Defining Your Own Data Type

Chun-Rong Huang

The Struct in C++

- A structure is a user-defined type that you define using the keyword struct, so it is often referred to as a struct
- A struct in C++ is functionally replaceable by a class
- Almost all the variables that you have seen up to now have been able to store a single type of entity
- Any physical object you can think of needs several items of data
  - Book
    - Title, author, publisher, date of publication, number of pages, price, topic or classification, and ISBN number

Defining a struct

- You could declare a structure to accommodate a book

```
struct BOOK
{
  char Title[80];
  char Author[80];
  char Publisher[80];
  int Year;
};
```

- It creates a new type for variables, and the name of the type is BOOK
- The elements Title, Author, Publisher, and Year enclosed between the braces in the definition above may also be referred to as members or fields of the BOOK structure

```
BOOK Novel;                                    // Declare variable Novel of type BOOK
```

Initializing a struct

- The first way to get data into the members of a struct is to define initial values in the declaration

```
BOOK Novel =
{
 "Paneless Programming",              // Initial value for Title
 "I.C. Fingers",                      // Initial value for Author
 "Gutter Press",                      // Initial value for Publisher
  1981                                // Initial value for Year
};
```

- The initializing values appear between braces, separated by commas, in much the same way that you defined initial values for members of an array

Accessing the Members of a struct

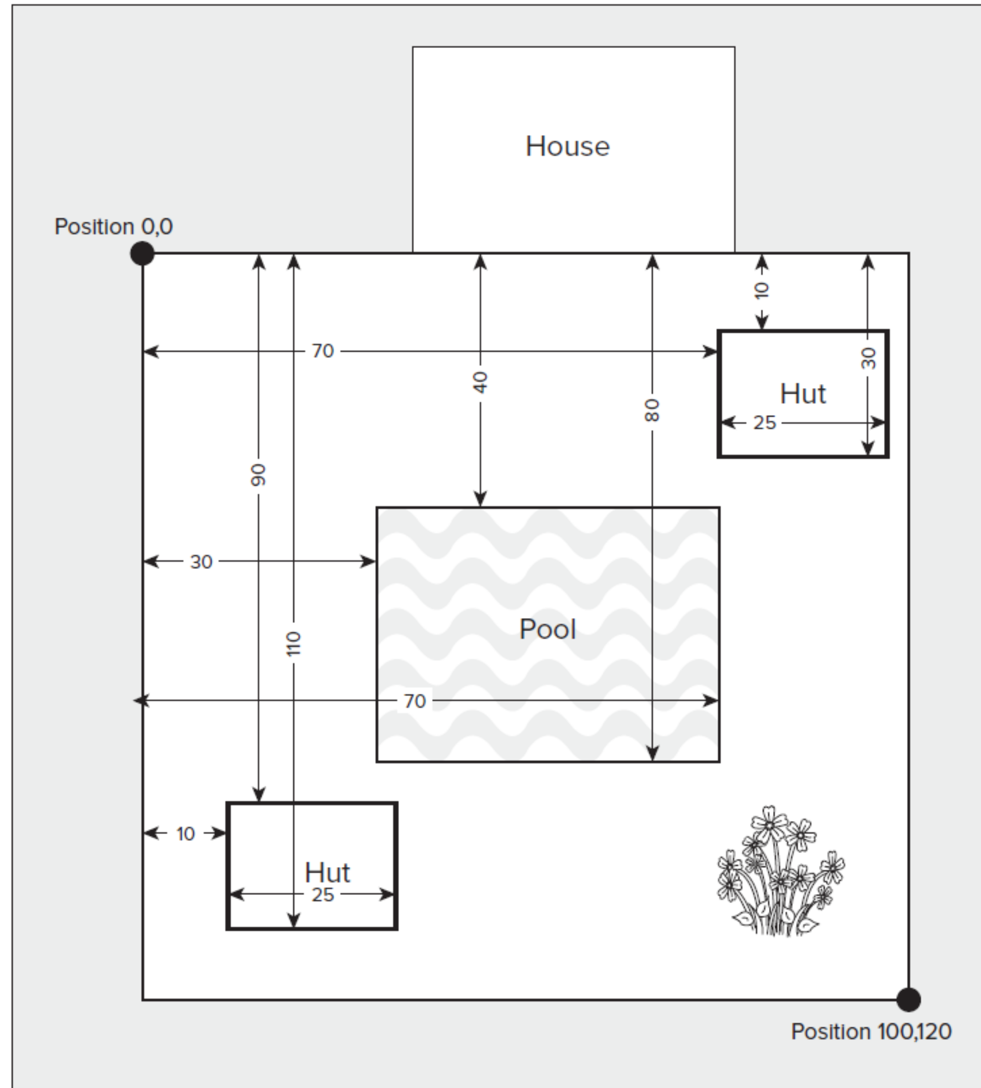
- To access individual members of a struct, you can use the member selection operator

```
Novel.Year = 1988;
```

- You can use a member of a structure in exactly the same way as any other variable of the same type as the member

```
Novel.Year += 2;
```

Using structs



Using structs

- Because the yard, huts, and pool are all rectangular, you could define a struct type to represent any of these objects

```
struct RECTANGLE
{
  int Left;                          // Top-left point
  int Top;                           // coordinate pair

  int Right;                         // Bottom-right point
  int Bottom;                        // coordinate pair
};
```

- Ex7_01

IntelliSense Assistance with Structures

```
58  }
59
60  // Function to Move a Rectangle
61  void MoveRect(RECTANGLE& aRect, int x, int y)
62  {
63     int length = aRect.Right - aRect.Left;  // Get length of rectangle
64     int width = aRect.Bottom - aRect.Top;   // Get width of rectangle
65
66     aRect.Left = x;                         // Set top left point
67     aRect.Top = y;                          // to new position
68     aRect.Right = x + length;               // Get bottom right point as
69     aRect.
70           Botto      coordinate pair
71           Left       File: ex7_01.cpp
72           Right
73           Top
74
75
```

The struct RECT

- Rectangles are used a great deal in Windows programs
- For this reason, there is a RECT structure predefined in the header file windows.h

```
struct RECT
{
  LONG left;                                    // Top-left point
  LONG top;                                     // coordinate pair

  LONG right;                                   // Bottom-right point
  LONG bottom;                                  // coordinate pair
};
```

- MFC also defines a class called CRect, which is the equivalent of a RECT structure

Using Pointers with a struct

- You can create a pointer to a variable of a structure type
- In fact, many of the functions declared in windows.h that work with RECT objects require pointers to a RECT as arguments
  - This avoids the copying of the whole structure when a RECT argument is passed to a function

```
RECT* pRect(nullptr);                    // Define a pointer to RECT
pRect = &aRect;                          // Set pointer to the address of aRect
```

Using Pointers with a struct

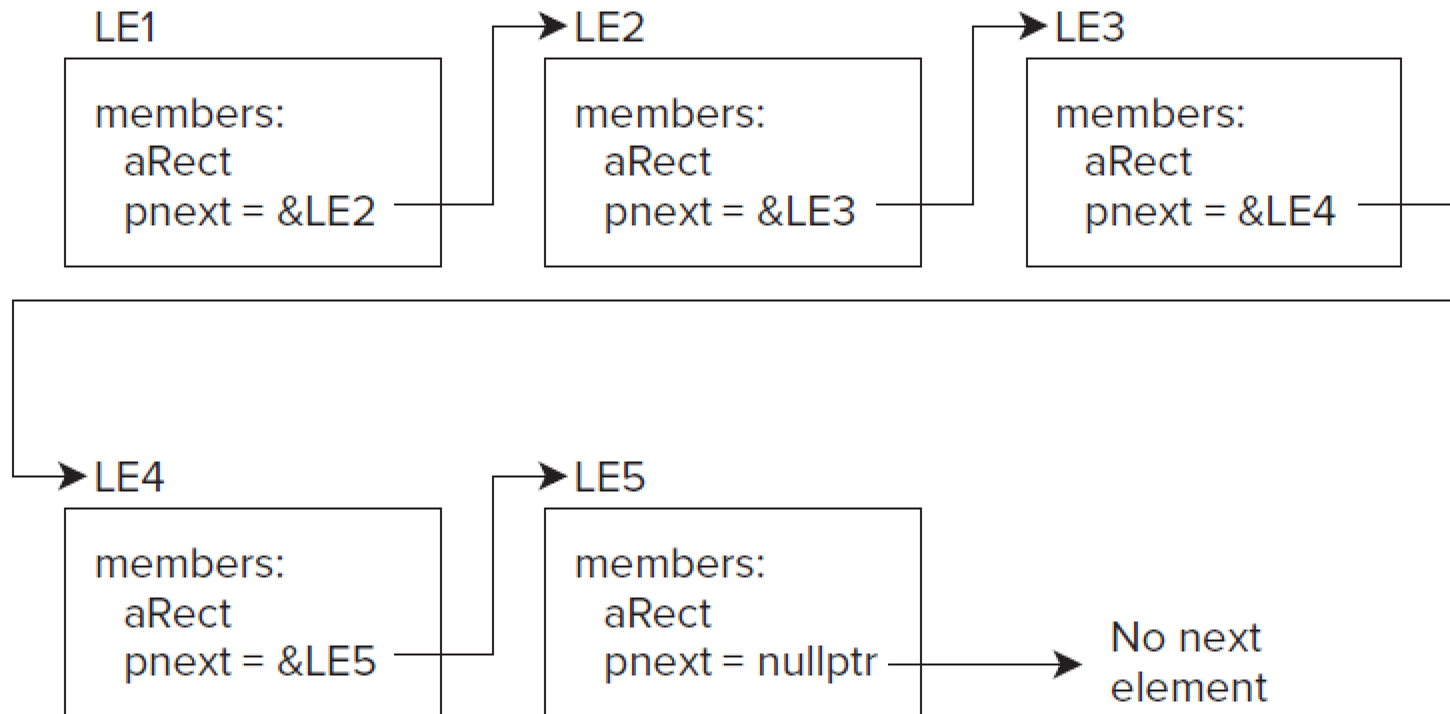
- A struct can't contain a member of the same type as the struct being defined
  - It can contain a pointer to a struct, including a pointer to a struct of the same type

```
struct ListElement
{
  RECT aRect;                                    // RECT member of structure
  ListElement* pNext;                            // Pointer to a list element
};
```

- This allows objects of type ListElement to be daisy-chained together, where
  - Each ListElement can contain the address of the next ListElement object in a chain
  - The last in the chain having the pointer as nullptr

Using Pointers with a struct

- Linked list



Accessing Structure Members through a Pointer

```
RECT aRect = {0, 0, 100, 100};
RECT* pRect(&aRect);
```

- You can now access the members of aRect through the pointer with a statement such as this

```
(*pRect).Top += 10;                    // Increment the Top member by 10
```

- The parentheses to dereference the pointer here are essential, because the member access operator takes precedence over the dereferencing operator
- The indirect member selection operator ->

```
pRect->Top += 10;                      // Increment the Top member by 10
```

Data Types, Objects, Classes and Instances

- Native C++ lets you create variables that can be any of a range of fundamental data types
  - **int**, **long**, **double**, and so on
- The variables of the fundamental types don't allow you to model real world objects
- It's hard to model a box in terms of an **int**, for example; however, you can use the members of a **struct** to define a set of attributes for such an object

```
struct Box
{
  double length;
  double width;
  double height;
};
```

Data Types, Objects, Classes and Instances

- With this definition of a new data type called Box, you define variables of this type just as you did with variables of the basic types
- You can then create, manipulate, and destroy as many Box objects as you need to in your program
- This means that you can model objects using structs and write your programs

Data Types, Objects, Classes and Instances

- Object oriented programming (OOP) is based on three basic concepts relating to object types
  - *Encapsulation*
  - *Polymorphism*
  - *Inheritance*
- The notion of a struct in C++ goes far beyond the original concept of struct in C — it incorporates the object-oriented notion of a class

Data Types, Objects, Classes and Instances

- This idea of classes, from which you can create your own data types and use them just like the native types, is fundamental to C++
  - The new keyword **class** was introduced into the language to describe this concept
- The keywords **struct** and class are almost identical in C++, except for the access control to the members
- The keyword **struct** is maintained for backwards compatibility with C
  - Everything that you can do with a **struct**, and more, you can achieve with a class

Data Types, Objects, Classes and Instances

- Take a look at how you might define a class representing boxes

```
class CBox
{
  public:
    double m_Length;
    double m_Width;
    double m_Height;
};
```

- The keyword public followed by a colon that precedes the definition of the members of the class
- It just specifies that the members of the class that follow the keyword are generally accessible, in the same way as the members of a structure are

Data Types, Objects, Classes and Instances

- The variables that you define as part of the class are called **data members** of the class, because they are variables that store data
- You have also called the class CBox instead of Box
- You could have called the class Box, but the MFC adopts the convention of using the prefix C for all class names
- MFC also prefixes data members of classes with m_ to distinguish them from other variables

```
CBox bigBox;
```

Operations on Classes

- In C++, you can create new data types as classes to represent whatever kinds of objects you like
- Classes (and structures) aren't limited to just holding data
  - You can also define member functions or even operations that act on objects of your classes using the standard C++ operators

```
CBox box1;
CBox box2;

if(box1 > box2)            // Fill the larger box
  box1.fill();
else
  box2.fill();
```

Terminology

- A **class** is a user-defined data type
- **Object-oriented programming (OOP)** is the programming style based on the idea of defining your own data types as classes, where the data types are specific to the domain of the problem you intend to solve
- Declaring an object of a class type is sometimes referred to as **instantiation** because you are creating an **instance** of a class

Terminology

- Instances of a class are referred to as **objects**
- The idea of an object containing the data implicit in its definition, together with the functions that operate on that data, is referred to as **encapsulation**

Understanding Class

- A class is a specification of a data type that you define
- It can contain data elements that can either be variables of the basic types in C++, or of other user-defined types
- The data elements of a class may be single data elements, arrays, pointers, arrays of pointers of almost any kind, or objects of other classes
- The data and functions within a class are called **members** of the class

Understanding Class

- The members of a class that are data items are called **data members** and the members that are functions are called **function members** or **member functions**
- The member functions of a class are also sometimes referred to as **methods**

Understanding Class

- The data members are also referred to as **fields**, and this terminology is used with C++/CLI
- 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
- To create a variable of a basic data type, you need to use a declaration statement

Defining a Class

- You defined the Cbox data type using the keyword class as follows

```
class CBox
{
  public:
    double m_Length;                    // Length of a box in inches
    double m_Width;                     // Width of a box in inches
    double m_Height;                    // Height of a box in inches
};
```

- The names of all the members of a class are local to the class
- You can therefore use the same names elsewhere in a program without causing any problems

Access Control in a Class

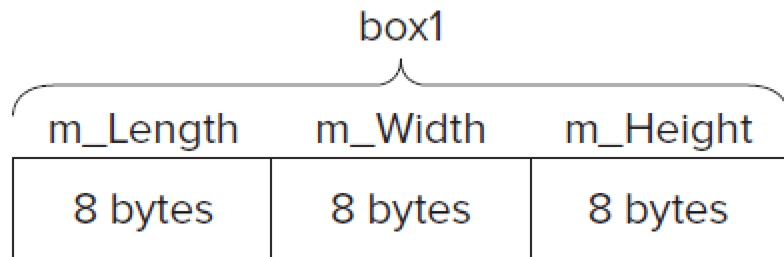
- The public keyword determines the access attributes of the members of the class that follow it
- Specifying the data members as public means that these members of an object of the class can be accessed anywhere within the scope of the class object to which they belong
- You can also specify the members of a class as
  - private
  - protected
- If you omit the access specification altogether, the members have the default attribute, private

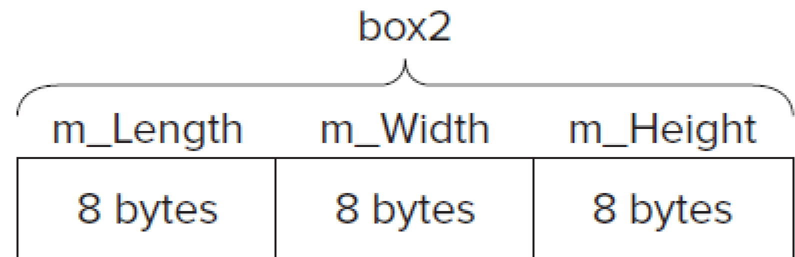
Declaring Objects of a Class

- You declare objects of a class with exactly the same sort of declaration that you use to declare objects of basic types

```
class CBox
{
  public:
    double m_Length;                    // Length of a box in inches
    double m_Width;                     // Width of a box in inches
    double m_Height;                    // Height of a box in inches
};

CBox box1;                              // Declare box1 of type CBox
CBox box2;                              // Declare box2 of type CBox
```





Accessing the Data Members of a Class

- You can refer to the data members of objects of a class using the **direct member selection operator** that you used to access members of a struct

```
box2.m_Height = 18.0;
```

- You can only access the data member in this way in a function that is outside the class, because the m_Height member was specified as having public access
- If it wasn’t defined as public, this statement would not compile
- Ex7_02

Member Functions of a Class

- A member function of a class is a function that has its definition or its prototype within the class definition
- 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
- Ex7_03

Positioning a Member Function Definition

- A member function definition need not be placed inside the class definition

```
class CBox                          // Class definition at global scope
{
  public:
    double m_Length;                // Length of a box in inches
    double m_Width;                 // Width of a box in inches
    double m_Height;                // Height of a box in inches
    double Volume(void);            // Member function prototype
};
```

- Telling the compiler that the function belongs to the class Cbox with the **scope resolution operator**, **::**

```
// Function to calculate the volume of a box
double CBox::Volume()
{
  return m_Length*m_Width*m_Height;
}
```

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
- This avoids much of the overhead of calling the function and, therefore, speeds up your code

Function declared as inline in a class

inline void function()
{ body }

The compiler replaces calls of inline function with body code for the function, suitably adjusted to avoid problems with variable names or scope.

int main(void)
{
...
~~function();~~
{ body }
...
~~function();~~
{ body }
....
}

Inline Functions

- It’s best used for very short, simple functions, such as our function Volume() in the CBox class
  - Because such functions execute faster and inserting the body code does not significantly increase the size of the executable module
- With the function definition outside of the class definition, the compiler treats the function as a normal function
- It’s also possible to tell the compiler that, if possible, you would like the function to be considered as inline

```
// Function to calculate the volume of a box
inline double CBox::Volume()
{
  return m_Length*m_Width*m_Height;
}
```

Class Constructors

- A class constructor is a special function in a class that is responsible for creating new objects when required
- A constructor provides the opportunity to initialize objects as they are created and to ensure that data members only contain valid values
- A class may have several constructors, enabling you to create objects in various ways
- The primary purpose of a class constructor is to assign initial values to the data elements of the class, and no return type for a constructor is necessary or permitted
- Ex7_04

Class Constructors

```
class CBox                          // Class definition at global scope
{
  public:
    double m_Length;                // Length of a box in inches
    double m_Width;                 // Width of a box in inches
    double m_Height;                // Height of a box in inches

    // Constructor definition
    CBox(double lv, double bv, double hv)
    {
      cout << endl << "Constructor called.";
      m_Length = lv;                // Set values of
      m_Width = bv;                    // data members
      m_Height = hv;
    }
```

The Default Constructor

- Try modifying the last example by adding the declaration for box2

```
CBox box2;
```

- You've left box2 without initializing values
  - When you rebuild this version of the program, you get the error message

```
error C2512: 'CBox': no appropriate default constructor available
```

- This means that the compiler is looking for a default constructor for box2
  - Referred to as the no arg constructor because you haven't supplied any initializing values for the data members

The Default Constructor

- A default constructor is one that does not require any arguments to be supplied, which can be either a constructor that has no parameters specified in the constructor definition, or one whose arguments are all optional

```
CBox()                          // Default constructor
{}                              // Totally devoid of statements
```

- Ex7_05

Assigning Default Parameter Values in a Class

- You can specify default values for the parameters to a function in the function prototype
- You can also do this for class member functions, including constructors
- If you put the definition of the member function inside the class definition, you can put the default values for the parameters in the function header

```
// Constructor definition
CBox(double lv = 1.0, double bv = 1.0, double hv = 1.0)
{
  cout << endl << "Constructor called.";
  m_Length = lv;                          // Set values of
  m_Width = bv;                           // data members
  m_Height = hv;
}
```

Assigning Default Parameter Values in a Class

- The declaration of box2 requires a constructor without parameters and either constructor can now be called without parameters

```
warning C4520: 'CBox': multiple default constructors specified
error C2668: 'CBox::CBox': ambiguous call to overloaded function
```

- Ex7_06

Using an Initialization List in a Constructor

- Initialization list

```
// Constructor definition using an initialization list
CBox(double lv = 1.0, double bv = 1.0, double hv = 1.0):
                                m_Length(lv), m_Width(bv), m_Height(hv)
{
  cout << endl << "Constructor called.";
}
```

- This technique for initializing parameters in a constructor is important
  - It’s the only way of setting values for certain types of data members of an object
- The MFC also relies heavily on the initialization list technique

Making a Constructor Explicit

- Implicit conversion

```
CBox(double side): m_Length(side), m_Width(side), m_Height(side) {}
```

```
CBox box;
box = 99.0;
```

- Explicit
  - With the constructor declared as explicit, the statement assigning the value 99.0 to the box object will not compile

```
explicit CBox(double side): m_Length(side), m_Width(side), m_Height(side) {}
```

Making a Constructor Explicit

- Implicit

```
CBox(double lv = 1.0, double bv = 1.0, double hv = 1.0):
                              m_Length(lv), m_Width(bv), m_Height(hv)
{
  cout << endl << "Constructor called.";
}
```

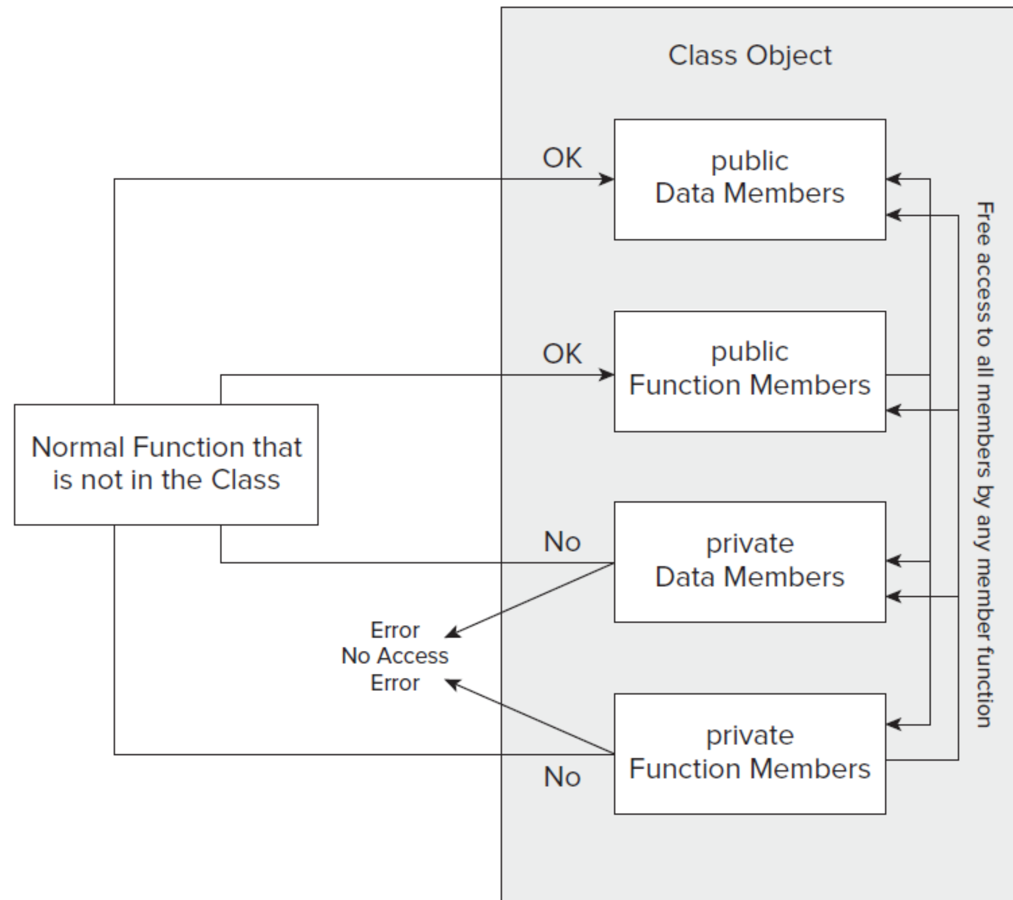
```
CBox box;
box = 99.0;
```

- Implicit call to the constructor above with the first argument value as 99.0 and the other two arguments with default values

```
explicit CBox(double lv = 1.0, double bv = 1.0, double hv = 1.0):
                              m_Length(lv), m_Width(bv), m_Height(hv)
{
  cout << endl << "Constructor called.";
}
```

Private Members of a Class

- Class members that are private can, in general, be accessed only by member functions of a class



Private Members of a Class

- To keep data and function members of a class safe from unnecessary meddling, it’s good practice to declare those that don’t need to be exposed as private
- Only make public what is essential to the use of your class
- Ex7_07

Accessing private Class Members

- It's all very well protecting them from unauthorized modification, but that's no reason to keep their values a secret
- All that's necessary is to write a member function to return the value of a data member

```
inline double CBox::GetLength()
{
  return m_Length;
}
```

```
double len = box2.GetLength();            // Obtain data member length
```

The friend Functions of a Class

- You want certain selected functions that are not members of a class to be able to access all the members of a class
- Such functions are called **friend functions** of a class and are defined using the keyword friend
- Friend functions are not members of the class
  - The access attributes do not apply to them
- Ex7_08

Placing friend Function Definitions Inside the Class

- You could have combined the definition of the function with its declaration as a friend of the Cbox class within the class definition

```
friend double BoxSurface(CBox aBox)
{
  return 2.0*(aBox.m_Length*aBox.m_Width +
              aBox.m_Length*aBox.m_Height +
              aBox.m_Height*aBox.m_Width);
}
```

The Default Copy Constructor

- Suppose that you declare and initialize a CBox object box1 with this statement

```
CBox box1(78.0, 24.0, 18.0);
```

- You now want to create another CBox object, identical to the first
- Ex7_09

THE POINTER this

- When any member function executes, it automatically contains a hidden pointer with the name this, which points to the object used with the function call
- Therefore, when the member m_Length is accessed in the Volume() function during execution
  - It's actually referring to this -> m_Length
- You can use the pointer this explicitly within a member function to return a pointer to the current object
- Ex7_10

CONST OBJECTS

- You will undoubtedly want to create class objects that are fixed from time to time
- You might define it with the following statement

```
const CBox standard(3.0, 5.0, 8.0);
```

- If you declare an object of a class as const, the compiler will not allow any member function to be called for it that might alter it

const Member Functions of a Class

- To make the this pointer in a member function const, you must declare the function as const within the class definition
- A const member function cannot call a non-const member function of the same class, since this would potentially modify the object
- The Compare() function calls Volume() , the Volume() member must also be declared as const
- Ex7_10A

Member Function Definitions Outside the Class

- When the definition of a const member function appears outside the class, the header for the definition must have the keyword const added

```
class CBox                                // Class definition at global scope
{
  public:
    // Constructor
    explicit CBox(double lv = 1.0, double bv = 1.0, double hv = 1.0);

    double Volume() const;                // Calculate the volume of a box
    bool Compare(CBox xBox) const;        // Compare two boxes

  private:
    double m_Length;                      // Length of a box in inches
    double m_Width;                       // Width of a box in inches
    double m_Height;                      // Height of a box in inches
};
```

Member Function Definitions Outside the Class

- Both the Volume() and Compare() members have been declared as const

```
double CBox::Volume() const
{
  return m_Length*m_Width*m_Height;
}

int CBox::Compare(CBox xBox) const
{
  return this->Volume() > xBox.Volume();
}

CBox::CBox(double lv, double bv, double hv):
                  m_Length(lv), m_Width(bv), m_Height(hv)
{
  cout << endl << "Constructor called.";
}
```

Arrays of Objects

- You can create an array of objects in exactly the same way as you created an ordinary array where the elements were one of the built-in types
- Each element of an array of class objects causes the default constructor to be called
- Ex7_11

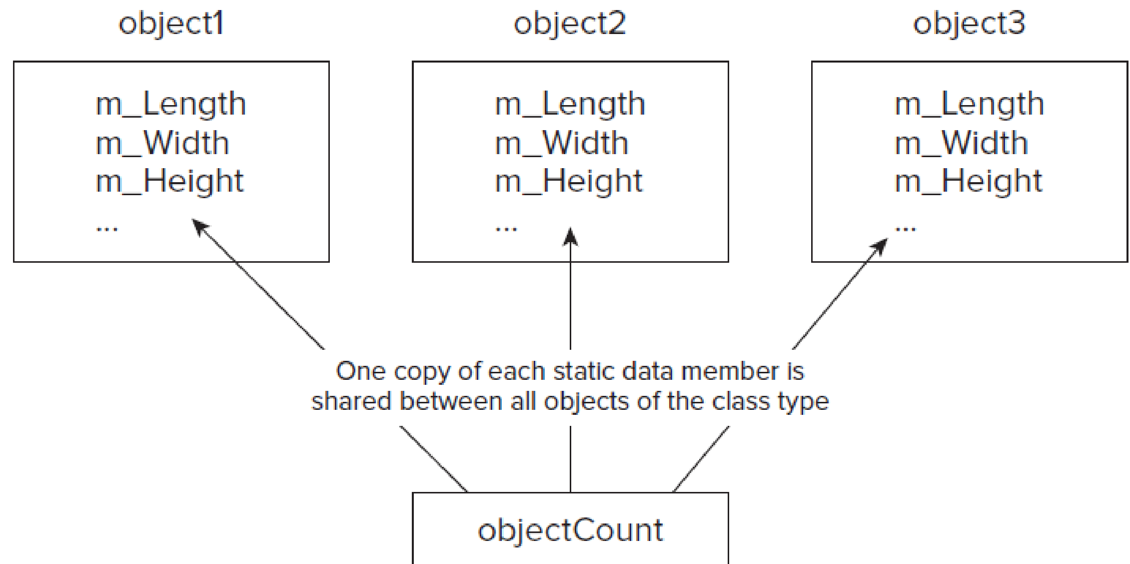
Static Data Members

- When you declare data members of a class to be static, the effect is that the static data members are defined only once and are shared between all objects of the class

Class Definition

```
class CBox
{
  public:
    static int objectCount;


      ...
  private:
    double m_Length;
    double m_Width;
    double m_Height;
      ...
```



Static Data Members

- One use for a static data member is to count how many objects actually exist
- You could add a static data member to the public section of the CBox class by adding the following statement to the previous class definition

```
static int objectCount;                    // Count of objects in existence
```

- You can't initialize the static data member in the class definition
- You don't want to initialize it in a constructor, because you want to increment it every time the constructor is called

Static Data Members

- You want it initialized before any object is created

```
int CBox::objectCount(0);          // Initialize static member of class CBox
```

- Ex7_12

Static Function Members of a Class

- By declaring a function member as static, you make it independent of any particular object of the class
- The static member function has the advantage that it exists, and can be called, even if no objects of the class exist
- In this case, only static data members can be used because they are the only ones that exist
  - You can call a static function member of a class to examine static data members, even when you do not know for certain that any objects of the class exist
- After the objects have been defined, a static member function can access private as well as public members of class objects

Static Function Members of a Class

- A static function might have this prototype

```
static void Afunction(int n);
```

- A static function can be called in relation to a particular object by a statement such as the following

```
aBox.Afunction(10);
```

- The same function could also be called without reference to an object

```
CBox::Afunction(10);
```

Pointer to Objects

- You declare a pointer to a class object in the same way that you declare other pointers

```
CBox* pBox(nullptr);                  // Declare a pointer to CBox
pBox = &cigar;                        // Store address of CBox object cigar in pBox
cout << pBox->Volume();               // Display volume of object pointed to by pBox
```

- Ex7_13

References to Class Objects

- To declare a reference to the object cigar

```
CBox& rcigar(cigar);                    // Define reference to object cigar
```

- To use a reference to calculate the volume of the object cigar, you would just use the reference name where the object name would otherwise appear

```
cout << rcigar.Volume();                // Output volume of cigar thru a reference
```

- A reference acts as an alias for the object it refers to, so the usage is exactly the same as using the original object name

Implementing a Copy Constructor

- The copy constructor is a constructor that creates an object by initializing it with an existing object of the same class

```
CBox(CBox initB);

CBox myBox(cigar);

CBox::CBox(cigar);
```

- Call by value->Call by reference

```
CBox(const CBox& initB);

CBox::CBox(const CBox& initB)
{
  m_Length = initB.m_Length;
  m_Width = initB.m_Width;
  m_Height = initB.m_Height;
}
```