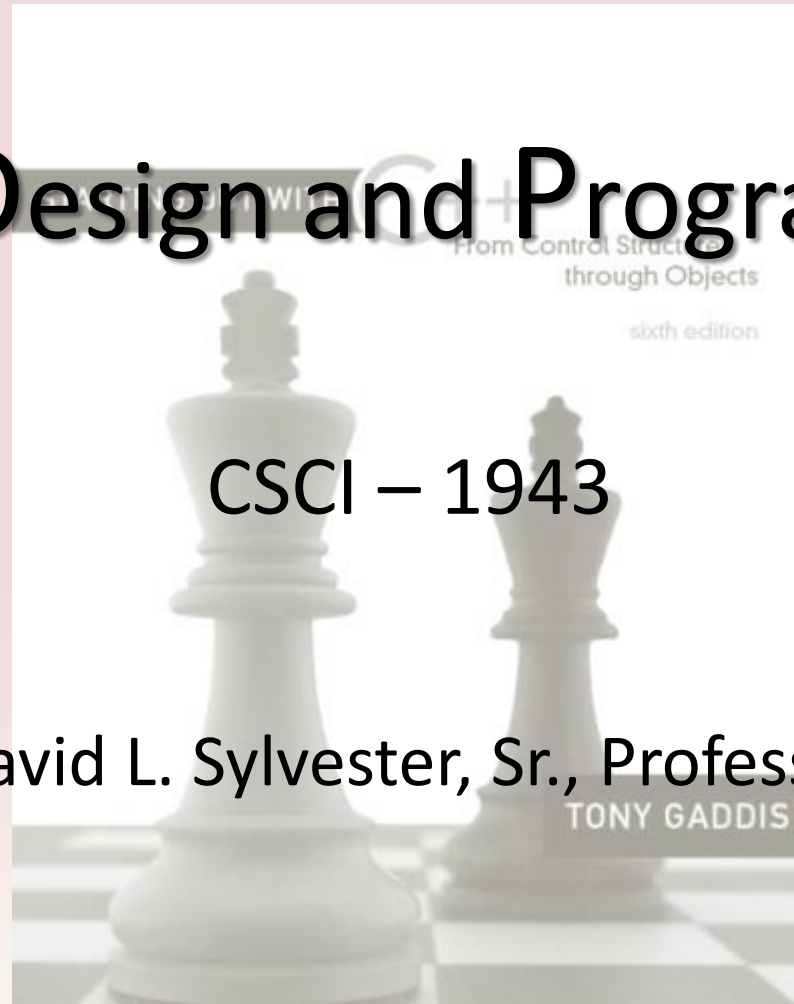


System Design and Programming II

CSCI – 1943

David L. Sylvester, Sr., Professor



Chapter 13

Introduction to Classes

Procedural and Object-Oriented Programming

Procedural programming is a method of writing software. It is a programming practice centered on the procedures or action that take place in a program. Object-oriented programming is centered around the object. Objects are created from abstract data types that encapsulate data and functions together.

Encapsulation is the process of combining data and functions into a single unit called class.

Two programming methods used today

Procedural programming - what we have been using so far

Object-oriented programming (OOP)

Typical procedural programming consists of data being stored in a collection of variables and/or structures, coupled with a set of functions that perform operations on the data. (The data and the functions are separate entities.

Ex: A program written to work with the geometry of a rectangle might have

Variables

```
double width; // to hold the rectangle's width
```

```
double height; // to hold the rectangle's height
```

Functions

```
setData() // to store values in width and length
```

```
displayWidth() // to display the rectangle's width
```

```
displayLength() // to display the rectangle's length
```

```
displayArea() // to display rectangle's area
```

Usually the variables and data structures in a procedural program are passed to the function that performs the desired operations. The focus on procedural programming is on creating the functions that operate on the program's data.

Procedural programming has worked well for software developers for many years. However, as programs become larger and more complex, the separation of a program's data and the code that operates on the data can lead to problems. (i.e. if the format of the data is altered.)

When the structure of the data changes, the code that operates on the data must also change to accept the new format.

Whereas procedural programming is centered on creating procedures or functions, object-oriented programming is centered on creating objects. An object is a software entity that contains both data and procedures. Procedures that an object performs are called member functions.

The object is a self-contained unit consisting of attributes (data) and procedures (functions).

In other programming languages, the procedures that an object performs are often called methods.

Object-Oriented Programming (OOP) addresses the problem that can result from separation of code and data through encapsulation and hiding data.

Data hiding refers to an object's ability to hide its data from code that is outside the object. Only the object's member functions may directly access and make changes to the object's data. An object typically hides its data, but allows outside code to access its member functions. When an object's internal data are hidden from outside code, and access to that data is restricted to the object's member functions, the data are protected from accidental corruption.

Also, the programming code outside the object does not need to know about the format or internal structure of the object's data. The code only needs to interact with the object's function. When a programmer changes the structure of an object's internal data, he or she also modifies the object's member function so they may properly operate on the data. The way in which outside code interacts with the member functions, does not change.

Object-oriented Example

Automobile

ignition switch, steering wheel, gas pedal, brake pedal and gear shift (each having simple user interfaces , starting vehicle, steering, etc.)

Users need not have any mechanical knowledge, meaning more people are likely to become customers. It's good for users of automobiles because they can learn just a few simple procedures and operate almost any vehicle.

A real-world program is rarely written by only one person. Even the programs you have created so far weren't written entirely by you. If you incorporated C++ library functions, or objects like `cin` and `cout`, you used code written by someone else. In the world of professional software development, programmers commonly work in teams, buy and sell their code, and collaborate on projects. With OOP, programmers can create objects with powerful engines tucked away "under the hood", protected by single interfaces that safeguard the object's algorithms.

Object Reusability

In addition to solving the problems of code/data separation, the use of OOP has also been encouraged by the trend of object reusability. An object is not a stand-alone program, but is used by programs that need its service.

Ex: Sharon is a programmer who has developed an object for rendering 3D images. She is a math whiz and knows a lot about computer graphics, so her object is coded to perform all the necessary 3D mathematical operations to handle the computer's video hardware. Tom, who is writing a program for an architectural firm, needs his application to display 3D images for buildings. Because he is working under a tight deadline and does not possess a great deal of knowledge about computer graphics, he can use Sharon's object to perform the 3D rendering. (*Normally this is done for a small fee.*)

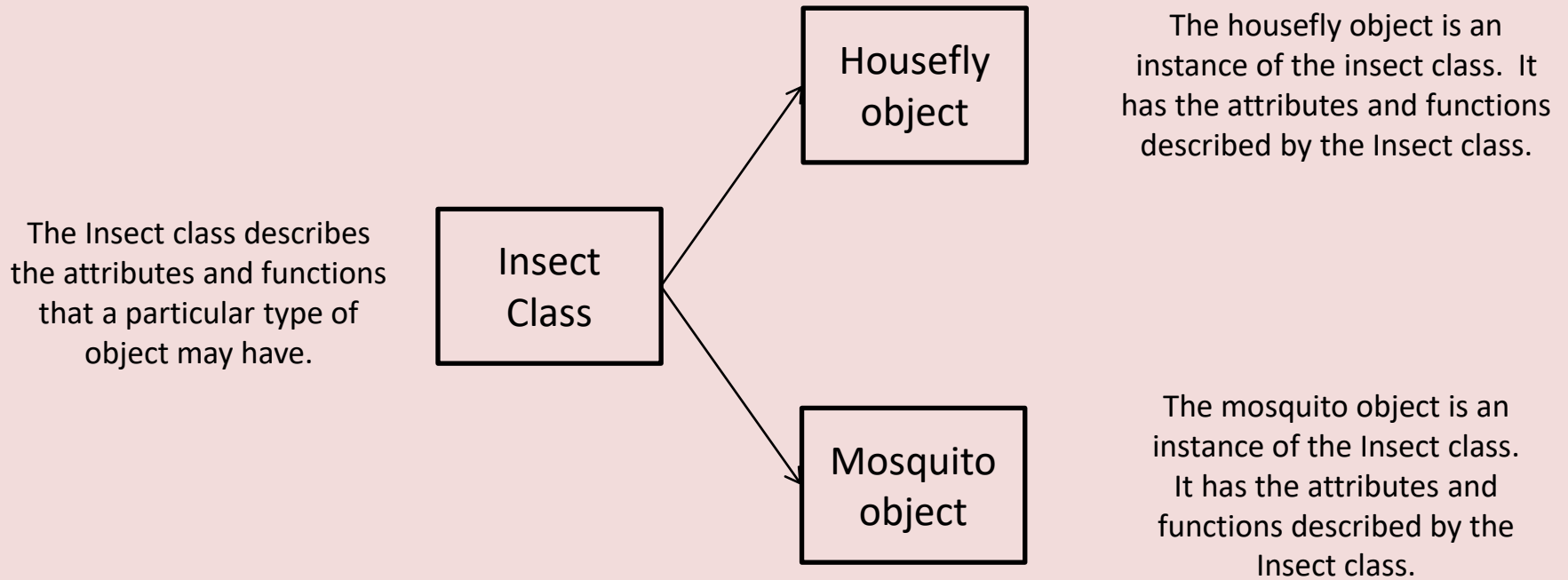
Classes and Objects

Before an object can be created, it must be designed by a programmer. The programmer determines the attributes and functions that are necessary, and then creates a class. A class is code that specifies the attributes and member functions that a particular type of object may have. A class can be thought of as a blueprint that objects may be created from; a blueprint being a detailed description of the object. Using this blueprint, several separate instances can be created.

A class is not an object, but it is a description of an object. When the program is running, it uses the class to create, in memory, as many objects of a specific type as needed. Each object that is created from a class is called an instance of the class.

Ex: An entomologist enjoys writing computer programs. She designs a program to catalog different types of insects. As part of the program, she creates a class named Insect, which specifies attributes and member functions for holding and manipulating data common to all types of insects. She then writes a programming statement that creates a housefly object, which is an instant of the Insect class. She then writes programming statements that create a mosquito object. The mosquito object is also an instance of the Insect class.

Even though the housefly and mosquito objects are two separate entities in the computer's memory, they were both created from the Insect class; each object having the attributes and member functions described by the Insect class.



Previously we discussed how a procedural program that works with rectangles might have variables and hold the rectangle's width and length, and separate functions to do things like store values in the variables and make decisions; passing variables to functions as needed. In an object-oriented program, we would create a Rectangle class which should encapsulate the data (width and length) and the functions that work with the data.

In the object-oriented approach, the variables and functions are all members of the Rectangle class.

Ex:

Member Variables double width; double length;
Member Functions void setWidth(double w) {. . . function code . . .} void setLength(double len) {. . . function code . . .} void getWidth() {. . . function code . . .} void getLength() {. . . function code . . .} void getArea() {. . . function code . . .}

When we need to work with a rectangle in our program, we create a Rectangle object, which is an instance of the Rectangle class. When we need to perform an operation on the Rectangle object's data, we use the object to call the appropriate private member function.

Ex: To get the area of a rectangle, we use the object to call the getArea member function. The getArea member function would be designed to calculate the area of that object's rectangle and return the value.

Using a Class You Already Know

When we need to work with a rectangle in our program, we create a Rectangle object, which is an instance of the Rectangle class. When we need to perform an operation on the Rectangle object's data, we use the object to call the appropriate private member function.

Ex: To get the area of a rectangle, we use the object to call the `getArea` member function. The `getArea` member function would be designed to calculate the area of that object's rectangle and return the value.

```
cityName = "Baton Rouge";
```

After this statement executes, the string "Baton Rouge" will be stored in the cityName object. "Baton Rouge" will become the object's data.

Introduction to Classes

The class is the construct primarily used to create objects. A class is similar to a structure. It is a data type defined by the programmer, consisting of variables and functions.

```
Ex:   class Classname
      {
          declaration;
          // more declarations
          //...
          // ...
      };
```

The declaration statements inside a class declaration are for variables and functions that are members of the class.

```
Ex:   class Rectangle
      {
          double width;
          double length;
      };           // don't forget the semicolon
```

There would be a problem with the previous defined class. Unlike structures, the members of a class are private by default. Private class member cannot be accessed by programming statements outside the class. So, no statements outside this Rectangle class can access the width and length members. Again, in C++, a class's private members are hidden, and can be accessed only by functions that are members of the same class. A class's public members may be accessed by code outside the class.

Access Specifiers

C++ provides the key words private and public which you may use in class declarations. These key words are known as access specifiers because they specify how class members may be accessed.

General format of a class declaration that uses the private and public access specifiers.

```
Ex:   class Classname
      {
      private:
          // declarations of private members
          // ...
      public:
          // Declarations of public member
          //
      };
```

Notice that the access specifiers are followed by a colon (:), and then followed by one or more member declaration.

Public Member Functions

To allow access to a class's private member variables, you create public member functions that work with the private member variables.

Ex:

```
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth( ) const;
        double getLength( ) const;
        double getArea( ) const;
};
```

const - Specifies that the function will not change any data stored in the calling object. Will get an error if changes are attempted.

Private member variables

Public member functions

In this declaration, the member variables width and length are declared as private. However, the member functions are declared as public, which means they can be called from statements outside the class. (*The public functions provide an interface for code outside the class to use Rectangle objects.*)

Even though the default access of a class is private, **it's still a good idea to use the private key word to explicitly declare private members.** This clearly documents the access specification of all the members of the class.

Using const with Member Functions

When the key word **const** appears after the parentheses in a member function declaration, it specifies that the function will not change any data stored in the calling object. If you inadvertently write code in the function that changes the calling object's data, the compiler will generate an error. Note: The const key word must also appear in the function header.

Placement of public and private Members

There is no rule requiring you to declare private member before public members. Also, it is not require that all members of the same access specification be declared in the same place.

Three Valid way to Declare a **class**

```
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth( ) const;
        double getLength( ) const;
        double getArea( ) const;
};
```

```
class Rectangle
{
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth( ) const;
        double getLength( ) const;
        double getArea( ) const;
    private:
        double width;
        double length;
};
```

```
class Rectangle
{
    private:
        double width;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth( ) const;
        double getLength( ) const;
        double getArea( ) const;
    private:
        double length;
};
```

Defining Member Functions

The Rectangle class declaration contains declarations (prototypes) of five member functions: (*setWidth*, *setLength*, *getWidth*, *getLength*, and *getArea*)

```
// setWidth: assigns its argument to the private member width.
```

```
void Rectangle::setWidth(double w)
{
    width = w;
}
```

```
// setLength: assigns its argument to the private member length.
```

```
void Rectangle::setLength(double len)
{
    length = len;
}
```

```
// getWidth: returns the value in the private member width.
```

```
double Rectangle::getWidth( ) const
{
    return width;
}
```

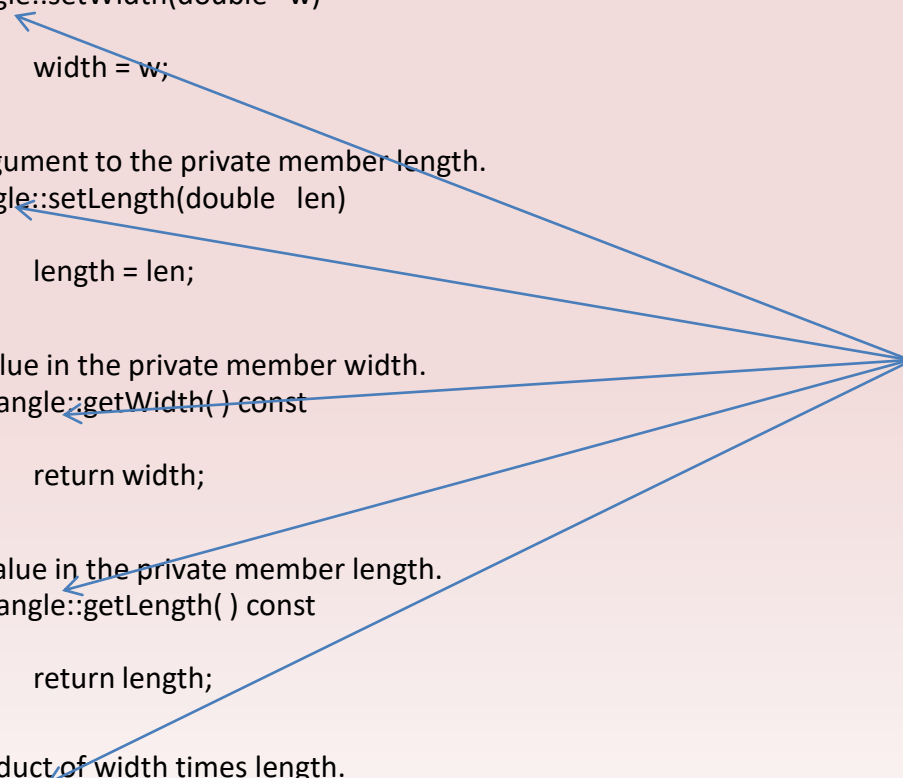
```
// getLength: returns the value in the private member length.
```

```
double Rectangle::getLength( ) const
{
    return length;
}
```

```
// getArea: returns the product of width times length.
```

```
double Rectangle::getArea( ) const
{
    return width * length;
}
```

classname:: precedes each function definition. Classname being Rectangle.



In each function definition, the following precedes the name of each function: `Rectangle::`

The two colons are called the scope resolution operator. When `Rectangle::` appears before the name of a function in a function header, it identifies the function as a member of the `Rectangle` class.

Ex: `ReturnType className::functionName(parameterList)`

In the general format, **ReturnType** is the function's return type. **className** is the name of the class that the function is a member of. **functionName** is the name of the member function. **parameterList** is an optional list of parameter variable declarations.

Accessors and Mutators

A member function that gets a value from a class's member variable but does not change it is known as an accessor. A member function that stores a value in member variables or changes the value of member variable in some other way is known as a mutator. In the Rectangle class, the member functions `getWidth` and `getLength` are accessors (only passes the values of width and length), and the member functions `setLength` and `setWidth` are mutators (assigns width and length to a value).

In the Rectangle class, all accessors are marked as constants (`const`). Being that accessors are member functions that do not change a member's value, it is a good practice to mark all accessor functions as `const`. This ensures that you do not inadvertently write code in the function that changes the calling object's data.

Defining an Instance of a Class

Like structure variables, class objects are not created in memory until they are defined. Remember, a class declaration by itself does not create an object, but is merely the description of an object. We must use the class to create one or more objects, which are called instances of the class.

Class objects are created with simple definition statements.

Ex: `ClassName objectName;`

ClassName being the name of a class and objectName being the name we are giving the object.

Ex: `Rectangle box;`

Defining a class object is called the instantiation of a class. In the above statement, box is an instance of the Rectangle class.

Accessing an Object's Member

The box object is an instance of the Rectangle class. To change the value in the box object's width variable, you must use the box object to call the setWidth member function.

Ex: `box.setWidth(12.7);`

Just as you use the dot operator to access a structure's member, you use the dot operator to call a class's member function.

Examples

<code>box.setWidth(12.7);</code>	Uses the box object to call the setWidth member function, passing 12.7 as an argument.
<code>box.setLength(4.8);</code>	Sets box's length to 4.8.
<code>x = box.getWidth();</code>	Assigns box's width to x.
<code>cout << box.getLength();</code>	Displays box's length.
<code>cout << box.getArea();</code>	Displays box's area.

Note: Inside the Rectangle class's member functions, the dot operator is not used to access any of the class's member variables. This is because when an object is used to call a member function, the member function has direct access to that object's member variable.

Sample Code

```
// This program demonstrates a simple class.
#include <iostream>
using namespace std;

// Rectangle class declaration.
class Rectangle
{
private:
    double width;
    double length;
public:
    void setWidth(double);
    void setLength(double);
    double getWidth() const;
    double getLength() const;
    double getArea() const;
};

// setWidth assigns a value to the width member. *
void Rectangle::setWidth(double w)
{
    width = w;
}

// setLength assigns a value to the length member. *
void Rectangle::setLength(double len)
{
    length = len;
}

// getWidth returns the value in the width member. *
double Rectangle::getWidth() const
{
    return width;
}
```

```
// getLength returns the value in the length member. *
double Rectangle::getLength() const
{
    return length;
}

// getArea returns the product of width times length. *
double Rectangle::getArea() const
{
    return width * length;
}

// Function main
int main()
{
    Rectangle box; // Define an instance of the Rectangle class
    double rectWidth; // Local variable for width
    double rectLength; // Local variable for length

    // Get the rectangle's width and length from the user.
    cout << "This program will calculate the area of a\n";
    cout << "rectangle. What is the width? ";
    cin >> rectWidth;
    cout << "What is the length? ";
    cin >> rectLength;

    // Store the width and length of the rectangle
    // in the box object.
    box.setWidth(rectWidth);
    box.setLength(rectLength);

    // Display the rectangle's data.
    cout << "Here is the rectangle's data:\n";
    cout << "Width: " << box.getWidth() << endl;
    cout << "Length: " << box.getLength() << endl;
    cout << "Area: " << box.getArea() << endl;
    return 0;
}
```

Sample Code

```
// Demonstrates the use of a class.
#include <iostream>
#include <iomanip>
#include <string>
#include <fstream>
#include <math.h>

using namespace std;

fstream datafile;
fstream outfile;
const int SIZE = 10;
class PersonallInfo
{
private:
    char fname[12];
    char lname[18];
    char sex;
    int weight;
    int age;
    int age_double;
    int age2();
    void getinfo();
    void putinfo() const;
public:
    void results();
    void results2();
};

void PersonallInfo::results()
{
    getinfo();
    age2();
}
```

```
void PersonallInfo::results2()
{
    putinfo();
}

// main function
int main()
{
    datafile.open("names2.txt", ios::in);
    outfile.open("namesout.dat", ios::out);

    PersonallInfo kids[SIZE];

    for (int x = 1; x < SIZE; x++)
        kids[x].results();

    cout << endl << endl << endl;
    for (int x = 1; x < SIZE; x++)
        kids[x].results2();
    return 0;
}

// Function to input data
void PersonallInfo::getinfo()
{
    datafile >> fname;
    datafile >> lname;
    datafile >> sex;
    datafile >> weight;
    datafile >> age;
}
```

```
// Function to double age
int PersonallInfo::age2()
{
    age_double = age * 2;
    return age_double;
}

// Function to output data
void PersonallInfo::putinfo() const
{
    outfile << fname;
    outfile << "\t" ;
    outfile << lname;
    outfile << "\t" ;
    outfile << sex;
    outfile << "\t" ;
    outfile << weight;
    outfile << "\t" ;
    outfile << age ;
    outfile << "\t" ;
    outfile << age_double;
    outfile << endl;
}
```

Inline Functions and Performance

A lot goes on behind the scenes each time a function is called. A number of special items, such as the function's return address in the program and the values of arguments, are stored in a section of memory called the stack. In addition, local variables are created and a location is reserved for the function's return value. All this overhead, which sets the stage for a function call, takes precious CPU time. Although the time needed is minuscule (small), it can add up if a function is called many times, as in a loop.

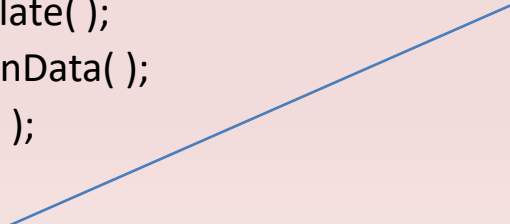
Inline functions are compiled differently than other functions. In the executable code, inline functions aren't called in the conventional sense. In a process known as inline expansion, the computer replaces the call to an inline function with the code of the function itself. This means that the overhead needed for a conventional function call is not necessary for an inline function. And can result in improved performance.

Functions that are defined inside of the struct or class declaration are considered to be inline. An inline function has its statements placed in the code at every location that calls the function. The result is that the object code is longer but the execution time is faster.

Ex:

```
class Invoice
{
    char stProduct[20];
    float fPrice, fAmtDue, fTaxRate;
    void iQuantity( );
    void Calculate( );
    void ObtainData( );
    void Print( );
public:
    Invoice( )
    {
        // This function is inline because the definition
        // of the function is within the class declaration.
        fTaxRate = 0.875;
    }
    void Run( );
};
```

Notice that there is no semicolon after the function header `Invoice()`. This is not a prototype, it is the actual function header and the function definition.



Constructors

A constructor is a member function that has the same name as the class. It is automatically called when the object is created in memory, or instantiated. They are very useful for initializing member variables or performing other setup operations.

```
// This program demonstrates a constructor.
#include <iostream>
using namespace std;
// Demo class declaration.
class Demo
{
public:
    Demo(); // Constructor
};
Demo::Demo()
{
    cout << "Welcome to the constructor!\n";
}
// Function main.
int main()
{
    Demo demoObject; // Define a Demo object;
    cout << "This program demonstrates an object\n";
    cout << "with a constructor.\n";
    return 0;
}
```

Notice that the constructor's function header looks different than that of a regular member function. There is no return type – **not even void**. This is because constructors are not executed by explicit function calls and cannot return a value.

demoObject's constructor executes automatically when the object is defined. Because the object is defined before the cout statement in the function main, the constructor displays its message first.

Program Output

```
Welcome to the constructor!
This program demonstrates an object
with a constructor.
```


Destructors

Destructors are member functions that are automatically called when an object is destroyed. They have the same name as the class, preceded by a tilde (~) character.

```
// This program demonstrates a constructor.
#include <iostream>
using namespace std;

// Demo class declaration.

class Demo
{
    public:
    Demo( ); // Constructor
    ~Demo( ); // Destructor
};

Demo::Demo( )
{
    cout << "Welcome to the constructor!\n";
}

Demo::~~Demo( )
{
    cout << "The destructor is now running.\n";
}
```

```
// Function main.
int main()
{
    Demo demoObject; // Define a Demo object;

    cout << "This program demonstrates an object\n";
    cout << "with a constructor and destructor.\n";
    return 0;
}
```

Program Output

```
Welcome to the constructor!
This program demonstrates an object
with a constructor and destructor.
The destructor is now running.
```