains character strings.

Sorted associative containers organize objects based on a key for quick retrieval of data. The sorted associative containers are:
- **set** – a data structure that maintains unique elements as an ordered representation.
- **multiset** – like **set**, except the elements do not have to be unique.
- **map** – similar to **vector** and **deque**, except the elements are associated with unique key values.
- **multimap** – like **map**, except the key values do not have to be unique.

Other containers in the STL include:

- **`stack`** – a data structure that inserts/deletes elements in a last-in, first-out (LIFO) manner.
- **`queue`** – a data structure that inserts/deletes elements in a first-in, first-out (FIFO) manner.

A container is primarily chosen by how well it can perform certain operations such as:

- Add elements to the container
- Remove elements from the container
- Rearrange elements within the container
- Inspect elements within the container

The table below summarizes the complexity of these containers in Big-O notation:

	`Vector`	**`deque`**	**`list`**	**`set/map`**
Insert/erase	$O(n)$	$O(n)$	$O(1)$	$O(n \log_2 n)$
Prepend	$O(n)$*	$O(1)$	$O(1)$	$O(n \log_2 n)$*
find(val)	$O(n)$*	$O(n)$*	$O(n)$*	$O(n \log_2 n)$
$X[n]$	$O(1)$	$O(1)$	$O(n)$*	$O(n)$*
No. of Pointers	0	1	2	3

* not directly supported by supported by member functions.

Iterators

An ***iterator*** is a generalization of C/C++ pointers used to access elements within an ordered sequence. An ordered sequence can be an STL container or an array of elements. Iterators are considered the "glue" that pastes together containers and algorithms in the STL.

To conform to the C++ standard, all of the STL containers must provide their own iterators. An iterator from a container or sequence may be declared using either of the following statements:

```
class name::iterator;
class name::const_iterator;
```

STL containers must also provide iterators to the beginning and end of their collections. These may be accessed using the class members, **`begin()`** and **`end()`**. For example,

```
typedef vector<int>::iterator iterator;
vector<int> v(10);
iterator first = v.begin();
iterator last = v.end()
while(first != last)
    {
    cout << *first << "\n";
    ++first;
    }
```

assigns the iterator pointing to the first element of the vector to **`first`** and assigns an iterator pointing to the last element of the vector to **`last`**. The **`while`** loop accesses each element of the vector by dereferencing and incrementing **`first`**.

Every iterator type guarantees that there is an iterator value that points past the last element of a corresponding container. This value is called the ***past-the-end*** value. No guarantee is made that this value is dereferencable.

There are five types of iterators:

- Input – read only access to elements in a container, i.e., access stored values.
- Output – write only access to elements in a container, i.e., generate a new sequence of values.
- Forward – read/write access to elements within a container, but can only move forward through the sequence.
- Bi-directional – same as a forward iterator, but has the ability to move backwards through the sequence as well.
- Random-Access – same as a bi-directional iterator, but can perform various pointer arithmetic and subscripting.

Algorithms

The STL provides ***generic algorithms*** for performing various operations on containers such as searching, sorting, and transforming. The list of algorithms is quite large. They can be grouped in terms of operation type or by mutating/non-mutating function.

The table below is a small sampling of the generic algorithms grouped by mutating/non-mutating function:

Non-mutating algorithms	**Mutating algorithms**
• `binary_search`	• `copy`
• `count`	• `fill`
• `equal`	• `generate`
• `find`	• `remove`
• `max_element`	• `reverse`
• `min_element`	• `sort`

7 References

- P.J. Plauger. "The Standard Template Library", *C/C++ Users Journal*, December 1995.
- P.J. Plauger. "Associative Containers", *C/C++ Users Journal*, April 1997.
- Dan Saks. "An Introduction to Namespaces", *C/C++ Users Journal*, January 1998.
- Dan Saks. "Generic Programming in C++ and the STL", *Software Development '98 East*, August 1998.
- Dan Saks. "Namespaces and Their Impact on C++", *Software Development '99 East*, November 1999.
- Chuck Allison. "C++ Exceptions in Depth", *Software Development '99 East*, November 1999.