Classes

Chapter 11

Classes and Object-Oriented Programming

Encapsulation

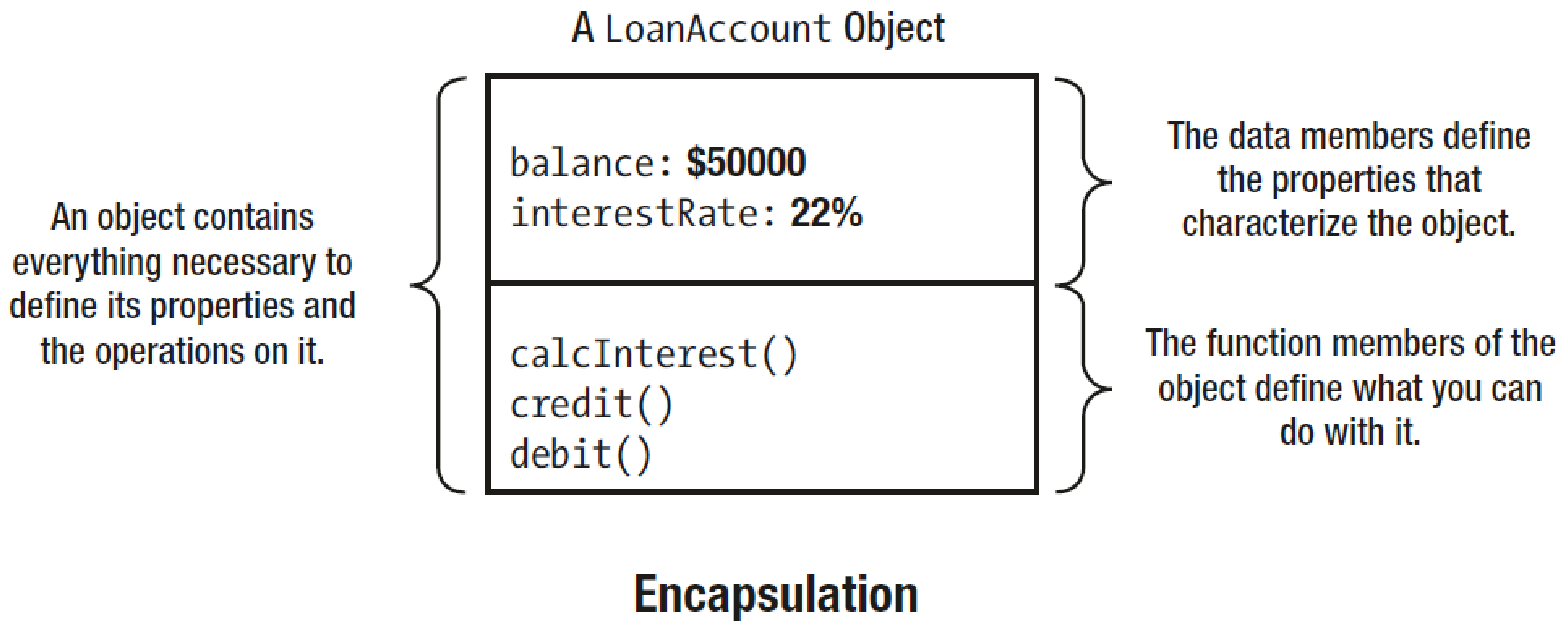
A LoanAccount Object
balance: $50000
interestRate: 22%
calcInterest()
credit()
debit()
An object contains everything necessary to define its properties and the operations on it.
The data members define the properties that characterize the object.
The function members of the object define what you can do with it.
Encapsulation

Figure 11-1. An example of encapsulation

Classes and Object-Oriented Programming

Data Hiding

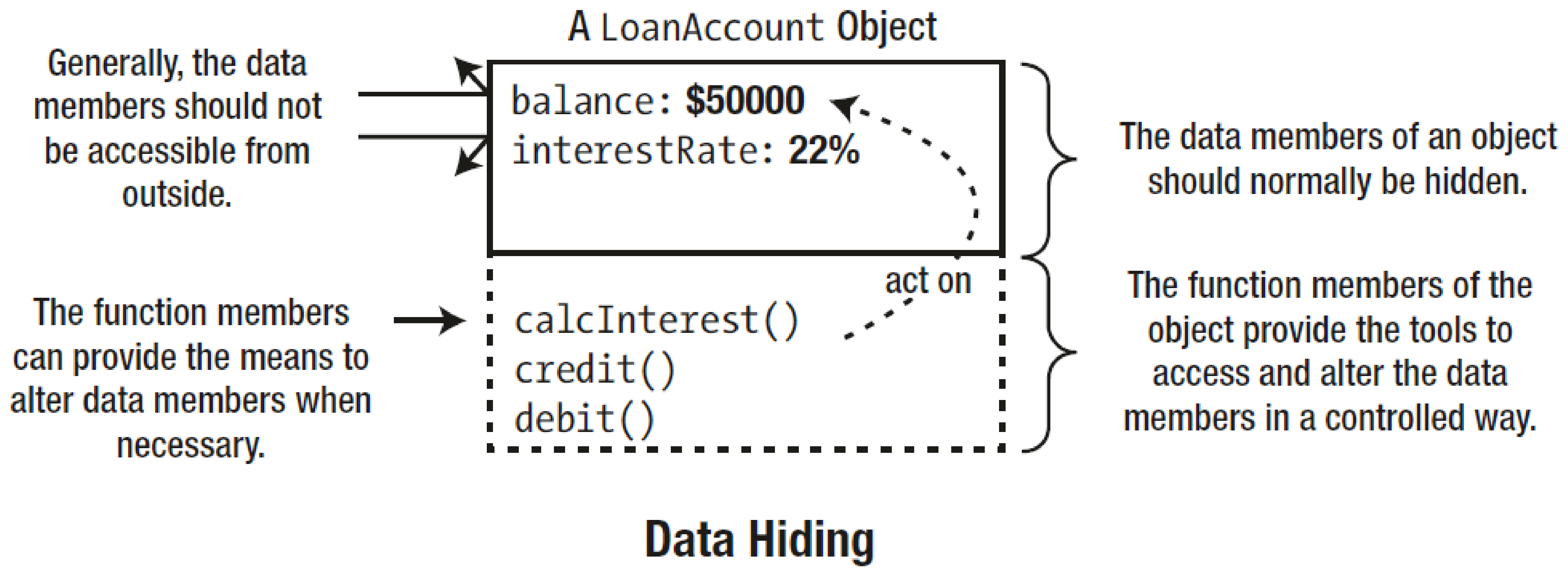
A LoanAccount Object
Generally, the data members should not be accessible from outside.
balance: $50000
interestRate: 22%
The data members of an object should normally be hidden.
act on
The function members can provide the means to alter data members when necessary.
calcInterest()
credit()
debit()
The function members of the object provide the tools to access and alter the data members in a controlled way.
Data Hiding

Figure 11-2. *An example of data hiding*

Classes and Object-Oriented Programming

Data Hiding

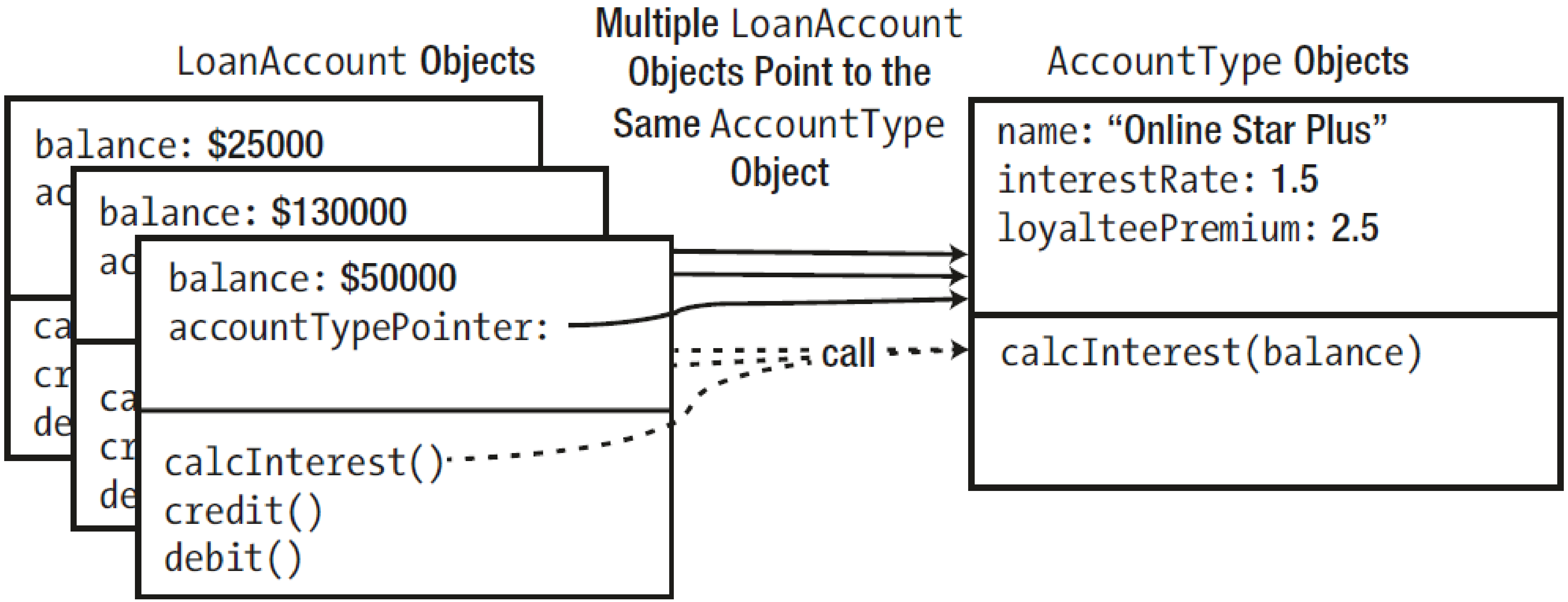

Figure 11-3. *Reworking the representation of the internal state of objects while preserving their interface*

Data Hiding

- Data hiding facilitates maintaining the integrity of an object. It allows you to make sure that an object's internal state—the combination of all its member variables—remains valid at all times.
- Data hiding, combined with a well-thought-out interface, allows you to rework both an object's internal representation (that is, its *state*) and the implementation of its member functions (that is, its *behavior*) without having to rework the rest of the program as well. In object-oriented speak we say that data hiding reduces the *coupling* between a class and the code that uses it. Interface stability is, of course, even more critical if you are developing a software library that is used by external customers.

Terminology

- A *class* is a user-defined data type.
- The variables and functions defined within a class are *members* of the class. The variables are *member variables*, and the functions are *member functions*. Member functions are also often referred to as *methods*; member variables are called either *data members* or *fields*.
- Variables of a class type store *objects*. Objects are sometimes called *instances* of the class. Defining an instance of a class is referred to as *instantiation*.
- *Object-oriented programming* is a programming style based on the idea of defining your own data types as classes. It involves the ideas of *encapsulation*, *data hiding*, class *inheritance*, and *polymorphism*.

Defining a Class

```
class ClassName
{
private:
  // Code that specifies members that are not accessible from outside the class...

public:
  // Code that specifies members that are accessible from outside the class...
};
```

Defining a Class

```
class Box
{
private:
  double length {1.0};
  double width {1.0};
  double height {1.0};

public:
  // Function to calculate the volume of a box
  double volume()
  {
    return length * width * height;
  }
};
```

Defining a Class

```
class Box
{
private:
  double length {1.0};
  double width {1.0};
  double height {1.0};

public:
  // Function to calculate the volume of a box
  double volume()
  {
    return length * width * height;
  }
};
```

Constructors

```
 class Box
 {
 private:
   double length {1.0};
   double width {1.0};
   double height {1.0};

 public:
   // Constructor
   Box(double lengthValue, double widthValue, double heightValue)
   {
     std::cout << "Box constructor called."  << std::endl;
     length = lengthValue;
     width = widthValue;
     height = heightValue;
   }
 // Function to calculate the volume of a box
 double volume()
 {
   return length * width * height;
 }
};

int main()
{
 Box firstBox {80.0, 50.0, 40.0};                // Create a box
 double firstBoxVolume {firstBox.volume()};      // Calculate the box volume
 std::cout << "Volume of Box object is " << firstBoxVolume << std::endl;

 // Box secondBox;                        // Causes a compiler error message
}
```

Default Constructors

```
Box() {}                          // Default constructor
```

Using the default Keyword

```
Box() = default;                  // Default constructor
```

Tip If there is nothing to do in a default constructor's body (or initializer list, as we'll encounter this later), always prefer `= default;` over `{}`. Not only does this make it more apparent that it concerns a default default constructor, there are also a few subtle technical reasons outside the scope of this discussion that make the compiler-generated version the better choice.

Defining Functions and Constructors Outside the Class

```
// Box.h
#ifndef BOX_H
#define BOX_H

class Box
{
private:
  double length {1.0};
  double width {1.0};
  double height {1.0};
 public:
   // Constructors
   Box(double lengthValue, double widthValue,  double heightValue);
   Box() = default;

   double volume();           // Function to calculate the volume of a box
 };

 #endif
```

- The header file defines the interface

Defining Functions and Constructors Outside the Class

```
// Box.cpp
#include "Box.h"
#include <iostream>

// Constructor definition
Box::Box(double lengthValue, double widthValue, double heightValue)
{
  std::cout << "Box constructor called." << std::endl;
  length = lengthValue;
  width = widthValue;
  height = heightValue;
}

// Function to calculate the volume of a box
double Box::volume()
{
  return length * width * height;
}
```

- The .cpp file defines the implementation

Defining Functions and Constructors Outside the Class

```
// Ex11_01B.cpp
// Defining functions and constructors outside the class definition
#include <iostream>
#include "Box.h"
int main()
{
  Box firstBox{80.0, 50.0, 40.0};              // Create a box
  double firstBoxVolume{firstBox.volume()};    // Calculate the box volume
  std::cout << "Volume of the first Box object is " << firstBoxVolume << std::endl;

  Box secondBox;                               // Uses compiler-generated default constructor
  double secondBoxVolume{secondBox.volume()};  // Calculate the box volume
  std::cout << "Volume of the second Box object is " << secondBoxVolume << std::endl;
}
```

■ **Note** Defining a member function outside a class is actually not quite the same as placing the definition inside the class. One subtle difference is that function definitions *within* a class definition are implicitly `inline`. (This doesn't necessarily mean they will be *implemented* as inline functions, though—the compiler still decides that, as we discussed in Chapter 8.)

Default Constructor Parameter Values

```
class Box
{
private:
  double length, width, height;

public:
  // Constructors
  Box(double lv = 1.0, double wv = 1.0, double hv = 1.0);
  Box() = default;

  double volume();          // Function to calculate the volume of a box
};
```

- Gives a compiler error – why?

Default Constructor Parameter Values

```
class Box
{
private:
  double length, width, height;

public:
  // Constructors
  Box(double lv = 1.0, double wv = 1.0, double hv = 1.0);
  Box() = default;

  double volume();          // Function to calculate the volume of a box
};
```

- Gives a compiler error – why?

Using a Member Initializer List

```
// Constructor definition using a member initializer list
Box::Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv}
{
  std::cout << "Box constructor called." << std::endl;
}
```

Tip As a rule, prefer to initialize all member variables in the constructor's member initializer list. This is generally more efficient. To avoid any confusion, you ideally put the member variables in the initializer list in the same order as they are declared in the class definition. You should initialize member variables in the body of the constructor only if either more complex logic is required or the order in which they are initialized is important.

Using the explicit Keyword

```
class Cube
{
private:
  double side;

public:
  Cube(double aSide);                      // Constructor
  double volume();                         // Calculate volume of a cube
  bool hasLargerVolumeThan(Cube aCube);    // Compare volume of a cube with another
};
#endif
```

You can define the constructor in `Cube.cpp` as follows:

```
Cube::Cube(double aSide) : side{aSide}
{
  std::cout << "Cube constructor called." << std::endl;
}
```

The definition of the function that calculates the volume will be as follows:

```
double Cube::volume() { return side * side * side; }
```

One `Cube` object is greater than another if its volume is the greater of the two. The `hasLargerVolumeThan()` member can thus be defined as follows:

```
bool Cube::hasLargerVolumeThan(Cube aCube) { return volume() > aCube.volume(); }
```

Using the explicit Keyword

Now comes the problem of implicit type conversion

```
// Ex11_02.cpp
// Problems of implicit object conversions
#include <iostream>
#include "Cube.h"

int main()
{
  Cube box1 {7.0};
  Cube box2 {3.0};
  if (box1.hasLargerVolumeThan(box2))
    std::cout << "box1 is larger than box2." << std::endl;
  else
    std::cout << "Volume of box1 is less than or equal to that of box2." << std::endl;

  std::cout << "volume of box1 is " << box1.volume() << std::endl;
  if (box1.hasLargerVolumeThan(50.0))
    std::cout << "Volume of box1 is greater than 50"<< std::endl;
  else
    std::cout << "Volume of box1 is less than or equal to 50"<< std::endl;
}
```

Here's the output:

```
Cube constructor called.
Cube constructor called.
box1 is larger than box2.
volume of box1 is 343
Cube constructor called.
Volume of box1 is less than or equal to 50
```

Solution: Using the explicit Keyword

```
class Cube
{
public:
  double side;

  explicit Cube(double aSide);               // Constructor
  double volume();                           // Calculate volume of a cube
  bool hasLargerVolumeThan(Cube aCube);      // Compare volume of a cube with another
};
```

Delegating Constructors

```cpp
class Box
{
private:
  double length {1.0};
  double width {1.0};
  double height {1.0};

public:
  // Constructors
  Box(double lv, double wv, double hv);
  explicit Box(double side);                    // Constructor for a cube
  Box() = default;                              // No-arg constructor

  double volume();                              // Function to calculate the volume of a box
};

Box::Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv}
{
  std::cout << "Box constructor 1 called." << std::endl;
}

Box::Box(double side) : Box{side, side, side}
{
  std::cout << "Box constructor 2 called." << std::endl;
}
```

Delegating Constructors

```
// Ex11_03.cpp
// Using a delegating constructor
#include <iostream>
#include "Box.h"

int main()
{
  Box box1 {2.0, 3.0, 4.0};              // An arbitrary box
  Box box2 {5.0};                        // A box that is a cube
  std::cout << "box1 volume = " << box1.volume() << std::endl;
  std::cout << "box2 volume = " << box2.volume() << std::endl;
}
```

The complete code is in the download. The output is as follows:

```
Box constructor 1 called.
Box constructor 1 called.
Box constructor 2 called.
box1 volume = 24
box2 volume = 125
```

The Copy Constructor

Suppose you add the following statement to main() in Ex11_03.cpp:

```
Box box3 {box2};
std::cout << "box3 volume = " << box3.volume() << std::endl;   // Volume = 125
```

Implementing the Copy Constructor

- A bad idea:

```
Box::Box(Box box) : length {box.length}, width {box.width}, height {box.height} // Wrong!!
{}
```

- This is how to do it:

```
Box::Box(const Box& box) : length {box.length}, width {box.width}, height {box.height}
{}
```

- Notice class privacy not object privacy

The this Pointer

```
double Box::volume()
{
  return this->length * this->width * this->height;
}
```

This way of accessing members is required in many company coding standards

- Cascading member function calls by returning this pointer:

```
Box* Box::setLength(double lv)
{
  if (lv > 0) length = lv;
  return this;
}
```

```
Box* Box::setWidth(double wv)
{
  if (wv > 0) width = wv;
  return this;
}
```

```
Box* Box::setHeight(double hv)
{
  if (hv > 0) height = hv;
  return this;
}
```

Now you can modify all the dimensions of a Box object in a single statement:

```
Box myBox{3.0, 4.0, 5.0};                              // Create a box
myBox.setLength(-20.0)->setWidth(40.0)->setHeight(10.0);   // Set all dimensions of myBox
```

The this Pointer

- A more prefered way of cascading calls:
  - cascading member function calls by returning a reference to the object itself
  - Then the dot operator can be used for the cascading calls

```
Box& Box::setLength(double lv)
{
  if (lv > 0) length = lv;
  return *this;
}
```

```
myBox.setLength(-20.0).setWidth(40.0).setHeight(10.0);     // Set all dimensions of myBox
```

const Objects

- The principle of least privilege

```
const Box myBox {3.0, 4.0, 5.0};
std::cout << "The length of myBox is " << myBox.length << std::endl;       // ok
myBox.length = 2.0;        // Error! Assignment to a member variable of a const object...
myBox.width *= 3.0;        // Error! Assignment to a member variable of a const object...
```

const Member Functions

```
class Box
{
  // Rest of the class as before...

  double volume() const;           // Function to calculate the volume of a box

  // Functions to provide access to the values of member variables
  double getLength() const { return length; }
  double getWidth() const  { return width; }
  double getHeight() const { return height; }

  // Functions to set member variable values
  void setLength(double lv) { if (lv > 0) length = lv;}
  void setWidth(double wv)  { if (wv > 0) width = wv; }
  void setHeight(double hv) { if (hv > 0) height = hv; }
};
```

Next, you must change the function definition in `Box.cpp` accordingly:

```
double Box::volume() const
{
  return length * width * height;
}
```

Casting Away const

- Consider whether you really want to do this !

```
const_cast<Type*>(expression)
const_cast<Type&>(expression)
```

For the first form, the type of `expression` must be either `const Type*`; or `Type*`; for the second, it can be either `const Type*`, `const Type&`, `Type`, or `Type&`.

■ **Caution** The use of `const_cast` is nearly always frowned upon because it can be used to misuse objects. You should never use this operator to undermine the `const`-ness of an object. If an object is `const`, it normally means that you are not expected to modify it. And making unexpected changes is a perfect recipe for bugs. The only situations in which you should use `const_cast` are those where you are sure the `const` nature of the object won't be violated as a result, such as because someone else forgot to add a `const` in a function declaration, even though you are positive the function doesn't modify the object. Another example is when you implement the idiom we branded `const`-and-back-again, which you'll learn about in Chapter 16.

Using the mutable Keyword

- Only use mutable members for special purposes

```
class Box
{
private:
  double length{1.0};
  double width{1.0};
  double height{1.0};
  mutable unsigned count{};   // Counts the amount of time printVolume() is called

public:
  // Constructors
  Box() = default;
  Box(double length, double width, double height);

  double volume() const;      // Function to calculate the volume of a box
  void printVolume() const;   // Function to print out the volume of a box

  // Getters and setters like before...
};
```

```
void Box::printVolume() const
{
  // Count how many times printVolume() is called using a mutable member in a const function
  std::cout << "The volume of this box is " << volume() << std::endl;
  std::cout << "printVolume() has been called " << ++count << " time(s)" << std::endl;
}
```

Friends

- Have access to your private parts

Friend Functions of a Class

```
class Box
{
private:
  double length;
  double width;
  double height;

public:
  // Constructor
  Box(double lv = 1.0, double wv = 1.0, double hv = 1.0);

  double volume() const;                        // Function to calculate the volume of a box

  friend double surfaceArea(const Box& aBox);   // Friend function for the surface area
};

// friend function to calculate the surface area of a Box object
double surfaceArea(const Box& aBox)
{
  return 2.0*(aBox.length*aBox.width + aBox.length*aBox.height +aBox.height*aBox.width);
}
```

Friend Classes

- Used for highly intertwinend classes

```
class Box
{
  // Public members of the class...

  friend class Carton;

  // Private members of the class...
};
```

■ **Caution** Friend declarations risk undermining one of the cornerstones of object-oriented programming: data hiding. They should therefore be used only when absolutely necessary, and this need does not arise that often. You'll meet one circumstance where it is needed in the next chapter when you learn about operator overloading. Nevertheless, only most classes should not need any friends at all. While that may sound somewhat sad and lonely, the following humorous definition of the C++ programming language should forever remind you why, in C++, one should choose his friends very wisely indeed: "C++: where your friends can access your private parts."

Static Members of a Class

```
class Box
{
  private:
    static inline size_t objectCount {};
    double length;
    double width;
    double height;
    ...
};
```

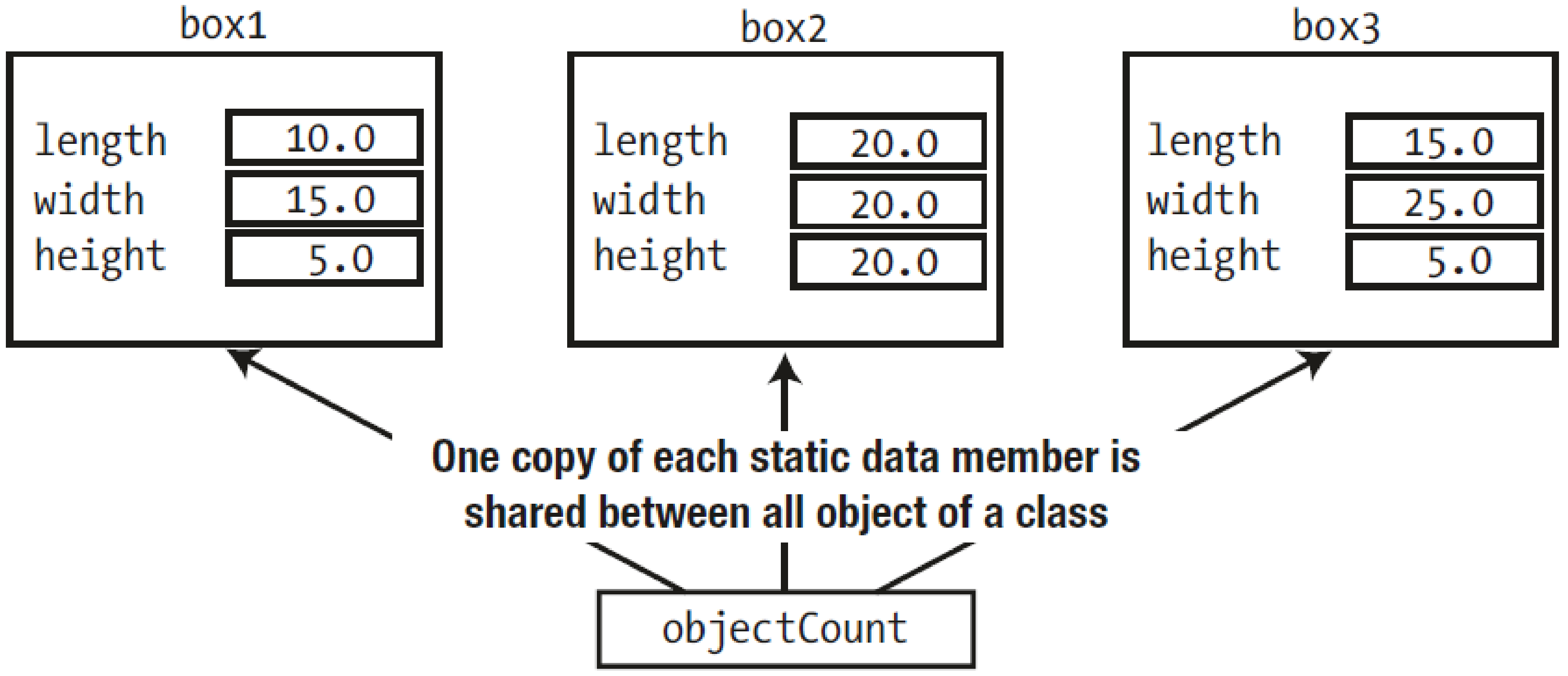

Figure 11-6. *Static class members are shared between objects.*

Static Members of a Class

```
class Box
{
private:
  double length {1.0};
  double width {1.0};
  double height {1.0};
  static inline size_t objectCount {};   // Count of objects in existence

public:
  // Constructors
  Box(double lv, double wv, double hv);
  Box(double side);                      // Constructor for a cube
  Box();                                 // Default constructor
  Box(const Box& box);                   // Copy constructor

  double volume() const;                 // Function to calculate the volume of a box

  size_t getObjectCount() const { return objectCount; } // ought to be declared as a static function
};
```

Static Members of a Class

```
// Constructor definition
Box::Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv}
{
  ++objectCount;
  std::cout << "Box constructor 1 called." << std::endl;
}

Box::Box(double side) : Box {side, side, side}  // Constructor for a cube
{
  std::cout << "Box constructor 2 called." << std::endl;
}

Box::Box()                                      // Default constructor
{
  ++objectCount;
  std::cout << "Default Box constructor called." << std::endl;
}

Box::Box(const Box& box)                        // Copy constructor
  : length {box.length}, width {box.width}, height {box.height}
{
  ++objectCount;
  std::cout << "Box copy constructor called." << std::endl;
}
```

declared as static

Static Members of a Class

```
// Ex11_11.cpp
// Using a static member variable
#include <iostream>
#include "Box.h"

int main()
{
  const Box box1 {2.0, 3.0, 4.0};                                  // An arbitrary box
  Box box2 {5.0};                                                  // A box that is a cube
  std::cout << "box1 volume = " << box1.volume() << std::endl;
  std::cout << "box2 volume = " << box2.volume() << std::endl;
  Box box3 {box2};
  std::cout << "box3 volume = " << box3.volume() << std::endl;   // Volume = 125

  std::cout << std::endl;

  Box boxes[6] {box1, box2, box3, Box {2.0}};

  std::cout << "\nThere are now " << box1.getObjectCount() << " Box objects." << std::endl;
}
```

- How many boxes are created?

```
~ClassName() {}
```

The name of the destructor for a class is always the class name prefixed with a tilde, ~. The destructor cannot have parameters or a return type. The default destructor in the Box class is as follows:

Destructors

```
~Box() {}
```

Of course, if the definition is placed outside the class, the name of the destructor would be prefixed with the class name:

```
Box::~Box() {}
```

If the body of your destructor is to be empty, however, you are again better off using the default keyword:

```
Box::~Box() = default;        // Have the compiler generate a default destructor
```

```
Box::~Box()                                    // Destructor
{
  std::cout << "Box destructor called." << std::endl;
  --objectCount;
}
```

Destructors

```cpp
int main()
{
  std::cout << "There are now " << Box::getObjectCount() << " Box objects." << std::endl;

  const Box box1 {2.0, 3.0, 4.0};     // An arbitrary box
  Box box2 {5.0};                     // A box that is a cube

  std::cout << "There are now " << Box::getObjectCount() << " Box objects." << std::endl;

  for (double d {} ; d < 3.0 ; ++d)
  {
    Box box {d, d + 1.0, d + 2.0};
    std::cout << "Box volume is " << box.volume() << std::endl;
  }

  std::cout << "There are now " << Box::getObjectCount() << " Box objects." << std::endl;

  auto pBox = std::make_unique<Box>(1.5, 2.5, 3.5);
  std::cout << "Box volume is " << pBox->volume() << std::endl;
  std::cout << "There are now " << pBox->getObjectCount() << " Box objects." << std::endl;
}
```

Nested Classes

```
class Truckload
{
private:
  class Package
  {
  public:
    SharedBox pBox;               // Pointer to the Box object contained in this Package
    Package* pNext;               // Pointer to the next Package in the list

    Package(SharedBox pb) : pBox{pb}, pNext{nullptr} {}     // Constructor
    ~Package() { delete pNext; }                            // Destructor
  };

  Package* pHead {};                                   // First in the list
  Package* pTail {};                                   // Last in the list
  Package* pCurrent {};                                // Last retrieved from the list

public:
  // Exact same public member functions as before...
};
```

- Objects of class Package can only be made inside the scope of class Truckload because it is defined in the private part. If it was defined in the public part then objects of it could be made from the outside
- The inner class has full access to the outer class members but not vice versa

Summary

- A *class* provides a way to define your own data types. Classes can represent whatever types of *objects* your particular problem requires.
- A class can contain *member variables* and *member functions*. The member functions of a class always have free access to the member variables of the same class.
- Objects of a class are created and initialized using member functions called *constructors*. A constructor is called automatically when an object declaration is encountered. Constructors can be overloaded to provide different ways of initializing an object.
- A copy constructor is a constructor for an object that is initialized with an existing object of the same class. The compiler generates a default copy constructor for a class if you don't define one.
- Members of a class can be specified as `public`, in which case they are freely accessible from any function in a program. Alternatively, they can be specified as `private`, in which case they may be accessed only by member functions, `friend` functions of the class, or members of nested classes.
- Member variables of a class can be `static`. Only one instance of each static member variable of a class exists, no matter how many objects of the class are created.
- Although `static` member variables of a class are accessible in a member function of an object, they aren't part of the object and don't contribute to its size.
- Every non-`static` member function contains the pointer `this`, which points to the current object for which the function is called.
- `static` member functions can be called even if no objects of the class have been created. A static member function of a class doesn't contain the pointer `this`.
- `const` member functions can't modify the member variables of a class object unless the member variables have been declared as `mutable`.
- Using references to class objects as arguments to function calls can avoid substantial overheads in passing complex objects to a function.
- A `destructor` is a member function that is called for a class object when it is destroyed. If you don't define a class destructor, the compiler supplies a default destructor.
- A nested class is a class that is defined inside another class definition.