Class Inheritance and Virtual Functions

Classes & Windows Programming

- The concept of “class” is essential to Windows programming.
  - We have several objects in Windows programming. Each of them belongs to different classes.
  - You need to master the concept of classes to utilize these classes in Windows programming.

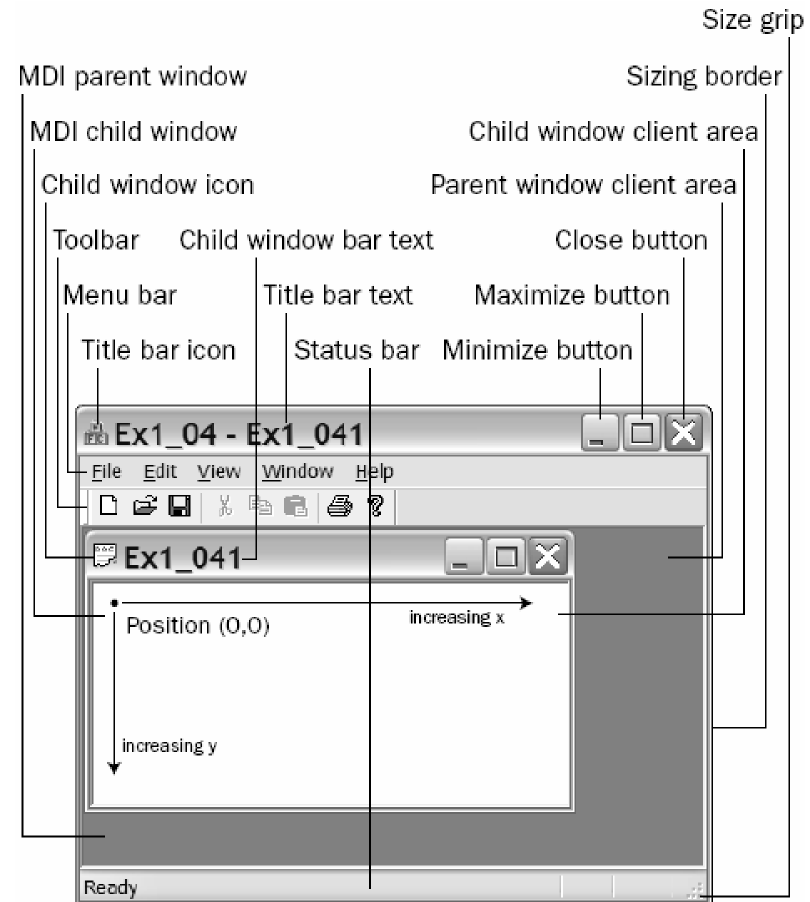


Figure 11-1

Object-Oriented Programming

- Real-world objects belongs to a particular class, which are specified by a common set of parameters (data members) and share a common set of operations (member functions).
- Take the "Box" class for example. There may be different kinds of boxes:
  - Carton, candy boxes, cereal boxes, …
- You may define a class CBox with common attributes,
- And add some additional attributes to obtain new classes.
- Re-use is the fundamental philosophy of OOP.

Figure 9-1

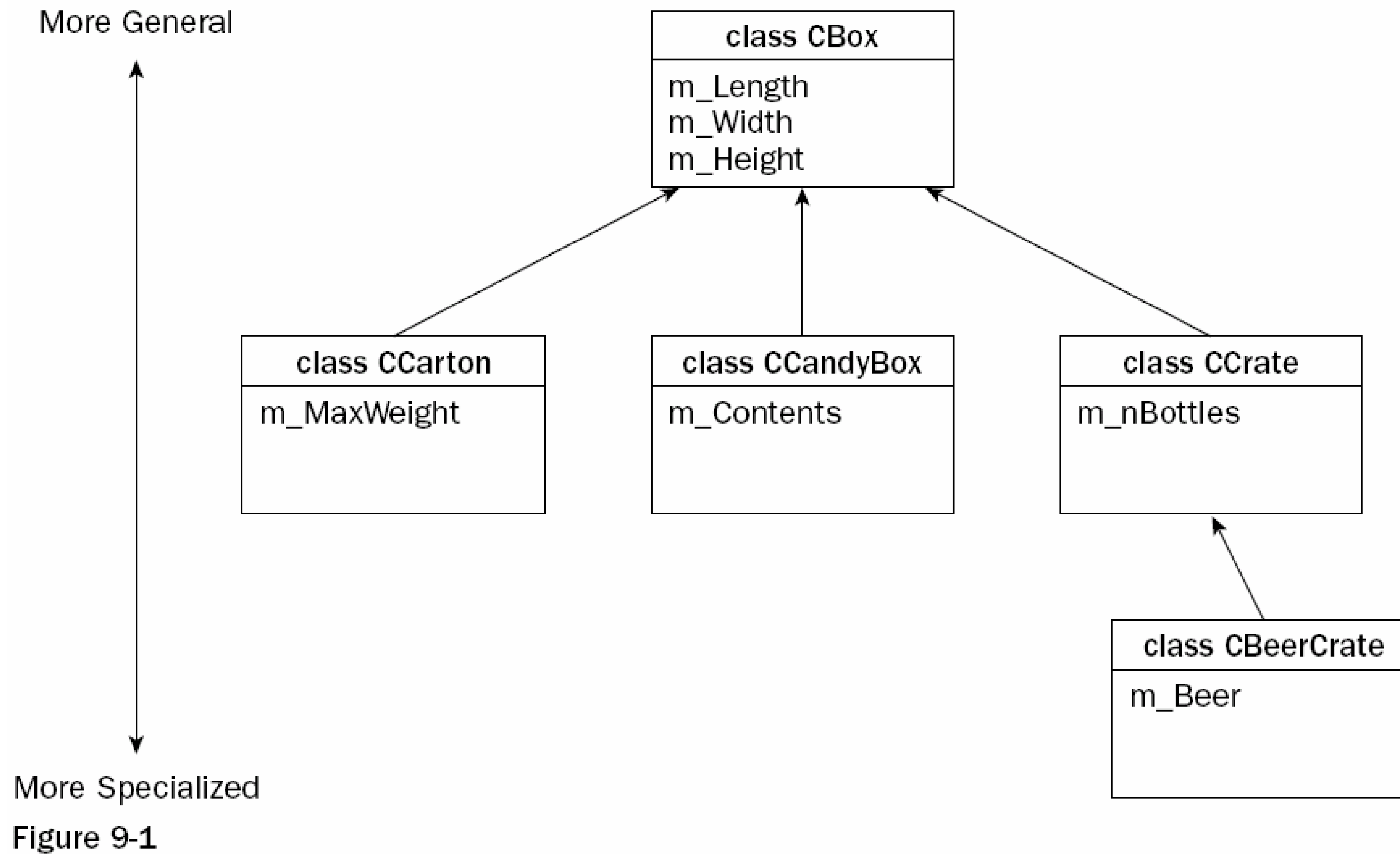


Figure 9-1

Inheritance in Classes

- When you define one class based on another, the former is referred to as a **derived class**, the latter is referred to as a **base class**.
  - CCandyBox is derived from CBox
  - CBeerCrate is derived from CCrate
- The derived class will inherit from the base class the data members and functions members.
- The only members of a base class that are not inherited by a derived class are
  - the destructor
  - the constructors
  - any member functions overloading the assignment operator.

Figure 9-2

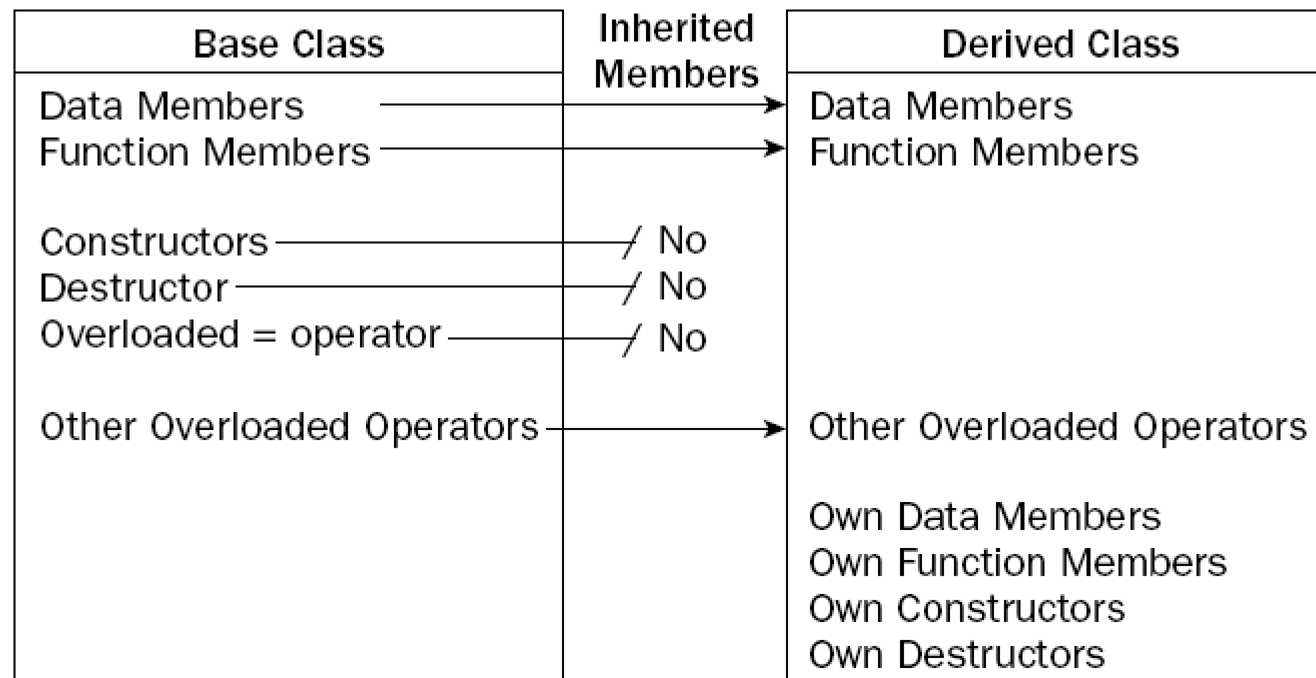


Figure 9-2

Suppose We Have a Base Class

```
// Box.h in Ex9_01
#pragma once

class CBox
{
  public:
    CBox(double lv = 1.0, double wv = 1.0, double hv = 1.0):
                               m_Length(lv), m_Width(wv), m_Height(hv){}

    double m_Length;
    double m_Width;
    double m_Height;
};
```

Define CCandyBox as a Derived Class

```
// Header file CandyBox.h in project Ex9_01
#pragma once
#include "Box.h"
class CCandyBox: CBox
{
  public:
    char* m_Contents;

    CCandyBox(char* str = "Candy")                    // Constructor
    {
      std::cout << "Constructor called.\n";
          m_Contents = new char[ strlen(str) + 1 ];
      strcpy_s(m_Contents, strlen(str)+1, str);
    }

    ~CCandyBox()                                      // Destructor
        { std::cout << "Destructor called.\n";
          delete[] m_Contents; };
};
```

Using a Derived Class

```
// Ex9_01.cpp
// Using a derived class
#include <iostream>                       // For stream I/O
#include <cstring>                        // For strlen() and strcpy()
#include "CandyBox.h"                     // For CBox and CCandyBox
using std::cout;
using std::endl;

int main()
{
  CBox myBox(4.0, 3.0, 2.0);                    // Create CBox object
  CCandyBox myCandyBox;
  CCandyBox myMintBox("Wafer Thin Mints"); // Create CCandyBox object
```

Ex9_01.cpp

```
cout << endl
      << "myBox occupies " << sizeof myBox   // Show how much memory
      << " bytes" << endl                    // the objects require
      << "myCandyBox occupies " << sizeof myCandyBox
       << " bytes" << endl
      << "myMintBox occupies " << sizeof myMintBox
      << " bytes";

  cout << endl
      << "myBox length is " << myBox.m_Length;

  myBox.m_Length = 10.0;

  // myCandyBox.m_Length = 1

  cout << endl;
  return 0;
}
```

myBox occupies 24 bytes
myCandyBox occupies 32 bytes
myMintBox occupies 32 bytes
myBox length is 4

Q: Isn't the size of myCandyBox 24+4=28 bytes?

Uncomment a Line

```
 cout << endl
       << "myBox occupies " << sizeof myBox  // Show how much memory
       << " bytes" << endl                   // the objects require
       << "myCandyBox occupies " << sizeof myCandyBox
        << " bytes" << endl
       << "myMintBox occupies " << sizeof myMintBox
       << " bytes";

  cout << endl
       << "myBox length is " <

  myBox.m_Length = 10.0;

  myCandyBox.m_Length = 10.0;
  cout << endl;
  return 0;
}
```

error C2247: ‘CBox::m_Length’ no accessible because ‘CCandyBox’ uses ‘private’ to inherit from ‘CBox’

m_Length is public in CBox, but becomes private in the derived class CCandyBox

Declare the `public` access specifier for the base class

```
class CCandyBox: public CBox
{
  public:
    char* m_Contents;

    CCandyBox(char* str = "Candy")          // Constructor
    {
      m_Contents = new char[ strlen(str) + 1 ];
      strcpy_s(m_Contents, strlen(str)+1, str);
    }

    ~CCandyBox(                                         r
        { de
};
```

Now the `m_Length` member is inherited as public in the derived class, and thus accessible in the function main().

If the data members are private

```
class CBox
{
public:
    CBox(double lv = 1.0, double wv = 1.0, double hv = 1.0):
                                    m_Length(lv), m_Width(wv), m_Height(hv){}
private:
    double m_Length;
    double m_Width;
    double m_Height;
};

class CCandyBox: CBox
{
  public:
    char* m_Contents;

    // Function to calculate the volume of a CCandyBox object
    double Volume() const  // Error - members not accessible
    { return m_Length*m_Width*m_Height; }

    CCandyBox(char* str = "Candy")                    // Constructor
    {
     m_Contents = new char[ strlen(str) + 1 ];
      strcpy_s(m_Contents, strlen(str)+1, str);
    }

    ~CCandyBox()                                      // Destructor
    { delete[] m_Contents; };
};
```

The function `Volume()` attempts to access the private members of the base class, which is not legal.

Ex9_02

- Move the definition of the function Volume() to the public section of the base class.
- The output will be
  - myBox occupies 24 bytes
  - myCandyBox occupies 32 bytes,
    - containing “Candy”
  - myMintBox occupies 32 bytes
    - containing “Wafer Thin Mints”
  - myMintBox volume is 1,
    - `CBox()` default constructor was called to create the base part of the object.

Constructor Operator in a Derived Class

```
class CCandyBox

    double m_Length;
    double m_Width;
    double m_Height;

    char* m_Contents;
```

base part of a derived class

- Private members of a base class are inaccessible in a derived class object, so responsibility for these has to lie with the base class constructors.
- The default base class constructor was called automatically in the last example.

Calling Constructors

- Ex9_03 in P.517
- In CandyBox.h
  - Calling the base class constructor
  - CCandyBox(double lv, double wv, double hv, char* str = "Candy") :CBox(lv, wv, hv)
- Output:
  - CBox constructor called
  - CBox constructor called
  - CCandyBox constructor1 called
  - CBox constructor called
  - CCandyBox constructor2 called
  - myBox occupies 24 bytes
  - myCandyBox occupies 32 bytes
  - myMintBox occupies 32 bytes
  - myMintBox volume is 6

CBox myBox(4.0, 3.0, 2.0);

CCandyBox myCandyBox;

CCandyBox myMintBox(1.0, 2.0, 3.0, "Wafer Thin Mints");

Declaring Class Members to be Protected

- Members in protected section cannot be accessed by ordinary global function,
  - but they can be accessible to member functions of a derived class.
- See Ex9_04 in P.520
  - In the previous example, `Volume()` must be defined in `CBox` to access `m_Length`.
  - Notice the calling sequence of constructors and destructors as shown in the output in P.521.
    - Destructors for a derived class object are called in the reverse order.

Access Level of Inherited Class Members

- If you have no access specifier, the default specification is private.
  - The inherited `public` and `protected` members become `private` in the derived class.
  - The `private` members will not be inherited.
- If you use `public` as the specifier for a base class,
  - The `public` members remain `public`, and `protected` members remain `protected`.
- If you declare a base class as `protected`,
  - The `public` members are inherited as `protected`,
  - The `protected` and `private` members retain their original access level

Figure 9-3

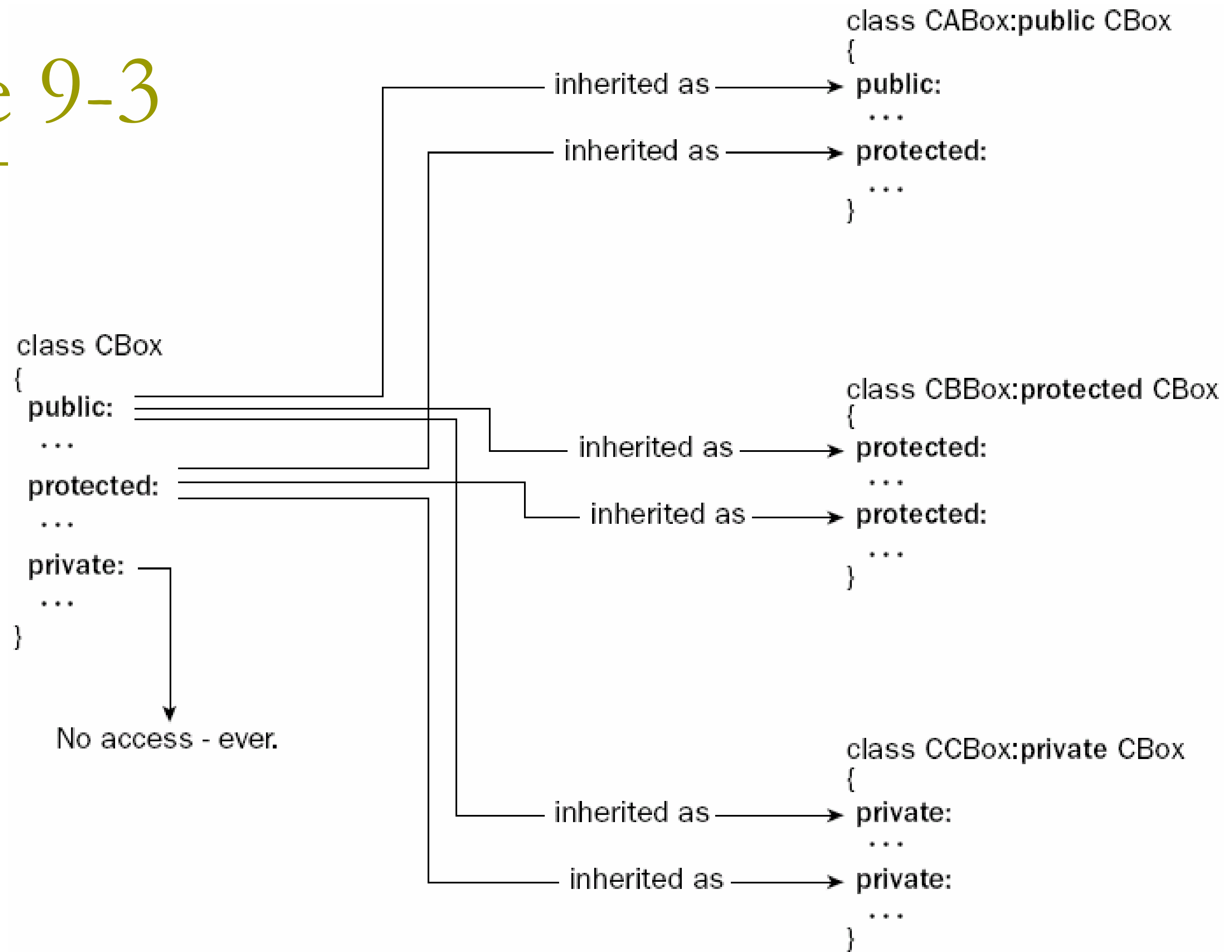
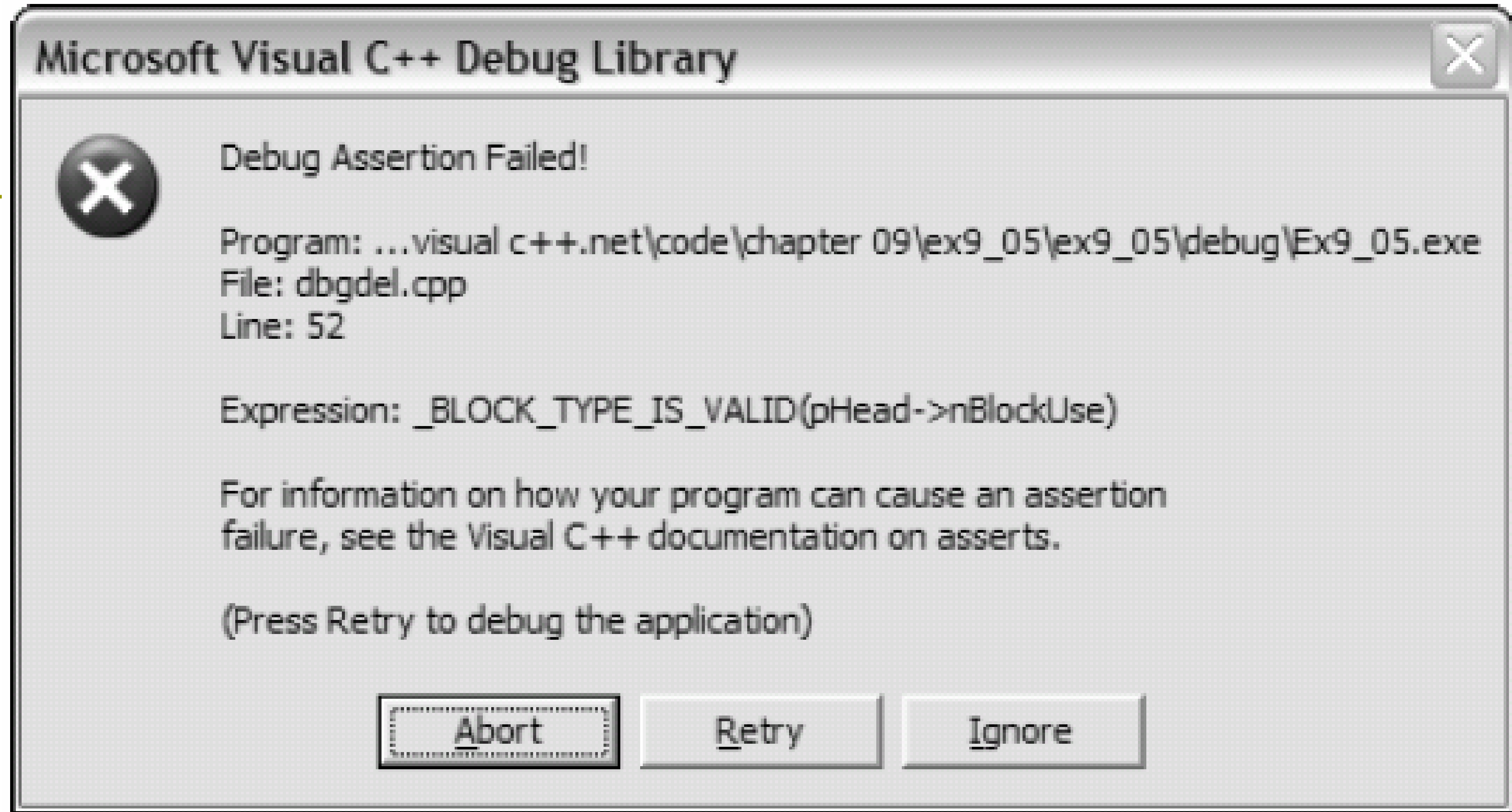


Figure 9-3

The Copy Constructor in a Derived Class

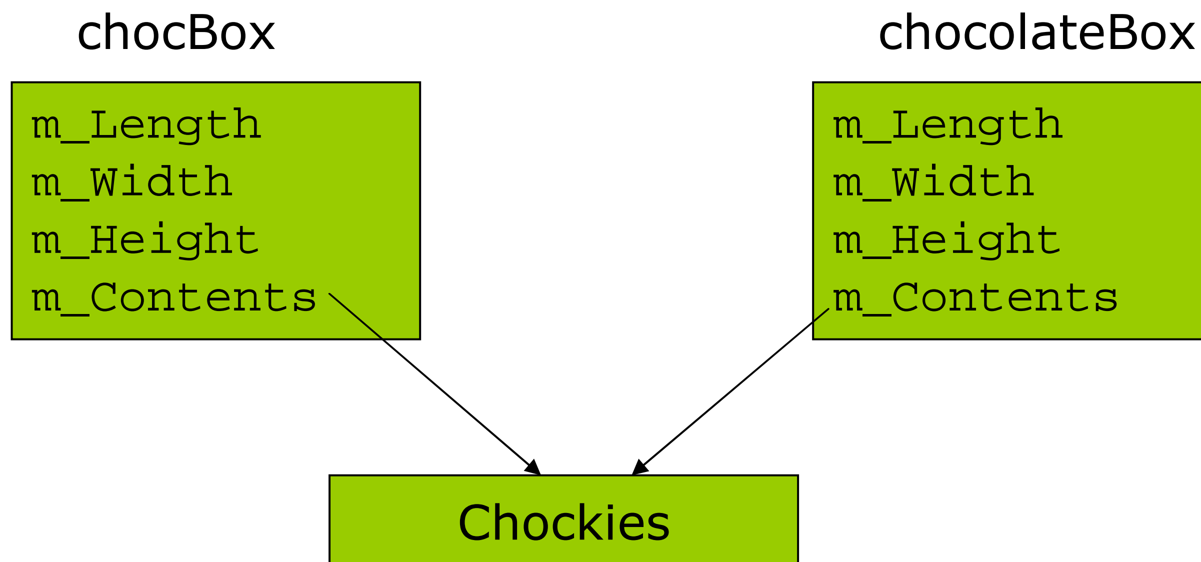
- The copy constructor is called automatically when you declare an object that is initialized with an object of the same class.
  - CBox myBox(2.0, 3.0, 4.0);
    - // Calls constructor
  - CBox copyBox(myBox);
    - // Calls copy constructor
- The compiler supplies a default copy constructor, which copies the initializing object member by member to the new object.
- You may define your own copy constructor, but remember that the copy constructor must have its parameter specified as a reference to avoid an infinite number of calls to itself (P.379).



- Click Abort, and you'll see the output that you expect in the console window.

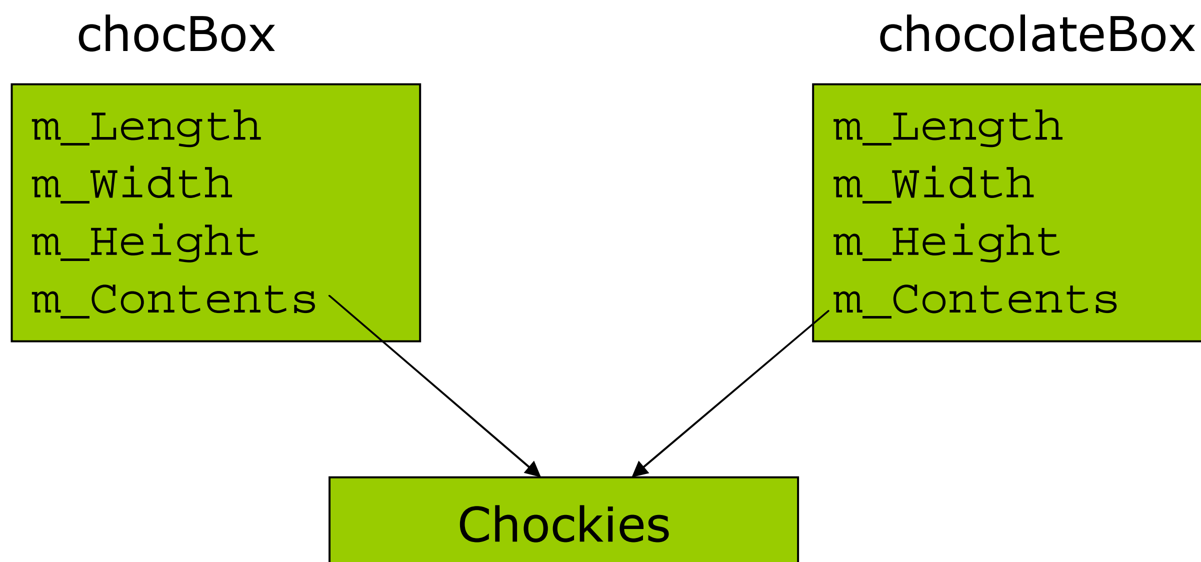
Why It Doesn't Work

- The `m_Contents` member in the second object points to the same memory as the one in the first object.



Why It Doesn't Work (2)

- When the first object is destroyed, it releases the memory occupied by the text.



- When the second object is destroyed, the destructor attempts to release some memory that has already been freed.

Fixing the Copy Constructor Problem

```
// Derived class copy constructor
CCandyBox(const CCandyBox& initCB)
{
  cout << endl << "CCandyBox copy constructor
   called";

  // Get new memory
  m_Contents = new char[ strlen(initCB.m_Contents) +
   1 ];

  // Copy string
  strcpy_s(m_Contents, strlen(initCB.m_Contents) +
   1 , initCB.m_Contents);
}
```

The problem of dynamic memory allocation is solved, but …

- You see the output as in P.527

```
CBox constructor called
CCandyBox constructor2 called
CBox constructor called
CCandyBox copy constructor called
Volume of chocBox is 24
Volume of chocolateBox is 1
CCandyBox destructor called
CBox destructor called
CCandyBox destructor called
CBox destructor called
```

Solution

- Call the copy constructor for the base part of the class in the initialization list for the copy constructor for the `CCandyBox` class.

```
CCandyBox(const CCandyBox& initCB): CBox(initCB)
{
  cout << endl << "CCandyBox copy constructor called";

  // Get new memory
  m_Contents = new char[ strlen(initCB.m_Contents) + 1 ];

  // Copy string
  strcpy_s(m_Contents, strlen(initCB.m_Contents) + 1,
    initCB.m_Contents);
}
```

The Copy Constructor is Correctly Called

```
CBox constructor called
CCandyBox constructor2 called
CBox copy constructor called
CCandyBox copy constructor called
Volume of chocBox is 24
Volume of chocolateBox is 24
CCandyBox destructor called
CBox destructor called
CCandyBox destructor called
CBox destructor called
```

Class Members as Friends

- A friend function has the privilege to access any of the class members (private or public).
- See the example on P.528.
  - We need a carton to package a dozen bottles.
  - The constructor `CCarton::CCarton()` tries to access the height of the bottle.
  - This doesn't work because the data members of the CBottle class are private.
- Declare the carton constructor in CBottle:
  - `friend CCarton::CCarton(const CBottle& aBottle);`
- You may also allow all the function member in CCarton to access the members in CBottle
  - `friend CCarton;`
- Class friendship is not inherited.
  - If you define another class with CBottle as a base, member of CCarton class will not have access to its data members.

Virtual Functions

- Let us look closely at inherited member functions.
- `Box.h` in P.531
  - `Volume()` – Calculate the volume of a CBox object
  - `ShowVolume()` - Output the volume of a CBox object
  - `CBox()` – The constructor sets the data member values in the initialization list
    `:m_Length(lv), m_Width(wv), m_Height(hv)`
    so no statements are necessary in the body of the function.
  - Data members are specified as `protected`, so they are accessible to the member functions of any derived class.

GlassBox.h

- Suppose we need a different kind of box to hold glassware.
- The contents would be fragile.
  - Some packaging material is added to protect them, so the capacity of the box is less than the capacity of a basic CBox object.
  - You need a different `Volume()` function to calculate the volume.
    - GlassBox.h in P.531
    - return 0.85*m_Length*m_Width*m_Height;

Ex9_06.cpp

- Try It Out

```
int main()
{
  CBox myBox(2.0, 3.0, 4.0);             // Declare a base box
  CGlassBox myGlassBox(2.0, 3.0, 4.0); // Declare derived box -
                                             same size

  myBox.ShowVolume();              // Display volume of base box
  myGlassBox.ShowVolume();         // Display volume of derived box

  cout << endl;
  return 0;
}
```

- The output is
  - CBox usable volume is 24
  - CBox usable volume is 24

Why Doesn't It Work?

- The volume of a `CGlassBox` object should be only 85% of a `CBox` with the same dimension.
- Reason:
  - The call of the `Volume()` function in the function `ShowVolume()` is being set once and for all by the compiler (as the version defined in the base class).
  - The compiler has no knowledge of any other `Volume()` function.
  - This is sometimes called early binding.

What Are We Hoping For?

- We want the actual version of the function `Volume()` invoked by `ShowVolume()` to be determined by the object being processed.
- This is sometimes called dynamic linkage, or late binding.
- C++ provides the mechanism of virtual function to support this.

Fixing the CGlassBox

- Box.h & GlassBox.h in P.533
  - The keyword `virtual` is added to the definitions of the `Volume()` function in the two classes.
- The result is what we expected
  - `CBox usable volume is 24`
  - `CBox usable volume is 20.4`
- The ability to use virtual functions for late binding is referred to as the mechanism of *Polymorphism* in Object-Oriented Programming.

Using Pointers to Class Objects

- Pointers to Base and Derived Classes

```
CBox myBox(2.0, 3.0, 4.0);                 // Declare a base box
CGlassBox myGlassBox(2.0, 3.0, 4.0);  // Declare derived box of
                                              same size
CBox* pBox = 0;     // Declare a pointer to base class objects

pBox = &myBox;      // Set pointer to address of base object
pBox->ShowVolume(); // Display volume of base box
pBox = &myGlassBox; // Set pointer to derived class object
pBox->ShowVolume(); // Display volume of derived box

cout << endl;
return 0;
```

- A pointer to a base class object can be assigned the address of a derived class object as well.

Figure 9-5

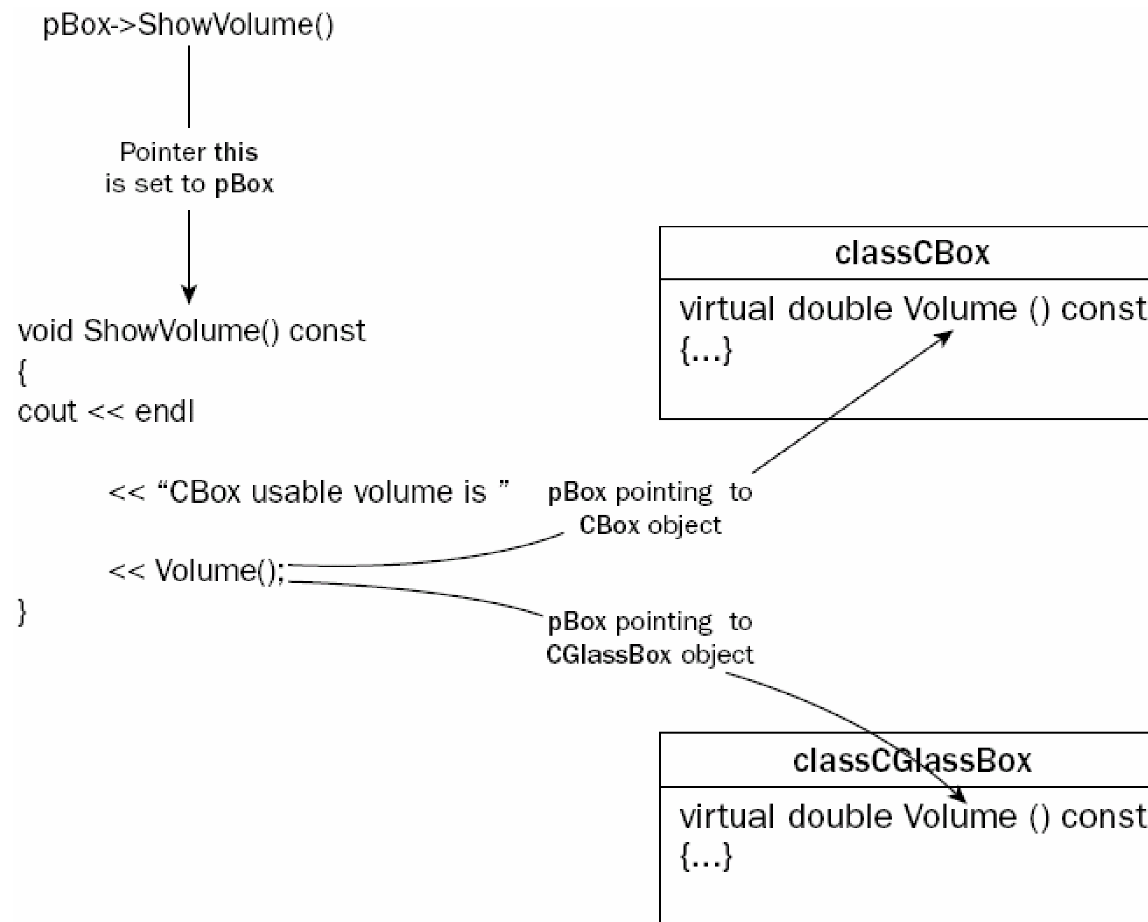


Figure 9-5

Pure Virtual Functions

- `Container.h` in P.538
  - `virtual double Volume() const = 0`
  - This statement declares a pure virtual function.
- A class containing a pure virtual function is called an **abstract class**.
  - It is called abstract because you cannot define objects of a class containing a pure virtual function.
  - It exists only for the purpose of defining classes that are derived from it.

Indirect Base Classes

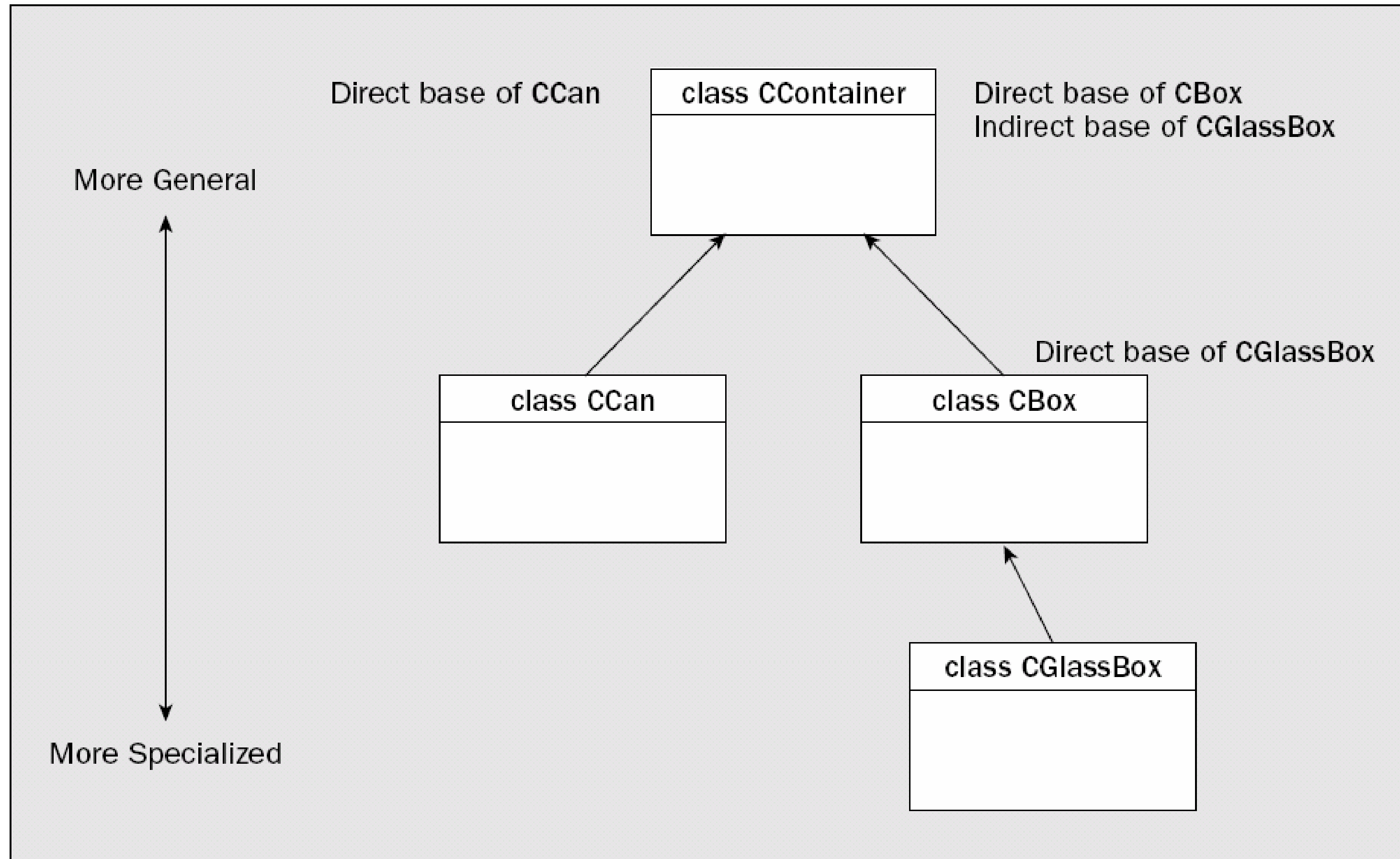


Figure 9-6

Ex9_11

- GlassBox.h in P.542
  - It does not define ShowVolume().
- Box.h in P.539
  - Class CBox derived from CContainer
- Can.h in P.540
  - Class CCan derived from CContainer
- Ex9_11.cpp in P.543
  - CBox usable volume is 24
  - Volume is 45.9458
  - CBox usable volume is 20.4

Virtual Destructors

- One problem of derived classes using a pointer to the base class is that, the correct destructor may not be called.
- Let us output a message for tracking in the destructor.
  - Container.h in P.545
  - Can.h
  - Box.h
  - GlassBox.h

Wrong Destructors Are Called

- CBox usable volume is 24
- Delete CBox
- CContainer destructor called

We expect a CBox destructor

- CBox usable volume is 102
- Delete CGlassBox
- CContainer destructor called

We expect a CGlassBox destructor

- Volume is 45.9458
- CBox usable volume is 20.4
- CGlassBox destructor called
- CBox destructor called
- CContainer destructor called
- CCan destructor called
- CContainer destructor called

Correcting the Problem

- The reason is that, the compiler only know the pointer type is a pointer to the base class `CContainer`.
- We can declare the destructor of the base class to be virtual, so that it will be resolved dynamically.
  - P.549

Casting Between Class Types

- You have seen how to store the address of a `CBox` object to a variable of type `CContainer*`.
  - Store the address of a derived class object in a variable of a base class type.
- How do you store an address of type `CContainer*` to a variable of type `CBox*`
  - `CBox* pBox = dynamic_cast<CBox*>(pContainer);`
- The difference between `dynamic_cast` and `static_cast`:
  - `dynamic_cast` operator checks the validity of a cast at run-time, while `static_cast` operator does not.
  - If a `dynamic_cast` operation is not valid, the result is null.