Crossroads
The ACM Student Magazine

Welcome! This is the first installment in a series called "Objective Viewpoint" that will teach you about C++ and Java. You can go to an index of the series by clicking on the banner immediately above, or you can follow the tour at the bottom of this document. Enjoy!

An Introduction to C++

by ***Saveen Reddy*** and ***G. Bowden Wise***

Welcome to the inaugural edition of the ObjectiveViewPoint column! Here we will touch on many aspects of object-orientation. The word object has surfaced in more ways than you can count. There are OOPLs (Object-Oriented Programming Languages) and OODBs (Object-Oriented Databases), OOA (object-oriented analysis), and OOD (object-oriented design). We are sure you can come up with some OOisms of your own.

Our goal in this column is to explore object-orientation through practical object-oriented programming. This time, we look at C++, but in the future we will explore other areas of object-orientation. Learning an object-oriented language-a whole new way of programming-will pave the way for many exciting topics down the road.

Our intended audience consists of humble beginners to seasoned hackers. We assume that you have programmed in at least one procedural language, such as C or Pascal. Even if you are familiar with C++, please stay with us, you may learn some interesting new language features. Also, we will illustrate our points with many self-contained

examples that you may later wish to incorporate into your own programs.

C++: A Historical Perspective

We begin our journey of C++ with a little history. C, the predecessor to C++, has become one of the most popular programming languages. Originally designed for systems programming, C enables programmers to write efficient code and provided close access to the machine. C compilers, found on practically every Unix system, are now available with most operating systems.

During the 1980s and into the 1990s, an explosive growth in object-oriented technology began with the introduction of the Smalltalk language. Object-Oriented Programming (OOP) began to replace the more traditional structured programming techniques. This explosion led to the development of languages which support programming with objects. Many new object-oriented programming languages appeared: Object-Pascal, Modula-2, Mesa, Cedar, Neon, Objective-C, LISP with the Common List Object System (CLOS), and, of course, C++. Although many of these languages appeared in the 1980s, many ideas of OOP were taken from Simula-67. Yes! OOP has been around since 1967.

C++ originated with Bjarne Stroustrop. In the simplest sense, if not the most accurate, we can consider it to be a better C. Although it is not an entirely new language, C++ represents a significant extension of C abilities. We might then consider C to be a subset of C++. C++ supports essentially every desirable behavior and most of the undesirable ones of its predecessor, but provides general language improvements as well as adding OOP capability. Note that using C++ does not imply that your are doing OOP. C++ does not force you to use its OOP features. You can simply create structured code that uses only C++'s non-OOP features.

C++: A Better C

The designers of C++ wanted to add object-oriented mechanisms without compromising the efficiency and simplicity that made C so popular. One of the driving principles for the language designers was to hide complexity from the programmer, allowing her to concentrate on the problem at hand.

Because C++ retains C as a subset, it gains many of the attractive features of the C language, such as efficiency, closeness to the machine, and a variety of built-in types.

A number of new features were added to C++ to make the language even more robust, many of which are not used by novice programmers. By introducing these new features here, we hope that you will begin to use them in your own programs early on and gain their benefits. Some of the features we will look at are the role of constants, inline expansion, references, declaration statements, user defined types, overloading, and the free store.

Most of these features can be summarized by two important design goals: strong compiler type checking and a user-extensible language.

By enforcing stricter type-checking, the C++ compiler makes us acutely aware of data types in our expressions. Stronger type checking is provided through several mechanisms, including: function argument type checking, conversions, and a few other features we will examine below.

C++ also enables programmers to incorporate new types into the language, through the use of classes. A class is a user-defined type. The compiler can treat new types as if they are one of the built-in types. This is a very powerful feature. In addition, the class provides the mechanism for data abstraction and encapsulation, which are key to object-oriented programming. As we examine some of the new features of C++ we will see these two goals resurface again and again.

A NEW FORM FOR COMMENTS.

It is always good practice to provide comments within your code so that it can be read and understood by others. In C, comments were placed between the tokens ***/ **** and **** /*** like this:

```
/* This is a traditional C comment */
```

C++ supports traditional C comments and also provides an easier comment mechanism, which only requires an initial comment delimiter:

```
// This is a C++ comment
```

Everything after the ***/ /*** and to the end of the line is a comment.

THE CONST KEYWORD.

In C, constants are often specified in programs using **#define** . The **#define** is essentially a macro expansion facility, for example, with the definition:

```
#define PI 3.14159265358979323846
```

the preprocessor will substitute **3.14159265358979323846** wherever **PI** is encountered in the source file. C++ allows any variable to be declared a constant by adding the const keyword to the declaration. For the **PI** constant above, we would write:

```
const double PI = 3.14159265358979323846;
```

A **const** object may be initialized, but its value may never change. The fact that an object will never change allows the compiler to ensure that constant data is not modified and to generate more efficient code. Since each const element also has an associated type, the compiler can also do more explicit type checking.

A very powerful use of const is found when it is combined with pointers. By declaring a ``pointer to const'', the pointer cannot be used to change the pointed-to object. As an example, consider:

```
int i = 10;
 const int *pi = &i;
 *pi = 15;
// Not allowed! pi is a const pointer!
```

It is not possible to change the value of i through the pointer because ***pi** is constant. A pointer used in this way can be thought of as a read-only pointer; the pointer can be used to read the data to which it points, but the data cannot be changed via the pointer. Read-only pointers are often used by class member functions to return a pointer to private data stored within the class. The pointer allows the user to read, but not change, the private data.

Unfortunately, the user can still modify the data pointed at by the read-only pointer by using a type cast. This is called ``casting away the const-ness''. Using the above example, we can still change the value of i like this:

```
// Cast away the constness of the pi pointer and  modify i
```

```
*((int*) pi) = 15;
```

By returning a const pointer we are telling users to keep their hands off of internal data. The data can still be modified, but only with extra work (the type cast). So, in most cases users will realize they are not to modify that data, but can do so at their own risk.

There are two ways to add the const keyword to a pointer declaration. Above, when const comes before the ***** , what the pointer points to is constant. It is not possible to change the variable that is pointed to by the pointer. When when const comes after the *****, like this:

```
int i = 10;
int j = 11;
int* const ptr = &i;
// Pointer initialized to point to i
```

the pointer itself becomes constant. This means that the pointer cannont be changed to point to some other variable after it has been initialized. In the above example, the pointer ptr must always point at the variable **i**. So, statements such as:

```
ptr = &j;
// Not allowed, since the pointer is const!
```

are not allowed and are caught by the compiler. However, it is possible to modify the variable that the pointer points to:

```
*ptr = 15;
// This is ok, what is pointed at is not const
```

If we want to prevent modification of what the pointer points to and prevent the value of the pointer from being changed, we must provide a **const** on both sides of the ***** like this:

```
const int * const ptr = &i;
```

Remember that adding **const** to a declaration simply invokes extra compile time type

checking; it does not cause the compiler to generate any extra code. Another advantage of using the **const** mechanism is that the C++ construct will be available to a symbolic debugger, while the preprocessing symbols generally are not.

INLINE EXPANSION

Another common use of the C **#define** macro expansion facility is to avoid function call overhead for small functions. Some functions are so small that the overhead of invoking the function call takes more time than the body of the function itself. C++ provides the inline keyword to inform the compiler to place the function inline rather than generate the code for calling the routine. For example, the macro

```
#define max (x, y) ((x)>(y)?(x):(y))
```

can be replaced for integers vy the C++ inline function

```
inline int max (int x, int y)
{
  return (x > y ? x : y);
}
```

When a similar function is needed for multiple types, the C++ template mechanism can be used.

Macro expansion can lead to notorious results when encountering an expression with side effects, such as

```
max (f(x), z++);
```

which, after macro expansion becomes:

```
((f(x)) > (z++) ? (f(x) : (z++));
```

The variable z will be incremented once or twice, depending on the values of the x and y arguments to the function **max()**. Such errors are avoided when using the inline mechanism.

When defining a C++ class, the body of a class member function can also be specified. This code is also treated as inline code provided it does not contain any loops (e.g.,

while). For example:

```
class A {
  int a;

  public:
    A() { }
    // inline
    int Value()
    {
      return a;
    }
    // inline
}
```

Since the code for both the constructor **A()** and the member function **Value()** are specified as part of the class definition, the code between the braces will be expanded inline whenever these functions are invoked.

REFERENCES

Unlike C, C++ provides true call-by-reference through the use of reference types. A reference is an alias or a name to an existing object. They are simliar to pointers in that they must be initialized before they can be used. For example, let's declare an integer:

```
int n = 10;
```

and then declare a reference to it:

```
int& r= n;
```

Now **r** is an alias for **n**; both identify the same object and can be used interchangeably. Hence, the assignment

```
r = - 10;
```

changes the value of both **r** and **n** to -10.

It is important to note that initialization and assignment are completely different for references. A reference must have an initializer. Initialization is an operator that operates only on the reference itself. The initialization

```
int& r = n;
```

establishes the correspondence between the reference and the data object that it names. Assignment behaves like we expect an operation to, and operates through the reference on the object referred to. The assignment,

```
r = -10;
```

is the same for references as for any other lvalue, and simply assigns a new value to the designated data object.

C programmers know that C uses the call-by-value parameter mechanism. In order to enable functions to modify the values of their parameters, pointers to the parameters must be used as the ``value'', which is passed. For example, a routine **Swap()**, which swaps its parameters would be written like this in C:

```
void Swap (int* a, int* b)
{
        int tmp;
        tmp = *a;
        *a = *b;
        *b = tmp;
}
```

The routine would be invoked like this:

```
int x = 1;
int y = 2;
Swap (&x, &y);
```

C programmers are all too familiar with what happens when one of the ampersands is forgotten; the program usually ends with a core dump!

Now consider the C++ version of **Swap()** which makes use of true call-by-reference.

```
void Swap (int& a, int& b)
{
        int tmp;
        tmp = a;
        a = b;
        b = tmp;
}
```

The routine would be invoked like this:

```
int x = 1;
int y = 2;
Swap (x, y);
```

The compiler ensures that the parameters of **Swap()** will be passed by reference. In C, often a run-time error results if the value of a parameter is passed instead of its address. References eliminates these errors and is syntactically more pleasing.

Another use for references is as return types. Consider this routine:

```
int& FindByIndex (int* theArray,int index)
{
        return theArray[index];
}
```

Note that the **FindByIndex()** returns a reference to the element in the array rather than its value. The expression **FindByIndex (A, i)** yields a reference to the ith element of the array **A**. Now, because a reference is an lvalue, it can be used on the left hand side of an expression, we can write:

```
FindByIndex(A, i) = 25;
```

which will assign 25 to the ith element of the array **A**.

Note that if **FindByIndex()** is made inline, the overhead due to the function call is eliminated. Inline functions that return references are attractive for the sake of efficiency.

DECLARATIONS AS STATEMENTS.

In a C++ program, a declaration can be placed wherever a statement can appear, which can be anywhere within a program block. Any initializations are done each time their declaration statement is executed. Suppose we are searching a linked list for a certain key:

```
int IsMember (const int key)
{
        int found = 0;
        if (NotEmpty())
        {
                List* ptr = head;
                // Declaration

                while (ptr && !found)
                {
                        int item = ptr->data;
                        // Declaration

                        ptr = ptr->next;

                        if (item == key)
                                found = 1;
                }
        }
        return found;
}
```

By putting declarations closer to where the variables are used, you write more legible code.

IMPROVED TYPE SYSTEM.

Through the use of classes, user-defined types may be created, and if properly defined, C++ will behave as if they are one of the built-in types: **int**, **char**, **float**, and **double**. It is possible to define a **Vector** type and perform operations such as addition and multiplication just as easily as is done with **int**s:

```
// Define some arrays of doubles
```

```
double a[3] = { 11, 12, 13 };
double b[3] = { 21, 22, 23 };

// Initialize vectors from the
// double arrays
Vector v1 = a;
Vector v2 = b;

// Add the two matrices.
Vector v3 = v1 + v2;
```

The **Vector** class has been defined with all of the appropriate arithmetic operations so that it can be treated as a built-in type. It is even possible to define conversion operators so that we can convert the **Vector** to a **double**, we get the magnitude, or norm, of the **Vector**:

```
double norm = (double) v3;
```

OVERLOADING.

One of the many strengths of C++ is the ability to overload functions and operators. By overloading, the same function name or operator symbol can be given several different definitions. The number and types of the arguments supplied to a function or operator tell the compiler which definition to use. Overloading is most often used to provide different definitions for member functions of a class. But overloading can also be used for functions that are not a member of any class.

Suppose we need to search different types of arrays for a certain value. We can provide implementations for searching arrays of **int**egers, **float**s, and **double**s:

```
int Search (
        const int* data,
        const int key);

int Search (
        const float* data,
        const float key);

int Search (
```

```
        const double* data,
        const double key);
```

The compiler will ensure that the correct function is called based on the types of the arguments passed to **Search()**. When arguments do not exactly match the formal parameter types, the compiler will perform implicit type conversions (e.g., **int** to **float**) in an attempt to find a match.

Overloading is most often used for member functions and operators of classes. Most classes have overloaded constructors, for there is often more than one way to create a given object. All of the built-in types also have operators such as addition, subtraction, multiplication, and division. In fact, we can mix different types and still add them together:

```
int i = 1;
char c = 'a';
float f = -1.0;
double d = 100.0;
int result = i + c + f + d;
```

The compiler takes applies the type conversions appropriate for the above calculation. When we define our own types, we can inform the compiler which operations and type conversions can be applied to our type. The compiler will allow our type to blend in with the built-in types. We will see more examples of this when we look at classes in detail.

A FREE STORE IS PROVIDED.

In C, variables are placed in the free store by using the **sizeof()** macro to determine the needed allocation size and then calling **malloc()** with that size. Variables are removed from the free store by calling **free()**. With classes, using **malloc()** and **free()** becomes tedious. C++ provides the operators **new** and **delete**, which can allocate not only built-in types but also user-defined types. This provides a uniform mechanism for allocating and deallocating memory from the free store.

For example, to allocate an integer:

```
int *pi;
pi = new int;
```

```
*pi = 1;
```

and to allocate an array of 10 ints:

```
int *array = new int [10];
for (int i=0;i < 10; i++)
        array[i] = i;
```

Just as with **malloc()** the memory returned by **new** is not initialized; only static memory has a default initial value of zero.

Suppose we have defined a type for complex numbers, called **complex**. We can dynamically allocate a **complex** number as follows:

```
complex* pc = new complex (1, 2);
```

In this case, the complex pointer **pc** will point to the **complex** number 1 + 2i.

All memory allocated using **new** should be deallocated using **delete**. However, **delete** takes on different forms depending on whether the variable being deleted is an array or a simple variable. For the complex number above, we simply call delete:

```
delete pc;
```

Delete calls the destructor for the object to be deleted. However, to delete each element of an array, you must explicitly inform delete that an array is to be deleted:

```
delete [] array;
```

The C++ compiler maintains information about the size and number of objects in an array and retrieves this information when deleting an array. The empty bracket pair informs the compiler to call the class destructor for each element in the array.

Be careful, attempting to **delete** a pointer that has not been initialized by new results in undefined program behavior. However, it is safe to apply the delete operator to a null pointer.

New and **delete** are global C++ operators and can be redefined (e.g., if it is desirable

to trap every memory allocation). This is useful in debugging, but is not recommended for general programming. More often, the operators **new** and **delete** are overridden by providing new and delete operators for a specific class.

When C++ allocates memory for a user-defined class, the **new** operator for that class is used if it exists, otherwise the global **new** is used. Most often, programmers define new for certain classes to achieve improved memory management (i.e., reference counting for a class).

The Class: Data Encapsulation, Data Hiding, and Objects

Like a C structure, a C++ class is a data type. An object is simply an instantiation of a class. C++ classes have additional capabilities as the following example should show:

```
Vector v1(1,2),
Vector v2(2,3),
Vector vr;
vr = v1 + v2;
```

Vector is a class. **v1**, **v2**, and **vr** are objects of class **Vector**. **v1** and **v2** are given initial values through their constructor. **vr** is also initialized through its constructor to certain default values. The example illustrates a major power of C++. Namely, we can define functions on a class as well as data members. Here, we have an overloaded addition operator which makes our expression involving **Vector**s seem much more natural than the equivalent C code:

```
Vector v1, v2, vr;
add_vector( &vr , &v1, &v2 );
```

The ability to define these member functions allows us to have a constructor for **Vector**, code that creates an object of class Vector. The constructor ensures proper initialization of our **Vector**s.

Though not illustrated in the above example, a class can limit the use of its data members and member functions by non-member code. This is encapsulation. If class K defines member M as private, then only members of class K can use M. Defining M as public means any other class or function can use M.

Let's take a look at a trivial implementation of Vector that will show is a little about constructors, operators, and references.

```
#include <iostream.h>
class Vector
{
 public:
        Vector(double new_x=0.0,double new_y=0.0)       {
                if ((new_x<100.0) && (new_y<100.0))
                {
                        x=new_x;
                        y=new_y;
                }
                else
                {
                        x=0;
                        y=0;
                }
        }
        Vector operator +
                ( const Vector & v)
        {
                return
                (Vector (x + v.x, y + v.y));
        }
        void PrintOn (ostream& os)
        {
                os << "["
                        << x
                        << ", "
                        << y
                        << "]";
        }

private:
        double x, y;
};

int main()
{
```

```
        Vector v1, v2, v3(0.0,0.0);
        v1=Vector(1.1,2.2);
        v2=Vector(1.1,2.2);
        v3=v1+v2;
        cout << "v1 is ";
        v1.PrintOn (cout);
        cout << endl;
        cout << "v2 is ";
        v2.PrintOn (cout);
        cout << endl;
        cout << "v3 is ";
        v3.PrintOn (cout);
        cout << endl;
}
```

Encapsulation of **x** and **y** means that they cannot be altered without the help of specific member functions. Any member function or data member of **Vector** can use **x** and **y** freely. For everyone else, the member functions provide a strict interface. They ensure a particular behavior in our objects. In the example above, no **Vector** can be created that has an **x** or **y** component that exceeds 100. If at some point code tries to do this, then the constructor performs bounds-checking and sets **x** and **y** both to zero. In a normal C structure we can simply do the following:

```
Vector v1;
InitVector( & v1, 99 , 99 );
v1.x = 1000;
```

InitVector() closely approximates a C++ constructor. Assume it tries to behave like the constructor Vector() in example three. This C code demonstrates how without encapsulation we can easily violate the rules set up in our pseudo-constructor. With class **Vector**, both **x** and **y** are private. As a result, they can only be accessed by member functions. If our goal is to prevent **x** and **y** from exceeding 100, we simply have all accessor functions perform bounds-checking. In fact, once created and outside of the addition operation there's no way to modify **x** or **y**. They are private and no member function, outside of the constructor, sets their values. Notice how the constructor **Vector()** limits our Vector component values. By returning a new object, the addition operator uses the constructor to check for overflow.

We could have made `**+**' do multiplication instead. Though such manipulation is

atypical, it can be quite useful. For example, C++ comes standard with a streams library which uses the **<<** operator to provide output.

There is one useful thing about the addition operator: we don't have to pass the addresses of arguments. The arguments for the addition operator specify that the parameters are references (using the reference operator &-not the same as the address-of operator &). Recall that the reference operator allows us to use the same calling syntax as call-by-value and yet modify the value of an argument. The reference operator avoids the overhead of actually creating a new object. Thus, we can avoid a lot of indirection.

However, the most powerful OOP extension C++ provides is probably inheritance. Classes can inherit data and functions from other classes. For this purpose we can declare members in the base class as protected: not usable publicly but usable by derived classes.

In conclusion, we looked at some of the features that make C++ a better C. C++ provides stronger type checking by checking arguments to functions and reduces syntactic errors through the use of reference types. Programmers can also add new types to the language by defining classes. Although we have only taken a brief look at classes, we will see more abstract discussion of C++ object-orientation as well as general OOP concepts in upcoming columns.