23. CLASSES AND OBJECTS

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types. A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class. **C++ Class Definitions** When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object. A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we define the Box data type using the keyword **class** as follows: `class Box { public: double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box };` The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section. **Define Objects** A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of

class Box: `Box Box1; // Declare Box1 of type Box`

C++
162

`Box Box2; // Declare Box2 of type Box` Both of the objects Box1 and Box2 will have their own copy of data members. **Accessing the Data Members** The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear: `#include <iostream> using namespace std; class Box { public: double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box }; int main( ) { Box Box1; // Declare Box1 of type Box Box Box2; // Declare Box2 of type Box double volume = 0.0; // Store the volume of a box here // box 1 specification Box1.height = 5.0; Box1.length = 6.0; Box1.breadth = 7.0; // box 2 specification Box2.height = 10.0; Box2.length = 12.0; Box2.breadth = 13.0;`

C++
163

`// volume of box 1 volume = Box1.height * Box1.length * Box1.breadth; cout << "Volume of Box1 : " << volume <<endl; // volume of box 2 volume = Box2.height * Box2.length * Box2.breadth; cout << "Volume of Box2 : " << volume <<endl;`

return 0; } When the above code is compiled and executed, it produces the following result: Volume of Box1 : 210 Volume of Box2 : 1560 It is important to note that private and protected members cannot be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed. **Classes & Objects in Detail** So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below: **Concept Description** Class member functions A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. Class access modifiers A class member can be defined as public, private or protected. By default members would be assumed as private. Constructor & destructor A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.

C++ copy constructor The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. C++ friend functions A **friend** function is permitted full access to private and protected members of a class. C++ inline functions With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function. The this pointer in C++ Every object has a special pointer **this** which points to the object itself. Pointer to C++ classes A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it. Static members of a class Both data members and function members of a class can be declared as static. **Class member functions** A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object. Let us take previously defined class to access the members of the class using a member function instead of directly accessing them: class Box { public: double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box double getVolume(void);// Returns box volume };

Member functions can be defined within the class definition or separately using **scope resolution operator, ::**. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier. So either you can define**Volume()** function as below: class Box { public: double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box double getVolume(void) { return length * breadth * height; } }; If you like, you can define the same function outside the class using the **scope resolution operator** (::) as follows: double Box::getVolume(void) { return length * breadth * height; } Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (**.**) on a object where it will manipulate data related to that object only as follows: Box myBox; // Create an object myBox.getVolume(); // Call member function for

the object Let us put above concepts to set and get the value of different class members in a class: #include <iostream> using namespace std;
C++
166
class Box { public: double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box // Member functions declaration double getVolume(void); void setLength(double len); void setBreadth(double bre); void setHeight(double hei); }; // Member functions definitions double Box::getVolume(void) { return length * breadth * height; } void Box::setLength(double len) { length = len; } void Box::setBreadth(double bre) { breadth = bre; } void Box::setHeight(double hei) { height = hei;
C++
167
} // Main function for the program int main( ) { Box Box1; // Declare Box1 of type Box Box Box2; // Declare Box2 of type Box double volume = 0.0; // Store the volume of a box here // box 1 specification Box1.setLength(6.0); Box1.setBreadth(7.0); Box1.setHeight(5.0); // box 2 specification Box2.setLength(12.0); Box2.setBreadth(13.0); Box2.setHeight(10.0); // volume of box 1 volume = Box1.getVolume(); cout << "Volume of Box1 : " << volume <<endl; // volume of box 2 volume = Box2.getVolume(); cout << "Volume of Box2 : " << volume <<endl; return 0; } When the above code is compiled and executed, it produces the following result: Volume of Box1 : 210 Volume of Box2 : 1560
C++
168
Class Access Modifiers Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. The access restriction to the class members is specified by the labeled **public, private,** and **protected** sections within the class body. The keywords public, private, and protected are called access specifiers. A class can have multiple public, protected, or private labeled sections. Each section remains in effect until either another section label or the closing right brace of the class body is seen. The default access for members and classes is private. class Base { public: // public members go here protected: // protected members go here private: // private members go here }; **The public Members** A **public** member is accessible from anywhere outside the class but within a program. You can set and get the value of public variables without any member function as shown in the following example: #include <iostream> using namespace std;
C++
169
class Line { public: double length; void setLength(double len); double getLength(void); }; // Member functions definitions double Line::getLength(void) { return length ; } void Line::setLength(double len) { length = len; } // Main function for the program int main( ) { Line line; // set line length line.setLength(6.0); cout << "Length of line : " << line.getLength() <<endl; // set line length without member function line.length = 10.0; // OK: because length is public cout << "Length of line :

" << line.length <<endl; return 0; } When the above code is compiled and executed, it produces the following result:
C++
170
Length of line : 6 Length of line : 10 **The private Members** A **private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members. By default all the members of a class would be private, for example in the following class**width** is a private member, which means until you label a member, it will be assumed a private member: class Box { double width; public: double length; void setWidth(double wid); double getWidth(void); }; Practically, we define data in private section and related functions in public section so that they can be called from outside of the class as shown in the following program. #include <iostream> using namespace std; class Box { public: double length; void setWidth(double wid); double getWidth(void); private:
C++
171
double width; }; // Member functions definitions double Box::getWidth(void) { return width ; } void Box::setWidth(double wid) { width = wid; } // Main function for the program int main( ) { Box box; // set box length without member function box.length = 10.0; // OK: because length is public cout << "Length of box : " << box.length <<endl; // set box width without member function // box.width = 10.0; // Error: because width is private box.setWidth(10.0); // Use member function to set it. cout << "Width of box : " << box.getWidth() <<endl; return 0; } When the above code is compiled and executed, it produces the following result: Length of box : 10
C++
172
Width of box : 10 **The protected Members** A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes. You will learn derived classes and inheritance in next chapter. For now you can check following example where I have derived one child class **SmallBox** from a parent class **Box**. Following example is similar to above example and here **width** member will be accessible by any member function of its derived class SmallBox. #include <iostream> using namespace std; class Box { protected: double width; }; class SmallBox:Box // SmallBox is the derived class. { public: void setSmallWidth(double wid); double getSmallWidth(void); }; // Member functions of child class double SmallBox::getSmallWidth(void) { return width ; } void SmallBox::setSmallWidth(double wid)
C++
173
{ width = wid; } // Main function for the program int main( ) { SmallBox box; // set box width using member function box.setSmallWidth(5.0); cout << "Width of box : "<< box.getSmallWidth() << endl; return 0; } When the above code is compiled and executed, it produces the following result: Width of box : 5
Constructor & Destructor A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class. A constructor will have exact same name as the class and it does not have any return type at all, not

even void. Constructors can be very useful for setting initial values for certain member variables. Following example explains the concept of constructor: #include <iostream> using namespace std; class Line { public: void setLength(double len);

C++

174

double getLength(void); Line(); // This is the constructor private: double length; }; // Member functions definitions including constructor Line::Line(void) { cout << "Object is being created" << endl; } void Line::setLength(double len) { length = len; } double Line::getLength(void) { return length; } // Main function for the program int main( ) { Line line; // set line length line.setLength(6.0); cout << "Length of line : " << line.getLength() <<endl; return 0; }

C++

175

When the above code is compiled and executed, it produces the following result: Object is being created Length of line : 6 **Parameterized Constructor** A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example: #include <iostream> using namespace std; class Line { public: void setLength(double len); double getLength(void); Line(double len); // This is the constructor private: double length; }; // Member functions definitions including constructor Line::Line(double len) { cout << "Object is being created, length = " << len << endl; length = len; } void Line::setLength(double len) { length = len; }

C++

176

double Line::getLength(void) { return length; } // Main function for the program int main( ) { Line line(10.0); // get initially set length. cout << "Length of line : " << line.getLength() <<endl; // set line length again line.setLength(6.0); cout << "Length of line : " << line.getLength() <<endl; return 0; } When the above code is compiled and executed, it produces the following result: Object is being created, length = 10 Length of line : 10 Length of line : 6 **Using Initialization Lists to Initialize Fields** In case of parameterized constructor, you can use following syntax to initialize the fields: Line::Line(double len): length(len) { cout << "Object is being created, length = " << len << endl; } Above syntax is equal to the following syntax: Line::Line(double len)

C++

177

{ cout << "Object is being created, length = " << len << endl; length = len; } If for a class C, you have multiple fields X, Y, Z, etc., to be initialized, then use can use same syntax and separate the fields by comma as follows: C::C(double a, double b, double c): X(a), Y(b), Z(c) { .... } **The Class Destructor** A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class. A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming

out of the program like closing files, releasing memories etc. Following example explains the concept of destructor: #include <iostream> using namespace std; class Line { public: void setLength(double len); double getLength(void); Line(); // This is the constructor declaration ~Line(); // This is the destructor: declaration private:

C++

178

double length; }; // Member functions definitions including constructor Line::Line(void) { cout << "Object is being created" << endl; } Line::~Line(void) { cout << "Object is being deleted" << endl; } void Line::setLength(double len) { length = len; } double Line::getLength(void) { return length; } // Main function for the program int main( ) { Line line; // set line length line.setLength(6.0); cout << "Length of line : " << line.getLength() <<endl; return 0; }

C++

179

When the above code is compiled and executed, it produces the following result: Object is being created Length of line : 6 Object is being deleted **Copy Constructor** The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to: • Initialize one object from another of the same type. • Copy an object to pass it as an argument to a function. • Copy an object to return it from a function. If a copy constructor is not defined in a class, the compiler itself defines one.If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here: classname (const classname &obj) { // body of constructor } Here, **obj** is a reference to an object that is being used to initialize another object. #include <iostream> using namespace std; class Line { public: int getLength(void); Line(int len); // simple constructor Line(const Line &obj); // copy constructor

C++

180

~Line(); // destructor private: int *ptr; }; // Member functions definitions including constructor Line::Line(int len) { cout << "Normal constructor allocating ptr" << endl; // allocate memory for the pointer; ptr = new int; *ptr = len; } Line::Line(const Line &obj) { cout << "Copy constructor allocating ptr." << endl; ptr = new int; *ptr = *obj.ptr; // copy the value } Line::~Line(void) { cout << "Freeing memory!" << endl; delete ptr; } int Line::getLength(void) { return *ptr; } void display(Line obj)

C++

181

{ cout << "Length of line : " << obj.getLength() <<endl; } // Main function for the program int main( ) { Line line(10); display(line); return 0; } When the above code is compiled and executed, it produces the following result: Normal constructor allocating ptr Copy constructor allocating ptr. Length of line : 10 Freeing memory! Freeing memory! Let us see the same example but with a small change to create another object using existing object of the same type: #include <iostream> using namespace std; class Line { public: int getLength(void);

```
Line( int len ); // simple constructor Line( const Line &obj); // copy
constructor ~Line(); // destructor
```
C++
182
```
private: int *ptr; }; // Member functions definitions including constructor
Line::Line(int len) { cout << "Normal constructor allocating ptr" << endl; //
allocate memory for the pointer; ptr = new int; *ptr = len; }
Line::Line(const Line &obj) { cout << "Copy constructor allocating ptr." <<
endl; ptr = new int; *ptr = *obj.ptr; // copy the value } Line::~Line(void) {
cout << "Freeing memory!" << endl; delete ptr; } int Line::getLength( void )
{ return *ptr; } void display(Line obj) {
```
C++
183
```
cout << "Length of line : " << obj.getLength() <<endl; } // Main function for
the program int main( ) { Line line1(10); Line line2 = line1; // This also
calls copy constructor display(line1); display(line2); return 0; }
```
When the above code is compiled and executed, it produces the following result:
```
Normal constructor allocating ptr Copy constructor allocating ptr. Copy constructor
allocating ptr. Length of line : 10 Freeing memory! Copy constructor
allocating ptr. Length of line : 10 Freeing memory! Freeing memory! Freeing
memory!
```
Friend Functions A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions. A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.
C++
184
To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows:
```
class Box { double width; public:
double length; friend void printWidth( Box box ); void setWidth( double wid
); };
```
To declare all member functions of class ClassTwo as friends of class ClassOne, place a following declaration in the definition of class ClassOne:
```
friend class ClassTwo;
```
Consider the following program:
```
#include <iostream> using namespace std; class Box { double width; public: friend void printWidth( Box
box ); void setWidth( double wid ); }; // Member function definition void
Box::setWidth( double wid ) { width = wid; }
```
C++
185
```
// Note: printWidth() is not a member function of any class. void printWidth(
Box box ) { /* Because printWidth() is a friend of Box, it can directly
access any member of this class */ cout << "Width of box : " << box.width
<<endl; } // Main function for the program int main( ) { Box box; // set box
width without member function box.setWidth(10.0); // Use friend function to
print the wdith. printWidth( box ); return 0; }
```
When the above code is compiled and executed, it produces the following result:
```
Width of box : 10
```
Inline Functions C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time. Any change to

an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality. To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

C++

186

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier. Following is an example, which makes use of inline function to return max of two numbers: `#include <iostream> using namespace std; inline int Max(int x, int y) { return (x > y)? x : y; } // Main function for the program int main( ) { cout << "Max (20,10): " << Max(20,10) << endl; cout << "Max (0,200): " << Max(0,200) << endl; cout << "Max (100,1010): " << Max(100,1010) << endl; return 0; }` When the above code is compiled and executed, it produces the following result: `Max (20,10): 20 Max (0,200): 200 Max (100,1010): 1010` **this Pointer** Every object in C++ has access to its own address through an important pointer called **this**pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object. Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

C++

187

Let us try the following example to understand the concept of this pointer: `#include <iostream> using namespace std; class Box { public: // Constructor definition Box(double l=2.0, double b=2.0, double h=2.0) { cout <<"Constructor called." << endl; length = l; breadth = b; height = h; } double Volume() { return length * breadth * height; } int compare(Box box) { return this->Volume() > box.Volume(); } private: double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box }; int main(void) { Box Box1(3.3, 1.2, 1.5); // Declare box1`

C++

188

`Box Box2(8.5, 6.0, 2.0); // Declare box2 if(Box1.compare(Box2)) { cout << "Box2 is smaller than Box1" <<endl; } else { cout << "Box2 is equal to or larger than Box1" <<endl; } return 0; }` When the above code is compiled and executed, it produces the following result: `Constructor called. Constructor called. Box2 is equal to or larger than Box1` **Pointer to C++ Classes** A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class you use the member access operator **->** operator, just as you do with pointers to structures. Also as with all pointers, you must initialize the pointer before using it. Let us try the following example to understand the concept of pointer to a class: `#include <iostream> using namespace std; class Box { public: // Constructor definition Box(double l=2.0, double b=2.0, double h=2.0)`

C++

189

{ cout <<"Constructor called." << endl; length = l; breadth = b; height = h; } double Volume() { return length * breadth * height; } private: double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box }; int main(void) { Box Box1(3.3, 1.2, 1.5); // Declare box1 Box Box2(8.5, 6.0, 2.0); // Declare box2 Box *ptrBox; // Declare pointer to a class. // Save the address of first object ptrBox = &Box1; // Now try to access a member using member access operator cout << "Volume of Box1: " << ptrBox->Volume() << endl; // Save the address of first object ptrBox = &Box2; // Now try to access a member using member access operator cout << "Volume of Box2: " << ptrBox->Volume() << endl;

C++

190

return 0; } When the above code is compiled and executed, it produces the following result: Constructor called. Constructor called. Volume of Box1: 5.94 Volume of Box2: 102 **Static Members of a Class** We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member. A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to. Let us try the following example to understand the concept of static data members: #include <iostream> using namespace std; class Box { public: static int objectCount; // Constructor definition Box(double l=2.0, double b=2.0, double h=2.0) { cout <<"Constructor called." << endl; length = l; breadth = b;

C++

191

height = h; // Increase every time object is created objectCount++; } double Volume() { return length * breadth * height; } private: double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box }; // Initialize static member of class Box int Box::objectCount = 0; int main(void) { Box Box1(3.3, 1.2, 1.5); // Declare box1 Box Box2(8.5, 6.0, 2.0); // Declare box2 // Print total number of objects. cout << "Total objects: " << Box::objectCount << endl; return 0; } When the above code is compiled and executed, it produces the following result: Constructor called. Constructor called. Total objects: 2

C++

192

Static Function Members By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**. A static member function can only access static data member, other static member functions and any other functions from outside the class. Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not. Let us try the following example to understand the concept of static function members: #include <iostream> using namespace std; class Box {

```
public: static int objectCount; // Constructor definition Box(double l=2.0,
double b=2.0, double h=2.0) { cout <<"Constructor called." << endl; length =
l; breadth = b; height = h; // Increase every time object is created
objectCount++; } double Volume() { return length * breadth * height; }
```

C++

193

```
static int getCount() { return objectCount; } private: double length; //
Length of a box double breadth; // Breadth of a box double height; // Height
of a box }; // Initialize static member of class Box int Box::objectCount =
0; int main(void) { // Print total number of objects before creating object.
cout << "Inital Stage Count: " << Box::getCount() << endl; Box Box1(3.3, 1.2,
1.5); // Declare box1 Box Box2(8.5, 6.0, 2.0); // Declare box2 // Print total
number of objects after creating object. cout << "Final Stage Count: " <<
Box::getCount() << endl; return 0; }
```

When the above code is compiled and executed, it produces the following result:

```
Inital Stage Count: 0 Constructor called.
```

C++

194

```
Constructor called. Final Stage Count: 2
```

24. INHERITANCE

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time. When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class. The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on. **Base & Derived Classes** A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form: `class derived-class: access-specifier base-class` Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default. Consider a base class **Shape** and its derived class **Rectangle** as follows: `#include <iostream> using namespace std; // Base class class Shape { public: void`

`setWidth(int w) { width = w;`

C++
196

`} void setHeight(int h) { height = h; } protected: int width; int height; }; // Derived class class Rectangle: public Shape { public: int getArea() { return (width * height); } }; int main(void) { Rectangle Rect; Rect.setWidth(5); Rect.setHeight(7); // Print the area of the object. cout << "Total area: " << Rect.getArea() << endl; return 0; }`

C++
197

When the above code is compiled and executed, it produces the following result: `Total area: 35` **Access Control and Inheritance** A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class. We can summarize the different access types according to - who can access them, in the following way: **Access public protected private** Same class yes yes yes Derived classes yes yes no Outside classes yes no no A derived class inherits all base class methods with the following exceptions: • Constructors, destructors and copy constructors of the base class. • Overloaded operators of the base class. • The friend functions of the base class. **Type of** When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above. We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied: • **Public Inheritance:** When deriving a class from a **public** base

class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.

C++

198

• **Protected Inheritance:** When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class. • **Private Inheritance:** When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class. **Multiple Inheritance** A C++ class can inherit members from more than one class and here is the extended syntax: `class derived-class: access baseA, access baseB....` Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example: `#include <iostream> using namespace std; // Base class Shape class Shape { public: void setWidth(int w) { width = w; } void setHeight(int h) { height = h; } protected: int width; int height; };`

C++

199

```
// Base class PaintCost class PaintCost { public: int getCost(int area) {
return area * 70; } }; // Derived class class Rectangle: public Shape, public
PaintCost { public: int getArea() { return (width * height); } }; int
main(void) { Rectangle Rect; int area; Rect.setWidth(5); Rect.setHeight(7);
area = Rect.getArea(); // Print the area of the object. cout << "Total area:
" << Rect.getArea() << endl; // Print the total cost of painting
```

C++

200

```
cout << "Total paint cost: $" << Rect.getCost(area) << endl; return 0; }
```

When the above code is compiled and executed, it produces the following result: `Total area: 35 Total paint cost: $2450`

25. OVERLOADING (OPERATOR & FUNCTION)

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively. An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation). When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**. **Function Overloading in C++** You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type. Following is the example where same function **print()** is being used to print different data types: `#include <iostream> using namespace std; class printData { public: void print(int i) { cout << "Printing int: " << i << endl; } void print(double f) { cout << "Printing`

`float: " << f << endl; }`

C++
202
`void print(char* c) { cout << "Printing character: " << c << endl; } }; int main(void) { printData pd; // Call print to print integer pd.print(5); // Call print to print float pd.print(500.263); // Call print to print character pd.print("Hello C++"); return 0; }` When the above code is compiled and executed, it produces the following result: `Printing int: 5 Printing float: 500.263 Printing character: Hello C++` **Operators Overloading in C++** You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well. Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list. `Box operator+(const Box&);` Declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above
C++

203
function as non-member function of a class then we would have to pass two arguments for each operand as follows: `Box operator+(const Box&, const Box&);` Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below: `#include <iostream> using namespace std; class Box { public: double getVolume(void) { return length * breadth * height; } void setLength(double len) { length = len; } void setBreadth(double bre) { breadth = bre; } void setHeight(double hei) { height = hei; } // Overload + operator to add two Box objects.`
C++
204
`Box operator+(const Box& b) { Box box; box.length = this->length + b.length; box.breadth = this->breadth + b.breadth; box.height = this->height + b.height; return box; } private: double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box }; // Main function for the program int main( ) { Box Box1; // Declare Box1 of type Box Box Box2; // Declare Box2 of type Box Box Box3; // Declare Box3 of type Box double volume = 0.0; // Store the volume of a box here // box 1 specification Box1.setLength(6.0); Box1.setBreadth(7.0); Box1.setHeight(5.0); // box 2 specification Box2.setLength(12.0); Box2.setBreadth(13.0); Box2.setHeight(10.0); // volume of box 1 volume = Box1.getVolume();`
C++
205
`cout << "Volume of Box1 : " << volume <<endl; // volume of box 2 volume = Box2.getVolume(); cout << "Volume of Box2 : " << volume <<endl; // Add two object as follows: Box3 = Box1 + Box2; // volume of box 3 volume = Box3.getVolume(); cout << "Volume of Box3 : " << volume <<endl; return 0; }` When the above code is compiled and executed, it produces the following result: `Volume of Box1 : 210 Volume of Box2 : 1560 Volume of Box3 : 5400` **Overloadable/Non-overloadable Operators** Following is the list of operators which can be overloaded: `+ - * / % ^ & | ~ ! , = < > <= >= ++ -- << >> == != && || += -= /= %= ^= &= |= *= <<= >>= [] ()`
C++
206
`-> ->* new new [] delete delete []` Following is the list of operators, which cannot be overloaded: `:: .* . ?:` **Operator Overloading Examples** Here are various operator overloading examples to help you in understanding the concept. **S.N. Operators and Example** 1 Unary operators overloading 2 Binary operators overloading 3 Relational operators overloading 4 Input/Output operators overloading 5 ++ and -- operators overloading 6 Assignment operators overloading 7 Function call () operator overloading 8 Subscripting [] operator overloading 9 Class member access operator -> overloading **Unary Operators** The unary operators operate on a single operand and following are the examples of Unary operators: • The increment (++) and decrement (--) operators. • The unary minus (-) operator.
C++
207

• The logical not (!) operator. The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--. Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage. #include <iostream> using namespace std; class Distance { private: int feet; // 0 to infinite int inches; // 0 to 12 public: // required constructors Distance(){ feet = 0; inches = 0; } Distance(int f, int i){ feet = f; inches = i; } // method to display distance void displayDistance() { cout << "F: " << feet << " I:" << inches <<endl; } // overloaded minus (-) operator Distance operator- () { feet = -feet; inches = -inches;

C++

208

return Distance(feet, inches); } }; int main() { Distance D1(11, 10), D2(-5, 11); -D1; // apply negation D1.displayDistance(); // display D1 -D2; // apply negation D2.displayDistance(); // display D2 return 0; } When the above code is compiled and executed, it produces the following result: F: -11 I:-10 F: 5 I:-11 Hope above example makes your concept clear and you can apply similar concept to overload Logical Not Operators (!). **Increment (++) and Decrement (- -) Operators** The increment (++) and decrement (--) operators are two important unary operators available in C++. Following example explain how increment (++) operator can be overloaded for prefix as well as postfix usage. Similar way, you can overload operator (--). #include <iostream> using namespace std; class Time { private: int hours; // 0 to 23

C++

209

int minutes; // 0 to 59 public: // required constructors Time(){ hours = 0; minutes = 0; } Time(int h, int m){ hours = h; minutes = m; } // method to display time void displayTime() { cout << "H: " << hours << " M:" << minutes <<endl; } // overloaded prefix ++ operator Time operator++ () { ++minutes; // increment this object if(minutes >= 60) { ++hours; minutes -= 60; } return Time(hours, minutes); } // overloaded postfix ++ operator Time operator++(int) { // save the orignal value Time T(hours, minutes); // increment this object

C++

210

++minutes; if(minutes >= 60) { ++hours; minutes -= 60; } // return old original value return T; } }; int main() { Time T1(11, 59), T2(10,40); ++T1; // increment T1 T1.displayTime(); // display T1 ++T1; // increment T1 again T1.displayTime(); // display T1 T2++; // increment T2 T2.displayTime(); // display T2 T2++; // increment T2 again T2.displayTime(); // display T2 return 0; } When the above code is compiled and executed, it produces the following result: H: 12 M:0 H: 12 M:1 H: 10 M:41 H: 10 M:42

C++

211

Binary Operators Overloading The unary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator. Following example explains how addition (+) operator can be overloaded. Similar way, you

can overload subtraction (-) and division (/) operators. #include <iostream> using namespace std; class Box { double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box public: double getVolume(void) { return length * breadth * height; } void setLength(double len) { length = len; } void setBreadth(double bre) { breadth = bre; } void setHeight(double hei) { height = hei;

C++

212

} // Overload + operator to add two Box objects. Box operator+(const Box& b) { Box box; box.length = this->length + b.length; box.breadth = this->breadth + b.breadth; box.height = this->height + b.height; return box; } }; // Main function for the program int main( ) { Box Box1; // Declare Box1 of type Box Box Box2; // Declare Box2 of type Box Box Box3; // Declare Box3 of type Box double volume = 0.0; // Store the volume of a box here // box 1 specification Box1.setLength(6.0); Box1.setBreadth(7.0); Box1.setHeight(5.0); // box 2 specification Box2.setLength(12.0); Box2.setBreadth(13.0); Box2.setHeight(10.0); // volume of box 1 volume = Box1.getVolume(); cout << "Volume of Box1 : " << volume <<endl;

C++

213

// volume of box 2 volume = Box2.getVolume(); cout << "Volume of Box2 : " << volume <<endl; // Add two object as follows: Box3 = Box1 + Box2; // volume of box 3 volume = Box3.getVolume(); cout << "Volume of Box3 : " << volume <<endl; return 0; } When the above code is compiled and executed, it produces the following result: Volume of Box1 : 210 Volume of Box2 : 1560 Volume of Box3 : 5400 **Relational Operators Overloading** There are various relational operators supported by C++ language like (<, >, <=, >=, ==, etc.) which can be used to compare C++ built-in data types. You can overload any of these operators, which can be used to compare the objects of a class. Following example explains how a < operator can be overloaded and similar way you can overload other relational operators. #include <iostream> using namespace std; class Distance { private: int feet; // 0 to infinite int inches; // 0 to 12

C++

214

public: // required constructors Distance(){ feet = 0; inches = 0; } Distance(int f, int i){ feet = f; inches = i; } // method to display distance void displayDistance() { cout << "F: " << feet << " I:" << inches <<endl; } // overloaded minus (-) operator Distance operator- () { feet = -feet; inches = -inches; return Distance(feet, inches); } // overloaded < operator bool operator <(const Distance& d) { if(feet < d.feet) { return true; } if(feet == d.feet && inches < d.inches) { return true; }

C++

215

return false; } }; int main() { Distance D1(11, 10), D2(5, 11); if(D1 < D2) { cout << "D1 is less than D2 " << endl; } else { cout << "D2 is less than D1 " << endl; } return 0; } When the above code is compiled and executed, it produces the following result: D2 is less than D1 **Input/Output Operators Overloading** C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<. The stream

insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object. Here, it is important to make operator overloading function a friend of the class because it would be called without creating an object. Following example explains how extraction operator >> and insertion operator <<. #include <iostream> using namespace std; class Distance

C++

216

{ private: int feet; // 0 to infinite int inches; // 0 to 12 public: // required constructors Distance(){ feet = 0; inches = 0; } Distance(int f, int i){ feet = f; inches = i; } friend ostream &operator<<(ostream &output, const Distance &D) { output << "F : " << D.feet << " I : " << D.inches; return output; } friend istream &operator>>(istream &input, Distance &D) { input >> D.feet >> D.inches; return input; } }; int main() { Distance D1(11, 10), D2(5, 11), D3; cout << "Enter the value of object : " << endl; cin >> D3;

C++

217

cout << "First Distance : " << D1 << endl; cout << "Second Distance :" << D2 << endl; cout << "Third Distance :" << D3 << endl; return 0; } When the above code is compiled and executed, it produces the following result: $./a.out Enter the value of object : 70 10 First Distance : F : 11 I : 10 Second Distance :F : 5 I : 11 Third Distance :F : 70 I : 10 **++ and - Operators Overloading** The increment (++) and decrement (--) operators are two important unary operators available in C++. Following example explain how increment (++) operator can be overloaded for prefix as well as postfix usage. Similar way, you can overload operator (--). #include <iostream> using namespace std; class Time { private: int hours; // 0 to 23 int minutes; // 0 to 59 public: // required constructors Time(){

C++

218

hours = 0; minutes = 0; } Time(int h, int m){ hours = h; minutes = m; } // method to display time void displayTime() { cout << "H: " << hours << " M:" << minutes <<endl; } // overloaded prefix ++ operator Time operator++ () { ++minutes; // increment this object if(minutes >= 60) { ++hours; minutes -= 60; } return Time(hours, minutes); } // overloaded postfix ++ operator Time operator++(int) { // save the orignal value Time T(hours, minutes); // increment this object ++minutes; if(minutes >= 60) { ++hours;

C++

219

minutes -= 60; } // return old original value return T; } }; int main() { Time T1(11, 59), T2(10,40); ++T1; // increment T1 T1.displayTime(); // display T1 ++T1; // increment T1 again T1.displayTime(); // display T1 T2++; // increment T2 T2.displayTime(); // display T2 T2++; // increment T2 again T2.displayTime(); // display T2 return 0; } When the above code is compiled and executed, it produces the following result: H: 12 M:0 H: 12 M:1 H: 10 M:41 H: 10 M:42 **Assignment Operators Overloading** You can overload the assignment operator (=) just as you can other operators and it can be used to create an object

just like the copy constructor. Following example explains how an assignment operator can be overloaded. #include <iostream> using namespace std;
C++
220
class Distance { private: int feet; // 0 to infinite int inches; // 0 to 12 public: // required constructors Distance(){ feet = 0; inches = 0; } Distance(int f, int i){ feet = f; inches = i; } void operator=(const Distance &D) { feet = D.feet; inches = D.inches; } // method to display distance void displayDistance() { cout << "F: " << feet << " I:" << inches << endl; } }; int main() { Distance D1(11, 10), D2(5, 11); cout << "First Distance : ";
C++
221
D1.displayDistance(); cout << "Second Distance :"; D2.displayDistance(); // use assignment operator D1 = D2; cout << "First Distance :"; D1.displayDistance(); return 0; } When the above code is compiled and executed, it produces the following result: First Distance : F: 11 I:10 Second Distance :F: 5 I:11 First Distance :F: 5 I:11 **Function Call () Operator Overloading** The function call operator () can be overloaded for objects of class type. When you overload ( ), you are not creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters. Following example explains how a function call operator () can be overloaded. #include <iostream> using namespace std; class Distance { private: int feet; // 0 to infinite int inches; // 0 to 12 public: // required constructors Distance(){
C++
222
feet = 0; inches = 0; } Distance(int f, int i){ feet = f; inches = i; } // overload function call Distance operator()(int a, int b, int c) { Distance D; // just put random calculation D.feet = a + c + 10; D.inches = b + c + 100 ; return D; } // method to display distance void displayDistance() { cout << "F: " << feet << " I:" << inches << endl; } }; int main() { Distance D1(11, 10), D2; cout << "First Distance : "; D1.displayDistance(); D2 = D1(10, 10, 10); // invoke operator() cout << "Second Distance :"; D2.displayDistance();
C++
223
return 0; } When the above code is compiled and executed, it produces the following result: First Distance : F: 11 I:10 Second Distance :F: 30 I:120 **Subscripting [ ] Operator Overloading** The subscript operator [] is normally used to access array elements. This operator can be overloaded to enhance the existing functionality of C++ arrays. Following example explains how a subscript operator [] can be overloaded. #include <iostream> using namespace std; const int SIZE = 10; class safearay { private: int arr[SIZE]; public: safearay() { register int i; for(i = 0; i < SIZE; i++) { arr[i] = i; } } int &operator[](int i) { if(i > SIZE)
C++
224
{ cout << "Index out of bounds" <<endl; // return first element. return arr[0]; } return arr[i]; } }; int main() { safearay A; cout << "Value of A[2] : " << A[2] <<endl; cout << "Value of A[5] : " << A[5]<<endl; cout << "Value

of A[12] : " << A[12]<<endl; return 0; } When the above code is compiled and executed, it produces the following result: Value of A[2] : 2 Value of A[5] : 5 Index out of bounds Value of A[12] : 0 **Class Member Access Operator - > Overloading** The class member access operator (->) can be overloaded but it is bit trickier. It is defined to give a class type a "pointer-like" behavior. The operator -> must be a member function. If used, its return type must be a pointer or an object of a class to which you can apply. The operator-> is used often in conjunction with the pointer-dereference operator * to implement "smart pointers." These pointers are objects that behave like normal pointers except they perform other tasks when you access an object through them, such as automatic object deletion either when the pointer is destroyed, or the pointer is used to point to another object.
C++
225
The dereferencing operator-> can be defined as a unary postfix operator. That is, given a class: class Ptr{ //... X * operator->(); }; Objects of class **Ptr** can be used to access members of class **X** in a very similar manner to the way pointers are used. For example: void f(Ptr p) { p->m = 10 ; // (p.operator->())->m = 10 } The statement p->m is interpreted as (p.operator->())->m. Using the same concept, following example explains how a class access operator -> can be overloaded. #include <iostream> #include <vector> using namespace std; // Consider an actual class. class Obj { static int i, j; public: void f() const { cout << i++ << endl; } void g() const { cout << j++ << endl; } }; // Static member definitions: int Obj::i = 10; int Obj::j = 12; // Implement a container for the above class class ObjContainer {
C++
226
vector<Obj*> a; public: void add(Obj* obj) { a.push_back(obj); // call vector's standard method. } friend class SmartPointer; }; // implement smart pointer to access member of Obj class. class SmartPointer { ObjContainer oc; int index; public: SmartPointer(ObjContainer& objc) { oc = objc; index = 0; } // Return value indicates end of list: bool operator++() // Prefix version { if(index >= oc.a.size()) return false; if(oc.a[++index] == 0) return false; return true; } bool operator++(int) // Postfix version { return operator++(); } // overload operator-> Obj* operator->() const {
C++
227
if(!oc.a[index]) { cout << "Zero value"; return (Obj*)0; } return oc.a[index]; } }; int main() { const int sz = 10; Obj o[sz]; ObjContainer oc; for(int i = 0; i < sz; i++) { oc.add(&o[i]); } SmartPointer sp(oc); // Create an iterator do { sp->f(); // smart pointer call sp->g(); } while(sp++); return 0; } When the above code is compiled and executed, it produces the following result: 10 12 11 13 12 14 13 15
C++
228
14 16 15 17 16 18 17 19 18 20 19 21

26. POLYMORPHISM

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance. C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function. Consider the following example where a base class has been derived by other two classes: `#include <iostream> using namespace std; class Shape { protected: int width, height; public: Shape( int a=0, int b=0) { width = a; height = b; } int area() { cout << "Parent class area :" <<endl; return 0; } }; class Rectangle: public Shape{ public: Rectangle( int a=0, int b=0):Shape(a, b) { }`

`int area () { cout << "Rectangle class area :" <<endl;`

C++
230

`return (width * height); } }; class Triangle: public Shape{ public: Triangle(int a=0, int b=0):Shape(a, b) { } int area () { cout << "Triangle class area :" <<endl; return (width * height / 2); } }; // Main function for the program int main( ) { Shape *shape; Rectangle rec(10,7); Triangle tri(10,5); // store the address of Rectangle shape = &rec; // call rectangle area. shape->area(); // store the address of Triangle shape = &tri; // call triangle area. shape->area(); return 0; }`

C++
231

When the above code is compiled and executed, it produces the following result: `Parent class area Parent class area` The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program. But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this: `class Shape { protected: int width, height; public: Shape( int a=0, int b=0) { width = a; height = b; } virtual int area() { cout << "Parent class area :" <<endl; return 0; } };` After this slight modification, when the previous example code is compiled and executed, it produces the following result: `Rectangle class area Triangle class area` This time, the compiler looks at the contents of the pointer instead of its type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

C++
232

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations. **Virtual Function** A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function. What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**. **Pure Functions** It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class. We can change the virtual function area() in the base class to the following: `class Shape { protected: int width, height; public: Shape( int a=0, int b=0) { width = a; height = b; } // pure virtual function virtual int area() = 0; };` The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

27. DATA ABSTRACTION

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details. Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation. Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen. Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals. In C++, classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally. For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work. In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this:

```
#include <iostream> using namespace std; int main( ) { cout << "Hello C++"


<<endl; return 0; }
```

C++
234

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of ‘cout’ is free to change. **Access Labels Enforce Abstraction** In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels: • Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members. • Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type. There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen. **Benefits of Data** Data abstraction provides two important advantages: • Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object. • The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code. By defining data members only in the private section of the class, the class author is free to make

changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken. **Example** Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example: `#include <iostream> using namespace std; class Adder{ public:`

C++

235

```
// constructor Adder(int i = 0) { total = i; } // interface to outside world
void addNum(int number) { total += number; } // interface to outside world
int getTotal() { return total; }; private: // hidden data from outside world
int total; }; int main( ) { Adder a; a.addNum(10); a.addNum(20);
a.addNum(30); cout << "Total " << a.getTotal() <<endl; return 0; }
```

When the above code is compiled and executed, it produces the following result: `Total 60`

C++

236

Above class adds numbers together, and returns the sum. The public members - **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly. **Designing Strategy** Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact. In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation.