Introduction to C++

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

This document is an introduction to the C++ programming language. C++ is an extension of the C programming language, which means that all of the C library functions can be used in a C++ application. C++ was finally standardized in June 1998, but its history can be traced back almost 20 years. This document will begin with how C++ has evolved over the years and introduce some of the language's features. Since C++ is an object-oriented programming language, it is important to understand the concepts of object-oriented programming. The remainder of this document will discuss object-oriented programming, C++ classes and how they are implemented, introduce some new keywords, and mention some basic I/O differences between C and C++.

An example C++ application was developed to demonstrate the content described in this document and the ***C++ Advanced Features*** 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 Evolution of C++

C++ was originally known as "C with Classes." Bjarne Stroustrup from AT&T Laboratories developed the language in 1980. Bjarne needed to add speed to simulations that were written in Simula-67. Since C was the fastest procedural language, he decided to add classes, function argument type checking and conversion, and other features to it. Around the 1983/1984 time frame, virtual functions and operator overloading were added to the language, and it was decided that "C with Classes" be renamed to C++. The language became available to the public in 1985 after a few refinements were made. Templates and exception handling were added to C++ in 1989. The Standard Template Library (STL) was developed by Hewlett-Packard in 1994, and was ultimately added to the draft C++ standard. The final draft was accepted by the X3J16 subcommittee in November 1997, and received final approval from the International Standards Organization (ISO) in June 1998 to officially declare C++ a standard.

3 Some Features of C++

C++ is an object-oriented programming (OOP) language. It offers all of the advantages of OOP by allowing the developer to create user-defined types for modeling real world situations. However, the real power within C++ is contained in its features. Since the scope of this document is strictly introductory, this chapter only briefly describes some of the features built-in to the language. A detailed overview of these features can be found in the ***C++ Intermediate and Advanced Features*** document.

Pass-By-Reference

Arguments passed to functions are strictly ***pass-by-value*** in C. That is, only a **copy** of the argument is passed to a function. If the argument's value is changed within the function that received it, the change is not saved when the application returns to the point of the function call. Large data structures passed as arguments will be copied as well. A pointer to a data structure is allowed in a function parameter list, but the argument name must be preceded with the address operator (**`&`**) when it is passed to the function. Inadvertently omitting the address operator in this case usually resulted with a run-time error and core dump.

With ***pass-by-reference*** parameter passing, only the **address** of the variable is passed. Any changes to the argument's value will be saved when the application returns to the point of the function call. Pass-by-reference parameter passing is nothing new to some programming languages such as Pascal. This feature was added to C++ so that references to data types (user-defined or built-in) could be specified in function parameter lists. This allows passing a complex data structure as an argument to a function without having to precede it with the address operator.

Operator Overloading

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, a conditional expression such as:

```
if(s1 == s2)
    {
    ...
    }
```

is much easier to read than

```
if(strcmp(s1.getStr(),s2.getStr()) == 0)
    {
    ...
    }
```

Operator overloading is often referred to as "syntactic sugar."

Generic Programming

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, then the same function must be rewritten using type **`float`** for every instance of type **`int`**. 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 was designed for generic programming.

Exception Handling

The ***exception handling*** mechanism 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. 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 is found.

Namespaces

A ***namespace*** is a mechanism that avoids global variable name conflicts that may arise due to using various libraries from different sources. All library functions in the C++ standard are defined in a namespace called **`std`**.

Default Arguments

Default arguments can be specified within parameter lists of class constructors and templates. For example, consider the following class constructor code fragment:

```
Sports::Sports(string str,int win,int loss,int tie = 0)
    {
    ...
    }
```

Only the first three parameters of the class constructor require arguments because parameter **`tie`** has a default value of **`0`**. An object created this way might look like:

```
Sports sp("Mets",94,68);
```

If a different value for **`tie`** is required, the fourth argument must be supplied to override the default value. For example:

```
Sports sp("Jets",8,8,0);
```

will assign the value **`0`** to **`tie`**. Most compilers support default arguments for class constructors however default arguments for templates is very new to the standard, and are not supported by all compilers.

4 Object-Oriented Programming

Please note this chapter is the same as the corresponding Object-Oriented Programming *chapter of the* ***Introduction to Java*** *document.*

Programming Paradigms

There are two programming paradigms:

- Procedure-Oriented
- Object-Oriented

Examples of procedure-oriented languages include:

- C
- Pascal
- FORTRAN

Examples of object-oriented languages include:

- C++
- SmallTalk
- Eiffel.

A side-by-side comparison of the two programming paradigms clearly shows how object-oriented programming is vastly different from the more conventional means of programming:

Procedure-Oriented Programming	**Object-Oriented Programming**
• Top Down/Bottom Up Design	• Identify objects to be modeled
• Structured programming	• Concentrate on what an object does
• Centered around an algorithm	• Hide how an object performs its tasks
• Identify tasks; how something is done	• Identify an object's behavior and attributes

Some Object-Oriented Programming (OOP) Definitions

An ***abstract data type*** (ADT) is a user-defined data type where objects of that data type are used through provided functions without knowing the internal representation. For example, an ADT is analogous to, say an automobile transmission. The car's driver knows how to operate the transmission, but does not know how the transmission works internally.

The ***interface*** is a set of functions within the ADT that allow access to data.

The ***implementation*** of an ADT is the underlying data structure(s) used to store data.

It is important to understand the distinction between a ***class*** and an ***object***. The two terms are often used interchangeably, however there are noteworthy differences. Classes will be formally introduced later in this document, but is mentioned here due to the frequent use of the nomenclature in describing OOP. The differences are summarized below:

Class	**Object**
• Defines a model	• An instance of a class
• Declares attributes	• Has state
• Declares behavior	• Has behavior
• An ADT	• There can be many ***unique*** objects of the same class

Main Attributes of OOP

There are four main attributes to object-oriented programming:
- Data Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism

Data Encapsulation

Data encapsulation separates the implementation from the interface. User access to data is only allowed through a defined interface. Data encapsulation combines information and an object's behavior.

Data Abstraction

Data abstraction defines a data type by its functionality as opposed to its implementation. For example, the protocol to use a double-linked list is made public through the supplied interface. Knowledge of the implementation is unnecessary and therefore hidden.

Inheritance

Inheritance is a means for defining a new class as an extension of a previously defined class. A ***derived*** class **inherits** all attributes and behavior of a ***base*** class, i.e., it provides access to all data members and member functions of the base class, and allows additional members and member functions to be added if necessary.

The base class and derived class have an "is a" relationship. For example,
- Baseball (a derived class) ***is a*** Sport (a base class)
- Pontiac (a derived class) ***is a*** Car (a base class)

Polymorphism

Polymorphism is the ability of different objects to respond differently to virtually the same function. For example, a base class provides a function to print the current contents of an object. Through inheritance, a derived class can use the same function without explicitly defining its own. However, if the derived class must print the contents of an object differently than the base class, it can override the base class's function definition with its own definition. In order to invoke polymorphism, the function's return type and parameter list must be identical. Otherwise, the compiler ignores polymorphism.

Polymorphism is derived from the Greek meaning "many forms." It is a mechanism provided by an object-oriented programming language, rather than a programmer-provided workaround.

Advantages of OOP

- The implementation of an ADT can be refined and improved without having to change the interface, i.e., existing code within an application doesn't have to be modified to accommodate changes in the implementation.
- Encourages modularity in application development.
- Better maintainability of code yielding less code "spaghetti."
- Existing code can be reused in other applications.

5 Some C++ Keywords

The keywords defined below are just a subset of the complete C++ keyword list.

- **`class`** – used for declaring/defining a class.
- **`new`** – allocate storage on the free store (heap).
- **`delete`** – deallocate the storage on the free store.
  - **`new`** and **`delete`** are more robust than the C library functions **`malloc`** and **`free`**.
- **`inline`** – used for inline member functions.
- **`private`**/**`protected`**/**`public`** – access specifiers used for data hiding which is a means of protecting data.
  - **`private`** – not visible outside of the class.
  - **`protected`** – like private except visible only to derived classes through inheritance.
  - **`public`** – visible to all applications.
- **`try`**/**`throw`**/**`catch`** – used in exception handling.
- **`friend`** – declares a class will full access rights to private and protected members of an outside class without being a member of that class.
- **`explicit`** – prevents implicit conversion of a data type to a particular class that may lead to unexpected surprises:
  - **`array::array(size_t n);`** creates an array with **n** elements.
  - **`float max(array const &a);`** a function that uses the array data type.
  - **`max(m);`** where **m** is an integer inadvertently passed to the function. A new array of **m** elements will be implicitly created automatically, which is not what was intended.
- **`virtual`** – a declaration specifier that invokes polymorphism on a function.
- **`bool`**/**`false`**/**`true`** – used for Boolean logic.
  - **`bool`** – new data type that can only accept the values **`true`** and **`false`**.
  - **`false`** – numerically zero.
  - **`true`** – numerically one.

6 Basic I/O Differences Between C and C++

Sending Formatted Output to the Standard Output (**`stdout`**) Device

In C, the library function **`printf()`** is available to display formatted output to **`stdout`**:

```
printf("%s%2d\n","The answer is: ",var);
```

Since C++ is an extension of C, the **`printf()`** function can still be used in a C++ application. However, the overloaded left shift operator (**`<<`**) directed toward the C++ library function **`cout`** provides an easier means of sending formatted output:

```
cout << "The answer is: " << var << "\n";
```

Obtaining Formatted Input from the Standard Input (**`stdin`**) Device

In C, the library function **`scanf()`** is available to obtain formatted input from **`stdin`**:

```
scanf("%2d",&var);
```

Again, the **`scanf()`** function can be used in a C++ application, but the overloaded right shift operator (**`>>`**) directed away from the C++ library function **`cin`** provides an easier means of obtaining formatted input:

```
cin >> var;
```

7 C++ Classes

As mentioned earlier, a C++ class is a user-defined ADT. It encapsulates a data type and any operations on it. A class is also an extension of a C structure, which is a collection of one or more variables defined under a single name. The biggest difference between the two is the default access to data members and member functions. By default, data members and member functions in a class are **private**, where they are **public** in a structure. An ***abstract class*** is one that contains at least one pure virtual member function.

A basic C++ class as well as a structure usually contains the following elements:

- Constructor(s) – creates an object.
- Destructor – destroys an object.
- Data members – object attributes.
- Member functions (methods) – operations on the attributes.

Each one of these is demonstrated in a simple example:

```
class Sports
    {
    private:
        // private data members:
        char *team;
        int win;
        int loss;

    public:
        // constructor and destructor declarations:
        Sports(char *,int,int);    // primary constructor
        ~Sports(void);             // destructor

        // public member functions:
        char *getTeam(void) const // constant member function
            {
            return team;
            }
        int getWin(void) const // constant member function
            {
            return win;
            }
        void setWin(int w)
            {
            win = w;
            }
        int getLoss(void) const // constant member function
            {
            return loss;
            }
        void setLoss(int l)
            {
            loss = l;
            }
    };
```

```
// constructor and destructor definitions:
Sports::Sports(char *str,int w,int l)
    {
    team = new char[strlen(str) + 1]; // allocate storage for type char *
    strcpy(team,str);
    setWin(w);
    setLoss(l);
    }

Sports::~Sports(void)
    {
    delete[] team; // deallocate storage; note use of '[]'
    }
```

C++ comments begin with a double slash (**//**). Anything after a double slash until the end of the current line is considered a comment by the compiler. C comments (**/* ... */**) can still be used in a C++ application as well.

Note that constructors and destructors have the same name as the class and have **no** return type. The destructor is declared/defined with a tilde (**~**) in front of its name.

Also note the use of the scope resolution operator (**::**) for the constructor and destructor definitions. They were defined outside of the class, and therefore required their ***fully-qualified member names*** so the compiler knows that these definitions belong to the **Sports** class.

More than one constructor can be written for a particular class. The different constructor types are:

- Default constructors
- Primary constructors
- Copy constructors

Default Constructors

A ***default constructor*** creates objects with specified default values. A default constructor added to **Sports** might look like:

```
Sports(void); // declaration

Sports::Sports(void) // definition
    {
    team = new char[8];
    strcpy(team,"No team");
    setWin(0);
    setLoss(0);
    }
```

The compiler will automatically generate a default constructor if one is not explicitly defined.

Primary Constructors

A ***primary constructor*** creates objects with the argument values passed in the constructor parameter list. More than one primary constructor may be defined for a class. The primary constructor in **Sports** is declared as:

```
Sports(char *,int,int);   // primary constructor
```

If the application requires, say, a floating-point value in the parameter list in place of one of the integer values, then a second constructor can be declared as:

```
Sports(float,char *,int); // another primary constructor
```

Note that the order of the parameter list has changed from the first primary constructor. This is to avoid ambiguity between the two constructor declarations and definitions. The compiler will generate an error message about ambiguity between constructor parameter lists if the order of the parameters is similar.

Copy Constructors

A ***copy constructor*** creates a copy of an object using the current object as a parameter. A copy constructor added to **`Sports`** might look like:

```
Sports(Sports const &); // declaration

Sports::Sports(Sports const &sp) // definition
    {
    team = new char[strlen(sp.getTeam()) + 1];
    strcpy(team,sp.getTeam());
    setWin(sp.getWin());
    setLoss(sp.getLoss());
    }
```

8 Class Instantiation

Classes can be instantiated both statically and dynamically. For example, consider a **`Baseball`** class that is derived from **`Sports`**. It has the following constructor declaration:

```
Baseball(string,int,int);
```

Dynamic Instantiation

An object of type **`Baseball`** is ***dynamically*** instantiated using operator **`new`** as shown in the following statement:

```
Baseball *bball = new Baseball("Mets",94,68);
```

This statement declares **`bball`** as a pointer to an object of type **`Baseball`** containing the values **`"Mets"`**, **`94`**, and **`68`**. Once the object is created, any public member functions are called using the name of the pointer to the object and the pointer indirection operator (**`->`**). For example,

```
bball->getWin();
```

calls the function **`getWin()`**. Since the object is a pointer, it must be deleted to free memory. This is accomplished using operator **`delete`** as shown in the following statement:

```
delete bball;
```

The destructor is invoked at this point.

Static Instantiation

An object of type **`Baseball`** is ***statically*** instantiated using the following statement:

```
Baseball bball("Mets",94,68);
```

This statement declares **bball** as an object of type **Baseball** containing the values **"Mets"**, **94**, and **68**. Once the object is created, any public member functions are called using the name of the object and the structure dot operator (**.**). For example,

```
bball.getWin();
```

calls the function **getWin()**. The object remains alive until the scope in which it was created is closed. The destructor is invoked and the object is deleted.

9 Popular Compilers

Some of the more commonly used compilers are listed below:

- Borland C++ 5.02
- Borland C++ Builder 4.0
  - **http://www.borland.com/**
- Microsoft Visual C++ +6.0
  - **http://www.microsoft.com/**
- Watcom C++ 11.0
- Metrowerks C++ (Mac)
  - **http://www.metrowerks.com/**
- g++ (UNIX)
  - **http://www.gnu.com/**

10 References for Further Reading

The references listed below are only a small sampling of resources where further information on C++ can be obtained:

- C & C++ Code Capsules (*book*)
  - Chuck Allison
  - ISBN 0-13-591785-9
  - **http://www.freshsources.com/**
- C/C++ Users Journal (*monthly periodical*)
  - **http://www.cuj.com/**
- The Annotated C++ Reference Manual (*book*)
  - Margaret Ellis and Bjarne Stroustrup
  - ISBN 0-201-51459-1
- 1997 C++ Public Review Document (*latest available on-line C++ standard documentation*)
  - **http://www.maths.warwick.ac.uk/cpp/pub/wp/html/cd2/**