**Chapter 13 Class Inheritance**

Often class libraries are available in source code, which means you can modify them to meet your needs. But C++ has a better method than code modification for extending and modifying classes. This method, called class inheritance, lets you derive new classes from old ones, with the derived class inheriting the properties, including the methods, of the old class, called a base class.

You don’t even need access to the source code to derive a class. Thus, if you purchase a class library that provides only the header files and the compiled code for class methods, you can still derive new classes based on the library classes.

* **Beginning with a Simple Base Class**

When one class inherits from another, the original class is called a base class, and the inheriting class is called a derived class.

**// tabtenn0.h** -- a table-tennis **base class**

#ifndef TABTENN0\_H\_

#define TABTENN0\_H\_

**#include <string>**

**using std::string;**

class **TableTennisPlayer**

{

private:

**string** firstname;

**string** lastname;

bool hasTable;

public:

TableTennisPlayer (const string & fn = “none", const string & ln = “none", bool ht = false);

void Name() const;

bool HasTable() const { return hasTable; };

void ResetTable(bool v) { hasTable = v; };

};

#endif

//**tabtenn0.cpp** -- simple base-class methods

#include “tabtenn0.h"

#include <iostream>

TableTennisPlayer::**TableTennisPlayer** (const string & fn, const string & ln, bool ht) :

firstname(fn), lastname(ln), hasTable(ht) {}

void TableTennisPlayer::Name() const

{

std::cout << lastname << “, “ << firstname;

}

All the *TableTennisPlayer* class does is keep track of the players’ names and whether they have tables. There are a couple of points to notice. **First, the class uses the standard string class to hold the names. This is more convenient, flexible, and safer than using a character array. Second, the constructor uses the member initializer list syntax introduced in Chapter 12.** You could also do this:

TableTennisPlayer::TableTennisPlayer (const string & fn, const string & ln, bool ht)

{

firstname = fn;

lastname = ln;

hasTable = ht;

}

However, **this approach has the effect of** first calling the **default string constructor** for firstname and then invoking the **string assignment operator** to reset firstname to fn. *But the member initializer list syntax saves a step by just using the string copy constructor to initialize firstname to fn*.

**// usett0.cpp** -- using a base class

#include <iostream>

#include "tabtenn0.h"

int main ( void )

{

using std::cout;

TableTennisPlayer player1("Chuck", "Blizzard", true); **// Use C-Style String when creating object instance**

TableTennisPlayer player2("Tara", "Boomdea", false);

player1.Name();

if (player1.HasTable())

cout << ": has a table.\n";

else

cout << ": hasn't a table.\n";

player2.Name();

if (player2.HasTable())

cout << ": has a table";

else

cout << ": hasn't a table.\n";

return 0;

}

Note that the program uses constructors with **C-style string arguments**:

TableTennisPlayer player1("Chuck", "Blizzard", true);

TableTennisPlayer player2("Tara", "Boomdea", false);

But the formal parameters for the constructor were declared as type **const string &**. This is a type mismatch, **but the string class**, much like the String class of Chapter 12, **has a constructor with a *const char \** parameter**, and that constructor is used automatically to create a string object initialized by the C-style string. In short, you can use either a string object or a C-style string as an argument to the TableTennisPlayer constructor. The first invokes a string constructor with a **const string &** parameter, and the second invokes a string constructor with a **const char \*** parameter.

**Deriving a Class**

Customers demand a class now that includes the point ratings they’ve earned through their play. Rather than start from scratch, you can derive a class from the *TableTennisPlayer* class. The first step is to have the *RatedPlayer* class declaration show that it derives from the TableTennisPlayer class:

// RatedPlayer derives from the TableTennisPlayer base class

class **RatedPlayer** : public TableTennisPlayer

{

...

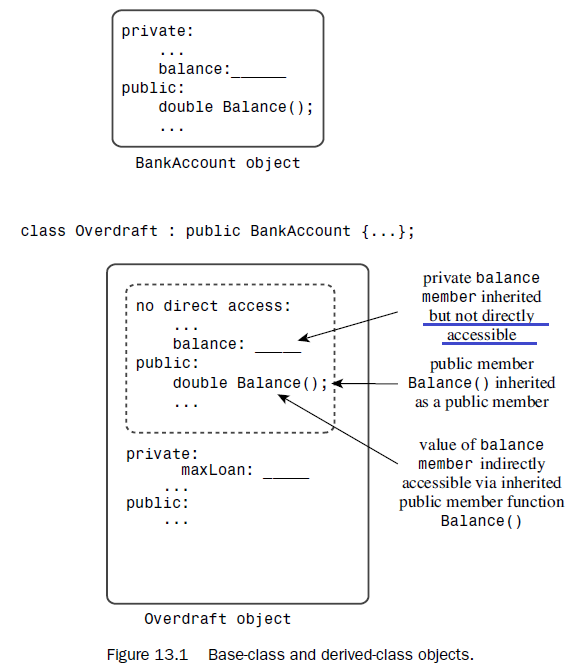
};

The colon indicates that the RatedPlayer class is based on the TableTennisPlayer class. This particular heading indicates that **TableTennisPlayer is a public base class**; this is termed ***public derivation***. An object of a derived class incorporates a base class object. **With *public derivation***, the public members of the base class become public members of the derived class. The private portions of a base class become part of the derived class, **but they can be accessed only through public and protected methods of the base class**. (We’ll get to protected members in a bit.)

If you declare a *RatedPlayer* object, it has the following special properties:

* An object of the derived type has stored within it the data members of the base type. (The derived class inherits the base-class implementation.)
* An object of the derived type can use the methods of the base type. (The derived class inherits the base-class interface.)

Thus, a RatedPlayer object can store the first name and last name of each player and whether the player has a table. Also a RatedPlayer object can use the Name(), HasTable(), and ResetTable() methods from the TableTennisPlayer class (see Figure 13.1 for another example).



What needs to be added to these inherited features?

* A derived class needs *its own constructors*;
* A derived class can add additional data members and member functions as needed;

In this particular case, the class needs one more data member to hold the ratings value. It should also have a method for retrieving the rating and a method for resetting the rating. So the class declaration could look like this:

// simple derived class

class RatedPlayer : public TableTennisPlayer

{

private:

unsigned int rating; // add a additional data member

public:

**RatedPlayer** (unsigned int r = 0, const string & fn = "none", const string & ln = "none", bool ht = false);

**RatedPlayer**(unsigned int r, const TableTennisPlayer & tp);

unsigned int Rating() const { return rating; } // add a method

void ResetRating (unsigned int r) {rating = r;} // add a method

};

**Constructors: Access Considerations**

**A derived class does not have direct access to the private members of the base class**; it has to work through the base-class methods. For example, the ***RatedPlayer* *constructors*** **cannot directly set the inherited members (firstname, lastname, and hasTable)**. Instead, they have to use public base-class methods to access private base-class members. In particular, *the derived-class constructors have to use the base-class constructors*.

When a program constructs a derived-class object, it **first constructs** the **base-class object**. Conceptually, that means **the base-class object should be constructed before the program enters the body of the derived-class constructor**. C++ uses the member initialize list syntax to accomplish this.

RatedPlayer::RatedPlayer(unsigned int r, const string & fn, const string & ln, bool ht) **: TableTennisPlayer(fn, ln, ht)**

{

rating = r;

}

The following part is the member initializer list:

: TableTennisPlayer(fn, ln, ht)

***It calls the TableTennisPlayer constructor***. Suppose, for example, a program has the following declaration:

RatedPlayer rplayer1(1140, "Mallory", "Duck", true);

The RatedPlayer constructor assigns the actual arguments "Mallory", "Duck", and true to the formal parameters fn, ln, and ht. It then passes these parameters on as actual arguments to the TableTennisPlayer constructor. **This constructor, in turn, creates the embedded TableTennisPlayer object and stores the data "Mallory", "Duck", and true in it**. Then the program enters the body of the RatedPlayer constructor, completes the construction of the RatedPlayer object, and assigns the value of the parameter r (that is, 1140) to the rating member.

What if you omit the member initializer list?

RatedPlayer::RatedPlayer(unsigned int r, const string & fn, const string & ln, bool ht) // what if no initializer list?

{

rating = r;

}

**The base-class object must be created first,** so if you omit calling a base-class constructor, the program uses the default base-class constructor. Therefore, the previous code is the same as the following:

RatedPlayer::RatedPlayer(unsigned int r, const string & fn, const string & ln, bool ht) // : TableTennisPlayer()

{

rating = r;

}

Unless you want the default constructor to be used, you should explicitly provide the correct base-class constructor call.

Now let’s look at code for the second constructor:

RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer **&** tp) : TableTennisPlayer(tp)

{

rating = r;

}

Again, the TableTennisPlayer information is passed on to a TableTennisPlayer constructor:

TableTennisPlayer(tp)

Because **tp** is type *const TableTennisPlayer &*, this call invokes the *base-class copy constructor*. The base class didn’t define a copy constructor, but recall from Chapter 12 that the compiler automatically generates a copy constructor if one is needed and you haven’t defined one already. In this case, the implicit copy constructor, which does memberwise copying, is fine because the class doesn’t directly use dynamic memory allocation. (The string members do use dynamic memory allocation, but, recall, memberwise copying will use the string class copy constructors to copy the string members.)

You may, if you like, also use member initializer list syntax for members of the derived class. In this case, you use the member name instead of the class name in the list. Thus, the second constructor can also be written in this manner:

// alternative version

RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp) : TableTennisPlayer(tp), rating(r)

{

}

These are the key points about constructors for derived classes:

1. The base-class object is constructed first;
2. The derived-class constructor should pass base-class information to a base-class constructor via a member initializer list;
3. The derived-class constructor should initialize the data members that were added to the derived class;

**This example doesn’t provide explicit destructors**, so the implicit destructors are used. **Destroying an object occurs in the opposite order used to construct an object**. That is, the body of the derived-class destructor is executed first, and then the base-class destructor is called automatically.

When creating an object of a derived class, a program first calls the base-class constructor and then calls the derived-class constructor. The base-class constructor is responsible for initializing the inherited data members. The derived-class constructor is responsible for initializing any added data members. A derived-class constructor always calls a base-class constructor. You can use the initializer list syntax to indicate which base-class constructor to use. Otherwise, the default base-class constructor is used.

**Using a Derived Class**

// **tabtenn1.h** -- a table-tennis base class

#ifndef TABTENN1\_H\_

#define TABTENN1\_H\_

#include <string>

using std::string;

// simple base class

class TableTennisPlayer

{

private:

string firstname;

string lastname;

bool hasTable;

public:

TableTennisPlayer (const string & fn = "none", const string & ln = "none", bool ht = false);

void Name() const;

bool HasTable() const { return hasTable; };

void ResetTable(bool v) { hasTable = v; };

};

// simple derived class

class RatedPlayer : **public** TableTennisPlayer

{

private:

unsigned int rating;

public:

RatedPlayer (unsigned int r = 0, const string & fn = "none", const string & ln = "none", bool ht = false);

RatedPlayer(unsigned int r, const TableTennisPlayer & tp);

unsigned int Rating() const { return rating; }

void ResetRating (unsigned int r) {rating = r;}

};

#endif

You could give each class its own header file, but because the two classes are related, it makes more organizational sense to keep the class declarations together.

//**tabtenn1.cpp** -- simple base-class methods

#include "tabtenn1.h"

#include <iostream>

TableTennisPlayer::TableTennisPlayer (const string & fn, const string & ln, bool ht) : firstname(fn),

lastname(ln), hasTable(ht) {}

void TableTennisPlayer::Name() const

{

std::cout << lastname << ", " << firstname;

}

// RatedPlayer methods

RatedPlayer::RatedPlayer(unsigned int r, **const string &** fn, **const string &** ln, bool ht) : TableTennisPlayer(fn, ln, ht)

{

rating = r;

}

RatedPlayer::RatedPlayer(unsigned int r, **const TableTennisPlayer &** tp) : TableTennisPlayer(tp), rating(r)

{

}

下面是测试的主程序:

// **usett1.cpp** -- using base class and derived class

#include <iostream>

#include "tabtenn1.h"

int main ( void )

{

using std::cout;

using std::endl;

TableTennisPlayer **player1**("Tara", "Boomdea", false);

RatedPlayer **rplayer1**(1140, "Mallory", "Duck", true);

rplayer1.Name(); // derived object uses base method

if (rplayer1.HasTable())

cout << ": has a table.\n";

else

cout << ": hasn't a table.\n";

player1.Name(); // base object uses base method

if (player1.HasTable())

cout << ": has a table";

else

cout << ": hasn't a table.\n";

cout << "Name: ";

rplayer1.Name();

cout << "; Rating: " << rplayer1.Rating() << endl;

*// initialize RatedPlayer using TableTennisPlayer object*

**RatedPlayer rplayer2(1212, player1);**

cout << "Name: ";

rplayer2.Name();

cout << "; Rating: " << rplayer2.Rating() << endl;

return 0;

}

**Special Relationships Between Derived and Base Classes**

A derived class has some special relationships with the base class. One, which you’ve just seen, is that a derived-class object can use base-class methods, **provided that the methods are not private**:

RatedPlayer rplayer1(1140, "Mallory", "Duck", true);

rplayer1.Name(); // derived object uses base method

**Two other important relationships** are that a **base-class pointer can point to a derived class object without an explicit type cast** and that a **base-class reference can refer to a derived-class object without an explicit type cast**:

RatedPlayer rplayer(1140, "Mallory", "Duck", true);

TableTennisPlayer & rt = rplayer;

TableTennisPlayer \* pt = &rplayer;

rt.Name(); // invoke Name() with reference

pt->Name(); // invoke Name() with pointer

However, a base-class pointer or reference can **invoke just base-class methods**, so you couldn’t use rt or pt to invoke, say, the derived-class ResetRanking() method.

Ordinarily, C++ requires that references and pointer types match the assigned types, but this rule is relaxed for inheritance. However, the rule relaxation is just in one direction. **You can’t assign base-class objects and addresses to derived-class references and pointers**:

TableTennisPlayer player("Betsy", "Bloop", true);

RatedPlayer & rr = player; // NOT ALLOWED

RatedPlayer \* pr = player; // NOT ALLOWED

**Both these sets of rules make sense**. For example, consider the implications of having a base-class reference refer to a derived object. In this case, you can **use the base-class reference to invoke base-class methods for the derived-class object**. Because the derived class inherits the base-class methods and data members, this causes no problems. Now consider what would happen if you could assign a base-class object to a derived-class reference. The derived-class reference would be able to invoke derived-class methods for the base object, and that could cause problems. For example, applying the RatedPlayer::Rating() method to a TableTennisPlayer object makes no sense because the TableTennisPlayer object doesn’t have a rating member.

**The fact that base-class references and pointers can refer to derived-class objects has some interesting consequences.** One is that functions defined with *base-class reference or pointer arguments* can be used with either base-class or derived-class objects. For instance, consider this function:

void Show(const TableTennisPlayer & rt)

{

using std::cout;

cout << "Name: ";

rt.Name();

cout << "\nTable: ";

if (rt.HasTable())

cout << "yes\n";

else

cout << "no\n";

}

The formal parameter rt is a reference to a base class, so it can refer to a base-class object or to a derived-class object. Thus, you can use Show() with either a TableTennis argument or a RatedPlayer argument:

TableTennisPlayer player1("Tara", "Boomdea", false);

**RatedPlayer rplayer1**(1140, "Mallory", "Duck", true);

Show(player1); // works with TableTennisPlayer argument

Show(rplayer1); // works with RatedPlayer argument

void Wohs(const TableTennisPlayer \* pt); // function with pointer parameter

...

TableTennisPlayer player1("Tara", "Boomdea", false);

RatedPlayer rplayer1(1140, "Mallory", "Duck", true);

Wohs(&player1); // works with TableTennisPlayer \* argument

Wohs(**&rplayer1**); // works with RatedPlayer \* argument

The reference compatibility property also **allows you to initialize a base-class object to a derived-class object**, although somewhat indirectly. Suppose you have this code:

RatedPlayer olaf1(1840, "Olaf", "Loaf", true);

TableTennisPlayer olaf2(olaf1);

**是说可以用derived class object去初始化base class object**;

The exact match for initializing olaf2 would be a constructor with this prototype:

TableTennisPlayer(const RatedPlayer &); // doesn't exist

The class definitions don’t include this constructor, but there is the ***implicit copy constructor***:

// implicit copy constructor

TableTennisPlayer(const TableTennisPlayer &);

The formal parameter is a reference to the base type, so it can refer to a derived type. Thus, the attempt to initialize olaf2 to olaf1 uses this constructor, which copies the firstname, lastname, and hasTable members. In other words, it initializes olaf2 to the TableTennisPlayer object embedded in the RatedPlayer object olaf1. **Similarly, you can assign a derived-class object to a base-class object**:

RatedPlayer olaf1(1840, "Olaf", "Loaf", true);

TableTennisPlayer winner;

**winner = olaf1**; // assign derived to base object

In this case, the program uses the ***implicit overloaded assignment operator***:

TableTennisPlayer & operator=(const TableTennisPlayer &) const;

Again, a base-class reference refers to a derived-class object, *and just the base-class portion of olaf1 is copied to winner*.

* **Inheritance: An Is-a Relationship**

The special relationship between a derived class and a base class is based on an underlying model for C++ inheritance. Actually, C++ has three varieties of inheritance: **public**, **protected**, and **private**. ***Public inheritance*** is the most common form, and it models an **is-a relationship**. This is shorthand for saying that an object of a **derived class** should also be an object of the **base class**. Anything you do with a base-class object, you should be able to do with a derived-class object.

To clarify is-a relationships, let’s look at some examples that don’t match that model. **Public inheritance** doesn’t model a *has-a* relationship. A lunch, for example, might contain a fruit. But a lunch, in general, is not a fruit. Therefore, you should not derive a Lunch class from the Fruit class in an attempt to place fruit in a lunch. The correct way to handle putting fruit into a lunch is to consider the matter as a has-a relationship: A lunch has a fruit. As you’ll see in Chapter 14, that’s most easily modeled by including a Fruit object as a data member of a Lunch class.

**Public inheritance** doesn’t model an *is-implemented-as-a* relationship. For example, you could implement a stack by using an array. However, it wouldn’t be proper to derive a Stack class from an Array class. A stack is not an array. For example, array indexing is not a stack property. Also a stack could be implemented in some other way, such as by using a linked list. **A proper approach would be to hide the array implementation by giving the stack a private Array object member**.

**Public inheritance** doesn’t model a *uses-a relationship*. For example, a computer can use a laser printer, but it doesn’t make sense to derive a Printer class from a Computer class, or vice versa. You might, however, devise friend functions or classes to handle communication between Printer objects and Computer objects.

* **Polymorphic Public Inheritance**

Objects of the derived class use the base-class methods without change. But you can encounter situations in which you want a method to behave differently for the derived class than it does for the base class. That is, *the way a particular method behaves may depend on the object that invokes it*. This more sophisticated behavior is termed polymorphic (“having many forms”) because you can have multiple behaviors for a method, depending on the context.

There are two key mechanisms for implementing polymorphic public inheritance:

1. Redefining base-class methods in a derived class;
2. Using virtual methods;

It’s time for another example. You become head programmer for the Pontoon National Bank. The first thing the bank asks you to do is develop two classes. One class will represent its basic checking plan, the Brass Account, and the second class will represent the Brass Plus checking account, which adds an overdraft protection feature. That is, if a user writes a check larger (but not too much larger) than his or her balance, the bank will cover the check, charging the user for the excess payment and adding a surcharge. You can characterize the two accounts in terms of data to be stored and operations to be allowed.

First, here is the information for a **Brass Account** checking plan:

1. Client name;
2. Account number;
3. Current balance;

And here are the operations to be represented:

1. Creating an account;
2. Depositing money into the account;
3. Withdrawing money from the account;
4. Displaying the account information;

For the **Brass Plus Account** checking plan, the Pontoon National Bank wants all the features of the Brass Account as well as the following additional items of information:

1. An **upper limit** to the overdraft protection;
2. An **interest rate** charged for overdraft loans;
3. The **overdraft amount** currently owed to the bank;

Two operations need to be implemented differently:

1. The withdrawing money operation has to incorporate overdraft protection for the Brass Plus Account;
2. The display operation has to show the additional information required by the Brass Plus Account;

**[With *public derivation***, the public members of the base class become public members of the derived class. The private portions of a base class become part of the derived class, **but they can be accessed only through public and protected methods of the base class.]**

还有一些额外信息: overdraft上限是500, 但可变; 利息率11.125%, 但可变; 付overdraft的不能通过deposit或者转账, 必须付cash(这一点只是为了让程序简单);

// **brass.h** -- bank account classes

#ifndef BRASS\_H\_

#define BRASS\_H\_

#include <string>

**// Brass Account Class**

class Brass

{

private:

std::string fullName;

long acctNum;

double balance;

public:

Brass(const std::string & s = "Nullbody", long an = -1, double bal = 0.0);

void Deposit(double amt);

*virtual* void Withdraw(double amt);

double Balance() const;

*virtual* void ViewAcct() const;

*virtual* ~Brass() {}

};

**//Brass Plus Account Class**

***class BrassPlus : public Brass***

{

private:

double maxLoan;

double rate;

double owesBank;

public:

BrassPlus( const std::string & s = "Nullbody",

long an = -1,

double bal = 0.0,

double ml = 500,

double r = 0.11125);

BrassPlus(const Brass & ba, double ml = 500, double r = 0.11125);

*virtual* void ViewAcct()const; // redefined here

*virtual* void Withdraw(double amt); // redefined here

void ResetMax(double m) { maxLoan = m; }

void ResetRate(double r) { rate = r; };

void ResetOwes() { owesBank = 0; }

};

#endif

Both the Brass class and the BrassPlus class declare the ViewAcct() and Withdraw() methods; these are the methods that will behave differently for a BrassPlus object than they do with a Brass object. The two ViewAcct() prototypes indicate that there will be two separate method definitions.The qualified name for the base-class version is Brass::ViewAcct(), and the qualified name for the derived-class version is BrassPlus::ViewAcct().A program will use the object type to determine which version to use:

Brass dom("Dominic Banker", 11224, 4183.45);

BrassPlus dot("Dorothy Banker", 12118, 2592.00);

dom.ViewAcct(); // use Brass::ViewAcct()

dot.ViewAcct(); // use BrassPlus::ViewAcct()

Methods that behave the same for both classes, such as Deposit() and Balance(), are declared only in the base class.

**The use of *virtual* determines** which method is used if the method is invoked by a reference or a pointer **instead of by an object**.

* If you **don’t use the keyword virtual**, the program chooses a method based on the reference type or pointer type;
* If you **do use the keyword virtual**, the program chooses a method based on the type of object the reference or pointer ***refers to***;

Here is how a program behaves if ViewAcct() is not virtual:

// behavior with non-virtual ViewAcct() method chosen according to reference type

Brass dom("Dominic Banker", 11224, 4183.45);

BrassPlus dot("Dorothy Banker", 12118, 2592.00);

Brass & b1\_ref = dom;

**Brass** & b2\_ref = dot;

b1\_ref.ViewAcct(); // use Brass::ViewAcct()

b2\_ref.ViewAcct(); **// use Brass::ViewAcct()**

The reference variables are type Brass, so Brass::ViewAccount() is chosen. Using pointers to Brass instead of references results in similar behavior.

In contrast, here is the behavior if ViewAcct() is virtual:

// behavior with virtual ViewAcct() method chosen according to object type

Brass dom("Dominic Banker", 11224, 4183.45);

BrassPlus dot("Dorothy Banker", 12118, 2592.00);

Brass & b1\_ref = dom;

**Brass** & b2\_ref = dot;

b1\_ref.ViewAcct(); // use Brass::ViewAcct()

b2\_ref.ViewAcct(); **// use BrassPlus::ViewAcct()**

In this case, both references are type Brass, but b2\_ref refers to a BrassPlus object, so BrassPlus::ViewAcct() is used for it. Using pointers to Brass instead of references results in similar behavior.

It turns out, as you’ll see in a bit, that this behavior of virtual functions is very handy. Therefore, it is the common practice to declare as virtual in the base class those methods that might be redefined in a derived class. When a method is declared virtual in a base class, it is automatically virtual in the derived class, but it is a good idea to document which functions are virtual by using the keyword virtual in the derived class declarations, too.

The base class declares a virtual destructor. This is to make sure that the correct sequence of destructors is called when a derived object is destroyed. We’ll discuss this point in more detail later in this chapter.

***[Note]***

**If you redefine a base-class method in a derived class, the usual practice is to declare the base-class method as virtual**. This makes the program choose the method version based on object type instead of the type of a reference or pointer. **It’s also the usual practice to declare a virtual destructor for the base class**.

// **brass.cpp** -- bank account class methods

#include <iostream>

#include "brass.h"

using std::cout;

using std::endl;

using std::string;

// formatting stuff

typedef std::ios\_base::fmtflags format;

typedef std::streamsize precis;

format setFormat();

void restore(format f, precis p);

**// Brass methods**

Brass::Brass(const string & s, long an, double bal)

{

fullName = s;

acctNum = an;

balance = bal;

}

void Brass::Deposit(double amt)

{

if (amt < 0)

cout << "Negative deposit not allowed; " << "deposit is cancelled.\n";

else

balance += amt;

}

void Brass::Withdraw(double amt)

{

// set up ###.## format

format initialState = setFormat();

precis prec = cout.precision(2);

if (amt < 0)

cout << "Withdrawal amount must be positive; " << "withdrawal canceled.\n";

else if (amt <= balance)

balance -= amt;

else

cout << "Withdrawal amount of $" << amt << " exceeds your balance.\n" << "Withdrawal canceled.\n";

restore(initialState, prec);

}

double Brass::Balance() const

{

return balance;

}

void Brass::ViewAcct() const

{

// set up ###.## format

format initialState = setFormat();

precis prec = cout.precision(2);

cout << "Client: " << fullName << endl;

cout << "Account Number: " << acctNum << endl;

cout << "Balance: $" << balance << endl;

restore(initialState, prec); // restore original format

}

**// BrassPlus Methods**

BrassPlus::BrassPlus(**const string & s, long an, double bal**, double ml, double r) ***: Brass(s, an, bal)***

{

maxLoan = ml;

owesBank = 0.0;

rate = r;

}

BrassPlus::BrassPlus(**const Brass & ba**, double ml, double r) ***: Brass(ba)*** ***// uses implicit copy constructor***

{

maxLoan = ml;

owesBank = 0.0;

rate = r;

}

// *redefine* how ViewAcct() works

void BrassPlus::**ViewAcct()** const

{

// set up ###.## format

format initialState = setFormat();

precis prec = cout.precision(2);

**Brass::ViewAcct();** // display base portion, ***explicitly call the Brass::ViewAcct() of base class;***

cout << "Maximum loan: $" << maxLoan << endl;

cout << "Owed to bank: $" << owesBank << endl;

cout.precision(3); // ###.### format

cout << "Loan Rate: " << 100 \* rate << "%\n";

restore(initialState, prec);

}

// *redefine* how Withdraw() works

void BrassPlus::**Withdraw(double amt)**

{

// set up ###.## format

format initialState = setFormat();

precis prec = cout.precision(2);

**double bal = Balance();**

if (amt <= bal)

**Brass::Withdraw(amt); // *explicitly call the Brass::ViewAcct() of base class;***

else if (amt <= bal + maxLoan - owesBank)

{

double advance = amt - bal;

owesBank += advance \* (1.0 + rate);

cout << "Bank advance: $" << advance << endl;

cout << "Finance charge: $" << advance \* rate << endl;

**Deposit(advance); *// no Brass:: because only class Brass has function Deposit()***

**Brass::Withdraw(amt);**

}

else

cout << "Credit limit exceeded. Transaction cancelled.\n";

restore(initialState, prec);

}

format setFormat()

{

// set up ###.## format

return cout.setf(std::ios\_base::fixed,

std::ios\_base::floatfield);

}

void restore(format f, precis p)

{

cout.setf(f, std::ios\_base::floatfield);

cout.precision(p);

}

Note that the keyword virtual is used just in the method prototypes in the class declaration, not in the method definitions in Listing above.

Keep in mind that the **derived class does not have direct access to *private* base-class data**; the derived class has to use base-class public methods to access that data. The means of access depends on the method. Constructors use one technique, and other member functions use a different technique.

The technique that **derived-class constructors use to initialize base-class private data** is the member initializer list syntax. 参见上面BrassPlus Class的那两个constructor. Each of these constructors uses the member initializer list syntax to pass base-class information to a base-class constructor and then uses the constructor body to initialize the new data items added by the BrassPlus class.

Non-constructors can’t use the member initializer list syntax. **But a derived-class method can call a public base-class method**. BrassPlus::ViewAcct() displays the added BrassPlus data members and calls on the base-class method Brass::ViewAcct() to display the base-class data members. Using the scope-resolution operator in a derived-class method to invoke a baseclass method is a standard technique.

It’s vital that the code use the scope-resolution operator. Suppose that, instead, you wrote the code this way:

// redefine erroneously how ViewAcct() works

void BrassPlus::ViewAcct() const

{

...

ViewAcct(); // oops! recursive call

...

}

**If the code doesn’t use the scope-resolution operator**, the compiler assumes that ViewAcct() is BrassPlus::ViewAcct(), and this creates a recursive function that has no termination—not a good thing.

// **usebrass1.cpp** -- testing bank account classes

#include <iostream>

#include "brass.h"

int main()

{

using std::cout;

using std::endl;

Brass Piggy("Porcelot Pigg", 381299, 4000.00);

BrassPlus Hoggy("Horatio Hogg", 382288, 3000.00);

Piggy.ViewAcct();

cout << endl;

Hoggy.ViewAcct();

cout << endl;

cout << "Depositing $1000 into the Hogg Account:\n";

Hoggy.Deposit(1000.00);

cout << "New balance: $" << Hoggy.Balance() << endl;

cout << "Withdrawing $4200 from the Pigg Account:\n";

Piggy.Withdraw(4200.00);

cout << "Pigg account balance: $" << Piggy.Balance() << endl;

cout << "Withdrawing $4200 from the Hogg Account:\n";

Hoggy.Withdraw(4200.00);

Hoggy.ViewAcct();

return 0;

}

In Listing above the methods are invoked by objects, not pointers or references, so this program doesn’t use the virtual method feature. Let’s look at an example for which the virtual methods do come into play. You can create an array of pointers-to-Brass. In that case, every element is of the same type, but because of the public inheritance model, a pointer-to-Brass can point to either a Brass or a BrassPlus object. Thus, in effect, you have a way of representing a collection of more than one type of object with a single array.

// **usebrass2.cpp** -- polymorphic example

#include <iostream>

#include <string>

#include "brass.h"

const int CLIENTS = 4;

int main()

{

using std::cin;

using std::cout;

using std::endl;

***Brass \* p\_clients[CLIENTS];***

std::string temp;

long tempnum;

double tempbal;

char kind;

for (int i = 0; i < CLIENTS; i++)

{

cout << "Enter client's name: ";

getline(cin,temp);

cout << "Enter client's account number: ";

cin >> tempnum;

cout << "Enter opening balance: $";

cin >> tempbal;

cout << "Enter 1 for Brass Account or " << "2 for BrassPlus Account: ";

while (cin >> kind && (kind != ‘1' && kind != ‘2'))

cout <<"Enter either 1 or 2: ";

if (kind == ‘1')

p\_clients[i] = *new* Brass(temp, tempnum, tempbal);

else

{

double tmax, trate;

cout << "Enter the overdraft limit: $";

cin >> tmax;

cout << "Enter the interest rate " << "as a decimal fraction: ";

cin >> trate;

p\_clients[i] = new BrassPlus(temp, tempnum, tempbal, tmax, trate);

}

while (cin.get() != ‘\n')

continue;

} // end of for loop

cout << endl;

for (int i = 0; i < CLIENTS; i++)

{

**p\_clients[i]->ViewAcct();** // polymorphic aspect here

cout << endl;

}

for (int i = 0; i < CLIENTS; i++)

{

**delete p\_clients[i];** // free memory

}

cout << "Done.\n";

return 0;

}

The code in Listing above that uses *delete* to free the objects allocated by new illustrates *why the base class should have a virtual destructor*, even if no destructor appears to be needed. If the destructors are not virtual, then just the destructor corresponding to the pointer type is called. 程序里, Brass Class有一个空的destructor, BrassPlus Class没有定义destructor.

This means that only the Brass destructor would be called, even if the pointer pointed to a BrassPlus object. **If the destructors are virtual**, the destructor corresponding to the object type is called. So if a pointer points to a BrassPlus object, the BrassPlus destructor is called. **And when a BrassPlus destructor finishes, it automatically calls the base-class constructor**. Thus, *using virtual destructors ensures that the correct sequence of destructors is called*. In Listing above, this correct behavior isn’t essential because the destructors do nothing. But if, say, BrassPlus had a do-something destructor, it would be vital for Brass to have a virtual destructor, even if it did nothing.

* **Static and Dynamic Binding**

通常, 在编译阶段compiler就知道一个函数对应的code (这叫static binding), 但是在上面的例子中, 选1或者2决定了初始化哪个class, 从而决定了后面调用哪个函数 (这叫dynamic binding).

**Pointer and Reference Type Compatibility**

*Dynamic binding* in C++ is associated with methods invoked by pointers and references, and this is governed, in part, by the inheritance process. One way public inheritance models the is-a relationship is in how it handles pointers and references to objects. **Normally, C++ does not allow you to assign an address of one type to a pointer of another type. Nor does it let a reference to one type refer to another type**:

double x = 2.5;

int \* pi = &x; // invalid assignment, mismatched pointer types

long & rl = x; // invalid assignment, mismatched reference type

**However**, as you’ve seen, a **reference or a pointer to a base class can refer to a derived class object without using an explicit type cast**.

BrassPlus dilly ("Annie Dill", 493222, 2000);

Brass \* pb = &dilly; // ok

Brass & rb = dilly; // ok

Base class的指针或者引用可以指向derived class;

Converting a derived-class reference or pointer to a base-class reference or pointer is called **upcasting**, and it is always allowed for public inheritance without the need for an explicit type cast. This rule is part of expressing the is-a relationship. A BrassPlus object is a Brass object in that it inherits all the data members and member functions of a Brass object. Therefore, **anything that you can do to a Brass object, you can do to a BrassPlus object**.

**Upcasting is transitive**. That is, if you derive a BrassPlusPlus class from BrassPlus, then a Brass pointer or reference can refer to a Brass object, a BrassPlus object, or a BrassPlusPlus object.

**The opposite process**, converting a base-class pointer or reference to a derived-class pointer or reference, is called **downcasting**, and it is not allowed without an explicit type cast. The reason for this restriction is that the is-a relationship is not, in general, symmetric.

Upcasting also takes place for function calls with base-class references or pointers as parameters. Consider the following code fragment, and suppose each function calls upon the virtual method ViewAcct():

void fr(Brass & rb); // uses rb.ViewAcct()

void fp(Brass \* pb); // uses pb->ViewAcct()

void fv(Brass b); // uses b.ViewAcct()

int main()

{

Brass b("Billy Bee", 123432, 10000.0);

BrassPlus bp("Betty Beep", 232313, 12345.0);

fr(b); // uses Brass::ViewAcct()

fr(bp); // uses **BrassPlus**::ViewAcct()

fp(b); // uses Brass::ViewAcct()

fp(bp); // uses **BrassPlus**::ViewAcct()

fv(b); // uses Brass::ViewAcct()

fv(bp); // **uses Brass::ViewAcct()**

...

}

***Passing by value*** causes only the Brass component of a BrassPlus object to be passed to the fv() function. But the **implicit upcasting that occurs with references and pointers** causes the fr() and fp() functions to use Brass::ViewAcct() for Brass objects and BrassPlus::ViewAcct() for BrassPlus objects.

Implicit upcasting makes it possible for a base-class pointer or reference to refer to either a base-class object or a derived-class object, and that produces the need for dynamic binding. Virtual member functions are the C++ answer to that need.

**Virtual Member Functions and Dynamic Binding**

Let’s revisit the process of invoking a method with a reference or pointer. Consider the following code:

BrassPlus ophelia; // derived-class object

Brass \* bp; // base-class pointer

bp = &ophelia; // Brass pointer to BrassPlus object

bp->ViewAcct(); **// which version?**

As discussed before, *if ViewAcct() is not declared as virtual in the base class*, **bp->ViewAcct() goes by the pointer type (Brass \*) and invokes Brass::ViewAcct()**. The pointer type is known at compile time, so the compiler can bind ViewAcct() to Brass::ViewAcct() at compile time. In short, the compiler uses **static binding** for nonvirtual methods.

But if ViewAcct() is declared as virtual in the base class, **bp->ViewAcct()** goes by the **object type** (BrassPlus) and invokes **BrassPlus::ViewAcct().** In this example, you can see that the object type is BrassPlus, but, in general, the object type might only be determined when the program is running. Therefore, the compiler generates code that binds ViewAcct() to Brass::ViewAcct() or BrassPlus::ViewAcct(), depending on the object type, while the program executes. In short, the compiler uses **dynamic binding** for virtual methods.

* **Why Two Kinds of Binding and Why Static Is the Default**

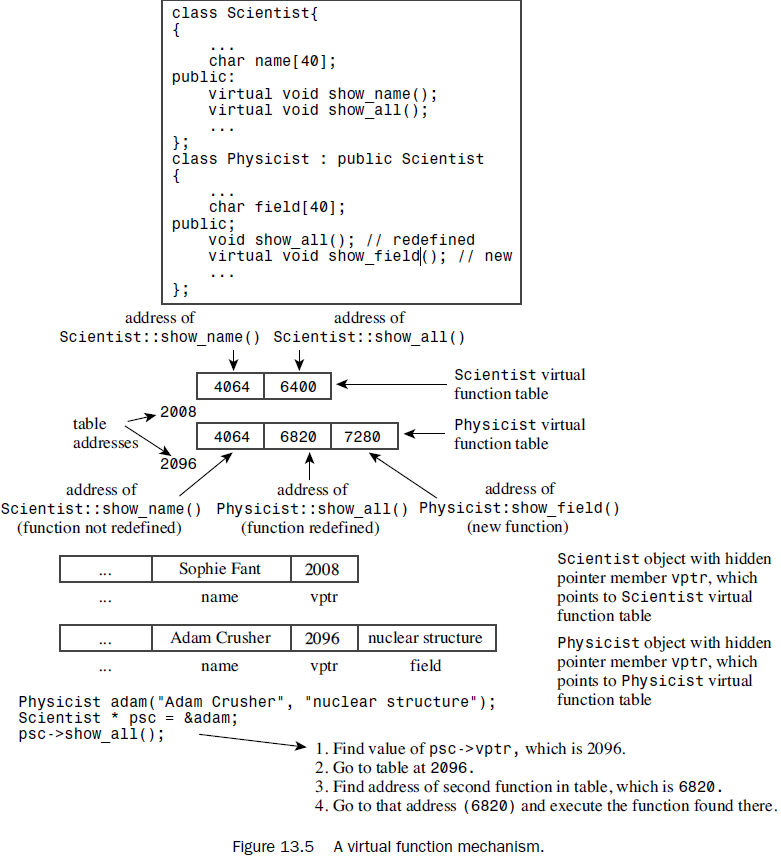
There are two reasons: efficiency and a conceptual model.

**First, let’s consider efficiency**. For a program to be able to make a runtime decision, it has to have some way to keep track of what sort of object a base-class pointer or reference refers to, and that entails some extra processing overhead. If, for example, you design a class that won’t be used as a base class for inheritance, you don’t need dynamic binding. Similarly, if you have a derived class, such as the RatedPlayer example, that does not redefine any methods, you don’t need dynamic binding. In these cases, it makes sense to use static binding and gain a little efficiency. The fact that static binding is more efficient is the reason it is the default choice for C++. You should therefore go to virtual functions only if the program design needs them.

**Next, let’s consider the conceptual model**. When you design a class, you may have member functions that you **don’t want redefined** in derived classes. For example, the Brass::Balance() function, which returns the account balance, seems like a function that **shouldn’t be redefined**. **By making this function non-virtual**, **you accomplish two things.** First, you make it *more efficient*. Second, you announce that *it is your intention that this function* ***not be redefined***. **That suggests reserving the virtual label just for methods you expect to be redefined**.

* **How Virtual Functions Work**

The usual way compilers handle virtual functions is to add a hidden member to each object. The hidden member holds a pointer to an array of function addresses. Such an array is usually termed a *virtual function table (vtbl)*. The vtbl holds the addresses of the virtual functions declared for objects of **that class**. For example, an object of a base class contains a pointer to a table of addresses of all the virtual functions for that class. An object of a derived class contains a pointer to a **separate table** of addresses. If the derived class provides a new definition of a virtual function, the vtbl holds the address of the new function. If the derived class doesn’t redefine the virtual function, the vtbl holds the address of the original version of the function. If the derived class defines a new function and makes it virtual, its address is added to the vtbl (see Figure 13.5). Note that whether you define 1 or 10 virtual functions for a class, you add just one address member to an object; it’s the table size that varies.



* **Things to Know About Virtual Methods**

Beginning a class method declaration with the keyword virtual in a base class makes the function virtual for the base class and all classes derived from the base class, including classes derived from the derived classes, and so on.

**Constructors can’t be virtual**. Creating a derived object invokes a derived-class constructor, not a base-class constructor. The derived-class constructor then uses a base-class constructor, but this sequence is distinct from the inheritance mechanism. Thus, a derived class doesn’t inherit the base-class constructors.

**Destructors should be virtual unless a class isn’t to be used as a base class**. For example, suppose Employee is a base class and Singer is a derived class that adds a char \* member that points to memory allocated by new.Then, when a Singer object expires, it’s vital that the ~Singer() destructor be called to free that memory. Now consider the following code:

Employee \* pe = new Singer; // legal because Employee is base for Singer

...

delete pe; // ~Employee() or ~Singer()?

If the default static binding applies, the delete statement invokes the ~Employee() destructor. This frees memory pointed to by the Employee components of the Singer object but not memory pointed to by the new class members. However, if the destructors are virtual, **the same code invokes the ~Singer() destructor**, which frees memory pointed to by the Singer component, and then calls the ~Employee() destructor to free memory pointed to by the Employee component.

Note that this implies that even if a base class doesn’t require the services of an explicit destructor, you shouldn’t rely on the default constructor. Instead, you should provide a virtual destructor, even if it has nothing to do:

virtual ~BaseClass() { }

By the way, it’s not an error for a class to have a virtual destructor even if it is not intended to be a base class; it’s just a matter of efficiency.

**Friends can’t be virtual functions because friends are not class members**, and only members can be virtual functions. If this poses a problem for a design, you might be able to sidestep it by having the friend function use virtual member functions internally.

**No Redefinition**

If a derived class fails to redefine a function (virtual or not), the class will use the base class version of the function. If a derived class is part of a long chain of derivations, it will use the most recently defined version of the function. The exception is if the base versions are hidden, as described next.

class Dwelling

{

public:

virtual void showperks**(int a)** const;

};

class Hovel : public Dwelling

{

public:

virtual void showperks**()** const;

};

This causes a problem. You might get a compiler warning similar to the following:

Warning: Hovel::showperks(void) hides Dwelling::showperks(int)

Or perhaps you won’t get a warning. Either way, the code has the following implications:

Hovel trump;

trump.showperks(); // valid

trump.showperks(5); // **invalid**

The new definition defines a showperks() function that takes no arguments. Rather than resulting in two overloaded versions of the function, **this redefinition hides the base class version that takes an int argument**. In short, **redefining inherited methods is not a variation of overloading**. If you redefine a function in a derived class, it doesn’t just override the base class declaration with the same function signature. Instead, it hides all base class methods of the same name, regardless of the argument signatures.

This fact of life leads to a couple rules of thumb. **First, if you redefine an inherited method, you need to make sure you match the original prototype exactly**. One relatively new exception to this rule is that a return type that is a reference or pointer to a **base class** can be replaced by a reference or pointer to the **derived class**.

class Dwelling

{

public:

// a base method

virtual **Dwelling &** build(int n);

};

class Hovel : public Dwelling

{

public:

// a derived method with a covariant return type

virtual **Hovel &** build(int n); // same function signature

};

Note that this exception applies only to return values, not to arguments.

Second, if the base class declaration is overloaded, you need to redefine all the baseclass versions in the derived class:

class Dwelling

{

public:

// three overloaded showperks()

virtual void showperks(int a) const;

virtual void showperks(double x) const;

virtual void showperks() const;

...

};

class Hovel : public Dwelling

{

public:

// three redefined showperks()

virtual void showperks(int a) const;

virtual void showperks(double x) const;

virtual void showperks() const;

...

};

If you redefine just one version, the other two become hidden and cannot be used by objects of the derived class. Note that if no change is needed, the redefinition can simply call the base-class version:

void Hovel::showperks() const {Dwelling::showperks();}

* **Access Control: protected**

The **protected** keyword **is like private** in that the outside world can access class members in a protected section only by using public class members. **The difference between private and protected** comes into play only within **classes derived** from the base class. **Members of a derived class can access protected members of a base class directly**, **but they cannot directly access private members of the base class**. So members in the protected category behave like private members as far as the outside world is concerned but behave like public members as far as derived classes are concerned.

Protected使得derived class容易错误的操作base class的protected member; you should prefer private to protected access control for class data members, and you should use base-class methods to provide derived classes access to base-class data.

However, **protected access control can be quite useful for member functions**, giving derived classes access to internal functions that are not available publicly.

* **Abstract Base Classes**

Sometimes applying the is-a rule is not as simple as it might appear. Suppose, for example, you are developing a graphics program that is supposed to represent, among other things, circles and ellipses. A circle is a special case of an ellipse: It’s an ellipse whose long axis is the same as its short axis. Therefore, all circles are ellipses, and it is tempting to derive a Circle class from an Ellipse class.

class Ellipse

{

private:

double x; // x-coordinate of the ellipse's center

double y; // y-coordinate of the ellipse's center

double a; // semimajor axis

double b; // semiminor axis

double angle; // orientation angle in degrees

...

public:

...

void Move(int nx, ny) { x = nx; y = ny; }

virtual double Area() const { return 3.14159 \* a \* b; }

virtual void Rotate(double nang) { angle += nang; }

virtual void Scale(double sa, double sb) { a \*= sa; b \*= sb; }

...

};

Although a circle is an ellipse, this derivation is awkward. For example, a circle needs only a single value, its radius, to describe its size and shape instead of having a semimajor axis (a) and semiminor axis (b). The Circle constructors can take care of that by assigning the same value to both the a and b members, but then you have redundant representation of the same information. The angle parameter and the Rotate() method don’t really make sense for a circle, and the Scale() method, as it stands, can change a circle to a non-circle by scaling the two axes differently.

**On the whole, it seems simpler to define a Circle class without using inheritance**:

class Circle // no inheritance

{

private:

double x; // x-coordinate of the circle's center

double y; // y-coordinate of the circle's center

double r; // radius

...

public:

...

void Move(int nx, ny) { x = nx; y = ny; }

double Area() const { return 3.14159 \* r \* r; }

void Scale(double sr) { r \*= sr; }

...

};

Now the class has only the members it needs. Yet this solution also seems weak. **The Circle and Ellipse classes have a lot in common, but defining them separately ignores that fact**.

There is another solution: You can abstract from the Ellipse and Circle classes what they have in common and place those features in an ABC. Next, you derive both the Circle and Ellipse classes from the ABC. Then, for example, **you can use an array of base class pointers to manage a mixture of Ellipse and Circle objects**, that is, you can use a polymorphic approach.

In this case, what the two classes have in common are the coordinates of the center of the shape; a Move() method, which is the same for both; and an Area() method, which works differently for the two classes. Indeed, the Area() method *can’t even be implemented for the ABC* because it doesn’t have the necessary data members **(因为圆与椭圆的共性里没有半径或者长短轴这个参数).** C++ has a way to provide an unimplemented function by using a pure virtual function. A pure virtual function has = 0 at the end of its declaration, as shown for the Area() method:

class BaseEllipse // abstract base class

{

private:

double x; // x-coordinate of center

double y; // y-coordinate of center

...

public:

BaseEllipse(double x0 = 0, double y0 = 0) : x(x0),y(y0) {}

virtual ~BaseEllipse() {}

void Move(int nx, ny) { x = nx; y = ny; }

virtual double Area() const **= 0**; // a *pure virtual function*

...

}

When a class declaration contains a pure virtual function, you **can’t create an object** of that class. The idea is that **classes with pure virtual functions exist solely to serve as base classes**. For a class to be a genuine ABC, it has to have at least one pure virtual function. It is the = 0 in the prototype that makes a virtual function a pure virtual function. In the case of the Area() method, the function has no definition, but C++ allows even a pure virtual function to have a definition.

You could make the prototype virtual first:

void Move(int nx, ny) = 0;

This makes the base class abstract. And then you could still provide a definition in the implementation file:

void BaseEllipse::Move(int nx, ny) { x = nx; y = ny; }

In short, the = 0 in the prototype indicates that the class is an abstract base class and that the class doesn’t necessarily have to define the function.

Now you can derive the Ellipse class and Circle class from the BaseEllipse class, adding the members needed to complete each class. One point to note is that the Circle class always represents circles, whereas the Ellipse class represents ellipses that can also be circles. However, an Ellipse class circle can be rescaled to a non-circle, whereas a Circle class circle must remain a circle.

**Applying the ABC Concept**

Let’s apply the concept to representing the Brass and BrassPlus accounts, starting with an ABC called *AcctABC*.

This class should contain all methods and data members that are common to both the *Brass* and the *BrassPlus* classes. The *methods that are to work differently* for the BrassPlus class than they do for the Brass class should *be declared as virtual functions*. *At least one virtual function* should be a pure virtual function in order to make the *AcctABC* class abstract.

Listing below is a header file that declares the *AcctABC* class (an ABC) and the *Brass* and *BrassPlus* classes (both concrete classes). To facilitate derived class access to base class data, AcctABC provides some protected methods. Recall that protected methods are methods that derived-class methods can call but that are not part of the public interface for derived-class objects.

// **acctabc.h** -- bank account classes

#ifndef ACCTABC\_H\_

#define ACCTABC\_H\_

#include <iostream>

#include <string>

**// Abstract Base Class**

class **AcctABC**

{

private:

std::string fullName;

long acctNum;

double balance;

**protected:**

**struct** Formatting

{

std::ios\_base::fmtflags flag;

std::streamsize pr;

};

const std::string & FullName() const {return fullName;}

long AcctNum() const {return acctNum;}

Formatting SetFormat() const;

void Restore(Formatting & f) const;

public:

**AcctABC**(const std::string & s = "Nullbody", long an = -1, double bal = 0.0);

void Deposit(double amt) ;

**virtual** void Withdraw(double amt) **= 0**; // pure virtual function

double Balance() const {return balance;};

**virtual** void ViewAcct() const **= 0**; // pure virtual function

**virtual** ~AcctABC() {}

};

**// Brass Account Class**

**class Brass :public AcctABC**

{

public:

**Brass**(const std::string & s = "Nullbody", long an = -1, double bal = 0.0) : **AcctABC(s, an, bal)** { }

**virtual** void Withdraw(double amt);

**virtual** void ViewAcct() const;

**virtual ~Brass() {}**

};

**//Brass Plus Account Class**

**class BrassPlus : public AcctABC**

{

private:

double maxLoan;

double rate;

double owesBank;

public:

BrassPlus(const std::string & s = "Nullbody", long an = -1, double bal = 0.0, double ml = 500, double r = 0.10);

BrassPlus(const Brass & ba, double ml = 500, double r = 0.1);

**virtual** void ViewAcct()const;

**virtual** void Withdraw(double amt);

void ResetMax(double m) { maxLoan = m; }

void ResetRate(double r) { rate = r; };

void ResetOwes() { owesBank = 0; }

};

#endif

四大基本功能:存钱(Deposit()), 取钱(Withdraw()), 获取账户余额(Balance()), 显示账户所有信息(ViewAcct());

因为取钱和显示账户所有信息对于两种账户会不一样, 所以在ABC和两个derived class里都定义了, 并且在ABC里设置成了pure virtual function; 另外两个函数对两种账户而言都是一样的, 所以只在ABC里定义;

// **acctabc.cpp** -- bank account class methods

#include <iostream>

#include "acctabc.h"

using std::cout;

using std::ios\_base;

using std::endl;

using std::string;

**// Abstract Base Class**

AcctABC::AcctABC(const string & s, long an, double bal)

{

fullName = s;

acctNum = an;

balance = bal;

}

void **AcctABC::Deposit**(double amt)

{

if (amt < 0)

cout << "Negative deposit not allowed; " << "deposit is cancelled.\n";

else

balance += amt;

}

void **AcctABC::Withdraw**(double amt)

{

balance -= amt;

}

// protected methods for formatting

AcctABC::Formatting AcctABC::SetFormat() const

{

// set up ###.## format

Formatting f;

f.flag = cout.setf(ios\_base::fixed, ios\_base::floatfield);

f.pr = cout.precision(2);

return f;

}

void AcctABC::Restore(Formatting & f) const

{

cout.setf(f.flag, ios\_base::floatfield);

cout.precision(f.pr);

}

**// Brass methods**

void Brass::Withdraw(double amt)

{

if (amt < 0)

cout << "Withdrawal amount must be positive; " << "withdrawal canceled.\n";

else if (amt <= Balance())

AcctABC::Withdraw(amt);

else

cout << "Withdrawal amount of $" << amt << " exceeds your balance.\n"

<< "Withdrawal canceled.\n";

}

void Brass::ViewAcct() const

{

Formatting f = SetFormat();

cout << "Brass Client: " << FullName() << endl;

cout << "Account Number: " << AcctNum() << endl;

cout << "Balance: $" << Balance() << endl;

Restore(f);

}

**// BrassPlus Methods**

BrassPlus::BrassPlus(const string & s, long an, double bal,double ml, double r)

: AcctABC(s, an, bal)

{

maxLoan = ml;

owesBank = 0.0;

rate = r;

}

BrassPlus::BrassPlus(const Brass & ba, double ml, double r)

: AcctABC(ba) // uses implicit copy constructor

{

maxLoan = ml;

owesBank = 0.0;

rate = r;

}

void BrassPlus::ViewAcct() const

{

Formatting f = SetFormat();

cout << "BrassPlus Client: " << FullName() << endl;

cout << "Account Number: " << AcctNum() << endl;

cout << "Balance: $" << Balance() << endl;

cout << "Maximum loan: $" << maxLoan << endl;

cout << "Owed to bank: $" << owesBank << endl;

cout.precision(3);

cout << "Loan Rate: " << 100 \* rate << "%\n";

Restore(f);

}

void BrassPlus::Withdraw(double amt)

{

Formatting f = SetFormat();

double bal = Balance();

if (amt <= bal)

**AcctABC::Withdraw(amt);**

else if ( amt <= bal + maxLoan - owesBank)

{

double advance = amt - bal;

owesBank += advance \* (1.0 + rate);

cout << "Bank advance: $" << advance << endl;

cout << "Finance charge: $" << advance \* rate << endl;

Deposit(advance);

**AcctABC::Withdraw(amt);**

}

else

cout << "Credit limit exceeded. Transaction cancelled.\n";

Restore(f);

}

A problem with the older version was that the original setFormat() and restore() were standalone functions, so those function names would conflict with client-defined functions of the same name. **There are several ways to solve that problem.** One is to declare both functions static, making them private to the brass.cpp file or to its successor, acctabc.cpp. A second is to place both functions and the struct Formatting definition into a namespace. But one of the topics for this example is protected access, so **this example places the structure definition and the functions in the protected part of the class definition**. This makes them available to the base class and the derived class while hiding them from the outside world.

上面的新implementation和之前的旧的主程序是完全兼容的, 只需要把Brass \* p\_clients[CLIENTS]改成AcctABC \* p\_clients[CLIENTS];

* **Inheritance and Dynamic Memory Allocation**

How does inheritance interact with dynamic memory allocation (the use of new and delete)? For example, if a base class uses dynamic memory allocation and redefines assignment and a copy constructor, how does that affect the implementation of the derived class? The answer depends on the nature of the derived class. If the derived class does not itself use dynamic memory allocation, you needn’t take any special steps. If the derived class does also use dynamic memory allocation, then there are a couple new tricks to learn.

**Case 1: Derived Class Doesn’t Use new**

Suppose you begin with the following base class that uses dynamic memory allocation:

// Base Class Using DMA

class **baseDMA**

{

private:

**char \* label;**

int rating;

public:

baseDMA(const char \* l = "null", int r = 0);

baseDMA(const baseDMA & rs);

virtual ~baseDMA();

baseDMA & operator=(const baseDMA & rs);

...

};

The declaration contains the special methods that are required when constructors use new: a destructor, a copy constructor, and an overloaded assignment operator. Now suppose you derive a lackDMA class from baseDMA and that lackDMA does not use new or have other unusual design features that require special treatment:

// derived class without DMA

class lacksDMA :public baseDMA

{

private:

**char color[40];**

public:

...

};

Do you now have to define an explicit destructor, copy constructor, and assignment operator for the lackDMA class? The answer is no.

**First, consider the need for a destructor**. If you don’t define one, the compiler defines a default destructor that does nothing. Actually, the default destructor for a derived class always does something; it calls the base-class destructor after executing its own code. Because the lackDMA members, we assume, don’t require any special action, the default destructor is fine.

**Next, consider the copy constructor**. Copying a class member or an inherited class component is done using the copy constructor for that class. Thus, the default copy constructor for the lacksDMA class uses the explicit baseDMA copy constructor to copy the baseDMA portion of a lacksDMA object. So the default copy constructor is fine for the new lacksDMA member.

**Essentially the same situation holds for assignment**. The default assignment operator for a class automatically uses the **base-class assignment operator for the base-class component**. So it, too, is fine.

**Case 2: Derived Class Does Use new**

Suppose that the derived class uses new:

// derived class with DMA

class **hasDMA** :public **baseDMA**

{

private:

**char \* style;** // use new in constructors

public:

...

};

In this case, of course, you do have to define an explicit destructor, copy constructor, and assignment operator for the derived class. Let’s consider these methods in turn.

A derived class destructor automatically calls the base-class destructor, so its own responsibility is to clean up after what the derived-class constructors do. Thus, the hasDMA destructor has to free the memory managed by the style pointer and can rely on the baseDMA destructor to free the memory managed by the label pointer:

baseDMA::**~baseDMA**() // takes care of baseDMA stuff

{

delete [] **label**;

}

hasDMA::**~hasDMA**() // takes care of hasDMA stuff

{

delete [] **style**;

}

Next, consider copy constructors. Example of **copy constructor** of baseDMA could be:

baseDMA::**baseDMA(const baseDMA & rs)**

{

label = new char[std::strlen(rs.label) + 1];

std::strcpy(label, rs.label);

rating = rs.rating;

}

The hasDMA copy constructor only has access to hasDMA data, so **it must invoke the baseDMA copy constructor to handle the baseDMA share of the data**:

hasDMA::hasDMA(const hasDMA & hs)**: baseDMA(hs)**

{

style = new char[std::strlen(hs.style) + 1];

std::strcpy(style, hs.style);

}

The point to note is that the member initializer list **passes a hasDMA reference to a baseDMA constructor**. There is no baseDMA constructor with a type hasDMA reference parameter, but none is needed. That’s because the baseDMA copy constructor has a baseDMA reference parameter, and a base class reference can refer to a derived type. Thus, **the baseDMA copy constructor uses the baseDMA portion of the hasDMA argument to construct the baseDMA portion of the new object**.

Next, consider assignment operators. The baseDMA assignment operator follows the usual pattern:

baseDMA & **baseDMA**::operator=(const baseDMA & rs)

{

if (this == &rs)

return \*this;

delete [] label;

label = new char[std::strlen(rs.label) + 1];

std::strcpy(label, rs.label);

rating = rs.rating;

return \*this;

}

Because hasDMA also uses dynamic memory allocation, it, too, needs an explicit assignment operator. Being a hasDMA method, it only has direct access to hasDMA data. Nonetheless, an explicit assignment operator for a derived class also has to take care of assignment for the inherited base class baseDMA object.也就是说, **copy** **constructor和destructor不需要take care of the inherited base class, 但assignment operator需要**.

hasDMA & hasDMA::operator=(const hasDMA & hs)

{

if (this == &hs)

return \*this;

**baseDMA::operator=(hs);** // copy base portion

delete [] style; // prepare for new style

style = new char[std::strlen(hs.style) + 1];

std::strcpy(style, hs.style);

return \*this;

}

In summary, when both the base class and the derived class use dynamic memory allocation, the derived-class destructor, copy constructor, and assignment operator all must use their base-class counterparts to handle the base-class component. This common requirement is accomplished three different ways. For a *destructor, it is done automatically*. For a *constructor, it is accomplished by invoking the base-class copy constructor in the member initialization list*, or else the default constructor is invoked automatically. For the *assignment operator, it is accomplished by using the scope-resolution operator in an explicit call of the base-class assignment operator*.

**An Inheritance Example with Dynamic Memory Allocation and Friends**

// **dma.h** -- inheritance and dynamic memory allocation

#ifndef DMA\_H\_

#define DMA\_H\_

#include <iostream>

// Base Class Using DMA

class **baseDMA**

{

private:

char \* label;

int rating;

public:

baseDMA(const char \* l = "null", int r = 0);

**baseDMA**(const baseDMA & rs);

**virtual** ~baseDMA();

**baseDMA &** operator=(const baseDMA & rs);

friend std::ostream & operator<<(std::ostream & os, **const baseDMA & rs**);

};

**// derived class without DMA**

// no destructor needed, uses implicit copy constructor, uses implicit assignment operator

**class lacksDMA : public baseDMA**

{

private:

enum { COL\_LEN = 40};

char color[COL\_LEN];

public:

lacksDMA(const char \* c = "blank", const char \* l = "null", int r = 0);

lacksDMA(const char \* c, **const baseDMA & rs**);

friend std::ostream & operator<<(std::ostream & os, **const lacksDMA & rs**);

};

// derived class with DMA

**class hasDMA :public baseDMA**

{

private:

char \* style;

public:

hasDMA(const char \* s = "none", const char \* l = "null", int r = 0);

hasDMA(const char \* s, const baseDMA & rs);

hasDMA(const hasDMA & hs);

~hasDMA();

hasDMA & operator=(const hasDMA & rs);

friend std::ostream & operator<<(std::ostream & os, **const hasDMA & rs**);

};

#endif

Code above adds a friend function that illustrates how derived classes can access friends to a base class.

// **dma.cpp** --dma class methods

#include "dma.h"

#include <cstring>

// baseDMA methods

baseDMA::baseDMA(const char \* l, int r)

{

label = new char[std::strlen(l) + 1];

std::strcpy(label, l);

rating = r;

}

baseDMA::baseDMA(**const baseDMA & rs**) // copy constructor of base class

{

label = new char[std::strlen(rs.label) + 1];

std::strcpy(label, rs.label);

rating = rs.rating;

}

baseDMA::~baseDMA()

{

delete [] label;

}

baseDMA & baseDMA::operator=(const baseDMA & rs)

{

if (this == &rs)

return \*this;

delete [] label;

label = new char[std::strlen(rs.label) + 1];

std::strcpy(label, rs.label);

rating = rs.rating;

return \*this;

}

std::ostream & operator<<(std::ostream & os, const baseDMA & rs)

{

os << "Label: " << rs.label << std::endl;

os << "Rating: " << rs.rating << std::endl;

return os;

}

// **lacksDMA methods**

lacksDMA::lacksDMA(const char \* c, const char \* l, int r) **: baseDMA(l, r)**

{

std::strncpy(color, c, 39);

color[39] = ‘\0';

}

lacksDMA::lacksDMA(const char \* c, const baseDMA & rs) **: baseDMA(rs)**

{

std::strncpy(color, c, COL\_LEN - 1);

color[COL\_LEN - 1] = ‘\0';

}

std::ostream & operator<<(std::ostream & os, **const lacksDMA & ls**)

{

os << **(const baseDMA &)** ls; **// 这里用的baseDMA里的 <<**

os << "Color: " << ls.color << std::endl;

return os;

}

**// hasDMA methods**

hasDMA::hasDMA(const char \* s, const char \* l, int r) : **baseDMA(l, r)**

{

style = new char[std::strlen(s) + 1];

std::strcpy(style, s);

}

hasDMA::hasDMA(const char \* s, const baseDMA & rs) : baseDMA(rs)

{

style = new char[std::strlen(s) + 1];

std::strcpy(style, s);

}

hasDMA::hasDMA(const hasDMA & hs) : **baseDMA(hs)** // invoke base class copy constructor

{

style = new char[std::strlen(hs.style) + 1];

std::strcpy(style, hs.style);

}

hasDMA::~hasDMA()

{

delete [] style;

}

hasDMA & hasDMA::operator=(const hasDMA & hs)

{

if (this == &hs)

return \*this;

baseDMA::operator=(hs); // copy base portion

delete [] style; // prepare for new style

style = new char[std::strlen(hs.style) + 1];

std::strcpy(style, hs.style);

return \*this;

}

std::ostream & operator<<(std::ostream & os, **const hasDMA & hs**)

{

os << **(const baseDMA &)** hs; **// 这里用的baseDMA里的 <<**

os << "Style: " << hs.style << std::endl;

return os;

}

这里针对<<的处理值得解释.

Consider, for example, the following friend to the hasDMA class:

friend std::ostream & operator<<(std::ostream & os, const hasDMA & rs);

Being a friend to the hasDMA class gives this function access to the style member. **But there’s a problem**: This function is not a friend to the *baseDMA* class, so how can it access the *label* and *rating* members? The solution is to use the operator<<() function that is a friend to the baseDMA class. **The next problem is that because friends are not member functions, you can’t use the scope-resolution operator to indicate which function to use**. The solution to this problem is to use a type cast so that prototype matching will select the correct function. Thus, the code type casts the type *const hasDMA &* parameter to a type *const baseDMA &* argument:

std::ostream & operator<<(std::ostream & os, const hasDMA & hs)

{

// type cast to match operator<<(ostream & , const baseDMA &)

os << **(const baseDMA &)** hs; **// type cast hs to baseDMA & type so that << will use the baseDMA**

**// version**

os << "Style: " << hs.style << endl;

return os;

}

书上这里给了个测试的主程序.

* **Class Design Review**

There are some guidelines that often apply, and this is as good a time as any to go over them by reviewing and amplifying earlier discussions.

**Member Functions That the Compiler Generates for You**

Compiler automatically generates certain public member functions, termed special member functions. The fact that it does so suggests that these special member functions are particularly important.

1. **Default Constructors**

A default constructor is one that has no arguments, or else one for which all the arguments have default arguments. If you don’t define any constructors, the compiler defines a default constructor for you. Its existence allows you to create objects. For example, suppose Star is a class. You need a default constructor to use the following:

Star rigel; // create an object without explicit initialization

Star pleiades[6]; // create an array of objects

Also if you write a derived-class constructor without explicitly invoking a base-class constructor in the member initializer list, the compiler uses the base class default constructor to construct the base class portion of the new object. If there is no base-class default constructor, you get a compile-time error in this situation.

If you define a constructor of any kind, the compiler does not define a default constructor for you. In that case, it’s up to you to provide a default constructor if one is needed.

1. **Copy Constructors**

A copy constructor for a class is a constructor that takes an object of the class type as its argument. Typically, the declared parameter is a constant reference to the class type.

**A class copy constructor is used in the following situations:**

* When a new object is initialized to an object of the same class;
* When an object is passed to a function by value;
* When a function returns an object by value;
* When the compiler generates a temporary object;

1. **Assignment Operators**

A default assignment operator handles assigning one object to another object of the same class. **Don’t confuse assignment with initialization**. If a statement creates a new object, it’s using initialization, and if a statement alters the value of an existing object, it’s assignment:

Star sirius;

Star alpha = sirius; // initialization (one notation)

Star dogstar;

dogstar = sirius; // assignment

Default assignment uses memberwise assignment. If a member is itself a class object, then default memberwise assignment uses the assignment operator defined for that particular class. If you need to define a copy constructor explicitly, you also need, for the same reasons, to define the assignment operator explicitly.

The compiler doesn’t generate assignment operators for assigning one type to another. Suppose you want to be able to assign a string to a Star object. One approach is to define such an operator explicitly:

Star & Star::operator=(const char \*) {...}

A second approach is to rely on a conversion function (see “Conversion Considerations” in the next section) to convert a string to a Star object and use the Star-to-Star assignment function. The first approach runs more quickly but requires more code. The conversion function approach can lead to compiler-befuddling situations.

**Other Class Method Considerations**

**Constructors** are different from other class methods in that they create new objects, whereas other methods are invoked by existing objects. This is one reason constructors aren’t inherited. Inheritance means a derived object can use a base-class method, but, in the case of constructors, the object doesn’t exist until after the constructor has done its work.

**You need to remember to define an explicit destructor** that deletes any memory allocated by new in the class constructors and takes care of any other special bookkeeping that destroying a class object requires. If the class is to be used as a base class, you should provide a virtual destructor even if the class doesn’t require a destructor.

* **Conversion Considerations**

Any constructor that can be invoked with exactly one argument defines conversion from the argument type to the class type.

Star(const char \*); // converts char \* to Star

Star(const Spectral &, int members = 1); // converts Spectral to Star

Conversion constructors are used, for example, when a convertible type is passed to a function that is defined as taking a class argument.

Star north;

north = "polaris";

The second statement would invoke the Star::operator=(const Star &) function, using Star::Star(const char \*) to generate a Star object to be used as an argument for the assignment operator function. This assumes that you haven’t defined a (char \*)-to-Star assignment operator.

Using explicit in the prototype for a one-argument constructor disables implicit conversions but still allows explicit conversions:

class Star

{

public:

**explicit Star(const char \*);**

...

};

Star north;

north = "polaris"; **// not allowed**

north = Star("polaris"); // allowed

To convert from a class object to some other type, you define a conversion function. A conversion function is a class member function with **no arguments** or declared **return type that has the name of the type to be converted to**. Despite having no declared return type, the function should return the desired conversion value. Here are some examples:

Star::Star double() {...} // converts star to double

Star::Star const char \* () {...} // converts to const char

* **Passing an Object by Value Versus Passing a Reference**

In general, if you write a function using an object argument, **you should pass the object by reference rather than by value**. One reason for this is efficiency. Passing an object by value involves generating a temporary copy, which means calling the copy constructor and then later calling the destructor. Calling these functions takes time, and copying a large object can be quite a bit slower than passing a reference. If the function doesn’t modify the object, you should declare the argument as a const reference.

Another reason for passing objects by reference is that, in the case of inheritance using virtual functions, a function defined as accepting a base-class reference argument can also be used successfully with derived classes, as you saw earlier in this chapter.

* **Returning an Object Versus Returning a Reference**

If it isn’t necessary to return object value, you should use a reference instead.

First, the only coding difference between returning an object directly and returning a reference is in the function prototype and header:

**Star** nova1(const Star &); // returns a Star object

**Star &** nova2(const Star &); // returns a reference to a Star

Next, the reason you should return a reference rather than an object is that returning an object involves generating a temporary copy of the returned object. Returning a reference saves time and memory use. Returning an object directly is analogous to passing an object by value: Both processes generate temporary copies. Similarly, returning a reference is analogous to passing an object by reference: Both the calling and the called function operate on the same object. 但是, 当object是在函数内部产生的时候,就不应该返回reference, 因为函数结束时这个reference就invalid了.

If a function returns an object that was passed to it via a reference or pointer, you should return the object by reference. For example, the following code returns, by reference, either the object that invokes the function or else the object passed as an argument:

const Stock & Stock::topval(const Stock & s) const

{

if (s.total\_val > total\_val)

return s; // argument object

else

return \*this; // invoking object

}

* **Using const**

Star::Star(**const** char \* s) {...} // won't change the string to which s points

void Star::show() **const** {...} // won't change invoking object

Here, in the second case, const means const Star \* this, where this points to the invoking object.

**Public Inheritance Considerations**

You should be guided by the is-a relationship. You shouldn’t derive a Programmer class from a Brain class. If you want to represent the belief that a programmer has a brain, you should use a Brain class object as a member of the Programmer class.

Remember that one expression of the is-a relationship is that a base class pointer can point to a derived-class object and that a base-class reference can refer to a derived-class object without an explicit type cast. Also remember that the reverse is not true.

* **What’s Not Inherited**

**Constructors are not inherited**. That is, creating a derived object requires calling a derived-class constructor. However, derived-class constructors typically use the member initializer list syntax to call on base-class constructors to construct the base class portion of a derived object. If the derived-class constructor doesn’t explicitly call a base-class constructor by using the member initializer list syntax, it uses the base class’s default constructor. In an inheritance chain, each class can use a member initializer list to pass back information to its immediate base class.

**Destructors are not inherited either**. However, when an object is destroyed, the program first calls the derived destructor and then the base destructor. If there is a default base class destructor, the compiler generates a default derived class destructor. Generally speaking, if a class serves as a base class, its destructor should be virtual.

**Assignment operators are not inherited**. The reason is simple. **An inherited method has the same function signature in a derived class as it does in the base class(比如之前银行账户的例子里withdraw()函数在两个class里都是double的, 即取款金额. 不同的是两个函数对这个金额处理方式不同).** However, an assignment operator has a function signature that changes from class to class because it has a formal parameter that is the class type.

**Assignment Operator Considerations**

As you’ve seen several times, you need to provide an explicit assignment operator if class constructors use *new* to initialize pointers. Because C++ uses the base-class assignment operator for the base part of derived objects, you don’t need to redefine the assignment operator for a derived class unless it adds data members that require special care. For example, the baseDMA class defines assignment explicitly, but the derived lacksDMA class uses the implicit assignment operator generated for that class.

Suppose, however, that a derived class does use new, and you have to provide an explicit assignment operator. The operator must provide for every member of the class, not just the new members.

**What about assigning a derived-class object to a base-class object? Yes!**

(Note that this is not the same as initializing a base-class reference to a derived-class object.)

Brass blips; // base class

BrassPlus snips("Rafe Plosh", 91191,3993.19, 600.0, 0.12); // derived class

blips = snips; // assign derived object to base object

Which assignment operator is used? Remember that the assignment statement is translated into a method that is invoked by **the left-hand object**:

blips.operator=(snips);

Here the left-hand object is a *Brass* object, so it invokes the **Brass::operator=(const Brass &)** function. The is-a relationship allows the Brass reference to refer to a derived-class object, such as snips. The assignment operator only deals with base-class members, so the maxLoan member and other BrassPlus members of snips are ignored in the assignment. *In short, you can assign a derived object to a base object, and only the base-class members are involved*.

**What about the reverse? Can you assign a base-class object to a derived object? Maybe!**

Brass gp("Griff Hexbait", 21234, 1200); // base class

BrassPlus temp; // derived class

temp = gp; **// possible?**

Here the assignment statement would be translated as follows:

temp.operator=(gp);

The left-hand object is a *BrassPlus* object, so it invokes the **BrassPlus::operator=(const BrassPlus &)** function. **However, a derived-class reference cannot automatically refer to a base-class object, so this code won’t run unless there is also a conversion constructor**:

BrassPlus(const Brass &);

It could be, as is the case for the BrassPlus class, that the conversion constructor is a constructor with a base-class argument plus additional arguments, provided that the additional arguments have default values:

BrassPlus(const Brass & ba, double ml = 500, double r = 0.1);

If there is a conversion constructor, *the program uses this constructor to create a temporary BrassPlus object from gp*, which is then used as an argument to the assignment operator.

**Alternatively, you could define an assignment operator for assigning a base class to a derived class**:

BrassPlus & BrassPlus ::operator=(const Brass &) {...}

Here the types match the assignment statement exactly, and no type conversions are needed.

**Private Versus Protected Members**

Remember that protected members act like public members as far as a derived class is concerned, but they act like private members for the world at large. A derived class can access protected members of a base class directly, but it can access private members only via base-class member functions. Stroustrup, in his book The Design and Evolution of C++, indicates that it’s better to use private data members than protected data members but that protected methods are useful.

**Virtual Method Considerations**

When you design a *base class*, you have to decide whether to make class methods virtual. If you want a derived class to be able to redefine a method, you define the method as **virtual** **in the base class**. This enables late, or dynamic, binding. If you don’t want the method to be redefined, you don’t make it virtual.

Note that inappropriate code can circumvent dynamic binding.

void show(**const Brass & rba**)

{

rba.ViewAcct();

cout << endl;

}

void inadequate(**Brass ba**)

{

ba.ViewAcct();

cout << endl;

}

**BrassPlus** buzz("Buzz Parsec", 00001111, 4300);

**show(buzz);**

**inadequate(buzz);**

The **show() function** call results in the rba argument being a **reference to** the BrassPlus object buzz, so ***rba.ViewAcct()* is interpreted as the *BrassPlus* version**, as it should be.

But in the **inadequate() function**, which passes an object by value, *ba* is a Brass object constructed by the Brass(const Brass &) constructor. (Automatic upcasting allows the constructor argument to refer to a BrassPlus object.) Thus, **in inadequate(),** ***ba.ViewAcct()* is the *Brass* version**, so only the Brass component of buzz is displayed. 传递value of object as paprameter to function时, 涉及到conversion和copy constructor的问题.

**Friend Considerations**

Friends因为不是class member, 所以不能inherited. 前面的例子里, <<已经说明了如何在derived class里用base class里的friend函数; 里面有一个类型转换; You can also use the dynamic\_cast<> operator, discussed in Chapter 15,“Friends, Exceptions, and More,” for the type cast:

os << dynamic\_cast<const baseDMA &> (hs);

**Observations on Using Base-Class Methods**

1. A derived object automatically uses inherited base-class methods if the derived class hasn’t redefined the method.
2. A derived-class destructor automatically invokes the base-class constructor.
3. A derived-class constructor automatically invokes the base-class default constructor if you don’t specify another constructor in a member-initialization list.
4. A derived-class constructor explicitly invokes the base-class constructor specified in a member-initialization list.
5. Derived-class methods can use the scope-resolution operator to invoke public and protected base-class methods.
6. Friends to a derived class can type cast a derived-class reference or pointer to a base-class reference or pointer and then use that reference or pointer to invoke a friend to the base class.