A Tutorial on C++

by
Prof. David Kaeli
Mr. Yue Liu
Ms. Judy Nortz

Northeastern University
Dept. of Electrical and Computer Engineering

Table of Contents

1.0 Introduction

C++ was developed in the early 1980's by Bjarne Stroustrup of AT&T Bell Laboratories. C++ has quickly become the most popular OOP programming language. C++ improves C by introducing several new programming constructs that directly support OOP techniques, such as *data abstraction*, *inheritance*, and *polymorphism*. In addition, structures such as *templates* and mechanism such as *generic functions* provide generics which make programming easier and more dynamic.

In C++, the *class* mechanism supports *data abstraction*, which is one of the motivation behind creating objects. You can encapsulate data with functions to define a new data type that is complete with its own operators. Through *inheritance*, you can express the differences among related classes as you share the functions and member data that implement common features. *Inheritance* also helps you to reuse existing code from one or multiple classes by simply deriving a new class from an existing class. Additionally, you can use *inheritance* to extend an existing class by adding new members to it. Polymorphic functions can work with many different argument types. C++ supports this kind of *polymorphism* through function overloading. Another type of *polymorphism* simplifies the syntax of performing the same operation with a hierarchy of classes. *Templates* allow container classes, such as lists, array, etc., to be simply defined and implemented without loss of static type checking and run-time efficiency. For example, a template can make a list of generic objects where the objects are specified at run time. Furthermore, several "types" of list can be created. A *generic function* simply refers to calling function by its address. Generic functions are more flexible in that a pointer's address can be changed dynamically (e.g., using a case statement) and passed into another function. A *global template function* is similar to the generic function, however, it takes a template as one of its argument.

2.0 Mastering C++

Mastering a new programming language such as C++ is not an easy task. This tutorial is geared towards C++ beginners. It is assumed that the student has sufficient background in the C programming language. Many of the new features provided by C++ are discussed. Fifteen example programs are given in this tutorial to demonstrate the important features of C++.

2.1 Classes in C++

A *class* is the most fundamental abstraction mechanism in C++. The C++ *class* construct is not simply an aggregation of data, as the C *struct* is. A class can group together both data and functions. Functions and data grouped in a class are called *member functions* and *member data* respectively. They are "members" of the class containing them. A class definition has the following syntax:

```
class Class_name
{
protected:
      data and functions, ...
private:
      data and functions, ...
public:
      data and functions, ...
}
```

The access to a class member is controlled using the keywords *private*, *public*, and *protected*. The appearance of these three keywords, followed by a colon, designates the access protection for the members following it, up to the end of the class definition or up to the next such designation. The default protection is in effect prior to the appearance of the first such designation. The access control attributes have the following semantics:

- *public* members of an object are accessible by any function having access to the declaration of that object's class and scope access to the object itself.

- *protected* members of an object are accessible only by member functions of that object's class or of its directly derived classes (its child classes). A derived class object cannot access the protected fields of its grandparent classes.

- *private* members of an object are accessible only by member functions of that object's class.

Figure 1 shows an example of a simple class definition.

```
1. #include <iostream.h>
2. //CONVERT SECONDS TO MINUTES AND SECONDS

3. const int modulus = 60;

4. //DEFINE Mod_Int Class
5. class Mod_Int
6. {
7.    private:
8.          int v;  // PRIVATE DATA
9.    public:
10.         Mod_Int(int i) {v = i%modulus;}      //  CONSTRUCTOR
11.         void assign(int i) {v = i%modulus;}
12.         void print() {cout << v << "\t";}
13.};

14.main()
15.{
16.   int seconds = 400;
17.   Mod_Int z(seconds); // DECLARE Z AND CALL CONSTRUCTOR
18.   cout << seconds << " seconds equals " << seconds/60
19.         << " minutes ";
20.   z.print();          // ACCESS MEMBER FUNCTION print()
21.   cout << "seconds\n";
22.}
```

```
Output:
400 seconds equals 6 minutes 40     seconds
```

Figure 1: Definition of a simple class

This example defines a class named Mod_Int. It has private data v and three member functions. The member function named Mod_Int() is called the *constructor* of the Mod_Int class. You can use a constructor to handle any specific requirements for initializing objects of a class. For example, if an object needs extra storage, you can allocate memory for it in the constructor. On line 17 in Figure 1, the constructor Mod_Int(400) is called when an object z of class Mod_Int is declared. It initializes private data v as 40 (400 % 60 = 40). The member function assign(int i) is not used in the example in Figure 1.

The way an object accesses its member function is similar to the way you would access members of a C structure. The operators **.** and → are used. If you use an object's name to index a function, the dot operator (**.**) should be used. If you use a pointer to an object to index a function, the arrow operator (→) should be used. We will see more examples later that use the dot and arrow operators.

C++ treats anything following the double forward slash (*//)* and everything up to the end of the line as a comment. C++ also recognizes the standard C comments that start with the characters /* and end after the characters ***/**. C++ also introduces new syntax and functions for producing I/O. These are contained in the iostream library. To use this library, your C++ program should include the header file <*iostream.h*>. This file contains the definitions of the classes that implement the stream objects and provides the buffering. The file <*iostream.h*> is analogous to <*stdio.h*> in C.

In *iostream* library, cout is an output stream connected to the standard output and is analogous to stdout in C. You should notice three things in the example in Figure 1:

- The << operator is a good choice to represent the output operation, because it points in the direction of data movement that, in this case, is toward the cout stream.

- You can concatenate multiple << operators in a single line, all feeding the same stream.

- You use the same syntax to print all the basic data types on a stream. The << operator automatically converts the internal representation of the variable into a textual representation. Contrast this with the need to use different format strings for printing different data types using printf.

The *const* keyword used to prefix the name of a variable indicates that the contents of the variable must not be modified by the program. Similarly, if a function's argument is a

pointer, and if the pointer is declared to be *const*, the function cannot modify the contents of the location referenced by that pointer.

Figure 2 illustrates more complex data structures and member functions.

```
1. #include <iostream.h>
2. struct listelem{ char data; listelem *next; };

3. class List {
4.    private:
5.          listelem *h;     // THE HEAD OF THE LIST
6.    public:
7.          List(){h = 0;}// THE CONSTRUCTOR
8.                      // 0 DENOTES AN EMPTY LIST
9.          ~List() {release();}   // DESTROYS LIST
10.         void prepend(char c); // ADDS TO THE HEAD
11.         void append(char c);  // ADDS TO THE TAIL
12.         void del() {
13.               listelem *temp = h;  // DELETES THE HEAD
14.               h = h->next; delete temp;}
15.         listelem *first() {return(h);} // RETURNS THE HEAD
16.         void print();              // PRINTS THE LIST
17.         void release();            // FREES THE LIST MEMORY
18.};

19.void List::prepend(char c) {
20.   listelem *temp = new listelem; //CREATE A NEW LIST ELEMENT
21.   temp->next = h;        // LINK TO THE LIST
22.   temp->data = c;
23.   h = temp;              // UPDATE THE HEAD OF THE LIST
24.}

25.void List::append(char c){
26.   listelem *ntemp = new listelem; //CREATE A NEW LIST ELEMENT
27.   listelem *temp = h;
28.   while (temp->next != 0)          //TRAVERSE THE LIST,
29.   temp = temp->next;               // LOOKING FOR THE END
30.   temp->next = ntemp;
31.   ntemp->data = c;                 // SET THE DATA VALUE
32.   ntemp->next = 0;                 // SET THE NEXT VALUE
33.}

34.void List::print(){
35.   listelem *temp = h;     // POINT TO BEGINNING OF LIST
36.   while (temp != 0) {     // WHILE NOT THE END OF LIST, PRINT
37.         cout << temp->data << "->";
38.         temp = temp->next;
39.   }
40.   cout << "\n###\n";
41.}
42.
43.void List::release() // RETURN STORAGE
44.{
45.   while (h != 0)    // WHILE NOT END OF THE LIST
46.   del();
47.}
```

```
48.main()
49.{
50.   List w;
51.   w.prepend('B'); // PREPEND B
52.   w.prepend('A'); // PREPEND A
53.   w.print();      // PRINT THE LIST
54.   w.append('C');  // APPEND C
55.   w.print();      // PRINT THE LIST
56.   cout << w.first()->data << "\n"; // PRINT THE HEAD
57.   cout << w.first()->next->data << "\n"; // PRINT THE 2nd ELEMENT
58.   cout << w.first()->next->next->data << "\n";
59.                                   // PRINT THE 3rd ELEMENT
60.}

Output:
A->B->
###
A->B->C->
###
A
B
C
```

Figure 2: A linked list

From this example we can see that C++ continues to support C's *struct* keyword. In fact, C++ expands the definition of *struct* by allowing inclusion of member functions. In C++, the only difference between a class and a structure is that the contents of a structure are always public by default. The instances of *List* class contain the pointer that points to *listelem* structure.

Note that the *List* class has a member function named *List()* and another named *~List()*. These two member functions are, respectively, the *constructor* and *destructor* of class *List*. You can define a destructor if there is any need to clean up after an object is destroyed, e.g. if you want to free memory allocated by the constructor.

Some member functions of the class *List* are defined inside the body of *List* (which is called *inline*) and some are outside. Both methods are acceptable syntax. You do not have to define a function inside a class to make it inline. C++ provides the inline keyword. This will be discussed in example 8. The syntax of a member function defined outside its class body is:

```
type Class_name::function_name( type arg1, type arg2, ...)
```

The double colon denotes which class the function belongs to and is called the *scope resolution operator*. Note the C++ function prototype differs from C. All arguments should have their type explicitly declared inside the parentheses.

Other new syntax in this example are the keywords *new* and *delete*. They are C++ operators. The operator *new* is used to dynamically allocate memory; the operator *delete*

is used to free up the memory allocated by new operator. Delete is always applied to a pointer. In lines 28 and 35 of Figure 2

```
listelem *temp = new listelem;
```

allocates memory of the size of *listelem* to pointer *temp*, and in line 20

```
delete temp;
```

releases the memory that pointer *temp* points to.

2.2 Overloading

Let us move on to Figure 3. Notice that there are now three constructors in class A and they all have the same name! This is called *overloading*. The constructor without any arguments is called the *default constructor*. Overloading causes no confusion or ambiguity; the compiler will call the appropriate constructor depending upon the *types* of the arguments. In other words, the types of arguments determine which constructor the C++ compiler calls. For example, if you define an instance but do not specify any initial value, the default constructor will be called. The destructor can not be overloaded.

```
1. #include <iostream.h> // EXAMPLE 3: MEMORY ALLOCATION
2. class A {
3.    private: int xx;  // PRIVATE DATA
4.    public:
5.          A(){xx=0;cout<<"A() called\n";} // A CONSTRUCTOR
6.          A(int n) { xx = n;      // ANOTHER CONSTRUCTOR
7.                cout << "A(int " << n << ") called\n"; }
8.          A(double y) { xx = y + 0.5;   // ANOTHER CONSTRUCTOR
9.                cout << "A(fl " << y << ") called\n"; }
10.         ~A(){cout<<"~A() called A::xx = "<<xx<< "\n";}
11.         // DEFAULT DESTRUCTOR
12.};
13.main(){  cout << "enter main\n";       // PRINT ENTER MAIN
14.   int x = 14; float y = 17.3;
15.   A z(11), zz(11.5), zzz(0);    // INVOKE CONSTRUCTORS
16.   A d[5] = {0, 1, 2, 3, 4};     // INVOKE CONSTRUCTOR
17.   cout << "\nOBJECT ALLOCATION LAYOUT\n";   // PRINT ADDRESSES
18.   cout << "\nx is at " << &x << "\ny is at " << &y;
19.   cout << "\nz is at " << &z << "\nzz is at " << &zz << "\nzzz is at
   " << &zzz;
20.   cout << "\n-------------------------\n";
21.   zzz = A(x);       // INSTANTIATE AND INITIALIZE
22.   zzz = A(y);       // INSTANTIATE AND INITIALIZE
23.   cout << "exit main\n";
24.}
Output:
enter main
A(int 11) called
A(fl 11.5) called
A(int 0) called
A(int 0) called
A(int 1) called
```

```
A(int 2) called
A(int 3) called
A(int 4) called

OBJECT ALLOCATION LAYOUT

x is at 0x7fffb848
y is at 0x7fffb84c
z is at 0x7fffb850
zz is at 0x7fffb858
zzz is at 0x7fffb860
------------------------
A(int 14) called
~A() called A::xx = 14
A(fl 17.3) called
~A() called A::xx = 17
exit main
~A() called A::xx = 4
~A() called A::xx = 3
~A() called A::xx = 2
~A() called A::xx = 1
~A() called A::xx = 0
~A() called A::xx = 17
~A() called A::xx = 12
~A() called A::xx = 11
```

Figure 3: Constructor overloading and memory allocation

Notice the sequence in which the constructor and its corresponding constructor are called. The destructors are invoked in the opposite order that the constructors are called. For example, object *z*'s constructor is called first and its destructor is called last. When instantiating *zzz*, which is accomplished by calling constructors *A(x)* and *A(y)* (lines 21, 22 in Figure 3), what actually happens is that a temporary object of class A is created, and the object *zzz* gets its new value of *xx* by copying from the temporary object. Then the temporary object is destroyed. When the destructor of *zzz* is called before the program finishes, the value of *xx* is changed to 17. (NOTE: Results may different on different computing platforms for this example.)

The *this* pointer is an important feature in C++. The *this* pointer points to the specific object that is being invoked. Figure 4 shows how to use the *this* pointer.

```
1. #include <iostream.h>
2. // EXAMPLE 4: THIS POINTER

3. class C_pair {  // DEFINE CLASS
4.          char c1, c2;
5.    public:
6.          void init(char b) { c2 = b; c1 = 1 + b;}
7.          C_pair &increment(){c1++; c2++;return(*this);}
8.          void *where_am_I() {return (this);}       // RETURN ADDRESS
9.          void print() {cout << c1 << c2 << "\t";}  // PRINT c1 AND c2
10.};
```

```
11.main()
12.{
13.   C_pair a, b, c; // CALLS THE DEFAULT CONSTRUCTOR
14.   a.init('A'); // CREATE BA
15.   b.init('B'); // CREATE CB
16.   c.init('D'); // CREATE ED
17.   a.print();
18.   cout << " is at " << a.where_am_I() << endl;
19.   b.print();
20.   cout << " is at " << b.where_am_I() << endl;
21.   c.print();
22.   cout << " is at " << c.where_am_I() << endl;
23.   c.increment().print(); // PRINT FE AND UPDATE
24.   cout << " is at " << c.where_am_I() << endl;
25.   c.print();
26.   cout << " is at " << c.where_am_I() << endl;
27.}
```

Output:

```
BA     is at 0x7fffb868
CB     is at 0x7fffb870
ED     is at 0x7fffb878
FE     is at 0x7fffb878
FE     is at 0x7fffb878
```

Figure 4: *This* Pointer

Note the difference between using **this* and *this*. The member function *increment()* updates the member variable *c1* and *c2*, but returns the same object that invokes this function. Thus calling *c.where_am_I()* and *c.increment().where_am_I()* will result in printing out the same value. The function *where_am_I()* returns the address of the object, not the object itself.

Some new syntax in this example is *endl* (lines 18, 20, 22, 24, and 26 in Figure 4). An *endl* sends a new line to *ostream* and flushes the buffer. It is similar as *\n* in C.

2.3 Derivation

C++ allows one class to be derived from another. This is called *inheritance*. The program in Figure 5 is an example of using *inheritance*.

```
1. #include <iostream.h>
2. // EXAMPLE 5: INHERITANCE EXAMPLE

3. class Student {// DEFINE THE Student CLASS
4.    protected:
5.          int   sid;
6.          int   year;
7.          char  name[13];
8.    public:
9.          Student(char *nm, int id, int y)  // CONSTRUCTOR
```

```
10.         {strcpy(name, nm); sid = id; year = y;}
11.         void print();
12.};

13.class Grad_student: public Student{// DEFINE THE Grad_student CLASS
14.   protected:              // INHERIT THE PUBLIC PART OF Student
15.         float support;
16.         char thesis[30];
17.   public:
18.         Grad_student(char *nm, int id, int y,  // CONSTRUCTOR
19.         float sup, char *th):
20.         Student(nm, id, y)
21.         { support = sup; strcpy(thesis, th);}
22.         void print();
23.};

24.void Student::print() // Student print MEMBER FUNCTION
25.{ cout << "\n" << name << " STUDENT NO. " << sid << " YEAR " << year
   << endl; }

26.void Grad_student::print() // Grad_student print MEMBER FUNCTION
27.{
28.   cout << "\n" << name << " STUDENT NO. " << sid
29.   << " YEAR " << year << " SUPPORT " << support
30.   << " THESIS " << thesis << endl;
31.}

32.main()
33.{
34.   Student  s("Joe Smith", 1234, 1994), *ps = &s;
35.   Grad_student  gs("Johnny Smart", 5678, 1999, 10000., "World
   Peace"), *pgs;
36.   ps->print();     // PRINT STUDENT INFORMATION
37.   ps = pgs = &gs;
38.   ps->print();     // PRINT STUDENT INFORMATION
39.   pgs->print();    // PRINT GRAD STUDENT INFORMATION
40.}
Output:
Joe Smith STUDENT NO. 1234 YEAR 1994

Johnny Smart STUDENT NO. 5678 YEAR 1999

Johnny Smart STUDENT NO. 5678 YEAR 1999 SUPPORT 10000 THESIS World Peace
```

Figure 5: Inheritance example

When you are deriving a new class from the existing class, you can add additional capabilities to the derived class by:

- defining new member variables
- defining new member functions
- overriding the definition of inherited member functions

In Figure 5, the class *Grad_student* is a derived class of the class *Student*. Therefore the *Grad-student* class has three member variables as does the *Student* class, and it adds two new member variables of its own. Both the *Grad_student* class and the *Student* class have

member function *print()*. In C++, an overloaded member function in the derived class hides all inherited member functions of the same name. Thus, when called by a pointer to an object of the *Grad_student* class (line 39 in Figure 5), the *print()* of *Grad_student* is invoked.

In C++, you can use a reference or a pointer to any derived class in place of a reference or a pointer to the base class without an explicit type cast. The opposite is not true. Therefore, when using *ps* to call *print()*, the member function of *Student* class is invoked.

Figure 6 illustrates how to implement multiple inheritance.

```
1. #include <iostream.h>
2. // A Construction Through Inheritance Example
3. class Labor {  // DEFINE Labor Class
4.    protected:
5.          int c;
6.    public:
7.          Labor(int cost) { c = cost; }     // Constructor
8.          int cost() { return(c); }
9. };
10.class Parts {  // DEFINE Parts Class
11.   protected:
12.         int c;
13.   public:
14.         Parts(int cost)  { c = cost; }  // Constructor
15.         int cost() { return(c); }
16.};
17.// DEFINE Service Class, Inherit from Labor and Parts
18.class Service: public Labor, public Parts {// MULTIPLE INHERITANCE
19.   private:
20.         int num_parts;
21.         char name[20];
22.   public:
23.         Service(int num,char *nm,int lc,int pc):Labor(lc),Parts(pc)
24.         {
25.               num_parts = num;
26.               strcpy(name, nm);
27.         }
28.         int cost()// Calculate Total Cost
29.         {return(num_parts *(Parts::cost()+Labor::cost()));}
30.         void print(){ cout << endl;
31.               cout << name << " cost " << Labor::c << " dollars per
   install "<<endl;
32.               cout << name << " cost " << Parts::c << " dollars per
   part " << endl;
33.               cout << num_parts << " "  << name << " installed cost
   " << cost() << " dollars "       << endl;
34.                     }
35.};

36.main()
37.{
38.   Service computer(10, "widgets", 50, 90); //Declare and Initialize
39.   Service display(20, "tubes", 500, 10);
40.   computer.print();    // Print the cost of computer service
41.   display.print();     // Print the cost of display service
```

```
42.}
Output:
widgets cost 50 dollars per install
widgets cost 90 dollars per part
10 widgets installed cost 1400 dollars

tubes cost 500 dollars per install
tubes cost 10 dollars per part
20 tubes installed cost 10200 dollars
```

Figure 6: Multiple inheritance

Service is a derived class of *Labor* and *Parts*. A point worth noting is how the base class is initialized. The constructor of *Service* has a initializer list (*Labor(lc), Parts(pc)*), which is used to initialize the base class (line 23 in Figure 6). You can also initialize member variables using an initializer list. By using an initializer list, you can avoid creating unnecessary temporary objects. It is an efficient way to initialize member variables.

```
1. // EXAMPLE 7: OVERLOADING
2. #include <iostream.h>
3. #include <math.h>
4. class Complex{// Define Complex Class
5.    private:
6.          double real, imag;
7.    public:
8.          Complex(double r) { real = r; imag = 0;} // Constructor
9.          void assign(double r, double i) {real = r; imag = i;}
10.         void print() {cout << real << " + " << imag << "i "; }
11.         operator double(){return(sqrt(real*real + imag*imag));}
12.};
13.// greater is an overloaded function
14.inline int greater(int i, int j){return( i > j ? i : j );}
15.inline double greater(double x, double y){return( x > y ? x : y);}
16.inline Complex greater(Complex w, Complex z){return(w > z ? w : z);}

17.main()
18.{
19.   int i = 10, j = 5;
20.   float x = 7.0;
21.   double y = 14.5;
22.   Complex w(0), z(0), zmax(0);
23.
24.   w.assign(x,y); // conversion from float to double is performed
25.   z.assign(i,j); // conversion from int to double is performed
26.   cout << " compare " << i << " and " << j << "   greater is " <<
   greater(i,j) << endl; // invokes int greater
27.   cout << " compare " << x << " and  " << y << "  greater is " <<
   greater(x,y) << endl; // invokes double greater
28.   cout << " compare " << y << " and " ;
29.   z.print();
30.   cout << "  greater is " << greater(y, double(z)) << endl;
31.                                        // invokes the double greater
32.   zmax = greater(w,z); // invokes the Complex greater
33.   cout << " compare ";
34.   w.print();
35.   cout << "  and  ";
36.   z.print();
```

```
37.   cout << "  greater is  ";
38.   zmax. print();
39.   cout << endl;
40.}
Output:
compare 10 and 5   greater is 10
compare 7 and  14.5  greater is 14.5
compare 14.5 and 10 + 5i   greater is 14.5
compare 7 + 14.5i   and  10 + 5i   greater is  7 + 14.5i
```

Figure 7: Overloading

In Figure 7, *greater()* is an overloaded function. Which *greater()* function is called depends on the types of the arguments. By giving the similar function the same name, overloaded functions can simplify your program and give your code better readability. Note in this function the presence of an include for math.h. You will need to compile with the -lm switch.

The *inline* keyword before each definition of *greater()* (lines 14, 15, and 16 in Figure 7) tells the C++ compiler to inline the body of *greater()* function wherever it is called. Remember that you can only use an inline function in the file in which it is defined. This is because the compiler needs the entire definition of an inline function.

For an overloaded inline function (as in this example), the compiler will pick up the appropriate function according to the argument type and then insert its body appropriately.

Type conversion is performed by the compiler when invoking *w.assign(x, y)* and *z.assign(i, j)* (lines 24 and 25 in Figure 7). The value of *float* variable *x* and *int* variables *i* and *j* are converted to *double*. But when calling an overloaded function, you should check the argument types to make sure that there is an overloaded function defined for those types. Otherwise, the compiler's automatic type conversion may generate an undesirable result.

2.4 Virtual Functions

Virtual function is the mechanism in C++ to realize *dynamic binding*. You can define a virtual function using the following syntax:

```
virtual type function_name( type arg1, type arg2, ...)
```

The *virtual* keyword preceding a function signals to the C++ compiler that the function should be defined in a derived class and that the compiler may have to call it indirectly through a pointer. In Figure 8, the member function *print_i()* of *A* class is a virtual function. Therefore, the *B* class, which is the only derived class of *A*, must also have a member function called *print_i()*. Both member functions are called through pointers.

```
1. // EXAMPLE 8: A VIRTUAL FUNCTION
```

```
2. #include <iostream.h>
3. class A {// Define Class A
4.    public:
5.          int i;
6.          virtual void print_i() { cout << i << " inside A\n"; }
7.          // print_i is a virtual function
8. };
9. class B: public A {  // Define Class B, inheriting from Class A
10.   public:
11.         void print_i() { cout << i << " inside B\n"; }
12.         // print_i is a virtual function
13.};

14.main(){
15.   A a;
16.   A* pa = &a;
17.   B f;
18.   f.i = 1 + (a.i = 1); // Assign i in 2 instances
19.   pa->print_i();    // What will this print?
20.   pa = &f;
21.   pa->print_i();    // What will this print?
22.}
Output:
1 inside A
2 inside B
```

Figure 8: Virtual function

Figure 9 shows an example of how to access another class's member functions.

```
1. #include <iostream.h>
2. class A {
3.    protected:
4.          int a;
5.    public:
6.          A(int x) { a = x;}
7.          int geta() {return(a);}
8.          void set_a(int x) { a = x;}
9. };
10.class B {
11.   protected:
12.         int b;
13.   public:
14.         B(int x) { b = x;}
15.         void printab(A ap) { cout << ap.geta() << b << endl;}
16.};

17.main(){
18.   A ai(1), &ap = ai;
19.   B bi(2);
20.   bi.printab(ap);
21.}
Output:
12
```

Figure 9: Access member functions

The member function *printab()* is called (line 20 of Figure 9) by an argument which has the type of an object class *A*. *Printab()* is defined in *B* class, but its argument is of type class *A*. *Printab()* prints out the protected member variables of *A* and *B*. The result of this example is 12.

2.5 Generic Functions

A function is normally called by its name and a list of arguments. However, a function can also be called with a pointer that points to a function as shown in Figure 10.

```
1.  void function(char *p)
2.  void (*funcptr) (char *);  // pointer to a function
3.  void sample()
4.    {
5.      funcptr = &function;   // funcptr points to function
6.      (*funcptr) ("string"); // calling function through funcptr
7.    }
```

Figure 10: Calling A Function Through a Void Pointer

To call a function through a void pointer we must first dereference the pointer. This is done in line 2 (note that line 2 is not **funcptr ("error");*, but rather **(funcptr("error"))*). Pointers to functions need to have the same argument types as the functions they call. Pointers to functions can be used to provide *polymorphic* routines. Polymorphic routines are routines that can be applied to objects of many different types, as shown in example 11 below.

```
1.  typedef int (*compare) (void *, void *);
2.  void sort(void* base, unsigned n, unsigned int sz, compare cmp)
3.   // Sort the "n" elements of vector "base"
     // into increasing order using the comparison
4.   // function pointed to by "cmp".  The elements are the size "sz".
5.    {
6.    for (int i=0; i<n-1; i++)
7.                for (int j=n-1; i<j; j--)
8.                {
9.                      char* elementj = (char*) base + j * sz;// b[j]
10.                     char* elementk = elementj - sz;        // b[j-1]
11.                     if ((*cmp) (elementj,elementk) < 0)
12.                     {               // cmp will return -1 if
                                        // elementj < elementk, else
                                        // will return +1
13.                     for (int k=0; k<sz; k++) // swap b[j] and b[j-1]
14.                       {
15.                       char temp = elementj[k];
16.                       elementj[k] = elementk[k];
17.                       elementk[k]= temp;
18.                       }
19.                     }
20.               }
21.   }
```

Figure 11: Polymorphic Pointer To A Sorting Function

The advantage of using a pointer to a function is that the *sort()* function does not need to know the type of objects that it is sorting.

2.6 Templates

C++ uses the keyword *template* to provide *parametric polymorphism*. Templates allow container classes to be simply defined and implemented. A *container class* holds objects of many different types such as lists, arrays, associative arrays and other sets. Static type checking and run-time efficiency are not lost. To make an analogy, a class template specifies how individual classes can be constructed similar to how a class declaration specifies how individual objects can be constructed. Figure 12 defines a stack of arbitrary type elements.

```
1.  template <class TYPE>
2.  class stack {
3.    TYPE* v;
4.    TYPE* p;
5.    int sz;

6.    public:
7.      stack(int s) { v=p=new TYPE[sz=s];  }
8.      ~stack() { delete[] v; }

9.     void push (TYPE a) { *p++ = a; }
10.    TYPE pop() { return *--p; }

11.    int size() const { return p-v; }
12. };
```

Figure 12: A Simple Stack Template

Here, a template <*class TYPE*> is declared. *TYPE* is the type defined in the instantiation, and it can take on a variety of types, such as a class, or a character (as shown below). Once the template is defined, classes conforming to this template can be instantiated. The name of a class template followed by a bracketed (< >) type defines a new class as specified by the template and is used exactly like other classes. Figure 13 defines object *sc* of class *stack*<*char*> where *TYPE* is of type char. The class definitions in Figures 13 and 14 are equivalent.

```
1.  stack<char> sc(100);            //stack of characters
```

Figure 13: A Class Defined By A Template

```
1.  class stack_char {
2.    char* v;
3.    char* p;
4.    int sz;
5.  public:
6.    stack_char (int s) { v=p=new char[sz=s]; }
7.    ~stack_char() { delete[] v; }
8.    void push(char a) { *p++ = a; }
9.    char pop() { return *--p; }
10.   int size() const { return p-v; }
11.  };
```

Figure 14: Equivalent Class Definition To Figure 13

```
1.  stack<int> si(100);              // stack of integer elements
2.  stack<char*> stk_str(50);        // stack of string elements
3.  stack<complex> stk_cmplx(10);    // stack of complex elements
```

Figure 15: A Class Defined By A Template

Figure 15 shows three more instantiations of different types of stacks. Templates save the programmer from rewriting class definitions when only a new type is to be considered. The template argument (TYPE) can be used throughout the class definition.

TIP: It is usually a good idea to thoroughly debug a class before converting it to a template.

2.7 Function Templates

Many times functions will have the same code, independent of type. For example, Figure 16 shows initialization of one array from another.

```
1.  template<class TYPE>
2.  void copy(TYPE a[], TYPE b[], int n)
3.  // copy the n elements from b[] to a[]
4.
5.    {
6.    for (int i=0; i<n; i++)
7.          a[i] = b[]i];
8.    }
```

Figure 16: A Template Function For Copying

We can then use copy this template function to successfully perform copies 1-4 shown below in Figure 17:

```
      double f1[50], f2[50];
      char   c1[10], c2[20];
      int    i1[4],  i2[4];
      char   *ptr1,  *ptr2;
1.    copy(f1, f2, 50);
2.    copy(c1, c2, 10);
3.    copy(i1, i2, 3);
4.    copy(ptr1, ptr2, 30);
5.    copy(i1, (int*)f2, 50);
```

Figure 17: A Global Template Function For Sorting

If we tried to perform the copy operation in 5, we might get an undesirable result. Instead we should perform a generic copy as follows in Figure 18:

```
1.  template<class T1, class T2>
2.  void copy(T1 a[], T2 b[], int n)
3.  // copy the n elements from b[] to a[]
4.
5.    {
6.    for (int i=0; i<n; i++)
7.          a[i] = b[]i];
8.    }
```

Figure 18: A Template Function For Copying

This form of copying is much safer, performing an element by element copy.

3.0 References

The provided programs are simple examples. They are examples of some of the features in C++ that help you develop object-oriented programs. It is recommended that you consult reference books for further information about C++. A list of recommended books is provided below.

1. H.M. Deitel and P.J. Deitel, *How to Program C++,* Prentice-Hall, 1994.

2. R. Sessions, *Class Construction in C and C++,* Prentice-Hall, 1992.

3. S.B. Lippman, *C++ Primer*, Addison-Wesley, 1989.

4. I. Pohn, *C++ for C Programmers*, Addison-Wesley, 1989.

5. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

6. B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, 2nd Edition, 1994.