

Classes - Constructor Initialization Lists

Declaration: A declaration tells the type of an object, but doesn't allocate storage.

Definition: A definition allocates storage for an object.

Instantiate: Definition of a (class) object.

C and Pascal are "Action Oriented" (functions) versus C++ which is "Object Oriented" (classes)

Ask not what I can do to a class, ask what a class can do to itself

Creating a class: A class is composed of a state and behaviors. The state is (are) the private variables that define the object. The behaviors are the methods that show, access or manipulate the state.

Here is the header file for a class (*rect.h*):

```
#ifndef _RECT_H           // Assembler directive to see if this header file has already been declared
#define _RECT_H         // If not already declared - declare it now

// Always start the assembler name with _ (underscore) and use capital letters. It cannot match any
// identifier in any of your files or system files.

class Rectangle          // Class Declaration
{
    public:              // These methods (functions) can be called from client programs

        // Constructors
        Rectangle();
        Rectangle(double, double);

        // Accessors
        double GetLength() const;
        double GetWidth() const;
        double GetArea() const;

        // Mutators
        void SetLength(double);
        void SetWidth(double);

    private:            // These methods or variables cannot be access from the client program
        double myLength;
        double myWidth;
};

#endif
```

Here is the code file (*rect.cpp*)

```
#include "rect.h"       // Need the declaration of the class

Rectangle::Rectangle() // :: is the scope resolution operator - See GetLength for details on ::
{
    myLength = 0;      // upon definition - set privates values to zero is no parameters
    myWidth = 0;
}
```

```

Rectangle::Rectangle(double len, double width)
{
    myLength = len;    // upon definition use parameters to set private values
    myWidth = width;
}

double Rectangle::GetLength() const    // Left of :: indicates the class the right side function belongs to
{                                       // Mays::Willie, means Willie belongs to the Mays family
    return myLength;
}

double Rectangle::GetWidth() const    // const at end means this function may not change the private
values                                // Any attempt will generate a compiler error
{
    return myWidth;
}

double Rectangle::GetArea() const
{
    return myLength * myWidth;
}

void Rectangle::SetLength(double len)  // Private values can only be set by methods belonging to the class
{
    myLength = len;
}

void Rectangle::SetWidth(double width)
{
    myWidth = width;
}

```

Here is a program that uses the class (*test.cpp*). When building the project you need to include *rect.cpp* in the project along with *test.cpp*.

```

#include <iostream.h>
#include "rect.h"

int main ()
{
    Rectangle r1;           // Definition - This is called the instantiation of an object
    Rectangle r2(2,3);     // Definition - This is called the instantiation of an object

    cout << r1.GetLength() << " " << r1.GetWidth() << endl;
    cout << r2.GetLength() << " " << r2.GetWidth() << endl;

    return 0;
}

```

Output is:

```

0 0
2 3

```

Array of Classes:

To have an array of a class you **MUST** have a constructor with **NO** parameters. For example if we took the default constructor away and from the client program stated:

```
apvector<Rectangle> r2(10);
```

We would get the following error message:

```
Error : class has no default constructor  
apvector.cpp line 31  myList(new itemType[size])
```

Constructor Initialization List:

Constructing an object sometimes causes a problem. If we add a class *Big*:

```
class Big
{
    public:
        Big();
        Big(int size);
        apvector<Rectangle> GetRectangles();
    private:
        apvector<Rectangle> myLots(20);           // I want 20 rectangles here
        int mySize;
};
```

This will produce the error:

```
Error : ';' expected
rect2.cpp line 48  apvector<Rectangle> myLots(20); // I want 20 rectangles here
```

Removing the (20) gets rid of the error, but now how do I create 20 rectangles in the private field? Use the "Constructor Initialization List". The constructors are:

```
Big::Big() : myLots(10), mySize(10)
{
    // No code here
}

Big::Big(int size) : myLots(size), mySize(size)
{
    // No code here
}
```

The default constructor will give 10 rectangles, whereas if I want 20 rectangles I need to pass that information while instantiating a *Big* variable. For example:

```
int main ()
{
    Big b(20);           // Definition - I want 20 rectangles
    apvector<Rectangle> r(20);

    r = b.GetRectangles();           // Pull out the 20 rectangles
    for (int i = 0; i < 20; i++)
        cout << "Rectangle #" << i << " " << r[i] << endl;

    return 0;
}
```

This is the only way to solve this problem. In general always use a "Constructor Initialization List" for member variables.

It is more efficient and usually better to initialize private variables using an initializer list rather than in the body of a constructor. All non-built-in types will be constructed before the statements in the body of the constructor are executed, so initializing them explicitly helps ensure that private variables have values in the body of the constructor.

When an initializer list is used, the private variables are constructed in the order in which they appear in the class declaration, not in the order they appear in the initialization list.

If you have a class with a private data member of another of another class that has no default constructor, then you must initialize it using an initializer list.

A data member is initialized and data allocated before the constructor code runs. When using either the default constructor if no initializer list is given, or with the initializer list, then the constructor code runs. If you do not use as initializer list, this means that a data member of a complex class will have its space allocated and initialized to default values, then in the code you would reassign values. A call to a constructor in the initializer list is more efficient, assigning values only once.

```
#include <iostream.h>
#include <iomanip.h>
```

```
class Test1
{
    public:
        Test1(int x) : size(x) {};
    private:
        int size;
};
```

```
class Test2
{
    public:
        Test2() : up(0) {};
    private:
        Test1 x;
        int up;
};
```

```
int main()
{
    Test2 y;

    return 0;
}
```

```
/*
```

```
Error : cannot construct direct member 'x'
test.cpp line 15 Test2() : up(0) {};
```

```
*/
```