

Classes

The Structure of Class Declarations

Class declarations are similar to `struct` declarations, except that they use the `class` keyword and require the members are private unless explicitly declared public. Programming convention calls for the declaration of a class to start with a capital letter to differentiate it from standard variables.

```
class ClassName
{
    public:
        public class members
    private:
        private class members
};
```

Class Declarations and Implementations: Example

```
#include <iostream>

using namespace std;

class Automobile
{
    public:
        Automobile(string newName);           // two constructors
        Automobile(string newName, int newPrice);
        void showAuto();                     // public member functions
        int findPrice();
    private:
        string brand;                        // private data members
        int price;
};

Automobile::Automobile(string newName)
{
    brand = newName;
    price = 20000;                           // assume the car costs $20,000
    cout << "Now creating a " << brand << endl;
}

Automobile::Automobile(string newName, int newPrice)
{
    price = newPrice;
    brand = newName;
    cout << "Now creating a $" << price << " " << brand << endl;
}

void Automobile::showAuto()
{
    cout << brand << " costs " << price << endl;
}

int Automobile::findPrice()
{
    return price;
}

int main()
{
    Automobile taurus("Ford"), jalopy("Lada", 500);
    taurus.showAuto();
    cout << "Used car: only $" << jalopy.findPrice() << endl;
    return 0;
}
```

Remark on Terminology

Technically, a *class* is a data type, while an *object* is a particular instance of a class.

The Simplest Classes

Classes in C++ have two types of members: data and member functions. The very simplest classes may not require any member functions at all. We have seen this in the struct examples.

Accessing Class Members

Modules can access public members of a class using the dot (.) operator (see example). Members of an object can access each other without the need for the dot; however, the dot is needed for a member of one object to access a member of a different object.

Class Declarations

A class header contains the declarations of all data members and the prototypes of all member functions. Members may be tagged as either public or private (see the example above for syntax). Public members of a class are accessible to all outside modules. Private members are accessible only to other members of the class. The implementation (code) for member functions should follow the class header. When defining a member function it is important to include the scope resolution operator (: :), which is used to specify the class to which a function belongs (see example).

Constructors and Destructors

In addition to ordinary member functions, a class may include one or more constructor functions. A class constructor is invoked whenever an instance of that class is declared. If a class contains multiple constructors, then the number and type of parameters determine the constructor used provided. Constructors always have the same name as the class that contains them (see example) and never have a return type. A default is provided in no others are defined, but a default (no parameters) is not provided if other constructors are declared.

Operator Overloading

C++ allows the programmer to define what should happen when a user-defined object is used in conjunction with a built-in operator (e.g., + or %). Thus, for example, the + operator can be overloaded so that it can be used to add fractions, even though fraction addition is not built into the language. An overloaded operator is defined much like a function. The best way to show this is by example:

```
class Fraction
{
    public:
        void show();
        fraction operator +(fraction addend);
    private:
        int num, denom;
};

void Fraction::show()
{
    cout << num << "/" << denom;
}

Fraction Fraction::operator + (Fraction addend)
{
    Fraction ans;
    ans.num = num * addend.denom + addend.num * denom;
    ans.denom = denom * addend.denom;
    return ans;
}
```

The first line of the operator definition gives the return type of the operation, the class to which the operator belongs, the operator that is being overloaded, and the parameter to the operation (the second operand). Some operators are inherently unary (they involve only one operand; for example, the `!` operator); these do not take parameters when overloaded.

Initialization Lists and `vector`-type Class Members

When we initialize `vector` objects, the syntax is slightly different than for real arrays declared in a main program. That's because `vectors` are actually classes, and in the variable declaration we call their single parameter constructor to initialize them to a certain size. For example, the statement

```
vector<int> code(3);
```

creates an `vector` of 3 ints. In this statement, we're calling the constructor `vector::vector(int size)`, and in this case it allocates memory for the array and everything's just peachy. Here's where the problem occurs:

```
class Padlock
{
    private:
        vector<int> code(3);
};
```

Knowing that this syntax causes a call to `vector`'s single parameter constructor, the obvious question is, when is this call actually made? Actually, this code will cause a compiler error, because a constructor cannot be called from within a class definition. The constructor for objects that are members of a class is called right before the constructor for that container class. The only acceptable syntax inside the class definition, then, is

```
class Padlock
{
    private:
        vector<int> code;
};
```

Now, right before `Padlock::Padlock()` is called (it's not defined, but it exists by default), `vector::vector()` is called. Of course, this isn't really what we want, because the default constructor for the `vector` class initializes the object to size 0. We could use `vector::resize(3)` to then set the size of the `vector`, but this is silly from the standpoints of both efficiency and elegance. We can call a non-default constructor for `vector`-type members by using initialization lists. Initialization lists are placed before the function body of the constructor of the container class:

```
class Padlock
{
    private:
        vector<int> code;
    public:
        Padlock();
};

Padlock::Padlock() : code(3) { /* no body */ }
```

Now, instead of calling `vector::vector()` to initialize `code`, `vector::vector(3)` is called, and `code` is initialized to size 3 as desired.

Inheritance

Inheritance provides us with the ability to create new classes based on existing classes. The original class is called the *base* class, and the newly created class is called the *derived* class. The derived class inherits data members and member functions from a previously defined base class. There are three types of inheritance that can occur: public, private and protected.

public: Derived objects are accessible by the base class objects

private: Derived objects are inaccessible by the base class

protected: Derived classes and friends can access protected members of the base class, this technically breaks encapsulation.

Some examples of Inherited class ideas:

Base Class	Derived Classes
Employee	FacultyMember StaffMember
Student	GraduateStudent UndergraduateStudent

Other information to keep in mind.

Friend functions are not inherited.

Objects of a derived class can be treated as objects of the base class, the reverse is not true.

To override a base-class member function: In the derived class, supply a new version of that function with the same signature, same function name, but a different definition

When the function is then mentioned by name in the derived class, the derived version is automatically called.

The scope-resolution operator may be used to access the base class version from the derived class

Below is an example of a base class Point and a derived class Circle demonstrating some of the above features.

Public Inheritance

Base class Point:

```
class Point
{
    public:
        Point( int = 0, int = 0 );           // constructor
        ~Point();
    protected:                             // accessible by derived classes
        int x, y;                           // x and y coordinates of Point
};

Point::Point( int a, int b )
{
    x = a;
    y = b;

    cout << "Point constructor: "
         << '[' << x << ", " << y << ']' << endl;
}

Point::~~Point()
{
    cout << "Point destructor: "
         << '[' << x << ", " << y << ']' << endl;
}
```

Derived class Circle:

```
class Circle : public Point                // class Circle inherits from class Point using
{
    public:
        Circle( double r = 0.0, int x = 0, int y = 0 );
        ~Circle();
    private:
        double radius;
};

Circle::Circle(double r, int a, int b): Point(a,b) // class Circle constructor calls
{                                                    // the base constructor
    radius = r;
    cout << "Circle constructor: radius is "
         << radius << " [" << x << ", " << y << ']' << endl;
}

Circle::~~Circle()
{
    cout << "Circle destructor: radius is "
         << radius << " [" << x << ", " << y << ']' << endl;
}
```