

注意，全局函数 `verify` 的声明在 `Screen` 类定义之前是不可见的。然而，名字查找的第 3 步要考虑那些出现在成员定义之前的外围作用域声明，并找到全局函数 `verify` 的声明。

450

习题

习题 12.17 如果将 `Screen` 类中的类型别名放到类中的最后一行，将会发生什么？

习题 12.18 解释下述代码。指出每次使用 `Type` 或 `initVal` 时用到的是哪个名字定义。如果存在错误，说明如何改正。

```
typedef string Type;
Type initVal();

class Exercise {
public:
    // ...
    typedef double Type;
    Type setVal(Type);
    Type initVal();
private:
    int val;
};

Type Exercise::setVal(Type parm) {
    val = parm + initVal();
}
```

成员函数 `setVal` 的定义有错。进行必要的修改以便类 `Exercise` 使用全局的类型别名 `Type` 和全局函数 `initVal`。

12.4 构造函数

构造函数（2.3.3 节）是特殊的成员函数，只要创建类类型的新对象，都要执行构造函数。构造函数的作用是保证每个对象的数据成员具有合适的初始值。7.7.3 节展示了如何定义构造函数：

```
class Sales_item {
public:
    // operations on Sales_item objects
    // default constructor needed to initialize members of built-in type
    Sales_item(): units_sold(0), revenue(0.0) { }
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

这个构造函数使用构造函数初始化列表来初始化 `units_sold` 和 `revenue` 成员。`isbn` 成员由 `string` 的默认构造函数隐式初始化为空串。

451

构造函数的名字与类的名字相同，并且不能指定返回类型。像其他任何函数一样，它们可以没有形参，也可以定义多个形参。

1. 构造函数可以被重载

可以为一个类声明的构造函数的数量没有限制，只要每个构造函数的形参表是唯一的。我们如何才能知道应该定义哪个或多少个构造函数？一般而言，不同的构造函数允许用户指定不同的方式来初始化数据成员。

例如，逻辑上可以通过提供两个额外的构造函数来扩展 `Sales_item` 类：一个允许用户提供 `isbn` 的初始值，另一个允许用户通过读取 `istream` 对象来初始化对象：

```
class Sales_item;
// other members as before
public:
    // added constructors to initialize from a string or an istream
    Sales_item(const std::string&);
    Sales_item(std::istream&);
    Sales_item();
};
```

2. 实参决定使用哪个构造函数

我们的类现在定义了几个构造函数。在定义新对象时，可以使用这些构造函数中的任意一个：

```
// uses the default constructor:
// isbn is the empty string; units_sold and revenue are 0
Sales_item empty;
// specifies an explicit isbn; units_sold and revenue are 0
Sales_item Primer_3rd_Ed("0-201-82470-1");
// reads values from the standard input into isbn, units_sold, and revenue
Sales_item Primer_4th_ed(cin);
```

用于初始化一个对象的实参类型决定使用哪个构造函数。在 `empty` 的定义中，没有初始化式，所以运行默认构造函数。接受一个 `string` 实参的构造函数用于初始化 `Primer_3rd_ed`；接受一个 `istream` 引用的构造函数初始化 `Primer_4th_ed`。

3. 构造函数自动执行

只要创建该类型的一个对象，编译器就运行一个构造函数：

```
// constructor that takes a string used to create and initialize variable
Sales_item Primer_2nd_ed("0-201-54848-8");
// default constructor used to initialize unnamed object on the heap
Sales_item *p = new Sales_item();
```

452

第一种情况下，运行接受一个 `string` 实参的构造函数，来初始化变量 `Primer_2nd_ed`。第二种情况下，动态分配一个新的 `Sales_item` 对象。假定分配成功，则通过运行默认构造函数初始化该对象。

4. 用于 `const` 对象的构造函数

构造函数不能声明为 `const` (7.7.1 节)：

```
class Sales_item {
public:
    Sales_item() const;    // error
};
```

`const` 构造函数是不必要的。创建类类型的 `const` 对象时，运行一个普通构造函数来初始化该

const 对象。构造函数的工作是初始化对象。不管对象是否为 const，都用一个构造函数来初始化该对象。

习题

习题 12.19 提供一个或多个构造函数，允许该类的用户不指定数据成员的初始值或指定所有数据成员的初始值：

```
class NoName {
public:
    // constructor(s) go here ...
private:
    std::string *pstring;
    int          ival;
    double       dval;
};
```

解释如何确定需要多少个构造函数以及它们应该接受什么样的形参。

习题 12.20 从下述抽象中选择一个（或一个自己定义的抽象），确定类中需要什么数据，并提供适当的构造函数集。解释你的决定：

- | | | |
|-------------|------------|--------------|
| (a) Book | (b) Date | (c) Employee |
| (d) Vehicle | (e) Object | (f) Tree |

12.4.1 构造函数初始化式

与任何其他函数一样，构造函数具有名字、形参表和函数体。与其他函数不同的是，构造函数也可以包含一个构造函数初始化列表：

453

```
// recommended way to write constructors using a constructor initializer
Sales_item::Sales_item(const string &book):
    isbn(book), units_sold(0), revenue(0.0) { }
```

构造函数初始化列表以一个冒号开始，接着是一个以逗号分隔的数据成员列表，每个数据成员后面跟一个放在圆括号中的初始化式。这个构造函数将 isbn 成员初始化为 book 形参的值，将 units_sold 和 revenue 初始化为 0。与任意的成员函数一样，构造函数可以定义在类的内部或外部。构造函数初始化式只在构造函数的定义中而不是声明中指定。



构造函数初始化列表是许多相当有经验的 C++ 程序员都没有掌握的一个特性。

构造函数初始化列表难以理解的一个原因在于，省略初始化列表并在构造函数的函数体内对数据成员赋值是合法的。例如，可以将接受一个 string 的 Sales_item 构造函数编写为：

```
// legal but sloppier way to write the constructor:
// no constructor initializer
Sales_item::Sales_item(const string &book)
{
```

```

    isbn = book;
    units_sold = 0;
    revenue = 0.0;
}

```

这个构造函数给类 `Sales_item` 的成员赋值，但没有进行显式初始化。不管是否有显式的初始化式，在执行构造函数之前，要初始化 `isbn` 成员。这个构造函数隐式使用默认的 `string` 构造函数来初始化 `isbn`。执行构造函数的函数体时，`isbn` 成员已经有值了。该值被构造函数函数体中的赋值所覆盖。

从概念上讲，可以认为构造函数分两个阶段执行：(1) 初始化阶段；(2) 普通的计算阶段。计算阶段由构造函数函数体中的所有语句组成。



不管成员是否在构造函数初始化列表中显式初始化，类类型的数据成员总是在初始化阶段初始化。初始化发生在计算阶段开始之前。

在构造函数初始化列表中没有显式提及的每个成员，使用与初始化变量 (2.3.4 节) 相同的规则来进行初始化。运行该类型的默认构造函数，来初始化类类型的数据成员。内置或复合类型的成员的初始值依赖于对象的作用域：在局部作用域中这些成员不被初始化，而在全局作用域中它们被初始化为 0。

454

在本节中编写的两个 `Sales_item` 构造函数版本具有同样的效果：无论是在构造函数初始化列表中初始化成员，还是在构造函数函数体中对它们赋值，最终结果是相同的。构造函数执行结束后，三个数据成员保存同样的值。不同之处在于，使用构造函数初始化列表的版本初始化数据成员，没有定义初始化列表的构造函数版本在构造函数函数体中对数据成员赋值。这个区别的重要性取决于数据成员的类型。

1. 有时需要构造函数初始化列表

如果没有为类成员提供初始化式，则编译器会隐式地使用成员类型的默认构造函数。如果那个类没有默认构造函数，则编译器尝试使用默认构造函数将会失败。在这种情况下，为了初始化数据成员，必须提供初始化式。



有些成员必须在构造函数初始化列表中进行初始化。对于这样的成员，在构造函数函数体中对它们赋值不起作用。没有默认构造函数的类类型的成员，以及 `const` 或引用类型的成员，不管是哪种类型，都必须在构造函数初始化列表中进行初始化。

因为内置类型的成员不进行隐式初始化，所以对这些成员是进行初始化还是赋值似乎都无关紧要。除了两个例外，对非类类型的数据成员进行赋值或使用初始化式在结果和性能上都是等价的。

例如，下面的构造函数是错误的：

```

class ConstRef {
public:
    ConstRef(int ii);
private:
    int i;
}

```

```

    const int ci;
    int &ri;
};
// no explicit constructor initializer: error ri is uninitialized
ConstRef::ConstRef(int ii)
{
    // assignments:
    i = ii;    // ok
    ci = ii;   // error: cannot assign to a const
    ri = i;    // assigns to ri which was not bound to an object
}

```

记住，可以初始化 `const` 对象或引用类型的对象，但不能对它们赋值。在开始执行构造函数的函数体之前，要完成初始化。初始化 `const` 或引用类型数据成员的唯一机会是在构造函数初始化列表中。编写该构造函数的正确方式为

```

// ok: explicitly initialize reference and const members
ConstRef::ConstRef(int ii): i(ii), ci(i), ri(ii) {}

```

455

建议：使用构造函数初始化列表

在许多类中，初始化和赋值严格来讲都是低效率的：数据成员可能已经被直接初始化了，还要对它进行初始化和赋值。比效率问题更重要的是，某些数据成员必须要初始化，这是一个事实。



必须对任何 `const` 或引用类型成员以及没有默认构造函数的类类型的任何成员使用初始化式。

当类成员需要使用初始化列表时，通过常规地使用构造函数初始化列表，就可以避免发生编译时错误。

2. 成员初始化的次序

每个成员在构造函数初始化列表中只能指定一次，这不会令人惊讶。毕竟，给一个成员两个初始值意味着什么？也许更令人惊讶的是，构造函数初始化列表仅指定用于初始化成员的值，并不指定这些初始化执行的次序。成员被初始化的次序就是定义成员的次序。第一个成员首先被初始化，然后是第二个，依次类推。



初始化的次序常常无关紧要。然而，如果一个成员是根据其他成员而初始化，则成员初始化的次序是至关重要的。

考虑下面的类：

```

class X {
    int i;
    int j;
public:
    // run-time error: i is initialized before j
    X(int val): j(val), i(j) {}
};

```

在这种情况下，构造函数初始化列表看起来似乎是用 `val` 初始化 `j`，然后再用 `j` 来初始化 `i`。然

而, `i` 首先被初始化。这个初始化列表的效果是用尚未初始化的 `j` 值来初始化 `i`!

如果数据成员在构造函数初始化列表中的列出次序与成员被声明的次序不同, 那么有的编译器非常友好, 会给出一个警告。



最佳
实践

按照与成员声明一致的次序编写构造函数初始化列表是个好主意。此外, 尽可能避免使用成员来初始化其他成员。

一般情况下, 通过(重复)使用构造函数的形参而不是使用对象的数据成员, 可以避免由初始化式的执行次序而引起的任何问题。例如, 下面这样为 `x` 编写构造函数可能更好:

```
x(int val): i(val), j(val) { }
```

在这个版本中, `i` 和 `j` 初始化的次序就是无关紧要的。

3. 初始化式可以是任意表达式

一个初始化式可以是任意复杂的表达式。例如, 可以给 `Sales_item` 类一个新的构造函数, 该构造函数接受一个 `string` 表示 `isbn`, 一个 `unsigned` 表示售出书的数目, 一个 `double` 表示每本书的售出价格:

```
Sales_item(const std::string &book, int cnt, double price):
    isbn(book), units_sold(cnt), revenue(cnt * price) { }
```

`revenue` 的初始化式使用表示价格和售出数目的形参来计算对象的 `revenue` 成员。

4. 类类型的数据成员的初始化式

初始化类类型的成员时, 要指定实参并传递给成员类型的一个构造函数。可以使用该类型的任意构造函数。例如, `Sales_item` 类可以使用任意一个 `string` 构造函数来初始化 `isbn` (9.6.1 节)。也可以用 ISBN 取值的极限值来表示 `isbn` 的默认值, 而不是用空字符串。可以将 `isbn` 初始化为由 10 个 9 构成的串:

```
// alternative definition for Sales_item default constructor
Sales_item(): isbn(10, '9'), units_sold(0), revenue(0.0) { }
```

这个初始化式使用 `string` 构造函数, 接受一个计数值和一个字符, 并生成一个 `string`, 来保存重复指定次数的字符。

习题

习题 12.21 使用构造函数初始化列表编写类的默认构造函数, 该类包含如下成员: 一个 `const string`, 一个 `int`, 一个 `double*` 和一个 `ifstream&`。初始化 `string` 来保存类的名字。

习题 12.22 下面的初始化式有错误。找出并改正错误。

```
struct X {
    X (int i, int j): base(i), rem(base % j) { }
    int rem, base;
};
```

习题 12.23 假定有个命名为 `NoDefault` 的类, 该类有一个接受一个 `int` 的构造函数, 但没有默认构造函数。定义有一个 `NoDefault` 类型成员类 `C`。为类 `C` 定义默认构造函数。

12.4.2 默认实参与构造函数

再来看看默认构造函数和接受一个 string 的构造函数的定义：

```
Sales_item(const std::string &book):
    isbn(book), units_sold(0), revenue(0.0) { }
Sales_item(): units_sold(0), revenue(0.0) { }
```

这两个构造函数几乎是相同的：唯一的区别在于，接受一个 string 形参的构造函数使用该形参来初始化 isbn。默认构造函数（隐式地）使用 string 的默认构造函数来初始化 isbn。

可以通过为 string 初始化式提供一个默认实参将这些构造函数组合起来：

```
class Sales_item {
public:
    // default argument for book is the empty string
    Sales_item(const std::string &book = ""):
        isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is);
    // as before
};
```

在这里，我们只定义了两个构造函数，其中一个为其形参提供一个默认实参。对于下面的任一定义，将执行为其 string 形参接受默认实参的那个构造函数：

```
Sales_item empty;
Sales_item Primer_3rd_Ed("0-201-82470-1");
```

在 empty 的情况下，使用默认实参，而 Primer_3rd_ed 提供了一个显式实参。

类的两个版本提供同一接口：给定一个 string 或不给定初始化式，它们都将一个 Sales_item 初始化为相同的值。



最佳
实践

我们更喜欢使用默认实参，因为它减少代码重复。

习题

习题 12.24 上面的 Sales_item 定义了两个构造函数，其中之一有一个默认实参对应其单个 string 形参。使用该 Sales_item 版本，确定用哪个构造函数来初始化下述的每个变量，并列出每个对象中数据成员的值：

```
Sales_item first_item(cin);

int main() {
    Sales_item next;
    Sales_item last("9-999-99999-9");
}
```

习题 12.25 逻辑上讲，我们可能希望将 cin 作为默认实参提供给接受一个 istream&形参的构造函数。编写使用 cin 作为默认实参的构造函数声明。

习题 12.26 对于分别接受一个 string 和接受一个 istream&的构造函数，具有默认实参都是合法的吗？如果不是，为什么？

12.4.3 默认构造函数

只要定义一个对象时没有提供初始化式，就使用默认构造函数。为所有形参提供默认实参的构造函数也定义了默认构造函数。

1. 合成的默认构造函数

458

一个类哪怕只定义了一个构造函数，编译器也不会再生成默认构造函数。这条规则的根据是，如果一个类在某种情况下需要控制对象初始化，则该类很可能在所有情况下都需要控制。



只有当一个类没有定义构造函数时，编译器才会自动生成一个默认构造函数。

合成的默认构造函数 (synthesized default constructor) 使用与变量初始化相同的规则来初始化成员。具有类类型的成员通过运行各自的默认构造函数来进行初始化。内置和复合类型的成员，如指针和数组，只对定义在全局作用域中的对象才初始化。当对象定义在局部作用域中时，内置或复合类型的成员不进行初始化。



最佳实践 如果类包含内置或复合类型的成员，则该类不应该依赖于合成的默认构造函数。它应该定义自己的构造函数来初始化这些成员。

此外，每个构造函数应该为每个内置或复合类型的成员提供初始化式。没有初始化内置或复合类型成员的构造函数，将使那些成员处于未定义的状态。除了作为赋值的目标之外，以任何方式使用一个未定义的成员都是错误的。如果每个构造函数将每个成员设置为明确的已知状态，则成员函数可以区分空对象和具有实际值的对象。

2. 类通常应定义一个默认构造函数

459

在某些情况下，默认构造函数是由编译器隐式应用的。如果类没有默认构造函数，则该类就不能用在这些环境中。为了例示需要默认构造函数的情况，假定有一个 NoDefault 类，它没有定义自己的默认构造函数，却有一个接受一个 string 实参的构造函数。因为该类定义了一个构造函数，因此编译器将不合成默认构造函数。NoDefault 没有默认构造函数，意味着：

(1) 具有 NoDefault 成员的所有类的每个构造函数，必须通过传递一个初始的 string 值给 NoDefault 构造函数来显式地初始化 NoDefault 成员。

(2) 编译器将不会为具有 NoDefault 类型成员类合成默认构造函数。如果这样的类希望提供默认构造函数，就必须显式地定义，并且默认构造函数必须显式地初始化其 NoDefault 成员。

(3) NoDefault 类型不能用作动态分配数组的元素类型。

(4) NoDefault 类型的静态分配数组必须为每个元素提供一个显式的初始化式。

(5) 如果有一个保存 NoDefault 对象的容器，例如 vector，就不能使用接受容器大小而没有同时提供一个元素初始化式的构造函数。



最佳实践 实际上，如果定义了其他构造函数，则提供一个默认构造函数几乎总是对的。通常，在默认构造函数中给成员提供的初始值应该指出该对象是“空”的。

3. 使用默认构造函数



初级 C++ 程序员常犯的一个错误是，采用以下方式声明一个用默认构造函数初始化的对象：

```
// oops! declares a function, not an object
Sales_item myobj();
```

编译 myobj 的声明没有问题。然而，当我们试图使用 myobj 时

```
Sales_item myobj(); // ok: but defines a function, not an object
if (myobj.same_isbn(Primer_3rd_ed)) // error: myobj is a function
```

编译器会指出不能将成员访问符号用于一个函数！问题在于 myobj 的定义被编译器解释为一个函数的声明，该函数不接受参数并返回一个 Sales_item 类型的对象——与我们的意图大相径庭！使用默认构造函数定义一个对象的正确方式是去掉最后的空括号：

```
// ok: defines a class object ...
Sales_item myobj;
```

另一方面，下面这段代码也是正确的：

```
// ok: create an unnamed, empty Sales_item and use to initialize myobj
Sales_item myobj = Sales_item();
```

在这里，我们创建并初始化一个 Sales_item 对象，然后用它来按值初始化 myobj。编译器通过运行 Sales_item 的默认构造函数来按值初始化一个 Sales_item。

460

习题

习题 12.27 下面的陈述中哪个是不正确的（如果有的话）？为什么？

- (a) 类必须提供至少一个构造函数。
- (b) 默认构造函数的形参列表中没有形参。
- (c) 如果一个类没有有意义的默认值，则该类不应该提供默认构造函数。
- (d) 如果一个类没有定义默认构造函数，则编译器会自动生成一个，同时将每个数据成员初始化为相关类型的默认值。

12.4.4 隐式类类型转换

在 5.12 节介绍过，C++ 语言定义了内置类型之间的几个自动转换。也可以定义如何将其他类型的对象隐式转换为我们的类类型，或将我们的类类型的对象隐式转换为其他类型。在 14.9 节将会看到如何定义从类类型到其他类型的转换。为了定义到类类型的隐式转换，需要定义合适的构造函数。



可以用单个实参来调用的构造函数定义了从形参类型到该类类型的一个隐式转换。

让我们再看看定义了两个构造函数的 Sales_item 版本：

```
class Sales_item {
public:
    // default argument for book is the empty string
    Sales_item(const std::string &book = ""):
        isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is);
    // as before
};
```

这里的每个构造函数都定义了一个隐式转换。因此，在期待一个 Sales_item 类型对象的地方，可以使用一个 string 或一个 istream：

```
string null_book = "9-999-99999-9";
// ok: builds a Sales_item with 0 units_sold and revenue from
// and isbn equal to null_book
item.same_isbn(null_book);
```

461

这段程序使用一个 string 类型对象作为实参传给 Sales_item 的 same_isbn 函数。该函数期待一个 Sales_item 对象作为实参。编译器使用接受一个 string 的 Sales_item 构造函数从 null_book 生成一个新的 Sales_item 对象。新生成的（临时的）Sales_item 被传递给 same_isbn。

这个行为是否我们想要的，依赖于我们认为用户将如何使用这个转换。在这种情况下，它可能是一个好主意。book 中的 string 可能代表一个不存在的 ISBN，对 same_isbn 的调用可以检测 item 中的 Sales_item 是否表示一个空的 Sales_item。另一方面，用户也许在 null_book 上错误地调用了 same_isbn。

更成问题的是从 istream 到 Sales_item 的转换：

```
// ok: uses the Sales_item istream constructor to build an object
// to pass to same_isbn
item.same_isbn(cin);
```

这段代码将 cin 隐式转换为 Sales_item。这个转换执行接受一个 istream 的 Sales_item 构造函数。该构造函数通过读标准输入来创建一个（临时的）Sales_item 对象。然后该对象被传递给 same_isbn。

这个 Sales_item 对象是一个临时对象（7.3.2 节）。一旦 same_isbn 结束，就不能再访问它。实际上，我们构造了一个在测试完成后被丢弃的对象。这个行为几乎肯定是一个错误。

1. 抑制由构造函数定义的隐式转换

可以通过将构造函数声明为 explicit，来防止在需要隐式转换的上下文中使用构造函数：

```
class Sales_item {
public:
    // default argument for book is the empty string
    explicit Sales_item(const std::string &book = ""):
        isbn(book), units_sold(0), revenue(0.0) { }
    explicit Sales_item(std::istream &is);
    // as before
};
```

explicit 关键字只能用于类内部的构造函数声明上。在类的定义体外部所做的定义上不再重复它：

```
// error: explicit allowed only on constructor declaration in class header
explicit Sales_item::Sales_item(istream& is)
{
    is >> *this; // uses Sales_item input operator to read the members
}
```

现在，两个构造函数都不能用于隐式地创建对象。前两个使用都不能编译：

```
item.same_isbn(null_book); // error: string constructor is explicit
item.same_isbn(cin);       // error: istream constructor is explicit
```

462



当构造函数被声明为 `explicit` 时，编译器将不使用它作为转换操作符。

2. 为转换而显式地使用构造函数

只要显式地按下面这样做，就可以用显式的构造函数来生成转换：

```
string null_book = "9-999-99999-9";
// ok: builds a Sales_item with 0 units_sold and revenue from
//      and isbn equal to null_book
item.same_isbn(Sales_item(null_book));
```

在这段代码中，从 `null_book` 创建一个 `Sales_item`。尽管构造函数为显式的，但这个用法是允许的。显式使用构造函数只是中止了隐式地使用构造函数。任何构造函数都可以用来显式地创建临时对象。



通常，除非有明显的理由想要定义隐式转换，否则，单形参构造函数应该为 `explicit`。将构造函数设置为 `explicit` 可以避免错误，并且当转换有用时，用户可以显式地构造对象。

习题

习题 12.28 解释一下接受一个 `string` 的 `Sales_item` 构造函数是否应该为 `explicit`。将构造函数设置为 `explicit` 的好处是什么？缺点是什么？

习题 12.29 解释在下面的定义中所发生的操作。

```
string null_isbn = "9-999-99999-9";
Sales_item null1(null_isbn);
Sales_item null("9-999-99999-9");
```

习题 12.30 编译如下代码：

```
f(const vector<int>&);
int main() {
    vector<int> v2;
    f(v2); // should be ok
    f(42); // should be an error
    return 0;
}
```

基于对 `f` 的第二个调用中出现的错误，我们可以对 `vector` 构造函数作出什么推断？如果该调用成功了，那么你能得出什么结论？

463

12.4.5 类成员的显式初始化

尽管大多数对象可以通过运行适当的构造函数进行初始化,但是直接初始化简单的非抽象类的数据成员仍是可能的。对于没有定义构造函数并且其全体数据成员均为 `public` 的类,可以采用与初始化数组元素相同的方式初始化其成员:

```
struct Data {
    int ival;
    char *ptr;
};
// val1.ival = 0; val1.ptr = 0
Data val1 = { 0, 0 };
// val2.ival = 1024;
// val2.ptr = "Anna Livia Plurabelle"
Data val2 = { 1024, "Anna Livia Plurabelle" };
```

根据数据成员的声明次序来使用初始化式。例如,因为 `ival` 在 `ptr` 之前声明,所以下面的用法是错误的:

```
// error: can't use "Anna Livia Plurabelle" to initialize the int ival
Data val2 = { "Anna Livia Plurabelle" , 1024 };
```

这种形式的初始化从 C 继承而来,支持与 C 程序兼容。显式初始化类类型对象的成员有三个重大的缺点。

- (1) 要求类的全体数据成员都是 `public`。
- (2) 将初始化每个对象的每个成员的负担放在程序员身上。这样的初始化是乏味且易于出错的,因为容易遗忘初始化式或提供不适当的初始化式。
- (3) 如果增加或删除一个成员,必须找到所有的初始化并正确更新。



最佳
实践

定义和使用构造函数几乎总是较好的。当我们为自己定义的类型提供一个默认构造函数时,允许编译器自动运行那个构造函数,以保证每个类对象在初次使用之前正确地初始化。

习题

习题 12.31 `pair` 的数据成员为 `public`,然而下面这段代码却不能编译,为什么?

464

```
pair<int, int> p2 = {0, 42}; // doesn't compile, why?
```

12.5 友元

在某些情况下,允许特定的非成员函数访问一个类的私有成员,同时仍然阻止一般的访问,这是很方便做到的。例如,被重载的操作符,如输入或输出操作符,经常需要访问类的私有数据成员。这些操作符不可能为类的成员,具体原因参见第 14 章。然而,尽管不是类的成员,它们仍是类的“接口的组成部分”。

友元 (friend) 机制允许一个类将对其非公有成员的访问权授予指定的函数或类。友元的声明以关键字 `friend` 开始。它只能出现在类定义的内部。友元声明可以出现在类中的任何地方：友元不是授予友元关系的那个类的成员，所以它们不受其声明出现部分的访问控制影响。

提示

通常，将友元声明成组地放在类定义的开始或结尾是个好主意。

1. 友元关系：一个例子

想像一下，除了 `Screen` 类之外，还有一个窗口管理器，管理给定显示器上的一组 `Screen`。窗口管理类在逻辑上可能需要访问由其管理的 `Screen` 对象的内部数据。假定 `Window_Mgr` 是该窗口管理类的名字，`Screen` 应该允许 `Window_Mgr` 像下面这样访问其成员：

```
class Screen {
    // Window_Mgr members can access private parts of class Screen
    friend class Window_Mgr;
    // ...rest of the Screen class
};
```

`Window_Mgr` 的成员可以直接引用 `Screen` 的私有成员。例如，`Window_Mgr` 可以有一个函数来重定位一个 `Screen`：

```
Window_Mgr&
Window_Mgr::relocate(Screen::index r, Screen::index c,
                    Screen& s)
{
    // ok to refer to height and width
    s.height += r;
    s.width += c;
    return *this;
}
```

缺少友元声明时，这段代码将会出错：将不允许使用形参 `s` 的 `height` 和 `width` 成员。因为 `Screen` 将友元关系授予 `Window_Mgr`，所以，`Window_Mgr` 中的函数都可以访问 `Screen` 的所有成员。

465

友元可以是普通的非成员函数，或前面定义的其他类的成员函数，或整个类。将一个类设为友元，友元类的所有成员函数都可以访问授予友元关系的那个类的非公有成员。

2. 使其他类的成员函数成为友元

如果不是将整个 `Window_Mgr` 类设为友元，`Screen` 就可以指定只允许 `relocate` 成员访问：

```
class Screen {
    // Window_Mgr must be defined before class Screen
    friend Window_Mgr&
        Window_Mgr::relocate(Window_Mgr::index,
                            Window_Mgr::index,
                            Screen&);
    // ...rest of the Screen class
};
```

当我们将成员函数声明为友元时，函数名必须用该函数所属的类名字加以限定。

3. 友元声明与作用域

为了正确地构造类，需要注意友元声明与友元定义之间的互相依赖。在前面的例子中，类

Window_Mgr 必须先定义。否则, Screen 类就不能将一个 Window_Mgr 函数指定为友元。然而,只有在定义类 Screen 之后,才能定义 relocate 函数——毕竟,它被设为友元是为了访问类 Screen 的成员。

更一般地讲,必须先定义包含成员函数的类,才能将成员函数设为友元。另一方面,不必预先声明类和非成员函数来将它们设为友元。



友元声明将已命名的类或非成员函数引入到外围作用域中。此外,友元函数可以在类的内部定义,该函数的作用域扩展到包围该类定义的作用域。

用友元引入的类名和函数(定义或声明),可以像预先声明的一样使用:

```
class X {
    friend class Y;
    friend void f() { /* ok to define friend function in the class body */ }
};
class Z {
    Y *ymem; // ok: declaration for class Y introduced by friend in X
    void g() { return ::f(); } // ok: declaration of f introduced by X
};
```

466

4. 重载函数与友元关系

类必须将重载函数集中每一个希望设为友元的函数都声明为友元:

```
// overloaded storeOn functions
extern std::ostream& storeOn(std::ostream &, Screen &);
extern BitMap& storeOn(BitMap &, Screen &);
class Screen {
    // ostream version of storeOn may access private parts of Screen objects
    friend std::ostream& storeOn(std::ostream &, Screen &);
    // ...
};
```

类 Screen 将接受一个 ostream& 的 storeOn 版本设为自己的友元。接受一个 BitMap& 的版本对 Screen 没有特殊访问权。

习题

习题 12.32 什么是友元函数? 什么是友元类?

习题 12.33 什么时候友元是有用的? 讨论使用友元的优缺点。

习题 12.34 定义一个增加两个 Sales_item 对象的非成员函数。

习题 12.35 定义一个非成员函数,读取一个 istream 并将读入的内容存储到一个 Sales_item 中。

12.6 static 类成员

对于特定类类型的全体对象而言,访问一个全局对象有时是必要的。也许,在程序的任意点需要统计已创建的特定类类型对象的数量;或者,全局对象可能是指向类的错误处理例程的一个

指针；或者，它是指向类类型对象的内存自由存储区的一个指针。

然而，全局对象会破坏封装：对象需要支持特定类抽象的实现。如果对象是全局的，一般的用户代码就可以修改这个值。类可以定义**类静态成员**，而不是定义一个可普遍访问的全局对象。

通常，非 static 数据成员存在于类类型的每个对象中。不像普通的数据成员，static 数据成员独立于该类的任意对象而存在；每个 static 数据成员是与类关联的对象，并不与该类的对象相关联。

正如类可以定义共享的 static 数据成员一样，类也可以定义 static 成员函数。static 成员函数没有 this 形参，它可以直接访问所属类的 static 成员，但不能直接使用非 static 成员。

467

1. 使用类的 static 成员的优点

使用 static 成员而不是全局对象有三个优点。

(1) static 成员的名字是在类的作用域中，因此可以避免与其他类的成员或全局对象名字冲突。

(2) 可以实施封装。static 成员可以是私有成员，而全局对象不可以。

(3) 通过阅读程序容易看出 static 成员是与特定类关联的。这种可见性可清晰地显示程序员的意图。

2. 定义 static 成员

在成员声明前加上关键字 static 将成员设为 static。static 成员遵循正常的公有/私有访问规则。

例如，考虑一个简单的表示银行账户的类。每个账户具有余额和拥有者，并且按月获得利息，但应用于每个账户的利率总是相同的。可以按下面这样编写这个类

```
class Account {
public:
    // interface functions here
    void applyint() { amount += amount * interestRate; }
    static double rate() { return interestRate; }
    static void rate(double); // sets a new rate
private:
    std::string owner;
    double amount;
    static double interestRate;
    static double initRate();
};
```

这个类的每个对象具有两个数据成员：owner 和 amount。对象没有与 static 数据成员对应的数据成员，但是，存在一个单独的 interestRate 对象，由 Account 类型的全体对象共享。

3. 使用类的 static 成员

可以通过作用域操作符从类直接调用 static 成员，或者通过对象、引用或指向该类类型对象的指针间接调用。

```
Account ac1;
Account *ac2 = &ac1;
// equivalent ways to call the static member rate function
double rate;
```

468

```

rate = ac1.rate();           // through an Account object or reference
rate = ac2->rate();          // through a pointer to an Account object
rate = Account::rate();      // directly from the class using the scope operator

```

像使用其他成员一样，类成员函数可以不用作用域操作符来引用类的 static 成员：

```

class Account {
public:
    // interface functions here
    void applyint() { amount += amount * interestRate; }
};

```

习题

习题 12.36 什么是 static 类成员？static 成员的优点是什么？它们与普通成员有什么不同？

习题 12.37 编写自己的 Account 类版本。

12.6.1 static 成员函数

Account 类有两个名为 rate 的 static 成员函数，其中一个定义在类的内部。当我们在类的外部定义 static 成员时，无须重复指定 static 保留字，该保留字只出现在类定义体内部的声明处：

```

void Account::rate(double newRate)
{
    interestRate = newRate;
}

```

static 函数没有 this 指针

static 成员是类的组成部分但不是任何对象的组成部分，因此，static 成员函数没有 this 指针。通过使用非 static 成员显式或隐式地引用 this 是一个编译时错误。

因为 static 成员不是任何对象的组成部分，所以 static 成员函数不能被声明为 const。毕竟，将成员函数声明为 const 就是承诺不会修改该函数所属的对象。最后，static 成员函数也不能被声明为虚函数。我们将在 15.2.4 节学习虚函数。

习题

习题 12.38 定义一个命名为 Foo 的类，具有单个 int 型数据成员。为该类定义一个构造函数，接受一个 int 值并用该值初始化数据成员。为该类定义一个函数，返回其数据成员的值。

习题 12.39 给定上题中定义的 Foo 类定义另一个 Bar 类。Bar 类具有两个 static 数据成员：一个为 int 型，另一个为 Foo 类型。

习题 12.40 使用上面两题中定义的类，给 Bar 类增加一对成员函数：第一个成员命名为 FooVal，返回 Bar 类的 Foo 类型 static 成员的值；第二个成员命名为 callsFooVal，保存 xval 被调用的次数。

12.6.2 static 数据成员

static 数据成员可以声明为任意类型，可以是常量、引用、数组、类类型，等等。

static 数据成员必须在类定义体的外部定义（正好一次）。不像普通数据成员，static 成员不是通过类构造函数进行初始化，而是应该在定义时进行初始化。

469



提示 保证对象正好定义一次的最好办法，就是将 static 数据成员的定义放在包含类的非内联成员函数定义的文件中。

定义 static 数据成员的方式与定义其他类成员和变量的方式相同：先指定类型名，接着是成员的完全限定名。

可以定义如下 interestRate:

```
// define and initialize static class member
double Account::interestRate = initRate();
```

这个语句定义名为 interestRate 的 static 对象，它是类 Account 的成员，为 double 型。像其他成员定义一样，一旦成员名出现，static 成员的定义就是在类作用域中。因此，我们可以没有限定地直接使用名为 initRate 的 static 成员函数，作为 interestRate¹ 的初始化式。注意，尽管 initRate 是私有的，我们仍然可以使用该函数来初始化 interestRate。像任意的其他成员定义一样，interestRate 的定义是在类的作用域中，因此可以访问该类的私有成员。



像使用任意的类成员一样，在类定义体外部引用类的 static 成员时，必须指定成员是在哪个类中定义的。然而，static 关键字只能用于类定义体内部的声明中，定义不能标示为 static。

1. 特殊的整型 const static 成员

一般而言，类的 static 成员，像普通数据成员一样，不能在类的定义体中初始化。相反，static 数据成员通常在定义时才初始化。

这个规则的一个例外是，只要初始化式是一个常量表达式，整型 const static 数据成员就可以在类的定义体中进行初始化：

470

```
class Account {
public:
    static double rate() { return interestRate; }
    static void rate(double); // sets a new rate
private:
    static const int period = 30; // interest posted every 30 days
    double daily_tbl[period]; // ok: period is constant expression
};
```

用常量值初始化的整型 const static 数据成员是一个常量表达式。同样地，它可以用在任何需要常量表达式的地方，例如指定数组成员 daily_tbl 的维。



const static 数据成员在类的定义体中初始化时，该数据成员仍必须在类的定义体之外进行定义。

1. 原文为 rate，应为 interestRate——译者注

在类内部提供初始化式时，成员的定义不必再指定初始值：

```
// definition of static member with no initializer;
// the initial value is specified inside the class definition
const int Account::period;
```

2. static 成员不是类对象的组成部分

普通成员都是给定类的每个对象的组成部分。static 成员独立于任何对象而存在，不是类类型对象的组成部分。因为 static 数据成员不是任何对象的组成部分，所以它们的使用方式对于非 static 数据成员而言是不合法的。

例如，static 数据成员的类型可以是该成员所属的类类型。非 static 成员被限定声明为其自身类对象的指针或引用：

```
class Bar {
public:
    // ...
private:
    static Bar mem1;    // ok
    Bar *mem2;          // ok
    Bar mem3;           // error
};
```

类似地，static 数据成员可用作默认实参：

```
class Screen {
public:
    // bkground refers to the static member
    // declared later in the class definition
    Screen& clear(char = bkground);
private:
    static const char bkground = '#';
};
```

471

非 static 数据成员不能用作默认实参，因为它的值不能独立于所属的对象而使用。使用非 static 数据成员作默认实参，将无法提供对象以获取该成员的值，因而是错误的。

习题

习题 12.41 利用 12.6.1 节的习题中编写的类 Foo 和 Bar，初始化 Foo 的 static 成员。将 int 成员初始化为 20，并将 Foo 成员初始化为 0。

习题 12.42 下面的 static 数据成员声明和定义中哪些是错误的（如果有的话）？解释为什么。

```
// example.h
class Example {
public:
    static double rate = 6.5;

    static const int vecSize = 20;
    static vector<double> vec(vecSize);
};
```

```
// example.C
#include "example.h"
double Example::rate;
vector<double> Example::vec;
```

472

小结

类是 C++ 中最基本的特征，允许定义新的类型以适应应用程序的需要，同时使程序更短且更易于修改。

数据抽象是指定义数据和函数成员的能力，而封装是指从常规访问中保护类成员的能力，它们都是类的基础。成员函数定义类的接口。通过将类的实现所用到的数据和函数设置为 `private` 来封装类。

类可以定义构造函数，它们是特殊的成员函数，控制如何初始化类的对象。可以重载构造函数。每个构造函数应初始化每个数据成员。初始化列表包含的是名-值对，其中的名是一个成员，而值则是该成员的初始值。

类可以将对其非 `public` 成员的访问权授予其他类或函数，并通过将其他的类或函数设为友元来授予其访问权。

类也可以定义 `mutable` 或 `static` 成员。`mutable` 成员永远都不能为 `const`；它的值可以在 `const` 成员函数中修改。`static` 成员可以是函数或数据，独立于类类型的对象而存在。

术语

abstract data type (抽象数据类型) 使用封装来隐藏其实现的数据结构，允许使用类型的程序员抽象地考虑该类型做什么，而不是具体地考虑类型如何表示。C++ 中的类可用来定义抽象数据类型。

access label (访问标号) `public` 或 `private` 标号，指定后面的成员可以被类的使用者访问或者只能被类的友元和成员访问。每个标号为在该标号到下一个标号之间声明的成员设置访问保护。标号可以在类中出现多次。

class (类) 是 C++ 中定义抽象数据类型的一种机制，可以有数据、函数或类型成员。一个类定义了新的类型和新的作用域。

class declaration (类声明) 类可以在定义之前声明。类声明用关键字 `class` (或 `struct`) 表示，后面加类名字和一个分号。已声明但没有定义的类型是一个不完整的类型。

class keyword (class 关键字) 用在 `class`

关键字定义的类中，初始的隐式访问标号是 `private`。

class scope (类作用域) 每个类定义一个作用域。类作用域比其他作用域复杂得多——在类的定义体内定义的成员函数可以使用出现在该定义之后的名字。

concrete class (具体类) 暴露其实现细节的类。

const member function (常量成员函数) 一种成员函数，不能改变对象的普通（即，既不是 `static` 也不是 `mutable`）数据成员。`const` 成员中的 `this` 指针指向 `const` 对象。成员函数是否可以被重载取决于该函数是否为 `const`。

constructor initializer list (构造函数初始化列表) 指定类的数据成员的初始值。在构造函数体执行前，用初始化列表中指定的值初始化成员。没有在初始化列表中初始化的类成员，使用它们的默认构造函数隐式初始化。

conversion constructor (转换构造函数) 可用单个实参调用的非 `explicit` 构造函数。隐式使用转换构造函数将实参的类型转换为类类型。

data abstraction (数据抽象) 注重类型接口的编程技术。数据抽象允许程序员忽略类型如何表示的细节,而只考虑该类型可以执行的操作。数据抽象是面向对象编程和泛型编程的基础。

default constructor (默认构造函数) 没有指定初始化式时使用的构造函数。

encapsulation (封装) 实现与接口的分离。封装隐藏了类型的实现细节。在 C++ 中,实施封装可以阻止普通用户访问类的 `private` 部分。

explicit constructor (显式构造函数) 可以用单个实参调用但不能用于执行隐式转换的构造函数。通过将关键字 `explicit` 放在构造函数的声明之前而将其设置为 `explicit`。

forward declaration (前向声明) 对尚未定义的名字的声明。大多用于引用出现在类定义之前的类声明。参见不完全类型。

friend (友元) 类授权访问其非 `public` 成员的机制。类和函数都可以被指定为友元。友元拥有与成员一样的访问权。

incomplete type (不完全类型) 已声明但未定义的类型。不能使用不完全类型来定义变量或类成员。定义指向不完全类型的引用或指针是合法的。

member function (成员函数) 类的函数成员。普通成员函数通过隐式的 `this` 指针绑定到类

类型的对象。`static` 成员函数不与对象绑定且没有 `this` 指针。成员函数可以被重载,只要该函数的版本可由形参的数目或类型来区别。

mutable data member (可变数据成员) 一种永远也不能为 `const` 对象的数据成员,即使作为 `const` 对象的成员,也不能为 `const` 对象。`mutable` 成员可以在 `const` 函数中改变。

name lookup (名字查找) 将名字的使用与其相应的声明相匹配的过程。

private member (私有成员) 在 `private` 访问标号之后定义的成员,只能被友元和其他的类成员访问。类所使用的数据成员和实用函数在不作为类型接口的组成部分时,通常声明为 `private`。

public member (公用成员) 在 `public` 访问标号之后定义的成员,可被类的任意使用者访问。一般而言,只有定义类接口的函数应定义在 `public` 部分。

static member (静态成员) 不是任意对象的组成部分、但由给定类的全体对象所共享的数据或函数成员。

struct keyword (struct 关键字) 用在 `struct` 关键字定义的类中,初始的隐式访问标号为 `public`。

synthesized default constructor (合成的默认构造函数) 编译器为没有定义任何构造函数的类创建(合成)的构造函数。这个构造函数通过运行该类的默认构造函数来初始化类类型的成员,内置类型的成员不进行初始化。

第 13 章

复制控制

目录

13.1 复制构造函数.....	406
13.2 赋值操作符.....	411
13.3 析构函数.....	412
13.4 消息处理示例.....	415
13.5 管理指针成员.....	419
小结.....	427
术语.....	427

每种类型，无论是内置类型还是类类型，都对该类型对象的一组（可能为空的）操作的含义进行了定义。比如，我们可以将两个 `int` 值相加，运行 `vector` 对象的 `size` 操作，等等。这些操作定义了用给定类型的对象可以完成什么任务。

每种类型还定义了创建该类型的对象时会发生什么——构造函数定义了该类类型对象的初始化。类型还能控制复制、赋值或撤销该类型的对象时会发生什么——类通过特殊的成员函数：复制构造函数、赋值操作符和析构函数来控制这些行为。本章将介绍这些操作。

当定义一个新类型的时候，需要显式或隐式地指定复制、赋值和撤销该类型的对象时会发生什么——这是通过定义特殊成员：复制构造函数、赋值操作符和析构函数来达到的。如果没有显式定义复制构造函数或赋值操作符，编译器（通常）会为我们定义。

复制构造函数（copy constructor）是一种特殊构造函数，具有单个形参，该形参（常用 `const` 修饰）是对该类类型的引用。当定义一个新对象并用一个同类型的对象对它进行初始化时，将显式使用复制构造函数。当将该类型的对象传递给函数或从函数返回该类型的对象时，将隐式使用复制构造函数。

析构函数（destructor）是构造函数的互补：当对象超出作用域或动态分配的对象被删除时，将自动应用析构函数。析构函数可用于释放对象时构造或在对象的生命期中所获取的资源。不管类是否定义了自己的析构函数，编译器都自动执行类中非 `static` 数据成员的析构函数。

在下一章我们将进一步学习操作符重载，本章中我们先介绍**赋值操作符**（assignment operator）。与构造函数一样，赋值操作符可以通过指定不同类型的右操作数而重载。右操作数为类类型的版本比较特殊：如果我们没有编写这种版本，编译器将为我们合成一个。

复制构造函数、赋值操作符和析构函数总称为**复制控制**（copy control）。编译器自动实现这些操作，但类也可以定义自己的版本。



复制控制是定义任意 C++ 类必不可少的部分。初学 C++ 的程序员常对必须定义在复制、赋值或撤销对象时发生什么感到困惑。因为如果我们没有显式定义这些操作，编译器将为我们定义它们（尽管它们也许不像我们期望的那样工作），这往往使初学者更加困惑。

通常，编译器合成的复制控制函数是非常精练的——它们只做必需的工作。但对某些类而言，依赖于默认定义会导致灾难。实现复制控制操作最困难的部分，往往在于识别何时需要覆盖默认版本。有一种特别常见的情况需要类定义自己的复制控制成员的：类具有指针成员。

13.1 复制构造函数

只有单个形参，而且该形参是对本类类型对象的引用（常用 `const` 修饰），这样的构造函数称为复制构造函数。与默认构造函数一样，复制构造函数可由编译器隐式调用。复制构造函数可用于：

- 根据另一个同类型的对象显式或隐式初始化一个对象。
- 复制一个对象，将它作为实参传给一个函数。
- 从函数返回时复制一个对象。
- 初始化顺序容器中的元素。
- 根据元素初始化式列表初始化数组元素。

1. 对象的定义形式

回忆一下，C++ 支持两种初始化形式（2.3.3 节）：直接初始化和复制初始化。复制初始化使用 `=` 符号，而直接初始化将初始化式放在圆括号中。

当用于类类型对象时，初始化的复制形式和直接形式有所不同：直接初始化直接调用与实参匹配的构造函数，复制初始化总是调用复制构造函数。复制初始化首先使用指定构造函数创建一个临时对象（7.3.2 节），然后用复制构造函数将那个临时对象复制到正在创建的对象：

```
string null_book = "9-999-99999-9";    // copy-initialization
string dots(10, '.');                  // direct-initialization
string empty_copy = string();           // copy-initialization
string empty_direct;                   // direct-initialization
```

对于类类型对象，只有指定单个实参或显式创建一个临时对象用于复制时，才使用复制初始化。

创建 `dots` 时，调用参数为一个数量和一个字符的 `string` 构造函数并直接初始化 `dots` 的成员。创建 `null_book` 时，编译器首先调用接受一个 C 风格字符串形参的 `string` 构造函数，创建一个临时对象，然后，编译器使用 `string` 复制构造函数将 `null_book` 初始化为那个临时对象的副本。

`empty_copy` 和 `empty_direct` 的初始化都调用默认构造函数。对前者初始化时，默认构造函数创建一个临时对象，然后复制构造函数用该对象初始化 `empty_copy`。对后者初始化时，直接运行 `empty_direct` 的默认构造函数。

支持初始化的复制形式主要是为了与 C 的用法兼容。当情况许可时，可以允许编译器跳过复制构造函数直接创建对象，但编译器没有义务这样做。

通常直接初始化和复制初始化仅在低级别优化上存在差异。然而，对于不支持复制的类型，或者使用非 `explicit` 构造函数（12.4.4 节）的时候，它们有本质区别：

```
ifstream file1("filename"); // ok: direct initialization
ifstream file2 = "filename"; // error: copy constructor is private
// This initialization is okay only if
// the Sales_item(const string&) constructor is not explicit
Sales_item item = string("9-999-99999-9");
```

`file1` 的初始化是正确的。`ifstream` 类定义了一个可用 C 风格字符串调用的构造函数，使用该构造函数初始化 `file1`。

看上去等效的 `file2` 初始化使用复制初始化，但该定义不正确。由于不能复制 IO 类型的对象（8.1 节），所以不能对那些类型的对象使用复制初始化。

`item` 的初始化是否正确，取决于正在使用哪个版本的 `Sales_item` 类。某些版本将参数为一个 `string` 的构造函数定义为 `explicit`。如果构造函数是显式的，则初始化失败；如果构造函数不是显式的，则初始化成功。

2. 形参与返回值

正如我们所知，当形参为非引用类型（7.2.1 节）的时候，将复制实参的值。类似地，以非引用类型作返回值时，将返回 `return` 语句中的值的副本（7.3.2 节）。

当形参或返回值为类类型时，由复制构造函数进行复制。例如，考虑 7.3.2 节的 `make_plural` 函数：

```
// copy constructor used to copy the return value;
// parameters are references, so they aren't copied
string make_plural(size_t, const string&, const string&);
```

这个函数隐式使用 `string` 复制构造函数返回给定单词的复数形式。形参是 `const` 引用，不能复制。

3. 初始化容器元素

复制构造函数可用于初始化顺序容器中的元素。例如，可以用表示容量的单个形参来初始化容器（3.3.1 节）。容器的这种构造方式使用了默认构造函数和复制构造函数：

```
// default string constructor and five string copy constructors invoked
vector<string> svec(5);
```

编译器首先使用 `string` 默认构造函数创建一个临时值来初始化 `svec`，然后使用复制构造函数将临时值复制到 `svec` 的每个元素。



提示 作为一般规则（9.1.1 节），除非你想使用容器元素的默认初始值，更有效的办法是，分配一个空容器并将已知元素的值加入容器。

4. 构造函数与数组元素

如果没有为类类型数组提供元素初始化式，则将用默认构造函数初始化每个元素。然而，如果使用常规的花括号括住的数组初始化列表（4.4.1 节）来提供显式元素初始化式，则使用复制初始化来初始化每个元素。根据指定值创建适当类型的元素，然后用复制构造函数将该值复制到相应元素：

```
Sales_item primer_eds[] = { string("0-201-16487-6"),
                             string("0-201-54848-8"),
                             string("0-201-82470-1"),
                             Sales_item()
                           };
```

如前三个元素的初始化式中所示可以直接指定一个值，用于调用元素类型的单实参构造函数。如果希望不指定实参或指定多个实参，就需要使用完整的构造函数语法，正如最后一个元素的初始化式那样。

习题

习题 13.1 什么是复制构造函数？何时使用它？

习题 13.2 下面第二个初始化不能编译。可以从 `vector` 的定义得出什么推断？

```
vector<int> v1(42); // ok: 42 elements, each 0
vector<int> v2 = 42; // error: what does this error tell us about vector?
```

习题 13.3 假定 `Point` 为类类型，该类类型有一个复制构造函数，指出下面程序段中每一个使用了复制构造函数的地方：

```
Point global;

Point foo_bar(Point arg)
{
    Point local = arg;
    Point *heap = new Point(global);
    *heap = local;
    Point pa[ 4 ] = { local, *heap };
    return *heap;
}
```

13.1.1 合成的复制构造函数

如果我们没有定义复制构造函数，编译器就会为我们合成一个。与合成的默认构造函数（12.4.2 节）不同，即使我们定义其他构造函数，也会合成复制构造函数。合成复制构造函数（synthesized copy constructor）的行为是，执行逐个成员初始化（memberwise initialize），将新对象初始化为原对象的副本。

所谓“逐个成员”，指的是编译器将现有对象的每个非 static 成员，依次复制到正创建的对象。只有一个例外，每个成员的类型决定了复制该成员的含义。合成复制构造函数直接复制内置类型成员的值，类类型成员使用该类的复制构造函数进行复制。数组成员的复制是个例外。虽然一般不能复制数组，但如果一个类具有数组成员，则合成复制构造函数将复制数组。复制数组时合成复制构造函数将复制数组的每一个元素。

479

逐个成员初始化最简单的概念模型是，将合成复制构造函数看作这样一个构造函数：其中每个数据成员在构造函数初始化列表中进行初始化。例如，对于我们的 Sales_item 类，它有三个数据成员：

```
class Sales_item {
    // other members and constructors as before
private:
    std::string isbn;
    int units_sold;
    double revenue;
};
```

合成复制构造函数如下所示：

```
Sales_item::Sales_item(const Sales_item &orig):
    isbn(orig.isbn),           // uses string copy constructor
    units_sold(orig.units_sold), // copies orig.units_sold
    revenue(orig.revenue)      // copy orig.revenue
{ }                             // empty body
```

13.1.2 定义自己的复制构造函数

复制构造函数就是接受单个类类型引用形参（通常用 const 修饰）的构造函数：

```
class Foo {
public:
    Foo();           // default constructor
    Foo(const Foo&); // copy constructor
    // ...
};
```

虽然也可以定义接受非 const 引用的复制构造函数，但形参通常是一个 const 引用。因为用于向函数传递对象和从函数返回对象，该构造函数一般不应设置为 explicit（12.4.4 节）。复制构造函数应将实参的成员复制到正在构造的对象。

对许多类而言，合成复制构造函数只完成必要的工作。只包含类类型成员或内置类型（但不是指针类型）成员的类，无须显式地定义复制构造函数，也可以复制。

然而，有些类必须对复制对象时发生的事情加以控制。这样的类经常有一个数据成员是指针，

480

或者有成员表示在构造函数中分配的其他资源。而另一些类在创建新对象时必须做一些特定工作。这两种情况下，都必须定义复制构造函数。

通常，定义复制构造函数最困难的部分在于认识到需要复制构造函数。只要能认识到需要复制构造函数，定义构造函数一般非常简单。复制构造函数的定义与其他构造函数一样：它与类同名，没有返回值，可以（而且应该）使用构造函数初始化列表初始化新创建对象的成员，可以在函数体中做任何其他必要工作。

后续章节中将给出一些需要定义复制构造函数的类的例子。13.4 节给出了一对类，它们需要显式复制构造函数，用于处理与简单消息处理应用程序相关的工作。具有指针成员的类在 13.5 节给出。

习题

习题 13.4 对于如下的类的简单定义，编写一个复制构造函数复制所有成员。复制 `pstring` 指向的对象而不是复制指针。

```
struct NoName {
    NoName(): pstring(new std::string), i(0), d(0) { }
private:
    std::string *pstring;
    int i;
    double d;
};
```

习题 13.5 哪个类定义可能需要一个复制构造函数？

- (a) 包含四个 `float` 成员的 `Point3w` 类。
- (b) `Matrix` 类，其中，实际矩阵在构造函数中动态分配，在析构函数中删除。
- (c) `Payroll` 类，在这个类中为每个对象提供唯一 ID。
- (d) `Word` 类，包含一个 `string` 和一个以行列位置对为元素的 `vector`。

习题 13.6 复制构造函数的形参并不限制为 `const`，但必须是一个引用。解释这个限制的基本原理，例如，解释为什么下面的定义不能工作。

```
Sales_item::Sales_item(const Sales_item rhs);
```

13.1.3 禁止复制

481

有些类需要完全禁止复制。例如，`iostream` 类就不允许复制（8.1 节）。如果想要禁止复制，似乎可以省略复制构造函数，然而，如果不定义复制构造函数，编译器将合成一个。



为了防止复制，类必须显式声明其复制构造函数为 `private`。

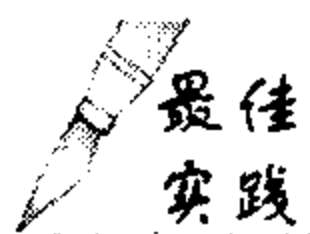
如果复制构造函数是私有的，将不允许用户代码复制该类类型的对象，编译器将拒绝任何进行复制的尝试。

然而，类的友元和成员仍可以进行复制。如果想要连友元和成员中的复制也禁止，就可以声明一个（`private`）复制构造函数但不对其定义。

声明而不定义成员函数是合法的，但是，使用未定义成员的任何尝试将导致链接失败。通过声明（但不定义）`private` 复制构造函数，可以禁止任何复制类类型对象的尝试：用户代码中的复制尝试将在编译时标记为错误，而成员函数和友元中的复制尝试将在链接时导致错误。

大多数类应定义复制构造函数和默认构造函数

不定义复制构造函数和/或默认构造函数，会严重局限类的使用。不允许复制的类对象只能作为引用传递给函数或从函数返回，它们也不能用作容器的元素。



最佳
实践

一般来说，最好显式或隐式定义默认构造函数和复制构造函数。只有不存在其他构造函数时才合成默认构造函数。如果定义了复制构造函数，也必须定义默认构造函数。

13.2 赋值操作符

与类要控制初始化对象的方式一样，类也定义了该类型对象赋值时会发生什么：

```
Sales_item trans, accum;
trans = accum;
```

与复制构造函数一样，如果类没有定义自己的赋值操作符，则编译器会合成一个。

1. 介绍重载赋值

在介绍合成赋值操作符之前，需要简单了解一下重载操作符（overloaded operator），我们将在第 14 章详细介绍。

重载操作符是一些函数，其名字为 `operator` 后跟着所定义的操作符的符号。因此，通过定义名为 `operator=` 的函数，我们可以对赋值进行定义。像任何其他函数一样，操作符函数有一个返回值和一个形参表。形参表必须具有与该操作符操作数数目相同的形参（如果操作符是一个成员，则包括隐式 `this` 形参）。赋值是二元运算，所以该操作符函数有两个形参：第一个形参对应着左操作数，第二个形参对应右操作数。

482

大多数操作符可以定义为成员函数或非成员函数。当操作符为成员函数时，它的第一个操作数隐式绑定到 `this` 指针。有些操作符（包括赋值操作符）必须是定义自己的类的成员。因为赋值必须是类的成员，所以 `this` 绑定到指向左操作数的指针。因此，赋值操作符接受单个形参，且该形参是同一类类型的对象。右操作数一般作为 `const` 引用传递。

赋值操作符的返回类型应该与内置类型赋值运算返回的类型相同（5.4.1 节）。内置类型的赋值运算返回对右操作数的引用，因此，赋值操作符也返回对同一类类型的引用。

例如，`Sales_item` 的赋值操作符可以声明为：

```
class Sales_item {
public:
    // other members as before
    // equivalent to the synthesized assignment operator
    Sales_item& operator=(const Sales_item &);
};
```

2. 合成赋值操作符

合成赋值操作符（synthesized assignment operator）与合成复制构造函数的操作类似。它会执

行**逐个成员赋值** (memberwise assignment): 右操作数对象的每个成员赋值给左操作数对象的对应成员。除数组之外, 每个成员用所属类型的常规方式进行赋值。对于数组, 给每个数组元素赋值。

例如, Sales_item 的合成赋值操作符可能如下所示:

```
// equivalent to the synthesized assignment operator
Sales_item&
Sales_item::operator=(const Sales_item &rhs)
{
    isbn = rhs.isbn;           // calls string::operator=
    units_sold = rhs.units_sold; // uses built-in int assignment
    revenue = rhs.revenue;      // uses built-in double assignment
    return *this;
}
```

合成赋值操作符根据成员类型使用适合的内置或类定义的赋值操作符, 依次给每个成员赋值, 该操作符返回*this, 它是对左操作数对象的引用。

3. 复制和赋值常一起使用

483

可以使用合成复制构造函数的类通常也可以使用合成赋值操作符。我们的 Sales_item 类无须定义复制构造函数或赋值操作符, 这些操作符的合成版本工作得很好。

然而, 类也可以定义自己的赋值操作符。一般而言, 如果类需要复制构造函数, 它也会需要赋值操作符。



实际上, 应将这两个操作符看作一个单元。如果需要其中一个, 我们几乎也肯定需要另一个。

我们将在 13.4 节和 13.5 节介绍类需要自定义赋值操作符的例子。

习题

习题 13.7 类何时需要定义赋值操作符?

习题 13.8 对于习题 13.5 中列出的每个类型, 指出类是否需要赋值操作符。

习题 13.9 习题 13.5 中包括 NoName 类的简单定义。确定这个类是否需要赋值操作符。如果需要, 实现它。

习题 13.10 定义一个 Employee 类, 包含雇员名字和一个唯一的雇员标识。为该类定义默认构造函数和参数为表示雇员名字的 string 的构造函数。如果该类需要复制构造函数或赋值操作符, 实现这些函数。

13.3 析构函数

构造函数的一个用途是自动获取资源。例如, 构造函数可以分配一个缓冲区或打开一个文件, 在构造函数中分配了资源之后, 需要一个对应操作自动回收或释放资源。析构函数就是这样的一个特殊函数, 它可以完成所需的资源回收, 作为类构造函数的补充。

1. 何时调用析构函数

撤销类对象时会自动调用析构函数：

```
// p points to default constructed object
Sales_item *p = new Sales_item;
{
    // new scope
    Sales_item item(*p); // copy constructor copies *p into item
    delete p;           // destructor called on object pointed to by p
}                       // exit local scope; destructor called on item
```

484

变量（如 item）在超出作用域时应该自动撤销。因此，当遇到右花括号时，将运行 item 的析构函数。

动态分配的对象只有在指向该对象的指针被删除时才撤销。如果没有删除指向动态对象的指针，则不会运行该对象的析构函数，对象就一直存在，从而导致内存泄漏，而且，对象内部使用的任何资源也不会释放。



当对象的引用或指针超出作用域时，不会运行析构函数。只有删除指向动态分配对象的指针或实际对象（而不是对象的引用）超出作用域时，才会运行析构函数。

撤销一个容器（不管是标准库容器还是内置数组）时，也会运行容器中的类类型元素的析构函数：

```
{
    Sales_item *p = new Sales_item[10]; // dynamically allocated
    vector<Sales_item> vec(p, p + 10); // local object
    // ...
    delete [] p; // array is freed; destructor run on each element
} // vec goes out of scope; destructor run on each element
```

容器中的元素总是按逆序撤销：首先撤销下标为 `size()-1` 的元素，然后是下标为 `size()-2` 的元素……直至最后撤销下标为 0 的元素。

2. 何时编写显式析构函数

许多类不需要显式析构函数，尤其是具有构造函数的类不一定需要定义自己的析构函数。仅在有些工作需要析构函数完成时，才需要析构函数。析构函数通常用于释放在构造函数或在对象生命期内获取的资源。



如果类需要析构函数，则它也需要赋值操作符和复制构造函数，这是一个有用的经验法则。这个规则常称为三法则（rule of three），指的是如果需要析构函数，则需要所有这三个复制控制成员。

析构函数并不仅限于用来释放资源。一般而言，析构函数可以执行任意操作，该操作是类设计者希望在该类对象的使用完毕之后执行的。

3. 合成析构函数

与复制构造函数或赋值操作符不同，编译器总是会为我们合成一个析构函数。合成析构函数按对象创建时的逆序撤销每个非 static 成员，因此，它按成员在类中声明次序的逆序撤销成员。

485 对于类类型的每个成员，合成析构函数调用该成员的析构函数来撤销对象。



撤销内置类型成员或复合类型的成员没什么影响。尤其是，合成析构函数并不删除指针成员所指向的对象。

4. 如何编写析构函数

`Sales_item` 类是类没有分配资源因此不需要自己的析构函数的一个例子。分配了资源的类一般需要定义析构函数以释放那些资源。析构函数是个成员函数，它的名字是在类名字之前加上一个代字号 (~)，它没有返回值，没有形参。因为不能指定任何形参，所以不能重载析构函数。虽然可以为一个类定义多个构造函数，但只能提供一个析构函数，应用于类的所有对象。

析构函数与复制构造函数或赋值操作符之间的一个重要区别是，即使我们编写了自己的析构函数，合成析构函数仍然运行。例如，可以为 `Sales_item` 类编写如下的空析构函数：

```
class Sales_item {
public:
    // empty; no work to do other than destroying the members,
    // which happens automatically
    ~Sales_item() { }
    // other members as before
};
```

撤销 `Sales_item` 类型的对象时，将运行这个什么也不做的析构函数，它执行完毕后，将运行合成析构函数以撤销类的成员。合成析构函数调用 `string` 析构函数来撤销 `string` 成员，`string` 析构函数释放了保存 `isbn` 的内存。`units_sold` 和 `revenue` 成员是内置类型，所以合成析构函数撤销它们不需要做什么。

习题

习题 13.11 什么是析构函数？合成析构函数有什么用？什么时候会合成析构函数？什么时候一个类必须定义自己的析构函数？

习题 13.12 确定在习题 13.4 中概略定义的 `NoName` 类是否需要析构函数，如果需要，实现它。

习题 13.13 确定在习题 13.10 中定义的 `Employee` 类是否需要析构函数，如果需要，实现它。

习题 13.14 理解复制控制成员和构造函数的一个良好方式是定义一个简单类，该类具有这些成员，每个成员打印自己的名字：

```
struct Exmpl {
    Exmpl() { std::cout << "Exmpl()" << std::endl; }
    Exmpl(const Exmpl&)
    { std::cout << "Exmpl(const Exmpl&)" << std::endl; }
    // ...
};
```

编写一个像 `Exmpl` 这样的类，给出复制控制成员和其他构造函数。然后写一个程序，用不同方式使用 `Exmpl` 类型的对象：作为非引用形参和引用形参传递，动态分配，放在容器中，等等。研究何时执行哪个构造函数和复制构造函数，可以帮助你融会贯通地理解这些概念。

习题 13.15 下面的代码段中发生了多少次析构函数的调用？

```

void fcn(const Sales_item *trans, Sales_item accum)
{
    Sales_item item1(*trans), item2(accum);
    if (!item1.same_isbn(item2)) return;
    if (item1.avg_price() <= 99) return;
    else if (item2.avg_price() <= 99) return;
    // ...
}

```

13.4 消息处理示例

有些类为了做一些工作需要复制进行控制, 为了给出这样的例子, 我们将概略定义两个类, 这两个类可用于邮件处理应用程序。Message 类和 Folder 类分别表示电子邮件 (或其他) 消息和消息所出现的目录, 一个给定消息可以出现在多个目录中。Message 上有 Save 和 Remove 操作, 用于在指定 Folder 中保存或删除该消息。

对每个 Message, 我们并不是在每个 Folder 中都存放一个副本, 而是使每个 Message 保存一个指针集 (set), set 中的指针指向该 Message 所在的 Folder。每个 Folder 也保存着一些指针, 指向它所包含的 Message。将要实现的数据结构如图 13-1 所示。

创建新的 Message 时, 将指定消息的内容但不指定 Folder。调用 save 将 Message 放入一个 Folder。

486

复制一个 Message 对象时, 将复制原始消息的内容和 Folder 指针集, 还必须给指向源 Message 的每个 Folder 增加一个指向该 Message 的指针。

将一个 Message 对象赋值给另一个, 类似于复制一个 Message: 赋值之后, 内容和 Folder 集将是相同的。首先从左边 Message 在赋值之前所处的 Folder 中删除该 Message。原来的 Message 去掉之后, 再将右边操作数的内容和 Folder 集复制到左边, 还必须在这个 Folder 集中的每个 Folder 中增加一个指向左边 Message 的指针。

撤销一个 Message 对象时, 必须更新指向该 Message 的每个 Folder。一旦去掉了 Message, 指向该 Message 的指针将失效, 所以必须从该 Message 的 Folder 指针集的每个 Folder 中删除这个指针。

487

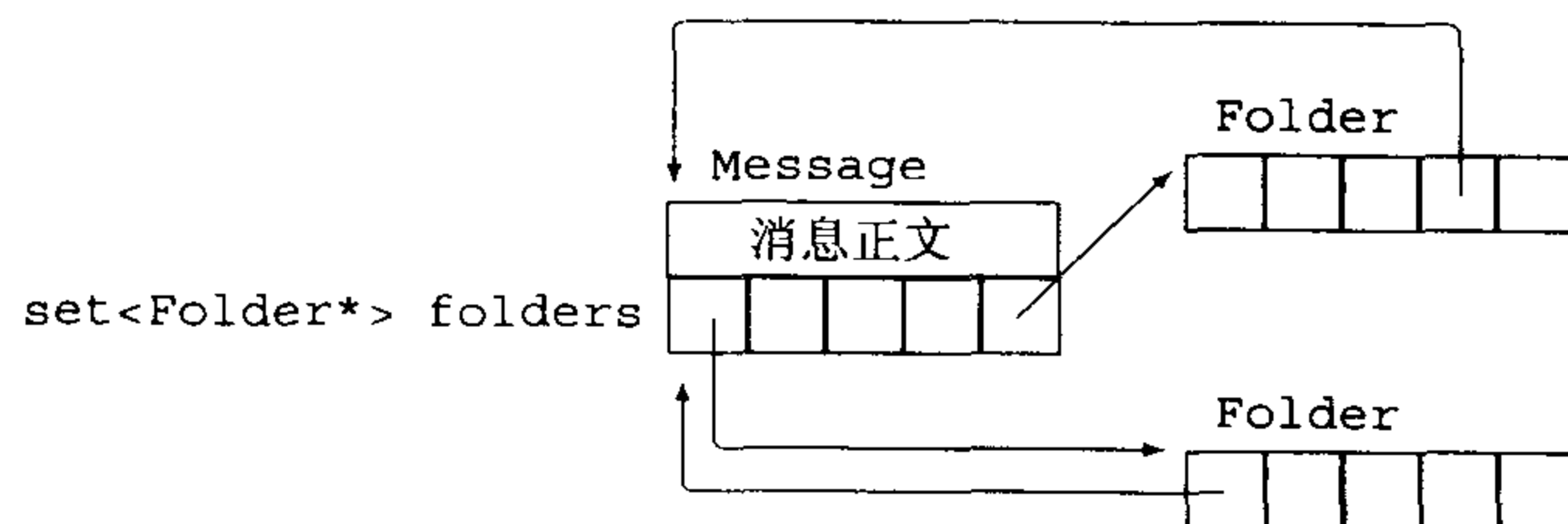


图 13-1 Message 和 Folder 类设计

查看这个操作列表, 可以看到, 析构函数和赋值操作符分担了从保存给定 Message 的 Folder 列表中删除消息的工作。类似地, 复制构造函数和赋值操作符分担将一个 Message 加到给定

Folder 列表的工作。我们将定义一对 private 实用函数完成这些任务。

1. Message 类

对于以上的设计，可以如下编写 Message 类的部分代码：

```
class Message {
public:
    // folders is initialized to the empty set automatically
    Message(const std::string &str = ""):
        contents (str) { }
    // copy control: we must manage pointers to this Message
    // from the Folders pointed to by folders
    Message(const Message&);
    Message& operator=(const Message&);
    ~Message();
    // add/remove this Message from specified Folder's set of messages
    void save (Folder&);
    void remove(Folder&);
private:
    std::string contents;           // actual message text
    std::set<Folder*> folders;      // Folders that have this Message
    // Utility functions used by copy constructor, assignment, and destructor:
    // Add this Message to the Folders that point to the parameter
    void put_Msg_in_Folders(const std::set<Folder*>&);
    // remove this Message from every Folder in folders
    void remove_Msg_from_Folders();
};
```

488

Message 类定义了两个数据成员：contents 是一个保存实际消息的 string，folders 是一个 set，包含指向该 Message 所在的 Folder 的指针。

构造函数接受单个 string 形参，表示消息的内容。构造函数将消息的副本保存在 contents 中，并（隐式）将 Folder 的 set 初始化为空集。这个构造函数提供一个默认实参（为空串），所以它也可以作为默认构造函数。

实用函数提供由复制控制成员共享的行为。put_Msg_in_Folders 函数将自身 Message 的一个副本添加到指向给定 Message 的各 Folder 中，这个函数执行完后，形参指向的¹每个 Folder 也将指向这个 Message。复制构造函数和赋值操作符都将使用这个函数。

remove_Msg_from_Folders 函数用于赋值操作符和析构函数，它从 folders 成员的每个 Folder 中删除指向这个 Message 的指针。

2. Message 类的复制控制

复制 Message 时，必须将新创建的 Message 添加到保存原 Message 的每个 Folder 中。这个工作超出了合成构造函数的能力范围，所以我们必须定义自己的复制构造函数：

```
Message::Message(const Message &m):
    contents(m.contents), folders(m.folders)
{
    // add this Message to each Folder that points to m
    put_Msg_in_Folders(folders);
}
```

1. 此处原文误为“指向形参的”。——译者注

}

复制构造函数将用旧对象成员的副本初始化新对象的数据成员。除了这些初始化之外（合成复制构造函数可以完成这些初始化），还必须用 `folders` 进行迭代，将这个新的 `Message` 加到那个集的每个 `Folder` 中。复制构造函数使用 `put_Msg_in_Folder` 函数完成这个工作。



编写自己的复制构造函数时，必须显式复制需要复制的任意成员。显式定义的复制构造函数不会进行任何自动复制。

像其他任何构造函数一样，如果没有初始化某个类成员，则那个成员用该成员的默认构造函数初始化。复制构造函数中的默认初始化不会使用成员的复制构造函数。

3. `put_Msg_in_Folder` 成员

`put_Msg_in_Folder` 通过形参 `rhs` 的成员 `folders` 中的指针进行迭代。这些指针表示指向 `rhs` 的每个 `Folder`，需要将指向这个 `Message` 的指针加到每个 `Folder` 中。

函数通过 `rhs.folders` 进行循环，调用命名为 `addMsg` 的 `Folder` 成员来完成这个工作，`addMsg` 函数将指向该 `Message` 的指针加到 `Folder` 中。

```
// add this Message to Folders that point to rhs
void Message::put_Msg_in_Folders(const set<Folder*> &rhs)
{
    for(std::set<Folder*>::const_iterator beg = rhs.begin();
        beg != rhs.end(); ++beg)
        (*beg)->addMsg(this);    // *beg points to a Folder
}
```

这个函数中唯一复杂的部分是对 `addMsg` 的调用：

```
(*beg) -> addMsg(this);    // *beg points to a Folder
```

那个调用以 `(*beg)` 开头，它解除迭代器引用。解除迭代器引用将获得一个指向 `Folder` 的指针。然后表达式对 `Folder` 指针应用箭头操作符以执行 `addMsg` 操作，将 `this` 传给 `addMsg`，该指针指向我们想要添加到 `Folder` 中的 `Message`。

4. `Message` 赋值操作符

赋值比复制构造函数更复杂。像复制构造函数一样，赋值必须对 `contents` 赋值并更新 `folders` 使之与右操作数的 `folders` 相匹配。它还必须将该 `Message` 加到指向 `rhs` 的每个 `Folder` 中，可以使用 `put_Msg_in_Folders` 函数完成赋值的这一部分工作。

在从 `rhs` 复制之前，必须首先从当前指向该 `Message` 的每个 `Folder` 中删除它。我们需要通过 `folders` 进行迭代，从 `folders` 的每个 `Folder` 中删除指向该 `Message` 的指针。命名为 `remove_Msg_from_Folders` 的函数将完成这项工作。

对于完成实际工作的 `remove_Msg_from_Folders` 和 `put_Msg_in_Folders`，赋值操作符本身相当简单：

```
Message& Message::operator=(const Message &rhs)
{
    if (&rhs != this) {
        remove_Msg_from_Folders(); // update existing Folders
    }
}
```

```

        contents = rhs.contents;    // copy contents from rhs
        folders = rhs.folders;      // copy Folder pointers from rhs
        // add this Message to each Folder in rhs
        put_Msg_in_Folders(rhs.folders);
    }
    return *this;
}

```

490

赋值操作符首先检查左右操作数是否相同。查看函数的后续部分可以清楚地看到进行这一检查的原因。假定操作数是不同对象，调用 `remove_Msg_from_Folders` 从 `folders` 成员的每个 `Folder` 中删除该 `Message`。一旦这项工作完成，必须将右操作数的 `contents` 和 `folders` 成员赋值给这个对象。最后，调用 `put_Msg_in_Folders` 将指向这个 `Message` 的指针添加至指向 `rhs` 的每个 `Folder` 中。

了解了 `remove_Msg_from_Folders` 的工作之后，我们来看看为什么赋值操作符首先要检查对象是否不同。赋值时需删除左操作数，并在撤销左操作数的成员之后，将右操作数的成员赋值给左操作数的相应成员。如果对象是相同的，则撤销左操作数的成员也将撤销右操作数的成员！



即使对象赋值给自己，赋值操作符的正确工作也非常重要。保证这个行为的通用方法是显式检查对自身的赋值。

5. `remove_Msg_from_Folders` 成员

除了调用 `remMsg` 从 `folders` 指向的每个 `Folder` 中删除这个 `Message` 之外，`remove_Msg_from_Folders` 函数的实现与 `put_Msg_in_Folders` 类似：

```

// remove this Message from corresponding Folders
void Message::remove_Msg_from_Folders()
{
    // remove this message from corresponding folders
    for(std::set<Folder*>::const_iterator beg =
        folders.begin(); beg != folders.end(); ++beg)
        (*beg)->remMsg(this); // *beg points to a Folder
}

```

6. `Message` 析构函数

剩下必须实现的复制控制函数是析构函数：

```

Message::~Message()
{
    remove_Msg_from_Folders();
}

```

有了 `remove_Msg_from_Folders` 函数，编写析构函数将非常简单。我们调用 `remove_Msg_from_Folders` 函数清除 `folders`，系统自动调用 `string` 析构函数释放 `contents`，自动调用 `set` 析构函数清除用于保存 `folders` 成员的内存，因此，`Message` 析构函数唯一要做的是调用 `remove_Msg_from_Folders`。



最佳
实践

赋值操作符通常要做复制构造函数和析构函数也要完成的工作。在这种情况下，通用工作应放在 `private` 实用函数中。

491

习题

习题 13.16 编写本节中描述的 `Message` 类。

习题 13.17 为 `Message` 类增加与 `Folder` 的 `addMsg` 和 `remMsg` 操作类似的函数。这些函数可以命名为 `addFldr` 和 `remFldr`，应接受一个指向 `Folder` 的指针并将该指针插入到 `folders`。这些函数可为 `private` 的，因为它们将仅在 `Message`¹ 类的实现中使用。

习题 13.18 编写相应的 `Folder` 类。该类应保存一个 `set<Message*>`，包含指向 `Message` 的元素。

习题 13.19 在 `Message` 类和 `Folder` 类中增加 `save` 和 `remove` 操作。这些操作应接受一个 `Folder`，并将该 `Folder` 加入到指向这个 `Message` 的 `Folder` 集中（或从其中删除该 `Folder`）。操作还必须更新 `Folder` 以反映它指向该 `Message`，这可以通过调用 `addMsg` 或 `remMsg` 完成。

13.5 管理指针成员

本书始终提倡使用标准库。这样做的一个原因是，使用标准库能够大大减少现代 C++ 程序中对指针的需要。然而，许多应用程序仍需要使用指针，特别是在类的实现中。包含指针的类需要特别注意复制控制，原因是复制指针时只复制指针中的地址，而不会复制指针指向的对象。

设计具有指针成员的类时，类设计者必须首先需要决定的是该指针应提供什么行为。将一个指针复制到另一个指针时，两个指针指向同一对象。当两个指针指向同一对象时，可能使用任一指针改变基础对象。类似地，很可能一个指针删除了一对象时，另一指针的用户还认为基础对象仍然存在。

指针成员默认具有与指针对象同样的行为。然而，通过不同的复制控制策略，可以为指针成员实现不同的行为。大多数 C++ 类采用以下三种方法之一管理指针成员：

(1) 指针成员采取常规指针型行为。这样的类具有指针的所有缺陷但无需特殊的复制控制。
(2) 类可以实现所谓的“智能指针”行为。指针所指向的对象是共享的，但类能够防止悬垂指针。

(3) 类采取值型行为。指针所指向的对象是唯一的，由每个类对象独立管理。

492

本节中介绍三个类，分别实现管理指针成员的三种不同方法。

1. 一个带指针成员的简单类

为了阐明所涉及的问题，我们将实现一个简单类，该类包含一个 `int` 值和一个指针：

```
// class that has a pointer member that behaves like a plain pointer
class HasPtr {
public:
    // copy of the values we're given
```

1. 此处原文误为 `Folder`。——译者注

```

HasPtr(int *p, int i): ptr(p), val(i) { }
// const members to return the value of the indicated data member
int *get_ptr() const { return ptr; }
int get_int() const { return val; }
// non const members to change the indicated data member
void set_ptr(int *p) { ptr = p; }
void set_int(int i) { val = i; }
// return or change the value pointed to, so ok for const objects
int get_ptr_val() const { return *ptr; }
void set_ptr_val(int val) const { *ptr = val; }
private:
    int *ptr;
    int val;
};

```

HasPtr 构造函数接受两个形参，将它们复制到 HasPtr 的数据成员。HasPtr 类提供简单的访问函数：函数 `get_int` 和 `get_ptr` 分别返回 `int` 成员和指针成员的值；`set_int` 和 `set_ptr` 成员则使我们能够改变这些成员，给 `int` 成员一个新值或使指针成员指向不同的对象。还定义了 `get_ptr_val` 和 `set_ptr_val` 成员，它们能够获取和设置指针所指向的基础值。

2. 默认复制/赋值与指针成员

因为 HasPtr 类没有定义复制构造函数，所以复制一个 HasPtr 对象将复制两个成员：

```

int obj = 0;
HasPtr ptr1(&obj, 42); // int* member points to obj, val is 42
HasPtr ptr2(ptr1);     // int* member points to obj, val is 42

```

复制之后，`ptr1` 和 `ptr2` 中的指针指向同一对象且两个对象中的 `int` 值相同。但是，因为指针的值不同于它所指对象的值，这两个成员的行为看来非常不同。复制之后，`int` 值是清楚和独立的，而指针则纠缠在一起。



具有指针成员且使用默认合成复制构造函数的类具有普通指针的所有缺陷。尤其是，类本身无法避免悬垂指针。

493

3. 指针共享同一对象

复制一个算术值时，副本独立于原版，可以改变一个副本而不改变另一个：

```

ptr1.set_int(0); // changes val member only in ptr1
ptr2.get_int();  // returns 42
ptr1.get_int();  // returns 0

```

复制指针时，地址值是可区分的，但指针指向同一基础对象。如果在任一对象上调用 `set_ptr_val`，则二者的基础对象都会改变：

```

ptr1.set_ptr_val(42); // sets object to which both ptr1 and ptr2 point
ptr2.get_ptr_val();   // returns 42

```

两个指针指向同一对象时，其中任意一个都可以改变共享对象的值。

4. 可能出现悬垂指针

因为类直接复制指针，会使用户面临潜在的问题：HasPtr 保存着给定指针。用户必须保证只要 HasPtr 对象存在，该指针指向的对象就存在：

```
int *ip = new int(42); // dynamically allocated int initialized to 42
HasPtr ptr(ip, 10);    // HasPtr points to same object as ip does
delete ip;             // object pointed to by ip is freed
ptr.set_ptr_val(0);    // disaster: The object to which HasPtr points was freed!
```

这里的问题是 `ip` 和 `ptr` 中的指针指向同一对象。删除了该对象时，`ptr`¹ 中的指针不再指向有效对象。然而，没有办法得知对象已经不存在了。

习题

习题 13.20 对于 `HasPtr` 类的原始版本（依赖于复制控制的默认定义），描述下面代码中会发生什么：

```
int i = 42;
HasPtr p1(&i, 42);
HasPtr p2 = p1;
cout << p2.get_ptr_val() << endl;
p1.set_ptr_val(0);
cout << p2.get_ptr_val() << endl;
```

习题 13.21 如果给 `HasPtr` 类添加一个析构函数，用来删除指针成员，会发生什么？

494

13.5.1 定义智能指针类

上节中我们定义了一个简单类，保存一个指针和一个 `int` 值。其中指针成员的行为与其他任意指针完全相同。对该指针指向的对象所做的任意改变都将作用于共享对象。如果用户删除该对象，则类就有一个悬垂指针，指向一个不复存在的对象。

除了使指针成员与指针完全相同之外，另一种方法是定义所谓的**智能指针**（`smart pointer`）类。智能指针除了增加功能外，其行为像普通指针一样。本例中让智能指针负责删除共享对象。用户将动态分配一个对象并将该对象的地址传给新的 `HasPtr` 类。用户仍然可以通过普通指针访问对象，但绝不能删除指针。`HasPtr` 类将保证在撤销指向对象的最后一个 `HasPtr` 对象时删除对象。

`HasPtr` 在其他方面的行为与普通指针一样。具体而言，复制对象时，副本和原对象将指向同一基础对象，如果通过一个副本改变基础对象，则通过另一对象访问的值也会改变。

新的 `HasPtr` 类需要一个析构函数来删除指针，但是，析构函数不能无条件地删除指针。如果两个 `HasPtr` 对象指向同一基础对象，那么，在两个对象都撤销之前，我们并不希望删除基础对象。为了编写析构函数，需要知道这个 `HasPtr` 对象是否为指向给定对象的最后一个。

1. 引入使用计数

定义智能指针的通用技术是采用一个**使用计数**（`use count`）。智能指针类将一个计数器与类指向的对象相关联。使用计数跟踪该类有多少个对象共享同一指针。使用计数为 0 时，删除对象。使用计数有时也称为**引用计数**（`reference count`）。

每次创建类的新对象时，初始化指针并将使用计数置为 1。当对象作为另一对象的副本而创建时，复制构造函数复制指针并增加与之相应的使用计数的值。对一个对象进行赋值时，赋值操

1. 此处原文误为 `HasPtr`。——译者注

作符减少左操作数所指对象的使用计数的值（如果使用计数减至 0，则删除对象），并增加右操作数所指对象的使用计数的值。最后，调用析构函数时，析构函数减少使用计数的值，如果计数减至 0，则删除基础对象。

唯一的创新在于决定将使用计数放在哪里。计数器不能直接放在 HasPtr 对象中，为什么呢？考虑下面的情况：

495

```
int obj;
HasPtr p1(&obj, 42);
HasPtr p2(p1); // p1 and p2 both point to same int object
HasPtr p3(p1); // p1, p2, and p3 all point to same int object
```

如果使用计数保存在 HasPtr 对象中，创建 p3 时怎样更新它？可以在 p1 中将计数增量并复制到 p3，但怎样更新 p2 中的计数？

2. 使用计数类

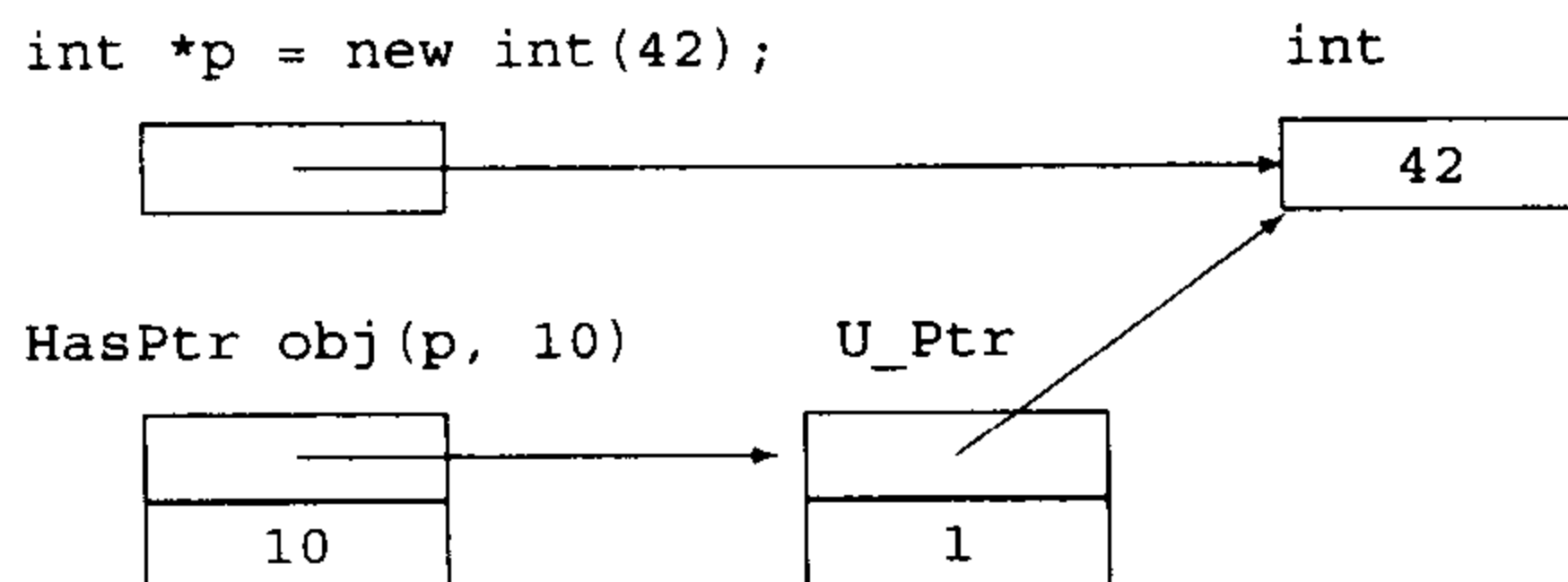
实现使用计数有两种经典策略，在这里将使用其中一种，另一种方法在 15.8.1 节中讲述。这里所用的方法中，需要定义一个单独的具体类用以封装使用计数和相关指针：

```
// private class for use by HasPtr only
class U_Ptr {
    friend class HasPtr;
    int *ip;
    size_t use;
    U_Ptr(int *p): ip(p), use(1) { }
    ~U_Ptr() { delete ip; }
};
```

这个类的所有成员均为 private。我们不希望普通用户使用 U_Ptr 类，所以它没有任何 public 成员。将 HasPtr 类设置为友元，使其成员可以访问 U_Ptr 的成员。

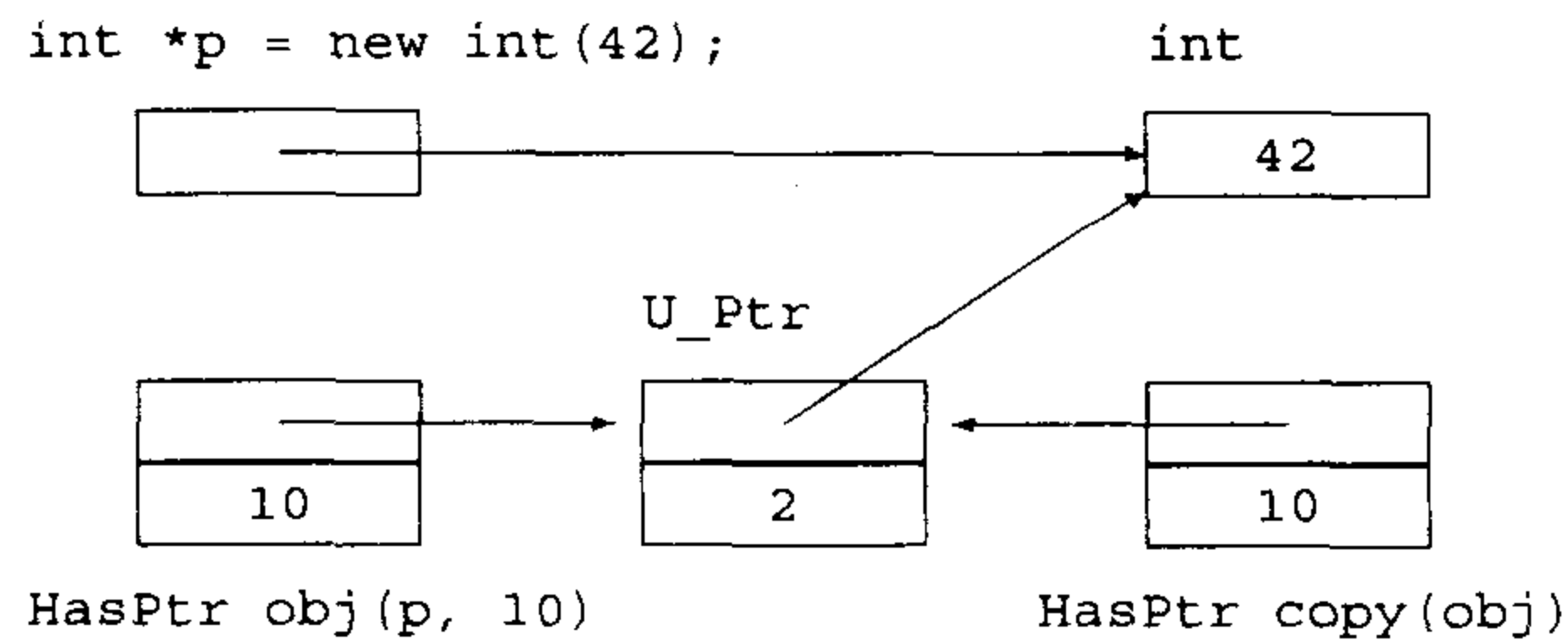
尽管该类的工作原理比较难，但这个类相当简单。U_Ptr 类保存指针和使用计数，每个 HasPtr 对象将指向一个 U_Ptr 对象，使用计数将跟踪指向每个 U_Ptr 对象的 HasPtr 对象的数目。U_Ptr 定义的仅有函数是构造函数和析构函数，构造函数复制指针，而析构函数删除它。构造函数还将使用计数置为 1，表示一个 HasPtr 对象指向这个 U_Ptr 对象。

假定刚从指向 int 值 42 的指针创建了一个 HasPtr 对象，可以画出这些对象，如下图：



496

如果复制这个对象，则对象如下图所示。



3. 使用计数类的使用

新的 HasPtr 类保存一个指向 U_Ptr 对象的指针，U_Ptr 对象指向实际的 int 基础对象。必须改变每个成员以说明 HasPtr 类指向一个 U_Ptr 对象而不是一个 int¹ 值。

先看看构造函数和复制控制成员：

```
/* smart pointer class: takes ownership of the dynamically allocated
 *      object to which it is bound
 * User code must dynamically allocate an object to initialize a HasPtr
 * and must not delete that object; the HasPtr class will delete it
 */
class HasPtr {
public:
    // HasPtr owns the pointer; p must have been dynamically allocated
    HasPtr(int *p, int i): ptr(new U_Ptr(p)), val(i) { }
    // copy members and increment the use count
    HasPtr(const HasPtr &orig):
        ptr(orig.ptr), val(orig.val) { ++ptr->use; }
    HasPtr& operator=(const HasPtr&);
    // if use count goes to zero, delete the U_Ptr object
    ~HasPtr() { if (--ptr->use == 0) delete ptr; }
private:
    U_Ptr *ptr;          // points to use-counted U_Ptr class
    int val;
};
```

接受一个指针和一个 int 值的 HasPtr 构造函数使用其指针形参创建一个新的 U_Ptr 对象。HasPtr 构造函数执行完毕后，HasPtr 对象指向一个新分配的 U_Ptr 对象，该 U_Ptr 对象存储给定指针。新 U_Ptr 中的使用计数为 1，表示只有一个 HasPtr 对象指向它。

复制构造函数从形参复制成员并增加使用计数的值。复制构造函数执行完毕后，新创建对象与原有对象指向同一 U_Ptr 对象，该 U_Ptr 对象的使用计数加 1。

析构函数将检查 U_Ptr 基础对象的使用计数。如果使用计数为 0，则这是最后一个指向该 U_Ptr 对象的 HasPtr 对象，在这种情况下，HasPtr 析构函数删除其 U_Ptr 指针。删除该指针将引起对 U_Ptr 析构函数的调用，U_Ptr 析构函数删除 int 基础对象。

4. 赋值与使用计数

赋值操作符比复制构造函数复杂一点：

1. 原文误为 int*。——译者注

```

HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    ++rhs.ptr->use;      // increment use count on rhs first
    if (--ptr->use == 0)
        delete ptr;     // if use count goes to 0 on this object, delete it
    ptr = rhs.ptr;      // copy the U_Ptr object
    val = rhs.val;      // copy the int member
    return *this;
}

```

在这里，首先将右操作数中的使用计数加 1，然后将左操作数对象的使用计数减 1 并检查这个使用计数。像析构函数中那样，如果这是指向 U_Ptr 对象的最后一个对象，就删除该对象，这会依次撤销 int 基础对象。将左操作数中的当前值减 1（可能撤销该对象）之后，再将指针从 rhs 复制到这个对象。赋值照常返回对这个对象的引用。



这个赋值操作符在减少左操作数的使用计数之前使 rhs 的使用计数加 1，从而防止自身赋值。

如果左右操作数相同，赋值操作符的效果将是 U_Ptr 基础对象的使用计数加 1 之后立即减 1。

5. 改变其他成员

现在需要改变访问 int* 的其他成员，以便通过 U_Ptr 指针间接获取 int：

```

class HasPtr {
public:
    // copy control and constructors as before
    // accessors must change to fetch value from U_Ptr object
    int *get_ptr() const { return ptr->ip; }
    int get_int() const { return val; }
    // change the appropriate data member
    void set_ptr(int *p) { ptr->ip = p; }
    void set_int(int i) { val = i; }
    // return or change the value pointed to, so ok for const objects
    // Note: *ptr->ip is equivalent to *(ptr->ip)
    int get_ptr_val() const { return *ptr->ip; }
    void set_ptr_val(int i) { *ptr->ip = i; }
private:
    U_Ptr *ptr;      // points to use-counted U_Ptr class
    int val;
};

```

498

获取和设置 int 成员的函数不变。那些使用指针操作的函数必须对 U_Ptr 解引用，以便获取 int* 基础对象。

复制 HasPtr 对象时，int 成员的行为与第一个类中一样。所复制的是 int 成员的值，各成员是独立的，副本和原对象中的指针仍指向同一基础对象，对基础对象的改变将影响通过任一 HasPtr 对象所看到的值。然而，HasPtr 的用户无须担心悬垂指针。只要他们让 HasPtr 类负责释放对象，HasPtr 类将保证只要有指向基础对象的 HasPtr 对象存在，基础对象就存在。

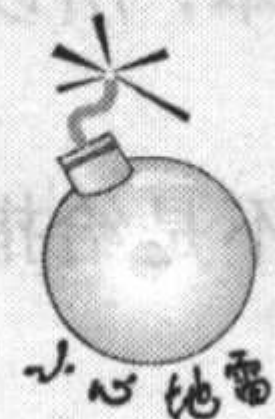
建议：管理指针成员

具有指针成员的对象一般需要定义复制控制成员。如果依赖合成版本，会给类的用户增加负担。用户必须保证成员所指向的对象存在，只要还有对象指向该对象。

为了管理具有指针成员类，必须定义三个复制控制成员：复制构造函数、赋值操作符和析构函数。这些成员可以定义指针成员的指针型行为或值型行为。

值型类将指针成员所指基础值的副本给每个对象。复制构造函数分配新元素并从被复制对象处复制值，赋值操作符撤销所保存的原对象并从右操作数向左操作数复制值，析构函数撤销对象。

作为定义值型行为或指针型行为的另一选择，是使用称为“智能指针”的一些类。这些类在对象间共享同一基础值，从而提供了指针型行为。但它们使用复制控制技术以避免常规指针的一些缺陷。为了实现智能指针行为，类需要保证基础对象一直存在，直到最后一个副本消失。使用计数（13.5.1 节）是管理智能指针类的通用技术。同一基础值的每个副本都有一个使用计数。复制构造函数将指针从旧对象复制到新对象时，会将使用计数加 1。赋值操作符将左操作数的使用计数减 1 并将右操作数的使用计数加 1，如果左操作数的使用计数减至 0，赋值操作符必须删除它所指向的对象，最后，赋值操作符将指针从右操作数复制到左操作数。析构函数将使用计数减 1，并且，如果使用计数减至 0，就删除基础对象。



管理指针的这些方法用得非常频繁，因此使用带指针成员类的程序员必须充分熟悉这些编程技术。

习题

习题 13.22 什么是使用计数？

习题 13.23 什么是智能指针？智能指针类如何与实现普通指针行为的类相区别？

习题 13.24 实现你自己的使用计数的 HasPtr 类的版本。

13.5.2 定义值型类

处理指针成员的另一个完全不同的方法，是给指针成员提供值语义（value semantics）。具有值语义的类所定义的对象，其行为很像算术类型的对象：复制值型对象时，会得到一个不同的新副本。对副本所做的改变不会反映在原有对象上，反之亦然。string 类是值型类的一个例子。

要使指针成员表现得像一个值，复制 HasPtr 对象时必须复制指针所指向的对象：

```
/*
 * Valuelike behavior even though HasPtr has a pointer member:
 * Each time we copy a HasPtr object, we make a new copy of the
 * underlying int object to which ptr points.
 */
class HasPtr {
```

```

public:
    // no point to passing a pointer if we're going to copy it anyway
    // store pointer to a copy of the object we're given
    HasPtr(const int &p, int i): ptr(new int(p)), val(i) {}
    // copy members and increment the use count
    HasPtr(const HasPtr &orig):
        ptr(new int (*orig.ptr)), val(orig.val) { }
    HasPtr& operator=(const HasPtr&);
    ~HasPtr() { delete ptr; }
    // accessors must change to fetch value from Ptr object
    int get_ptr_val() const { return *ptr; }
    int get_int() const { return val; }
    // change the appropriate data member
    void set_ptr(int *p) { ptr = p; }
    void set_int(int i) { val = i; }
    // return or change the value pointed to, so ok for const objects
    int *get_ptr() const { return ptr; }
    void set_ptr_val(int p) const { *ptr = p; }
private:
    int *ptr;          // points to an int
    int val;
};

```

复制构造函数不再复制指针，它将分配一个新的 `int` 对象，并初始化该对象以保存与被复制对象相同的值。每个对象都保存属于自己的 `int` 值的不同副本。因为每个对象保存自己的副本，所以析构函数将无条件删除指针。

赋值操作符不需要分配新对象，它只是必须记得给其指针所指向的对象赋新值，而不是给指针本身赋值：

```

HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    // Note: Every HasPtr is guaranteed to point at an actual int;
    // We know that ptr cannot be a zero pointer
    *ptr = *rhs.ptr;      // copy the value pointed to
    val = rhs.val;        // copy the int
    return *this;
}

```

换句话说，改变的是指针所指向的值，而不是指针。



即使要将一个对象赋值给它本身，赋值操作符也必须总是保证正确。本例中，即使左右操作数相同，操作本质上也是安全的，因此，不需要显式检查自身赋值。

习题

习题 13.25 什么是值型类？

习题 13.26 实现你自己的值型 `HasPtr` 类版本。

习题 13.27 值型 `HasPtr` 类定义了所有复制控制成员。描述将会发生什么，如果该类：

- (a) 定义了复制构造函数和析构函数但没有定义赋值操作符。
- (b) 定义了复制构造函数和赋值操作符但没有定义析构函数。

(c) 定义了析构函数但没有定义复制构造函数和赋值操作符。

习题 13.28 对于如下的类，实现默认构造函数和必要的复制控制成员。

```
(a) class TreeNode {
    public:
        // ...
    private:
        std::string value;
        int count;
        TreeNode *left;
        TreeNode *right;
};

(b) class BinStrTree {
    public:
        //...
    private:
        TreeNode *root;
};
```

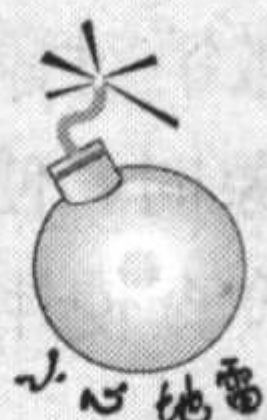
501

小结

类除了定义该类型对象上的操作，还需要定义复制、赋值或撤销该类型对象的含义。特殊成员函数（复制构造函数、赋值操作符和析构函数）可用于定义这些操作。这些操作统称为“复制控制”函数。

如果类没有定义这些操作中的一个或多个，编译器将自动定义它们。合成操作执行逐个成员初始化、赋值或撤销：合成操作依次取得每个成员，根据成员类型进行成员的复制、赋值或撤销。如果成员为类类型的，合成操作调用该类的相应操作（即，复制构造函数调用成员的复制构造函数，析构函数调用成员的析构函数，等等）。如果成员为内置类型或指针，则直接复制或赋值，析构函数对撤销内置类型或指针类型的成员没有影响。如果成员为数组，则根据元素类型以适当方式复制、赋值或撤销数组中的元素。

与复制构造函数和赋值操作符不同，无论类是否定义了自己的析构函数，都会创建和运行合成析构函数。如果类定义了析构函数，则在类定义的析构函数结束之后运行合成析构函数。



定义复制控制函数最为困难的部分通常在于认识到它们的必要性。

分配内存或其他资源的类几乎总是需要定义复制控制成员来管理所分配的资源。如果一个类需要析构函数，则它几乎也总是需要定义复制构造函数和赋值操作符。

术语

assignment operator (赋值操作符) 赋值操作符可以重载，对将某个类类型对象赋值给另一同类型对象的含义进行定义。赋值操作符必须是类的成员并且必须返回对所属类对象的引用。如果类没有定义赋值操作符，则编译器

将会合成一个。

copy constructor (复制构造函数) 将新对象初始化为另一同类型对象的副本的构造函数。显式应用复制构造函数进行对象的按值传递，向函数传递对象或从函数返回对象。如果没有

定义复制构造函数，编译器就为我们合成一个。

copy control (复制控制) 控制类类型对象的复制、赋值和撤销的特殊成员。如果类没有另外定义它们，则编译器将为这些操作合成适当定义。

destructor (析构函数) 当对象超出作用域或删除对象时，清除对象的特殊成员函数。编译器将自动撤销每个成员。类类型的成员通过调用它们的析构函数而撤销，撤销内置或复合类型的成员无须做显式工作。特别地，析构函数的自动工作不会删除指针成员所指向的对象。

memberwise copy (逐个成员复制) 用于描述合成赋值操作符如何工作的术语。赋值操作符将成员依次从旧对象赋值给新对象。内置或复合类型成员直接赋值，类类型成员使用成员的赋值操作符进行赋值。

memberwise initialization (逐个成员初始化) 用于描述合成复制构造函数如何工作的术语。构造函数将成员依次从旧对象复制到新对象。内置或复合类型成员直接复制，类类型成员使用成员的复制构造函数进行复制。

overloaded operator (重载操作符) 将一个 C++ 操作符重定义以操作类类型对象的函数。本章介绍了如何定义赋值操作符，第 14 章将更详细地介绍重载操作符。

reference count (引用计数) 使用计数的同义词。

Rule of Three (三法则) 一个经验原则的简写形式，即，如果一个类需要析构函数，则该类几乎也必然需要定义自己的复制构造函数和赋值操作符。

smart pointer (智能指针) 一个行为类似指针但也提供其他功能的类。智能指针的一个通

用形式接受指向动态分配对象的指针并负责删除该对象。用户分配对象，但由智能指针类删除它。智能指针类需要实现复制控制成员来管理指向共享对象的指针。只有在撤销了指向共享对象的最后一个智能指针后，才能删除该共享对象。使用计数是实现智能指针类最常用的方式。

synthesized assignment operator (合成赋值操作符) 由编译器为没有显式定义赋值操作符的类创建（合成）的赋值操作符版本。合成赋值操作符将右操作数逐个成员地赋值给左操作数。

synthesized copy constructor (合成复制构造函数) 由编译器为没有显式定义复制构造函数的类创建（合成）的复制构造函数。合成复制构造函数将原对象逐个成员地初始化新对象。

use count (使用计数) 复制控制成员中使用的编程技术。使用计数与共享对象一起存储。需要创建一个单独类指向共享对象并管理使用计数。由构造函数，而不是复制构造函数，设置共享对象的状态并将使用计数置为 1。每当由复制构造函数或赋值操作符生成一个新副本时，使用计数加 1。由析构函数撤销对象或作为赋值操作符的左操作数撤销对象时，使用计数减 1。赋值操作符和析构函数检查使用计数是否已减至 0，如果是，则撤销对象。

value semantics (值语义) 对类模拟算术类型复制方式的复制控制行为的描述。值型对象的副本是独立的：对副本的改变不会影响原有对象。具有指针成员的值型类必须定义自己的复制控制成员。复制控制操作复制指针所指向的对象。只包含其他值型类或内置类型的值型类，通常可以依赖合成的复制控制成员。

第 14 章

重载操作符与转换

目录

14.1 重载操作符的定义.....	430
14.2 输入和输出操作符.....	435
14.3 算术操作符和关系操作符.....	439
14.4 赋值操作符.....	441
14.5 下标操作符.....	442
14.6 成员访问操作符.....	443
14.7 自增操作符和自减操作符.....	446
14.8 调用操作符和函数对象.....	449
14.9 转换与类类型.....	454
小结.....	466
术语.....	467

第 5 章介绍过，C++定义了许多内置类型间的操作符和自动转换。使用这些设施程序员能够编写丰富的混合类型表达式。

C++允许我们重定义操作符用于类类型对象时的含义。如果需要，可以像内置转换那样使用类类型转换，将一个类型的对象隐式转换到另一类型。

通过操作符重载，程序员能够针对类类型的操作数定义不同的操作符版本。第 13 章阐述了赋值操作符的重要性并介绍了怎样定义赋值操作符。我们第一次使用重载操作符是在第 1 章，那时程序用移位操作符（>>和<<）进行输入输出，用加号操作符（+）将两个 Sales_item 相加。本章我们终于可以看到怎样定义这些重载操作符了。

通过操作符重载，可以重定义第 5 章介绍的大多数操作符，使它们用于类类型对象。明智地使用操作符重载可以使类类型的使用像内置类型一样直观。例如，标准库为容器类定义了几个重载操作符。这些容器类定义了下标操作符以访问数据元素，定义了*和->对容器迭代器解引用。这些标准库的类型具有相同的操作符，使用它们就像使用内置数组和指针一样。允许程序使用表达式而不是命名函数，可以使编写和阅读程序容易得多。将

```
cout << "The sum of " << v1 << " and " << v2
      << " is " << v1 + v2 << endl;
```

和以下更为冗长的代码相比较就能够看到。如果 IO 使用命名函数，类似下面的代码将无法避免：

```
// hypothetical expression if IO used named functions
cout.print("The sum of ").print(v1).
    print(" and ").print(v2).print(" is ").
    print(v1 + v2).print("\n").flush();
```

14.1 重载操作符的定义

重载操作符是具有特殊名称的函数：保留字 operator 后接需定义的操作符符号。像任意其他函数一样，重载操作符具有返回类型和形参表，如下语句：

```
Sales_item operator+(const Sales_item&, const Sales_item&);
```

声明了加号操作符，可用于将两个 Sales_item 对象“相加”并获得一个 Sales_item 对象的副本。

除了函数调用操作符之外，重载操作符的形参数目（包括成员函数的隐式 this 指针）与操作符的操作数数目相同。函数调用操作符可以接受任意数目的操作数。

1. 重载的操作符名

表 14-1 列出了可以重载的操作符，不能重载的在表 14-2 列出。

表 14-1 可重载的操作符					
+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

表 14-2 不能重载的操作符	
::	?:

通过连接其他合法符号可以创建新的操作符。例如，定义一个 `operator**` 以提供求幂运算是合法的。第 18 章将介绍重载 `new` 和 `delete`。

506

2. 重载操作符必须具有一个类类型操作数

用于内置类型的操作符，其含义不能改变。例如，内置的整型加号操作符不能重定义：

```
// error: cannot redefine built-in operator for ints
int operator+(int, int);
```

也不能为内置数据类型重定义加号操作符。例如，不能定义接受两个数组类型操作数的 `operator+`。



重载操作符必须具有至少一个类类型或枚举类型（2.7 节）的操作数。这条规则强制重载操作符不能重新定义用于内置类型对象的操作符的含义。

3. 优先级和结合性是固定的

操作符的优先级（5.10.1 节）、结合性或操作数数目不能改变。不管操作数的类型和操作符的功能定义如何，表达式

```
x == y + z;
```

总是将实参 `y` 和 `z` 绑定到 `operator+`，并且将结果用作 `operator==` 的右操作数。

有四个符号（`+`，`-`，`*` 和 `&`）既可作一元操作符又可作二元操作符，这些操作符有的在其中一种情况下可以重载，有的两种都可以，定义的是哪个操作符由操作数数目控制。除了函数调用操作符 `operator()` 之外，重载操作符时使用默认实参是非法的。

507

4. 不再具备短路求值特性

重载操作符并不保证操作数的求值顺序，尤其是，不会保证内置逻辑 AND、逻辑 OR（5.2 节）和逗号操作符（5.9 节）的操作数求值。在 `&&` 和 `||` 的重载版本中，两个操作数都要进行求值，而且对操作数的求值顺序不做规定。因此，重载 `&&`、`||` 或逗号操作符不是一种好的做法。

5. 类成员与非成员

大多数重载操作符可以定义为普通非成员函数或类的成员函数。



作为类成员的重载函数，其形参看起来比操作数数目少 1。作为成员函数的操作符有一个隐含的 `this` 形参，限定为第一个操作数。

重载一元操作符如果作为成员函数就没有（显式）形参，如果作为非成员函数就有一个形参。类似地，重载二元操作符定义为成员时有一个形参，定义为非成员函数时有两个形参。

类 `Sales_item` 中给出了成员和非成员二元操作符的良好例子。我们知道该类有一个加号操作符。因为它有一个加号操作符，所以也应该定义一个复合赋值（`+=`）操作符，该操作符将一个 `Sales_item` 对象的值加至另一个 `Sales_item` 对象。

一般将算术和关系操作符定义为非成员函数，而将赋值操作符定义为成员：

```
// member binary operator: left-hand operand bound to implicit this pointer
Sales_item& Sales_item::operator+=(const Sales_item&);
```

```
// nonmember binary operator: must declare a parameter for each operand
Sales_item operator+(const Sales_item&, const Sales_item&);
```

加和复合赋值都是二元操作符，但这些函数定义了不同数目的形参，差异的原因在于 `this` 指针。

当操作符为成员函数，`this` 指向左操作数，因此，非成员 `operator+` 定义两个形参，都引用 `const Sales_item` 对象。即使复合赋值是二元操作符，成员复合赋值操作符也只接受一个（显式的）形参。使用操作符时，一个指向左操作数的指针自动绑定到 `this`，而右操作数限定为函数的唯一形参。

复合赋值返回一个引用而加操作符返回一个 `Sales_item` 对象，这也没什么。当应用于算术类型时，这一区别与操作符的返回类型相匹配：加返回一个右值，而复合赋值返回对左操作数的引用。

508

6. 操作符重载和友元关系

操作符定义为非成员函数时，通常必须将它们设置为所操作类的友元（12.5 节）。在本章的后面部分，将给出操作符可以定义为非成员的两个原因。在这种情况下，操作符通常需要访问类的私有部分。

`Sales_item` 类也是说明为何有些操作符需要设置为友元的一个好例子。它定义了一个成员操作符，并且有三个非成员操作符。这些非成员操作符需要访问私有数据成员，声明为友元：

```
class Sales_item {
    friend std::istream& operator>>
        (std::istream&, Sales_item&);
    friend std::ostream& operator<<
        (std::ostream&, const Sales_item&);
public:
    Sales_item& operator+=(const Sales_item&);
};
Sales_item operator+(const Sales_item&, const Sales_item&);
```

输入和输出操作符需要访问 `private` 数据不会令人惊讶，毕竟，它们的作用是读入和写出那些成员。另一方面，不需要将加操作符设置为友元，它可以用 `public` 成员 `operator+=` 实现。

7. 使用重载操作符

使用重载操作符的方式，与内置类型操作数上使用操作符的方式一样。假定 `item1` 和 `item2` 是 `Sales_item` 对象，可以打印它们的和，就像打印两个 `int` 的和一样：

```
cout << item1 + item2 << endl;
```

这个表达式隐式调用为 `Sales_item` 类而定义的 `operator+`。

也可以像调用普通函数一样调用重载操作符函数，指定函数并传递适当类型适当数目的形参：

```
// equivalent direct call to nonmember operator function
cout << operator+(item1, item2) << endl;
```

这个调用与将 `item1` 和 `item2` 相加的表达式等效。

调用成员操作符函数与调用任意其他函数是一样的：指定运行函数的对象，然后使用点或箭头操作符获取希望调用的函数，同时传递所需数目和类型的实参。对于二元成员操作符函数的情况

况，我们必须传递一个操作数：

```
item1 += item2;           // expression based "call"
item1.operator+=(item2); // equivalent call to member operator function
```

两个语句都将 item2 的值加至 item1。第一种情况下，使用表达式语法隐式调用重载操作符函数；第二种情况下，在 item1 对象上调用成员操作符函数。

509

习题

习题 14.1 在什么情况下重载操作符与内置操作符不同？在什么情况下重载操作符与内置操作符相同？

习题 14.2 为 Sales_item 编写输入、输出、加以及复合赋值操作符的重载声明。

习题 14.3 解释如下程序，假定 Sales_item 构造函数的参数是一个 string，且不为 explicit。解释如果构造函数为 explicit 会怎样。

```
string null_book = "9-999-99999-9";
Sales_item item(cin);
item += null_book;
```

习题 14.4 string 和 vector 类都定义了一个重载的 ==，可用于比较这些类的对象。指出下面表达式中应用了哪个 == 版本：

```
string s; vector<string> svec1, svec2;
"cobble" == "stone"
svec1[0] == svec2[0];
svec1 == svec2
```

重载操作符的设计

设计类的时候，需要记住一些有用的经验原则，可以有助于确定应该提供哪些重载操作符（如果需要提供）。

1. 不要重载具有内置含义的操作符

赋值操作符、取地址操作符和逗号操作符对类类型操作数有默认含义。如果没有特定重载版本，编译器就自己定义以下这些操作符。

- 合成赋值操作符（13.2 节）进行逐个成员赋值：使用成员自己的赋值操作符依次对每个成员进行赋值。
- 默认情况下，取地址操作符（&）和逗号操作符（,）在类类型对象上的执行，与在内置类型对象上的执行一样。取地址操作符返回对象的内存地址，逗号操作符从左至右计算每个表达式的值，并返回最右边操作数的值。
- 内置逻辑与（&&）和逻辑或（||）操作符使用短路求值（5.2 节）。如果重新定义该操作符，将失去操作符的短路求值特征。

通过为给定类类型的操作数重定义操作符，可以改变这些操作符的含义。



最佳实践 重载逗号、取地址、逻辑与、逻辑或等操作符通常不是好做法。这些操作符具有有用的内置含义，如果我们定义了自己的版本，就不能再使用这些内置含义。

510

有时我们需要定义自己的赋值运算。这样做时，它应表现得类似于合成操作符：赋值之后，左右操作数的值应是相同的，并且操作符应返回对左操作数的引用。重载的赋值运算应在赋值的内置含义基础上进行定制，而不是完全绕开。

2. 大多数操作符对类对象没有意义

除非提供了重载定义，赋值、取地址和逗号操作符对于类类型操作数没有意义。设计类的时候，应该确定要支持哪些操作符。

为类设计操作符，最好的方式是首先设计类的公用接口。定义了接口之后，就可以考虑应将哪些操作符定义为重载操作符。那些逻辑上可以映射到某个操作符的操作可以考虑作为候选的重载操作符。例如：

- 相等测试操作应使用 `operator==`。
- 一般通过重载移位操作符进行输入和输出。
- 测试对象是否为空的操作可用逻辑非操作符 `operator!` 表示。

3. 复合赋值操作符

如果一个类有算术操作符（5.1 节）或位操作符（5.3 节），那么，提供相应的复合赋值操作符一般是个好的做法。例如，`Sales_item` 类定义了 `+` 操作符，逻辑上，它也应该定义 `+=`。不用说，操作符的行为应定义为与内置操作符一样：复合赋值的行为应与 `+` 之后接着 `=` 类似。

警告：审慎使用操作符重载

每个操作符用于内置类型都有关联的含义。例如，二元 `+` 与加法是完全相同的。将二元 `+` 对应到一个类类型的类似操作可提供方便的简写方法。例如，标准库的类型 `string`，遵循许多程序设计语言的通用规范，使用 `+` 表示连接——将一个串“加”至另一个串。

当内置操作符和类型上的操作存在逻辑对应关系时，操作符重载最有用。使用重载操作符而不是创造命名操作，可以令程序更自然、更直观，而滥用操作符重载会使得我们的类难以理解。

在实践中很少发生明显的操作符重载滥用。例如，不负责任的程序员可能会定义 `operator+` 来执行减法。更常见但仍不可取的是，改变操作符的“正常”含义以强行适应给定类型。操作符应该只用于对用户而言无二义的操作。在这里所谓有二义的操作符，就是指具有多个不同解释的操作符。

当一个重载操作符的含义不明显时，给操作取一个名字更好。对于很少用的操作，使用命名函数通常也比用操作符更好。如果不是普通操作，没有必要为简洁而使用操作符。



4. 相等和关系操作符

将要用作关联容器键类型的类应定义 `<` 操作符。关联容器默认使用键类型的 `<` 操作符。即使该类型将只存储在顺序容器中，类通常也应该定义相等 (`==`) 操作符和小于 (`<`) 操作符，理由是很多算法假定这些操作符存在。例如 `sort` 算法使用 `<` 操作符，而 `find` 算法使用 `==` 操作符。

如果类定义了相等操作符，它也应该定义不等操作符`!=`。类用户会假设如果可以进行相等比较，则也可以进行不等比较。同样的规则也应用于其他关系操作符。如果类定义了`<`，则它可能应该定义全部的四个关系操作符（`>`，`>=`，`<`，`<=`）。

5. 选择成员或非成员实现

为类设计重载操作符的时候，必须选择是将操作符设置为类成员还是普通非成员函数。在某些情况下，程序员没有选择，操作符必须是成员；在另一些情况下，有些经验原则可指导我们做出决定。下面是一些指导原则，有助于决定将操作符设置为类成员还是普通非成员函数：

- 赋值（`=`）、下标（`[]`）、调用（`()`）和成员访问箭头（`->`）等操作符必须定义为成员，将这些操作符定义为非成员函数将在编译时标记为错误。
- 像赋值一样，复合赋值操作符通常应定义为类的成员。与赋值不同的是，不一定非得这样做，如果定义非成员复合赋值操作符，不会出现编译错误。
- 改变对象状态或与给定类型紧密联系的其他一些操作符，如自增、自减和解引用，通常应定义为类成员。
- 对称的操作符，如算术操作符、相等操作符、关系操作符和位操作符，最好定义为普通非成员函数。

512

习题

习题 14.5 列出必须定义为类成员的操作符。

习题 14.6 解释下面操作符是否应该为类成员，为什么？

(a) + (b) += (c) ++ (d) -> (e) << (f) && (g) == (h) ()

14.2 输入和输出操作符

支持 I/O 操作的类所提供的 I/O 操作接口，一般应该与标准库 `iostream` 为内置类型定义的接口相同，因此，许多类都需要重载输入和输出操作符。

14.2.1 输出操作符`<<`的重载



为了与 IO 标准库一致，操作符应接受 `ostream&` 作为第一个形参，对类类型 `const` 对象的引用作为第二个形参，并返回对 `ostream` 形参的引用。

重载输出操作符一般的简单定义如下：

```
// general skeleton of the overloaded output operator
ostream&
operator <<(ostream& os, const ClassType &object)
{
    // any special logic to prepare object

    // actual output of members
    os << // ...
}
```

```

    // return ostream object
    return os;
}

```

513 第一个形参是对 ostream 对象的引用，在该对象上将产生输出。ostream 为非 const，因为写入到流会改变流的状态。该形参是一个引用，因为不能复制 ostream 对象。

第二个形参一般应是对要输出的类类型的引用。该形参是一个引用以避免复制实参。它可以是 const，因为（一般而言）输出一个对象不应该改变该对象。使形参成为 const 引用，就可以使用同一个定义来输出 const 和非 const 对象。

返回类型是一个 ostream 引用，它的值通常是输出操作符所操作的 ostream 对象。

1. Sales_item 输出操作符

现在可以编写 Sales_item 的输出操作符了：

```

ostream&
operator<<(ostream& out, const Sales_item& s)
{
    out << s.isbn << "\t" << s.units_sold << "\t"
        << s.revenue << "\t" << s.avg_price();
    return out;
}

```

输出 Sales_item，就需要输出它的三个数据成员以及计算得到的平均销售价格，每个成员用制表符间隔。输出值之后，该操作符返回对所写 ostream 对象的引用。

2. 输出操作符通常所做格式化应尽量少

关于输出，类设计者面临一个重要决定：是否格式化以及进行多少格式化。



最佳实践 一般而言，输出操作符应输出对象的内容，进行最小限度的格式化，它们不应该输出换行符。

用于内置类型的输出操作符所做的格式化很少，并且不输出换行符。由于内置类型的这种既定处理，用户预期类输出操作符也有类似行为。通过限制输出操作符只输出对象的内容，如果需要执行任意额外的格式化，我们让用户决定该如何处理。尤其是，输出操作符不应该输出换行符，如果该操作符输出换行符，则用户就不能将说明文字与对象输出在同一行上。尽量减少操作符所做的格式化，让用户自己控制输出细节。

3. IO 操作符必须为非成员函数

当定义符合标准库 iostream 规范的输入或输出操作符的时候，必须使它成为非成员操作符，为什么需要这样做呢？

我们不能将该操作符定义为类的成员，否则，左操作数将只能是该类类型的对象：

```

// if operator<< is a member of Sales_item
Sales_item item;
item << cout;

```

514

这个用法与为其他类型定义的输出操作符的正常使用方式相反。

如果想要支持正常用法，则左操作数必须为 `ostream` 类型。这意味着，如果该操作符是类的成员，则它必须是 `ostream` 类的成员，然而，`ostream` 类是标准库的组成部分，我们（以及任何想要定义 IO 操作符的人）是不能为标准库中的类增加成员的。

相反，如果想要使用重载操作符为该类型提供 IO 操作，就必须将它们定义为非成员函数。IO 操作符通常对非公用数据成员进行读写，因此，类通常将 IO 操作符设为友元。

习题

习题 14.7 为下面的 `CheckoutRecord` 类定义一个输出操作符：

```
class CheckoutRecord {
public:
    // ...
private:
    double book_id;
    string title;
    Date date_borrowed;
    Date date_due;
    pair<string,string> borrower;
    vector< pair<string,string>* > wait_list;
};
```

习题 14.8 12.4 节的习题中，你编写了下面某个类的框架：

(a) Book (b) Date (c) Employee
(d) Vehicle (e) Object (f) Tree

为所选择的类编写输出操作符。

14.2.2 输入操作符 >> 的重载

与输出操作符类似，输入操作符的第一个形参是一个引用，指向它要读的流，并且返回的也是对同一个流的引用。它的第二个形参是对要读入的对象的非 `const` 引用，该形参必须为非 `const`，因为输入操作符的目的是将数据读到这个对象中。



更重要但通常重视不够的是，输入和输出操作符有如下区别：输入操作符必须处理错误和文件结束的可能性。

515

1. `Sales_item` 的输入操作符

`Sales_item` 的输入操作符如下：

```
istream&
operator>>(istream& in, Sales_item& s)
{
    double price;
    in >> s.isbn >> s.units_sold >> price;
    // check that the inputs succeeded
    if (in)
        s.revenue = s.units_sold * price;
    else
        s = Sales_item(); // input failed: reset object to default state
```

```
    return in;
}
```

这个操作符从 `istream` 形参中读取三个值：一个 `string` 值，存储到 `Sales_item` 形参的 `isbn` 成员中；一个 `unsigned` 值，存储到 `Sales_item` 形参的 `units_sold` 成员中；一个 `double` 值，存储到 `Sales_item` 形参的 `price` 成员中。假定读取成功，操作符用 `price` 和 `units_sold` 来设置 `Sales_item` 对象的 `revenue` 成员。

2. 输入期间的错误

`Sales_item` 的输入操作符将读入所期望的值并检查是否发生错误。可能发生的错误包括如下种类：

(1) 任何读操作都可能因为提供的值不正确而失败。例如，读入 `isbn` 之后，输入操作符将期望下两项是数值型数据。如果输入非数值型数据，这次的读入以及流的后续使用都将失败。

(2) 任何读入都可能碰到输入流中的文件结束或其他一些错误。

我们无需检查每次读入，只在使用读入数据之前检查一次即可：

```
// check that the inputs succeeded
if (in)
    s.revenue = s.units_sold * price;
else
    s = Sales_item(); // input failed: reset object to default state
```

如果这些读入有一个失败了，则 `price` 可能没有初始化。因此，在使用 `price` 之前，我们需要检查输入流是否仍有效。如果有效，就进行计算并将结果存储到 `revenue` 中；如果出现了错误，我们不用关心是哪个输入失败了，相反，我们将整个对象复位，就好像它是一个空 `Sales_item` 对象，具体做法是创建一个新的、未命名的、用默认构造函数构造的 `Sales_item` 对象并将它赋值给 `s`。赋值之后，`s` 的 `isbn` 成员是一个空 `string`，它的 `revenue` 和 `units_sold` 成员为 0。

516

3. 处理输入错误

如果输入操作符检测到输入失败了，则确保对象处于可用和一致的状态是个好做法。如果对象在错误发生之前已经写入了部分信息，这样做就特别重要。

例如，在 `Sales_item` 的输入操作符中，可能成功地读入了一个新的 `isbn`，然后遇到了流错误。在读入 `isbn` 之后发生错误意味着旧对象的 `units_sold` 和 `revenue` 成员没变，结果会将另一个 `isbn` 与那个数据关联。

在这个操作符中，如果发生了错误，就将形参恢复为空 `Sales_item` 对象，以避免给它一个无效状态。用户如果需要知道输入是否成功，可以测试流。即使用户忽略了输入可能错误，对象仍处于可用状态——它的成员都已经定义。类似地，对象将不会产生令人误解的结果——它的数据是内在一致的。



最佳
实践

设计输入操作符时，如果可能，要确定错误恢复措施，这很重要。

4. 指出错误

除了处理可能发生的任何错误之外，输入操作符还可能需要设置输入形参的条件状态（8.2

节)。我们的输入操作符相当简单——我们只关心读入期间可能发生的错误。如果读入成功，则输入操作符就是正确的而且不需要进行附加检查。

有些输入操作符的确需要进行附加检查。例如，我们的输入操作符可以检查读到的 isbn 格式是否恰当。也许我们已成功读取了数据，但这些数据不能恰当解释为 ISBN，在这种情况下，尽管从技术上说实际的 IO 是成功的，但输入操作符仍可能需要设置条件状态以指出失败。通常输入操作符仅需设置 failbit。设置 eofbit 意思是文件耗尽，设置 badbit 可以指出流被破坏，这些错误最好留给 IO 标准库自己来指出。

习题

习题 14.9 给定下述输入，描述 Sales_item 输入操作符的行为。

- (a) 0-201-99999-9 10 24.95
- (b) 10 24.95 0-201-99999-9

习题 14.10 下述 Sales_item 输入操作符有什么错误？

```
istream& operator>>(istream& in, Sales_item& s)
{
    double price;
    in >> s.isbn >> s.units_sold >> price;
    s.revenue = s.units_sold * price;
    return in;
}
```

如果将上题中的数据作为输入，将会发生什么？

习题 14.11 为 14.2.1 节习题中定义的 CheckoutRecord 类定义一个输入操作符，确保该操作符处理输入错误。

14.3 算术操作符和关系操作符

一般而言，将算术和关系操作符定义为非成员函数，像下面给出的 Sales_item 加法操作符一样：

```
// assumes that both objects refer to the same isbn
Sales_item
operator+(const Sales_item& lhs, const Sales_item& rhs)
{
    Sales_item ret(lhs); // copy lhs into a local object that we'll return
    ret += rhs;           // add in the contents of rhs
    return ret;           // return ret by value
}
```

517

加法操作符并不改变操作数的状态，操作数是对 const 对象的引用；相反，它产生并返回一个新的 Sales_item 对象，该对象初始化为 lhs 的副本。我们使用 Sales_item 的复合赋值操作符来加入 rhs 的值。



最佳
实践

注意，为了与内置操作符保持一致，加法返回一个右值，而不是一个引用。

算术操作符通常产生一个新值，该值是两个操作数的计算结果，它不同于任一操作数且在一个局部变量中计算，返回对那个变量的引用是一个运行时错误。



最佳实践 既定义了算术操作符又定义了相关复合赋值操作符的类，一般应使用复合赋值实现算术操作符。

根据复合赋值操作符（如+=）来实现算术操作符（如+），比其他方式更简单且更有效。例如，我们的 Sales_item 操作符。如果我们调用+=来实现+，则可以不创建和撤销一个临时量来保存+的结果。

习题

习题 14.12 编写 Sales_item 操作符，用+进行实际加法，而+=调用+。与本节中操作符的实现方法相比较，讨论这个方法的缺点。

习题 14.13 如果有，你认为 Sales_item 还应该支持哪些其他算术操作符？定义你认为该类应包含的那些。

14.3.1 相等操作符

通常，C++中的类使用相等操作符表示对象是等价的。即，它们通常比较每个数据成员，如果所有对应成员都相同，则认为两个对象相等。与这一设计原则一致，Sales_item 的相等操作符应比较 isbn 以及销售数据：

```
inline bool
operator==(const Sales_item &lhs, const Sales_item &rhs)
{
    // must be made a friend of Sales_item
    return lhs.units_sold == rhs.units_sold &&
           lhs.revenue == rhs.revenue &&
           lhs.same_isbn(rhs);
}
inline bool
operator!=(const Sales_item &lhs, const Sales_item &rhs)
{
    return !(lhs == rhs); // != defined in terms of operator==
}
```

这些函数的定义并不重要，重要的是这些函数所包含的设计原则：

- 如果类定义了==操作符，该操作符的含义是两个对象包含同样的数据。
- 如果类具有一个操作，能确定该类型的两个对象是否相等，通常将该函数定义为 operator== 而不是创造命名函数。用户将习惯于用==来比较对象，而且这样做比记住新名字更容易。
- 如果类定义了 operator==，它也应该定义 operator!=。用户会期待如果可以用某个操作符，则另一个也存在。
- 相等和不等操作符一般应该相互联系起来定义，让一个操作符完成比较对象的实际工作，而另一个操作符只是调用前者。



定义了 `operator==` 的类更容易与标准库一起使用。有些算法，如 `find`，默认使用 `==` 操作符，如果类定义了 `==`，则这些算法可以无须任何特殊处理而用于该类类型。

519

14.3.2 关系操作符

定义了相等操作符的类一般也具有关系操作符。尤其是，因为关联容器和某些算法使用小于操作符，所以定义 `operator<` 可能相当有用。

我们也许认为 `Sales_item` 类应该支持关系操作符，但恰恰相反，它很可能不应该支持关系操作符，原因有些微妙，值得了解。

正如第 15 章将要介绍的，我们可能想要使用关联容器来保存 `Sales_item` 事务。将对象放在容器中时，我们会希望它们按 ISBN 排序，而不会关心两个记录中的销售数据是否不同。

但是，如果将 `operator<` 定义为对 `isbn` 的比较，该定义将与前面 `==` 的定义不相容。如果有两个针对同一 ISBN 的事务，其中任意一个都不会小于另一个，然而，如果这两个对象中的销售数据不同，则它们就会不相等。但是，一般说来，如果有两个对象，其中任意一个都不小于另一个，则认为它们相等。

因为 `<` 的逻辑定义与 `==` 的逻辑定义不一致，所以根本不定义 `<` 会更好。第 15 章将会介绍想要将 `Sales_item` 对象存储到关联容器中时，怎样使用单独的命名函数来比较 `Sales_item` 对象。



关联容器以及某些算法，使用默认 `<` 操作符。一般而言，关系操作符，诸如相等操作符，应定义为非成员函数。

14.4 赋值操作符

13.2 节讨论了类类型对象对同类型其他对象的赋值。类赋值操作符接受类类型形参，通常，该形参是对类类型的 `const` 引用，但也可以是类类型或对类类型的非 `const` 引用。如果没有定义这个操作符，则编译器将合成它。类赋值操作符必须是类的成员，以便编译器可以知道是否需要合成一个。

可以为一个类定义许多附加的赋值操作符，这些赋值操作符会因右操作数类型的不同而不同。例如，标准库的类 `string` 定义了 3 个赋值操作符：除了接受 `const string&` 作为右操作数的类赋值操作符之外，类还定义了接受 C 风格字符串或 `char` 作为右操作数的赋值操作符，这些操作符可以这样使用：

```
string car ("Volks");
car = "Studebaker";    // string = const char*
string model;
model = 'T';           // string = char
```

为了支持这些操作符，`string` 类包含如下的成员：

```
// illustration of assignment operators for class string
class string {
public:
```

520

```

string& operator=(const string &);      // s1 = s2;
string& operator=(const char *);       // s1 = "str";
string& operator=(char);               // s1 = 'c';
// ....
};

```



赋值操作符可以重载。无论形参为何种类型，赋值操作符必须定义为成员函数，这一点与复合赋值操作符有所不同。

赋值必须返回对*this的引用

string 赋值操作符返回 string 引用，这与内置类型的赋值一致。而且，因为赋值返回一个引用，就不需要创建和撤销结果的临时副本。返回值通常是左操作数的引用，例如，这是 Sales_item 复合赋值操作符的定义：

```

// assumes that both objects refer to the same isbn
Sales_item& Sales_item::operator+=(const Sales_item& rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}

```



一般而言，赋值操作符与复合赋值操作符应返回左操作数的引用。

习题

习题 14.14 定义一个赋值操作符，将 isbn 赋值给 Sales_item 对象。

习题 14.15 为 14.2.1 节习题中介绍的 CheckoutRecord 类定义赋值操作符。

习题 14.16 CheckoutRecord 类还应该定义其他赋值操作符吗？如果是，解释哪些类型应该用作操作数并解释为什么。为这些类型实现赋值操作符。

521

14.5 下标操作符

可以从容器中检索单个元素的容器类一般会定义下标操作符，即 operator[]。标准库的类 string 和 vector 均是定义了下标操作符的类的例子。



下标操作符必须定义为类成员函数。

1. 提供读写访问

定义下标操作符比较复杂的地方在于，它在用作赋值的左右操作数时都应该能表现正常。下标操作符出现在左边，必须生成左值，可以指定引用作为返回类型而得到左值。只要下标操作符返回引用，就可用作赋值的任意一方。

可以对 const 和非 const 对象使用下标也是个好主意。应用于 const 对象时，返回值应为 const 引用，因此不能用作赋值的目标。



最佳实践 类定义下标操作符时，一般需要定义两个版本：一个为非 const 成员并返回引用，另一个为 const 成员并返回 const 引用。

2. 原型下标操作符

下面的类定义了下标操作符。为简单起见，假定 Foo 所保存的数据存储在一个 vector<int>中：

```
class Foo {
public:
    int &operator[] (const size_t);
    const int &operator[] (const size_t) const;
    // other interface members
private:
    vector<int> data;
    // other member data and private utility functions
};
```

下标操作符本身可能看起来像这样：

```
int& Foo::operator[] (const size_t index)
{
    return data[index]; // no range checking on index
}
const int& Foo::operator[] (const size_t index) const
{
    return data[index]; // no range checking on index
}
```

522

习题

习题 14.17 14.2.1 节习题中定义了一个 CheckoutRecord 类，为该类定义一个下标操作符，从等待列表中返回一个名字。

习题 14.18 讨论用下标操作符实现这个操作的优缺点。

习题 14.19 提出另一种方法定义这个操作。

14.6 成员访问操作符

为了支持指针型类，例如迭代器，C++ 语言允许重载解引用操作符 (*) 和箭头操作符 (->)。



注解 箭头操作符必须定义为类成员函数。解引用操作符不要求定义为成员，但将它作为成员一般也是正确的。

1. 构建更安全的指针

解引用操作符和箭头操作符常用在实现智能指针 (13.5.1 节) 的类中。作为例子，假定想要定义一个类类型表示指向第 12 章中编写的 Screen 类型对象的指针，将该类命名为 ScreenPtr。

ScreenPtr 类将类似于我们的第二个 HasPtr 类。ScreenPtr 的用户将会传递一个指针，该指针指向动态分配的 Screen，ScreenPtr 类将拥有该指针，并安排在指向基础对象的最后一个

ScreenPtr 消失时删除基础对象。另外，不用为 ScreenPtr 类定义默认构造函数。因此，我们知道一个 ScreenPtr 对象将总是指向一个 Screen 对象，不会有未绑定的 ScreenPtr，这一点与内置指针不同。应用程序可以使用 ScreenPtr 对象而无须首先测试它是否指向一个 Screen 对象。

像 HasPtr 类一样，ScreenPtr 类将对其指针进行使用计数。我们将定义一个伙伴类保存指针及其相关使用计数：

```
// private class for use by ScreenPtr only
class ScrPtr {
    friend class ScreenPtr;
    Screen *sp;
    size_t use;
    ScrPtr(Screen *p): sp(p), use(1) { }
    ~ScrPtr() { delete sp; }
};
```

这个类看来很像 U_Ptr 类并且作用同样。ScrPtr 保存指针及其相关使用计数。将 ScreenPtr 设为友元，以便 ScreenPtr 可以访问使用计数。ScreenPtr 类将管理使用计数：

523

```
/*
 * smart pointer: Users pass to a pointer to a dynamically allocated Screen, which
 * is automatically destroyed when the last ScreenPtr goes away
 */
class ScreenPtr {
public:
    // no default constructor: ScreenPtrs must be bound to an object
    ScreenPtr(Screen *p): ptr(new ScrPtr(p)) { }
    // copy members and increment the use count
    ScreenPtr(const ScreenPtr &orig):
        ptr(orig.ptr) { ++ptr->use; }
    ScreenPtr& operator=(const ScreenPtr&);
    // if use count goes to zero, delete the ScrPtr object
    ~ScreenPtr() { if (--ptr->use == 0) delete ptr; }
private:
    ScrPtr *ptr; // points to use-counted ScrPtr class
};
```

因为没有默认构造函数，所以 ScreenPtr 类型的每个对象必须提供一个初始化函数，初始化函数必须是另一个 ScreenPtr 对象或指向动态分配的 Screen 的指针。构造函数分配一个新的 ScrPtr 对象以保存那个指针及相关的使用计数。

试图定义一个不带初始化式的 ScreenPtr 对象是错误的：

```
ScreenPtr p1; // error: ScreenPtr has no default constructor
ScreenPtr ps(new Screen(4,4)); // ok: ps points to a copy of myScreen
```

2. 支持指针操作

指针支持的基本操作有解引用操作和箭头操作。我们的类可以这样定义这些操作：

```
class ScreenPtr {
public:
    // constructor and copy control members as before
    Screen &operator*() { return *ptr->sp; }
    Screen *operator->() { return ptr->sp; }
```

```

    const Screen &operator*() const { return *ptr->sp; }
    const Screen *operator->() const { return ptr->sp; }
private:
    ScrPtr *ptr;          // points to use-counted ScrPtr class
};

```

3. 重载解引用操作符

解引用操作符是个一元操作符。在这个类中，解引用操作符定义为成员，因此没有显式形参，该操作符返回对 ScreenPtr 所指向的 Screen 的引用。

像下标操作符一样，我们需要解引用操作符的 const 和非 const 版本。它们的区别在于返回类型：const 成员返回 const 引用以防止用户改变基础对象。

524

4. 重载箭头操作符

箭头操作符与众不同。它可能表现得像二元操作符一样：接受一个对象和一个成员名。对对象解引用以获取成员。不管外表如何，箭头操作符不接受显式形参。

这里没有第二个形参，因为->的右操作数不是表达式，相反，是对应着类成员的一个标识符。没有明显可行的途径将一个标识符作为形参传递给函数，相反，由编译器处理获取成员的工作。

当这样编写时：

```
point -> action();
```

由于优先级规则，它实际等价于编写：

```
(point -> action)();
```

换句话说，我们想要调用的是对 point->action 求值的结果。编译器这样对该代码进行求值：

(1) 如果 point 是一个指针，指向具有名为 action 的成员的类对象，则编译器将代码编译为调用该对象的 action 成员。

(2) 否则，如果 action 是定义了 operator->操作符的类的一个对象，则 point->action 与 point.operator->()->action 相同。即，执行 point 的 operator->()，然后使用该结果重复这三步。

(3) 否则，代码出错。

5. 使用重载箭头

可以这样使用 ScreenPtr 对象访问 Screen 对象的成员：

```

ScreenPtr p(&myScreen);    // copies the underlying Screen
p->display(cout);

```

因为 p 是一个 ScreenPtr 对象，p->display 的含义与对 (p.operator->())->display 求值相同。对 p.operator->() 求值将调用 ScreenPtr 类的 operator->，它返回指向 Screen 对象的指针，该指针用于获取并运行 ScreenPtr 所指对象的 display 成员。

6. 对重载箭头的返回值的约束



重载箭头操作符必须返回指向类类型的指针，或者返回定义了自己的箭头操作符的类类型对象。

525

如果返回类型是指针，则内置箭头操作符可用于该指针，编译器对该指针解引用并从结果对象获取指定成员。如果被指向的类型没有定义那个成员，则编译器产生一个错误。

如果返回类型是类类型的其他对象（或是这种对象的引用），则将递归应用该操作符。编译器检查返回对象所属类型是否具有成员箭头，如果有，就应用那个操作符；否则，编译器产生一个错误。这个过程继续下去，直到返回一个指向带有指定成员的对象指针，或者返回某些其他值，在后一种情况下，代码出错。

习题

习题 14.20 在 `ScreenPtr` 类的概略定义中，声明但没有定义赋值操作符。请实现 `ScreenPtr` 赋值操作符。

习题 14.21 定义一个类，该类保存一个指向 `ScreenPtr` 的指针。为该类定义一个重载的箭头操作符。

习题 14.22 智能指针可能应该定义相等操作符和不等操作符，以便测试两个指针是否相等或不等。将这些操作加入到 `ScreenPtr` 类。

14.7 自增操作符和自减操作符

自增（++）和自减（--）操作符经常由诸如迭代器这样的类实现，这样的类提供类似于指针的行为来访问序列中的元素。例如，可以定义一个类，该类指向一个数组并为该数组中的元素提供访问检查。理想情况下，带访问检查的指针类可用于任意类型的数组，这一点的实现我们将在第 16 章介绍类模板时学习。现在，我们的类将处理 `int` 数组：

```
/*
 * smart pointer: Checks access to elements throws an out_of_range
 *                  exception if attempt to access a nonexistent element
 * users allocate and free the array
 */
class CheckedPtr {
public:
    // no default constructor; CheckedPtrs must be bound to an object
    CheckedPtr(int *b, int *e): beg(b), end(e), curr(b) { }
    // dereference and increment operations
private:
    int* beg;    // pointer to beginning of the array
    int* end;    // one past the end of the array
    int* curr;   // current position within the array
};
```

526

像 `ScreenPtr` 一样，这个类没有默认构造函数。创建一个 `CheckedPtr` 对象时，必须提供指向数组的指针。一个 `CheckedPtr` 对象有三个数据成员：`beg`，指向数组的第一个元素；`end`，指向数组的末端；`curr`，指向 `CheckedPtr` 对象当前引用的数组元素。

构造函数的参数是两个指针：一个指向数组的开头，另一个指向数组的末端。构造函数用这两个指针初始化 `beg` 和 `end`，并将 `curr` 初始化为指向第一个元素。

1. 定义自增/自减操作符



最佳
实践

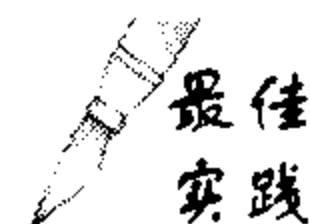
C++语言不要求自增操作符或自减操作符一定作为类的成员，但是，因为这些操作符改变操作对象的状态，所以更倾向于将它们作为成员。

在为类定义重载的自增操作符和自减操作符之前，还必须考虑另一件事情。对内置类型而言，自增操作符和自减操作符有前缀和后缀两种形式。毫不奇怪，也可以为我们自己的类定义自增操作符和自减操作符的前缀和后缀实例。我们首先介绍前缀形式，然后实现后缀形式。

2. 定义前自增/前自减操作符

前缀式操作符的声明看起来像这样：

```
class CheckedPtr {
public:
    CheckedPtr& operator++();           // prefix operators
    CheckedPtr& operator--();
    // other members as before
};
```



最佳
实践

为了与内置类型一致，前缀式操作符应返回被增量或减量对象的引用。

这个自增操作符根据 end 检查 curr，从而确保用户不能将 curr 增量到超过数组的末端。如果 curr 增量到超过 end，就抛出一个 out_of_range 异常；否则，将 curr 加 1 并返回对象引用：

```
// prefix: return reference to incremented/decremented object
CheckedPtr& CheckedPtr::operator++()
{
    if (curr == end)
        throw out_of_range
            ("increment past the end of CheckedPtr");
    ++curr;           // advance current state
    return *this;
}
```

除了将 curr 减 1 并检查是否会减到 beg，自减操作符的行为与自增操作符类似：

```
CheckedPtr& CheckedPtr::operator--()
{
    if (curr == beg)
        throw out_of_range
            ("decrement past the beginning of CheckedPtr");
    --curr;           // move current state back one element
    return *this;
}
```

527

3. 区别操作符的前缀和后缀形式

同时定义前缀式操作符和后缀式操作符存在一个问题：它们的形参数目和类型相同，普通重载不能区别所定义的是前缀式操作符还是后缀式操作符。

为了解决这一问题，后缀式操作符函数接受一个额外的（即，无用的）int 型形参。使用后缀式操作符时，编译器提供 0 作为这个形参的实参。尽管我们的前缀式操作符函数可以使用这个

额外的形参，但通常不应该这样做。那个形参不是后缀式操作符的正常工作所需要的，它的唯一目的是使后缀函数与前缀函数区别开来。

4. 定义后缀式操作符

现在将后缀式操作符加到 CheckedPtr:

```
class CheckedPtr {
public:
    // increment and decrement
    CheckedPtr operator++(int);          // postfix operators
    CheckedPtr operator--(int);
    // other members as before
};
```

 **最佳实践** 为了与内置操作符一致，后缀式操作符应返回旧值（即，尚未自增或自减的值），并且，应作为值返回，而不是返回引用。

后缀式操作符可以这样实现：

```
// postfix: increment/decrement object but return unchanged value
CheckedPtr CheckedPtr::operator++(int)
{
    // no check needed here, the call to prefix increment will do the check
    CheckedPtr ret(*this); // save current value
    ++*this;               // advance one element, checking the increment
    return ret;            // return saved state
}
CheckedPtr CheckedPtr::operator--(int)
{
    // no check needed here, the call to prefix decrement will do the check
    CheckedPtr ret(*this); // save current value
    --*this;               // move backward one element and check
    return ret;            // return saved state
}
```

528

操作符的后缀式比前缀式复杂一点，必须记住对象在加 1/减 1 之前的当前状态。这些操作符定义了一个局部 CheckedPtr 对象，将它初始化为 *this 的副本，即 ret 是这个对象当前状态的副本。

保存了当前状态的副本后，操作符调用自己的前缀式操作符分别进行加 1 或减 1：

```
++*this
```

调用这个对象的 CheckedPtr 前缀自增操作符，该操作符检查自增是否安全并将 curr 加 1 或抛出一个异常。假定不抛出异常，前自增操作符函数以返回存储在 ret 的副本而结束。因此，返回之后，对象本身加了 1，但返回的是尚未自增的原值。

因为通过调用前缀式版本实现这些操作符，不需要检查 curr 是否在范围之内，那个检查以及必要的 throw，在相应的前缀式操作符中完成。

 **注解**

因为不使用 int 形参，所以没有对其命名。

5. 显式调用前缀式操作符

正如在 14.1 节所见，可以显式调用重载操作符而不是将它作为操作符用在表达式中。如果想要使用函数调用来调用后缀式操作符，必须给出一个整型实参值：

```
CheckedPtr parr(ia, ia + size);    // ia points to an array of ints
parr.operator++(0);               // call postfix operator++
parr.operator++();                // call prefix operator++
```

所传递的值通常被忽略，但该值是必要的，用于通知编译器需要的是后缀式版本。



最佳
实践

一般而言，最好前缀式和后缀式都定义。只定义前缀式或只定义后缀式的类，将会让习惯于使用两种形式的用户感到奇怪。

529

习题

- 习题 14.23** CheckedPtr 类表示指向数组的指针。为该类重载下标操作符和解引用操作符。使操作符确保 CheckedPtr 有效：它应该不可能对超出数组末端的元素进行解引用或索引。
- 习题 14.24** 上题中定义和解引用操作符或下标操作符，是否也应该检查对数组起点之前的元素进行的解引用或索引？解释你的答案。
- 习题 14.25** 为了表现得像数组指针，CheckedPtr 类应实现相等和关系操作符，以便确定两个 CheckedPtr 对象是否相等，或者一个小于另一个，诸如此类。为 CheckedPtr 类增加这些操作。
- 习题 14.26** 为 CheckedPtr 类定义加法或减法，以便这些操作符实现指针运算（4.2.4 节）。
- 习题 14.27** 讨论允许将空数组实参传给 CheckedPtr 构造函数的优缺点。
- 习题 14.28** 没有定义自增和自减操作符的 const 版本，为什么？
- 习题 14.29** 我们也没有实现箭头操作符，为什么？
- 习题 14.30** 定义一个 CheckedPtr 版本，保存 Screen 数组。为该类实现重载的自增、自减、解引用、箭头等操作符。

14.8 调用操作符和函数对象

可以为类类型的对象重载函数调用操作符。一般为表示操作的类重载调用操作符。例如，可以定义名为 absInt 的结构，该结构封装将 int 类型的值转换为绝对值的操作：

```
struct absInt {
    int operator() (int val) {
        return val < 0 ? -val : val;
    }
};
```

这个类很简单，它定义了一个操作：函数调用操作符，该操作符有一个形参并返回形参的绝对值。

通过为类类型的对象提供一个实参表而使用调用操作符，所用的方式看起来像一个函数调用：

```
int i = -42;
```

530

```
absInt absObj; // object that defines function call operator
unsigned int ui = absObj(i); // calls absInt::operator(int)
```

尽管 `absObj` 是一个对象而不是函数，我们仍然可以“调用”该对象，效果是运行由 `absObj` 对象定义的重载调用操作符，该操作符接受一个 `int` 值并返回它的绝对值。



函数调用操作符必须声明为成员函数。一个类可以定义函数调用操作符的多个版本，由形参的数目或类型加以区别。

定义了调用操作符的类，其对象常称为**函数对象**（function object），即它们是行为类似函数的对象。

习题

习题 14.31 定义一个函数对象执行“如果-则-否则”操作：该函数对象应接受三个形参，它应该测试第一个形参，如果测试成功，就返回第二个形参，否则，就返回第三个形参。

习题 14.32 一个重载的函数调用操作符可以接受多少个操作数？

14.8.1 将函数对象用于标准库算法

函数对象经常用作通用算法的实参。在 11.2.3 节解决的问题就是这样一个例子。那个程序分析一组故事中的单词，计算有多少个单词长度在 6 字符以上。该解决方案的一个部分包括定义一个函数以确定给定 `string` 的长度是否大于 6 字符：

```
// determine whether a length of a given word is 6 or more
bool GT6(const string &s)
{
    return s.size() >= 6;
}
```

使用 `GT6` 作为传给 `count_if` 算法的实参，以计算使 `GT6` 返回 `true` 的单词的数目：

```
vector<string>::size_type wc =
    count_if(words.begin(), words.end(), GT6);
```

1. 函数对象可以比函数更灵活

我们的实现有个严重问题：它将 6 这个数字固化在 `GT6` 函数的定义中。`count_if` 算法运行只用一个形参且返回 `bool` 的函数。理想情况下，应传递 `string` 和我们想要的长度进行测试。

531 通过该方式，可以使用同一代码对不同长度的字符串进行计数。

通过将 `GT6` 定义为带函数调用成员的类，可以获得所需的灵活性。将这个类命名为 `GT_cls` 以区别于函数：

```
// determine whether a length of a given word is longer than a stored bound
class GT_cls {
public:
    GT_cls(size_t val = 0): bound(val) { }
    bool operator()(const string &s)
        { return s.size() >= bound; }
private:
```

```
std::string::size_type bound;
};
```

这个类有一个构造函数，该构造函数接受一个整型值并用名为 `bound` 的成员记住那个值。如果没有提供值，构造函数将 `bound` 置 0。该类也定义了调用操作符，接受一个 `string` 参数并返回一个 `bool`。调用操作符将 `string` 实参的长度与数据成员 `bound` 中存储的值相比较。

2. 使用 `GT_cls` 函数对象

可以像前面一样进行计数，但这一次使用 `GT_cls` 类型的对象而不是 `GT6` 函数：

```
cout << count_if(words.begin(), words.end(), GT_cls(6))
      << " words 6 characters or longer" << endl;
```

这个 `count_if` 调用传递一个 `GT_cls` 类型的临时对象而不再是名为 `GT6` 的函数。用整型值 6 来初始化那个临时对象，构造函数将这个值存储在 `bound` 成员中。现在，`count_if` 每次调用它的函数形参时，它都使用 `GT_cls` 的调用操作符，该调用操作符根据 `bound` 的值测试其 `string` 实参的长度。

使用函数对象，容易修改程序以根据其他值进行测试，只需为传给 `count_if` 的对象改变构造函数实参即可。例如，这样修改程序，就可以计算长度在 5 个字符以上的单词数：

```
cout << count_if(words.begin(), words.end(), GT_cls(5))
      << " words 5 characters or longer" << endl;
```

更为有用的是，还可以计算长度在 1 至 10 个字符的单词数：

```
for (size_t i = 0; i != 11; ++i)
    cout << count_if(words.begin(), words.end(), GT(i))
          << " words " << i
          << " characters or longer" << endl;
```

如果使用函数代替函数对象来编写这个程序，可能需要编写 10 个不同的函数，每个函数测试一个不同的值。

532

习题

- 习题 14.33 使用标准库算法和 `GT_cls` 类，编写一个程序查找序列中第一个比指定值大的元素。
- 习题 14.34 编写类似于 `GT_cls` 的函数对象类，但测试两个值是否相等。使用该对象和标准库算法编写程序，替换序列中给定值的所有实例。
- 习题 14.35 编写类似于 `GT_cls` 的类，但测试给定的长度是否与其边界相匹配。使用该对象重写 11.2.3 节中的程序，以便报告输入中有多少单词的长度在 1 到 10 之间（含 1 和 10）。
- 习题 14.36 修改前一程序以报告长度在 1 到 9 之间以及 10 以上的单词的数目。

14.8.2 标准库定义的函数对象

标准库定义了一组算术、关系与逻辑函数对象类，表 14-3 列出了这些类。标准库还定义了一组函数适配器，使我们能够特化或者扩展标准库所定义的以及自定义的函数对象类。这些标准库函数对象类型是在 `functional` 头文件中定义的。

表 14-3 标准库函数对象		
类 型	函数对象	所应用的操作符
算术函数对象类型	plus<Type>	+
	minus<Type>	-
	multiplies<Type>	*
	divides<Type>	/
	modulus<Type>	%
	negate<Type>	-
关系函数对象类型	equal_to<Type>	==
	not_equal_to<Type>	!=
	greater<Type>	>
	greater_equal<Type>	>=
	less<Type>	<
	less_equal<Type>	<=
	logical_and<Type>	&&
逻辑函数对象类型	logical_or<Type>	
	logical_not<Type>	!

1. 每个类表示一个给定操作符

每个标准库函数对象类表示一个操作符，即，每个类都定义了应用命名操作的调用操作符。例如，plus 是表示加法操作符的模板类型。plus 模板中的调用操作符对一对操作数应用+运算。不同的函数对象定义了执行不同操作的调用操作符。正如 plus 定义了执行+操作符的调用操作符，modulus 类定义了应用二元操作符%的调用操作符，equal_to 类应用==，等等。

有两个一元函数对象（unary function-object）类：一元减（negate<Type>）和逻辑非（logical_not<Type>）。其余的标准库函数对象都是表示二元操作符的二元函数对象（binary function-object）类。为二元操作符定义的调用操作符需要两个给定类型的形参，而一元函数对象类型定义了接受一个实参的调用操作符。

2. 表示操作数类型的模板类型

每个函数对象类都是一个类模板，我们需要为该模板提供一个类型。正如从诸如 vector 的顺序容器所了解的，类模板是可以用于不同类型的类。函数对象类的模板类型指定调用操作符的形参类型。

533

例如，plus<string>将 string 加法操作符应用于 string 对象，对于 plus<int>，操作数是 int 值，plus<Sales_item>将+应用于 Sales_item 对象，依次类推：

```
plus<int> intAdd;           // function object that can add two int values
negate<int> intNegate;      // function object that can negate an int value
// uses intAdd::operator(int, int) to add 10 and 20
int sum = intAdd(10, 20);   // sum = 30
// uses intNegate::operator(int) to generate -10 as second parameter
// to intAdd::operator(int, int)
sum = intAdd(10, intNegate(10)); // sum = 0
```

3. 在算法中使用标准库函数对象

函数对象常用于覆盖算法使用的默认操作符。例如，sort 默认使用 operator<按升序对容器进行排序。为了按降序对容器进行排序，可以传递函数对象 greater。该类将产生一个调用操

作符，调用基础对象的大于操作符。如果 `svec` 是一个 `vector<string>` 对象，以下代码

```
// passes temporary function object that applies > operator to two strings
sort(svec.begin(), svec.end(), greater<string>());
```

将按降序对 `vector` 进行排序。像通常那样，传递一对迭代器以指明被排序序列。第三个实参用于传递比较元素的谓词（11.2.3 节）函数。该实参是 `greater<string>` 类型的临时对象，是一个将 `>` 操作符应用于两个 `string` 操作数的函数对象。

534

14.8.3 函数对象的函数适配器

标准库提供了一组函数适配器（function adapter），用于特化和扩展一元和二元函数对象。函数适配器分为如下两类：

(1) 绑定器（binder），是一种函数适配器，它通过将一个操作数绑定到给定值而将二元函数对象转换为一元函数对象。

(2) 求反器（negator），是一种函数适配器，它将谓词函数对象的真值求反。

标准库定义了两个绑定器适配器：`bind1st` 和 `bind2nd`。每个绑定器接受一个函数对象和一个值。正如你可能想到的，`bind1st` 将给定值绑定到二元函数对象的第一个实参，`bind2nd` 将给定值绑定到二元函数对象的第二个实参。例如，为了计算一个容器中所有小于或等于 10 的元素的个数，可以这样给 `count_if` 传递值：

```
count_if(vec.begin(), vec.end(),
         bind2nd(less_equal<int>(), 10));
```

传给 `count_if` 的第三个实参使用 `bind2nd` 函数适配器，该适配器返回一个函数对象，该对象用 10 作右操作数应用 `<=` 操作符。这个调用计算输入范围中小于或等于 10 的元素的个数。

标准库还定义了两个求反器：`not1` 和 `not2`。你可能已经想到的，`not1` 将一元函数对象的真值求反，`not2` 将二元函数对象的真值求反。

为了对 `less_equal` 函数对象的绑定求反，可以编写这样的代码：

```
count_if(vec.begin(), vec.end(),
         not1(bind2nd(less_equal<int>(), 10)));
```

这里，首先将 `less_equal` 对象的第二个操作数绑定到 10，实际上是将该二元操作转换为一元操作。再用 `not1` 对操作的返回值求反，效果是测试每个元素是否 `<=10`。然后，对结果真值求反。这个 `count_if` 调用的效果是对不 `<=10` 的那些元素进行计数。

习题

习题 14.37 使用标准库函数对象和函数适配器，定义一个对象用于：

- (a) 查找大于 1024 的所有值。
- (b) 查找不等于 pooh 的所有字符串。
- (c) 将所有值乘以 2。

习题 14.38 最后一个 `count_if` 调用中，用 `not1` 将 `bind2nd(less_equal<int>(), 10)` 的结果

求反。为什么使用 not1 而不用 not2?

习题 14.39 使用标准库函数对象代替 GT_cls 来查找指定长度的单词。

14.9 转换与类类型

在 12.4.4 节介绍过，可用一个实参调用的非 explicit 构造函数定义一个隐式转换。当提供了实参类型的对象而需要一个类类型的对象时，编译器将使用该转换。这种构造函数定义了到类类型的转换。

除了定义到类类型的转换之外，我们还可以定义从类类型的转换。即，我们可以定义转换操作符，给定类类型的对象，该操作符将产生其他类型的对象。像其他转换一样，编译器将自动应用这个转换。在介绍如何定义这种转换之前，将说明它们为什么可能有用。

14.9.1 转换为什么有用

假定想要定义一个名为 SmallInt 的类，该类实现安全小整数，这个类将使我们能够定义对象以保存与 8 位 unsigned char 同样范围的值，即，0 到 255。这个类可以捕获下溢和上溢错误，因此使用起来比内置 unsigned char 更安全。

我们希望这个类定义 unsigned char 支持的所有操作。具体而言，我们想要定义 5 个算术操作符 (+, -, *, /, %) 及其对应的复合赋值操作符，4 个关系操作符 (<, <=, >, >=)，以及相等操作符 (==, !=)。显然，需要定义 16 个操作符。

1. 支持混合类型表达式

而且，我们希望可以在混合模式表达式中使用这些操作符。例如，应该可以将两个 SmallInt 对象相加，也可以将任意算术类型加到 SmallInt。通过为每个操作符定义三个实例来达到目标：

```
int operator+(int, const SmallInt&);
int operator+(const SmallInt&, int);
SmallInt operator+(const SmallInt&, const SmallInt&);
```

因为存在从任意算术类型到 int 的转换，这三个函数可以涵盖支持 SmallInt 对象的混合模式使用的要求。但是，这个设计仅仅接近内置整数运算的行为，它不能适当处理浮点类型的混合模式操作，也不能适当支持 long、unsigned int 或 unsigned long 的加运算。问题在于这个设计将所有算术类型（甚至包括那些比 int 大的）转换为 int 并进行 int 加运算。

2. 转换减少所需操作符的数目

即使忽略浮点或大整型操作数的问题，如果来实现这个设计，也必须定义 48 个操作符！幸好，C++ 提供了一种机制，利用这种机制，一个类可以定义自己的转换，应用于其类类型对象。对 SmallInt 而言，可以定义一个从 SmallInt 到 int 类型的转换。如果定义了该转换，则无须再定义任何算术、关系或相等操作符。给定到 int 的转换，SmallInt 对象可以用在任何可用 int 值的地方。

如果存在一个到 int 的转换，则以下代码：

```
SmallInt si(3);
```

```
si + 3.14159;           // convert si to int, then convert to double
```

可这样确定:

- (1) 将 `si` 转换为 `int` 值。
- (2) 将所得 `int` 值结果转换为 `double` 值并与双精度字面值常量 `3.14159` 相加, 得到 `double` 值。

14.9.2 转换操作符

转换操作符 (conversion operator) 是一种特殊的类成员函数。它定义将类类型值转变为其他类型值的转换。转换操作符在类定义体内声明, 在保留字 `operator` 之后跟着转换的目标类型:

```
class SmallInt {
public:
    SmallInt(int i = 0): val(i)
    { if (i < 0 || i > 255)
        throw std::out_of_range("Bad SmallInt initializer");
    }
    operator int() const { return val; }
private:
    std::size_t val;
};
```

转换函数采用如下通用形式:

```
operator type();
```

这里, `type` 表示内置类型名、类类型名或由类型别名所定义的名字。对任何可作为函数返回类型的类型 (除了 `void` 之外) 都可以定义转换函数。一般而言, 不允许转换为数组或函数类型, 转换为指针类型 (数据和函数指针) 以及引用类型是可以的。

537



转换函数必须是成员函数, 不能指定返回类型, 并且形参表必须为空。

下述所有声明都是错误的:

```
operator int(SmallInt &);           // error: nonmember

class SmallInt {
public:
    int operator int();               // error: return type
    operator int(int = 0);            // error: parameter list
    // ...
};
```

虽然转换函数不能指定返回类型, 但是每个转换函数必须显式返回一个指定类型的值。例如, `operator int` 返回一个 `int` 值; 如果定义 `operator Sales_item`, 它将返回一个 `Sales_item` 对象, 诸如此类。



转换函数一般不应该改变被转换的对象。因此, 转换操作符通常应定义为 `const` 成员。

1. 使用类类型转换

只要存在转换，编译器将在可以使用内置转换的地方自动调用它（5.12.1 节）：

- 在表达式中：

```
SmallInt si;
double dval;
si >= dval      // si converted to int and then convert to double
```

- 在条件中：

```
if (si)          // si converted to int and then convert to bool
```

- 将实参传给函数或从函数返回值：

```
int calc(int);
SmallInt si;
int i = calc(si); // convert si to int and call calc
```

- 作为重载操作符的操作数：

```
// convert si to int then call operator<< on the int value
cout << si << endl;
```

538

- 在显式类型转换中：

```
int ival;
SmallInt si = 3.541;
// instruct compiler to cast si to int
ival = static_cast<int>(si) + 3;
```

2. 类类型转换和标准转换

使用转换函数时，被转换的类型不必与所需要的类型完全匹配。必要时可在类类型转换之后跟上标准转换以获得想要的类型。例如，在一个 SmallInt 对象与一个 double 值的比较中：

```
SmallInt si;
double dval;
si >= dval      // si converted to int and then convert to double
```

首先将 si 从 SmallInt 对象转换为 int 值，然后将该 int 值转换为 double 值。

3. 只能应用一个类类型转换



类类型转换之后不能再跟另一个类类型转换。如果需要多个类类型转换，则代码将出错。

例如，假定有另一个类 Integral，它可以转换为 SmallInt 但不能转换为 int：

```
// class to hold unsigned integral values
class Integral {
public:
    Integral(int i = 0): val(i) { }
    operator SmallInt() const { return val % 256; }
private:
    std::size_t val;
};
```

可以在需要 `SmallInt` 的地方使用 `Integral`，但不能在需要 `int` 的地方使用 `Integral`：

```
int calc(int);
Integral intVal;
SmallInt si(intVal); // ok: convert intVal to SmallInt and copy to si
int i = calc(si);    // ok: convert si to int and call calc
int j = calc(intVal); // error: no conversion to int from Integral
```

创建 `si` 时使用 `SmallInt` 复制构造函数。首先调用 `Integral` 转换操作符产生一个 `SmallInt` 类型的临时值，将 `int_val` 对象转换为 `SmallInt` 对象。然后（合成的）复制构造函数使用该对象值初始化 `si`。

539

第一个 `calc` 调用也是正确的：将实参 `si` 自动转换为 `int`，然后将 `int` 值传给函数。

第二个 `calc` 调用是错误的：没有从 `Integral` 到 `int` 的直接转换。从 `Integral` 获得 `int` 需要两次类类型转换：首先从 `Integral` 到 `SmallInt`，然后从 `SmallInt` 到 `int`。但是，语言只允许一次类类型转换，所以该调用出错。

4. 标准转换可放在类类型转换之前

使用构造函数执行隐式转换（12.4.4 节）的时候，构造函数的形参类型不必与所提供的类型完全匹配。例如，下面的代码调用 `SmallInt` 类中定义的构造函数 `SmallInt(int)` 将 `sobj` 转换为 `SmallInt` 类型：

```
void calc(SmallInt);
short sobj;
// sobj promoted from short to int
// that int converted to SmallInt through the SmallInt(int) constructor
calc(sobj);
```

如果需要，在调用构造函数执行类类型转换之前，可将一个标准转换序列应用于实参。为了调用函数 `calc()`，应用标准转换将 `dobj` 从 `double` 类型转换为 `int` 类型，然后调用构造函数 `SmallInt(int)` 将转换结果转换为 `SmallInt` 类型。

习题

习题 14.40 编写可将 `Sales_item` 对象转换为 `string` 类型和 `double` 类型的操作符。你认为这些操作符应返回什么值？你认为定义这些操作符是个好办法吗？解释你的结论。

习题 14.41 解释这两个转换操作符之间的不同：

```
class Integral {
public:
    const int();
    int() const;
};
```

这两个转换操作符是否太严格了？如果是，怎样使得转换更通用一些？

习题 14.42 为 14.2.1 节习题中的 `CheckoutRecord` 类定义到 `bool` 的转换操作符。

习题 14.43 解释 `bool` 转换操作符做了什么。这是这个 `CheckoutRecord` 类型转换唯一可能的含义吗？解释你是否认为这个转换是一种转换操作的良好使用。

540

14.9.3 实参匹配和转换



本章其余部分讨论比较高级的主题。在第一次阅读时可跳过这些内容。

类类型转换可能是实现和使用类的一个好处。通过为 `SmallInt` 定义到 `int` 的转换，能够更容易实现和使用 `SmallInt` 类。`int` 转换使 `SmallInt` 的用户能够对 `SmallInt` 对象使用所有算术和关系操作符，而且，用户可以安全编写将 `SmallInt` 和其他算术类型混合使用的表达式。定义一个转换操作符就能代替定义 48 个（或更多）重载操作符，类实现者的工作就简单多了。

类类型转换也可能是编译时错误的一大来源。当从一个类型转换到另一类型有多种方式时，问题就出现了。如果有几个类类型转换可以使用，编译器必须决定对给定表达式使用哪一个。在这一节，我们介绍怎样用类类型转换将实参和对应形参相匹配。首先介绍非重载函数的形参匹配，然后介绍重载函数的形参匹配。



如果小心使用，类类型转换可以大大简化类代码和用户代码。如果使用得太过自由，类类型转换会产生令人迷惑的编译时错误，这些错误难以理解而且难以避免。

1. 实参匹配和多个转换操作符

为了举例说明类类型值的转换怎样与函数匹配相互作用，我们给 `SmallInt` 类加上另外两个转换，包括接受一个 `double` 参数的构造函数和一个将 `SmallInt` 转换为 `double` 的转换操作符：

```
// unwise class definition:
// multiple constructors and conversion operators to and from the built-in types
// can lead to ambiguity problems
class SmallInt {
public:
    // conversions to SmallInt from int and double
    SmallInt(int = 0);
    SmallInt(double);
    // Conversions to int or double from SmallInt
    // Usually it is unwise to define conversions to multiple arithmetic types
    operator int() const { return val; }
    operator double() const { return val; }
    // ...
private:
    std::size_t val;
};
```

541



一般而言，给出一个类与两个内置类型之间的转换是不好的做法，在这里这样做是为了举例说明所包含的缺陷。

考虑最简单的调用非重载函数的情况：

```
void compute(int);
```

```

void fp_compute(double);
void extended_compute(long double);
SmallInt si;
compute(si);           // SmallInt::operator int() const
fp_compute(si);        // SmallInt::operator double() const
extended_compute(si);  // error: ambiguous

```

任一转换操作符都可用于 compute 调用中:

(1) operator int 产生对形参类型的完全匹配。

(2) 首先调用 operator double 进行转换, 后跟从 double 到 int 的标准转换与形参类型匹配。

完全匹配转换比需要标准转换的其他转换更好, 因此, 第一个转换序列更好, 选择转换函数 SmallInt::operator int() 来转换实参。

类似地, 在第二个调用中, 可用任一转换调用 fp_compute。但是, 到 double 的转换是一个完全匹配, 不需要额外的标准转换。

最后一个对 extended_compute 的调用有二义性。可以使用任一转换函数, 但每个都必须跟上一个标准转换来获得 long double, 因此, 没有一个转换比其他的更好, 调用具有二义性。



如果两个转换操作符都可用在一个调用中, 而且在转换函数之后存在标准转换 (7.8.4 节), 则根据该标准转换的类别选择最佳匹配。

2. 实参匹配和构造函数转换

正如可能存在两个转换操作符, 也可能存在两个构造函数可以用来将一个值转换为目标类型。考虑 manip 函数, 它接受一个 SmallInt 类型的实参:

```

void manip(const SmallInt &);
double d; int i; long l;
manip(d);      // ok: use SmallInt(double) to convert the argument
manip(i);      // ok: use SmallInt(int) to convert the argument
manip(l);      // error: ambiguous

```

542

在第一个调用中, 可以用任一构造函数将 d 转换为 SmallInt 类型的值。int 构造函数需要对 d 的标准转换, 而 double 构造函数是完全匹配。因为完全匹配比标准转换更好, 所以用构造函数 SmallInt(double) 进行转换。

在第二个调用中, 情况恰恰相反, 构造函数 SmallInt(int) 提供完全匹配——不需要附加的转换, 调用接受一个 double 参数的 SmallInt 构造函数需要首先将 i 转换为 double 类型。对于这个调用, 用 int 构造函数转换实参。

第三个调用具有二义性。没有构造函数完全匹配于 long。使用每一个构造函数之前都需要对实参进行转换:

(1) 标准转换 (从 long 到 double) 后跟 SmallInt(double)。

(2) 标准转换 (从 long 到 int) 后跟 SmallInt(int)。

这些转换序列是不能区别的, 所以该调用具有二义性。



当两个构造函数定义的转换都可以使用时，如果存在构造函数实参所需的标准转换，就用该标准转换的类别选择最佳匹配。

3. 当两个类定义了转换时的二义性

当两个类定义了相互转换时，很可能存在二义性：

```
class Integral;
class SmallInt {
public:
    SmallInt(Integral); // convert from Integral to SmallInt
    // ...
};
class Integral {
public:
    operator SmallInt() const; // convert from SmallInt to Integral
    // ...
};
void compute(SmallInt);
Integral int_val;
compute(int_val); // error: ambiguous
```

实参 `int_val` 可以用两种不同方式转换为 `SmallInt` 对象，编译器可以使用接受 `Integral` 对象的构造函数，也可以使用将 `Integral` 对象转换为 `SmallInt` 对象的 `Integral` 转换操作。因为这两个函数没有高下之分，所以这个调用会出错。

在这种情况下，不能用显式类型转换来解决二义性——显式类型转换本身既可以使用转换操作又可以使用构造函数，相反，需要显式调用转换操作符或构造函数：

543

```
compute(int_val.operator SmallInt()); // ok: use conversion operator
compute(SmallInt(int_val));           // ok: use SmallInt constructor
```

而且，由于某些似乎微不足道的原因，我们认为可能有二义性的转换是合法的。例如，`SmallInt` 类构造函数复制它的 `Integral` 实参，如果改变构造函数以接受 `const Integral` 引用：

```
class SmallInt {
public:
    SmallInt(const Integral&);
};
```

则对 `compute(int_val)` 的调用不再有二义性！原因在于使用 `SmallInt` 构造函数需要将一个引用绑定到 `int_val`，而使用 `Integral` 类的转换操作符可以避免这个额外的步骤。这一小小区别足以使我们倾向于使用转换操作符。



避免二义性最好的方法是避免编写互相提供隐式转换的成对的类。

警告：避免转换函数的过度使用

与使用重载操作符一样，转换操作符的适当使用可以大大简化类设计者的工作并使得类的使用更简单。但是，有两个潜在的缺陷：定义太多转换操作符可能导致二义性代码，一些转

换可能利大于弊。

避免二义性最好的方法是，保证最多只有一种途径将一个类型转换为另一类型。做到这一点，最好的办法是限制转换操作符的数目，尤其是，到一种内置类型应该只有一个转换。

当转换操作符用于没有明显映射关系的类类型和转换类型之间时，容易引起误解，在这种情况下，提供转换函数可能会令类的使用者迷惑不解。

例如，如果有一个表示 Date 的类，我们可能会认为提供从 Date 到 int 的转换是个好主意，但是，这个转换函数应返回什么值？该函数可以返回公历日期，这是表示当前日期的一个顺序数，以 0 表示 1 月 1 日，但年份是否应放在日期之前或之后？即，1986 年 1 月 31 日是否应表示为 1986031 或 311986？作为一种选择，转换操作符可以返回一个表示从某个新纪元点开始计数的天数，计数器可以从 1971 年 1 月 1 日或其他起始点开始计算天数。

问题在于，无论怎样选择，Date 对象的使用将具有二义性，因为没有有一个 Date 类型对象与 int 类型值之间的一对一映射。在这种情况下，不定义转换函数更好。相反，这个类应该定义一个或多个普通成员从这些不同形式中抽取信息。

14.9.4 重载确定和类的实参

正如我们看到的，在需要转换函数的实参时，编译器自动应用类的转换操作符或构造函数。因此，应该在函数确定期间考虑类转换操作符。函数重载确定（7.8.2 节）由三步组成：

544

(1) 确定候选函数集合：这些是与被调用函数同名的函数。

(2) 选择可行的函数：这些是形参数目和类型与函数调用中的实参相匹配的候选函数。选择可行函数时，如果有转换操作，编译器还要确定需要哪个转换操作来匹配每个形参。

(3) 选择最佳匹配的函数。为了确定最佳匹配，对将实参转换为对应形参所需的类型转换进行分类。对于类类型的实参和形参，可能的转换的集合包括类类型转换。

1. 转换操作符之后的标准转换

哪个函数是最佳匹配，可能依赖于在匹配不同函数中是否涉及了一个或多个类类型转换。

如果重载集中的两个函数可以用同一转换函数匹配，则使用在转换之后或之前的标准转换序列的等级来确定哪个函数具有最佳匹配。



否则，如果可以使用不同转换操作，则认为这两个转换是一样好的匹配，不管可能需要或不需要的标准转换的等级如何。

14.9.3 节中介绍了类类型转换在非重载函数调用上的效果，现在，我们将看看类似的调用，但假定函数是重载的：

```
void compute(int);
void compute(double);
void compute(long double);
```

假定使用原来的 SmallInt 类，该类只定义了一个转换操作符——从 SmallInt 到 int 的转换，那么，如果将 SmallInt 对象传给 compute，该调用与接受一个 int 的 compute 版本相匹配。

三个函数都是可行的：

- `compute(int)`可行，因为 `SmallInt` 有到 `int` 的转换，该转换是对形参的完全匹配。
- `compute(double)`和 `compute(long double)`也是可行的，可以使用到 `int` 的转换，后面跟上适当的用于 `double` 或 `long double` 的标准转换。

因为可以用同一类类型转换来匹配这三个函数，如果存在标准转换，就用标准转换的等级确定最佳匹配。因为完全匹配比标准转换更好，所以选择 `compute(int)`函数作为最佳可行函数。

545



只有两个转换序列使用同一转换操作时，才用类类型转换之后的标准转换序列作为选择标准。

2. 多个转换和重载确定

现在可以看看为什么增加一个到 `double` 的转换是个坏主意。如果使用修改后的定义了到 `int` 和 `double` 的转换的 `SmallInt` 类，则用 `SmallInt` 值调用 `compute` 具有二义性：

```
class SmallInt {
public:
    // Conversions to int or double from SmallInt
    // Usually it is unwise to define conversions to multiple arithmetic types
    operator int() const { return val; }
    operator double() const { return val; }
    // ...
private:
    std::size_t val;
};

void compute(int);
void compute(double);
void compute(long double);
SmallInt si;
compute(si);    // error: ambiguous
```

在这个例子中，可以使用 `operator int` 转换 `si` 并调用接受 `int` 参数的 `compute` 版本，或者，可以使用 `operator double` 转换 `si` 并调用 `compute(double)`。

编译器将不会试图区别两个不同的类类型转换。具体而言，即使一个调用需要在类类型转换之后跟一个标准转换，而另一个是完全匹配，编译器仍会将该调用标记为错误。

3. 显式强制转换¹消除二义性

面对二义性转换，程序员可以使用强制转换来显式指定应用哪个转换操作：

```
void compute(int);
void compute(double);

SmallInt si;
compute(static_cast<int>(si)); // ok: convert and call compute(int)
```

这个调用现在是合法的，因为它显式指出了将哪个转换操作应用到实参。实参类型强制转换为 `int`，该类型与接受 `int` 参数的第一个 `compute` 版本完全匹配。

546

4. 标准转换和构造函数

现在来看存在多个转换构造函数的重载确定：

1. 原文此处误作“构造函数调用”。——译者注

```

class SmallInt {
public:
    SmallInt(int = 0);
};
class Integral {
public:
    Integral(int = 0);
};
void manip(const Integral&);
void manip(const SmallInt&);
manip(10);    // error: ambiguous

```

问题在于，Integral 和 SmallInt 这两个类都提供接受 int 参数的构造函数，其中任意一个构造函数都可以与 manip 的一个版本相匹配，因此，函数调用有二义性：它既可以表示将 Integral 转换为 int 并调用 manip 的第一个版本，也可以表示将 SmallInt 转换为 int 并调用 manip 的第二个版本。

即使其中一个类定义了实参需要标准转换的构造函数，这个函数调用也可能具有二义性。例如，如果 SmallInt 定义了一个构造函数，接受 short 而不是 int 参数，函数调用 manip(10) 将在使用构造函数之前需要一个从 int 到 short 的标准转换。在函数调用的重载版本中进行选择时，一个调用需要标准转换而另一个不需要，这一事实不是实质性的，编译器不会更喜欢直接构造函数，调用仍具有二义性。

5. 显式构造函数调用消除二义性

调用者可以通过显式构造所需类型的值而消除二义性：

```

manip(SmallInt(10));    // ok: call manip(SmallInt)
manip(Integral(10));    // ok: call manip(Integral)

```



在调用重载函数时，需要使用构造函数或强制类型转换来转换实参，这是设计拙劣的表现。

习题

习题 14.44 为下述每个初始化列出可能的类类型转换序列。每个初始化的结果是什么？

```

class LongDouble {
    operator double();
    operator float();
};
LongDouble ldObj;
(a) int ex1 = ldObj;    (b) float ex2 = ldObj;

```

习题 14.45 哪个 calc() 函数是如下函数调用的最佳可行函数？列出调用每个函数所需的转换序列，并解释为什么所选定的就是最佳可行函数。

```

class LongDouble {
public:
    LongDouble(double);
    // ...
};
void calc(int);

```

```
void calc(LongDouble);
double dval;

calc(dval); // which function?
```

14.9.5 重载、转换和操作符

重载操作符就是重载函数。使用与确定重载函数调用一样的过程来确定将哪个操作符（内置的还是类类型的）应用于给定表达式。给定如下代码：

547

```
ClassX sc;
int iobj = sc + 3;
```

有四种可能性：

- 有一个重载的加操作符与 ClassX 和 int 相匹配。
- 存在转换，将 sc 和/或 int 值转换为定义了+的类型。如果是这样，该表达式将先使用转换，接着应用适当的加操作符。
- 因为既定义了转换操作符又定义了+的重载版本，该表达式具有二义性。
- 因为既没有转换又没有重载的+可以使用，该表达式非法。

1. 重载确定和操作符



成员函数和非成员函数都是可能的，这一事实改变了选择候选函数集的方式。

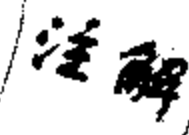
548

操作符的重载确定（7.8.2 节）遵循常见的三步过程：

- (1) 选择候选函数。
- (2) 选择可行函数，包括识别每个实参的潜在转换序列。
- (3) 选择最佳匹配的函数。

2. 操作符的候选函数

一般而言，候选函数集由所有与被使用的函数同名的函数构成，被使用的函数可以从函数调用处看到。对于操作符用在表达式中的情况，候选函数包括操作符的内置版本以及该操作符的普通非成员版本。另外，如果左操作数具有类类型，而且该类定义了该操作符的重载版本，则候选集将包含操作符的重载版本。



一般而言，函数调用的候选集只包括成员函数或非成员函数，不会两者都包括。而确定操作符的使用时，操作符的非成员和成员版本可能都是候选者。

确定指定函数的调用时，与操作符的使用相反，由调用本身确定所考虑的名字的作用域。如果是通过类类型的对象（或通过这种对象的引用或指针）的调用，则只需考虑该类的成员函数。具有同一名字的成员函数和非成员函数不会相互重载。使用重载操作符时，调用本身不会告诉我们与使用的操作符函数作用域相关的任何事情，因此，成员和非成员版本都必须考虑。

警告：转换和操作符

正确设计类的重载操作符、转换构造函数和转换函数需要多加小心。尤其是，如果类既定义了转换操作符又定义了重载操作符，容易产生二义性。下面几条经验规则会有所帮助：

(1) 不要定义相互转换的类，即如果类 Foo 具有接受类 Bar 的对象的构造函数，不要再为类 Bar 定义到类型 Foo 的转换操作符。

(2) 避免到内置算术类型的转换。具体而言，如果定义了到算术类型的转换，则

- 不要定义接受算术类型的操作符的重载版本。如果用户需要使用这些操作符，转换操作符将转换你所定义的类型对象，然后可以使用内置操作符。
- 不要定义转换到一个以上算术类型的转换。让标准转换提供到其他算术类型的转换。

最简单的规则是：对于那些“明显正确”的，应避免定义转换函数并限制非显式构造函数。

549

3. 转换可能引起内置操作符的二义性

我们再次扩展 SmallInt 类。这一次，除了到 int 的转换操作符和接受 int 参数的构造函数之外，将增加一个重载的加操作符：

```
class SmallInt {
public:
    SmallInt(int = 0); // convert from int to SmallInt
    // conversion to int from SmallInt
    operator int() const { return val; }
    // arithmetic operators
    friend SmallInt
    operator+(const SmallInt&, const SmallInt&);
private:
    std::size_t val;
};
```

现在，可以用这个类将两个 SmallInt 对象相加，但是，如果试图进行混合模式运算，将会遇到二义性问题：

```
SmallInt s1, s2;
SmallInt s3 = s1 + s2; // ok: uses overloaded operator+
int i = s3 + 0;        // error: ambiguous
```

第一个加使用接受两个 SmallInt 值的+的重载版本。第二个加有二义性，问题在于，可以将 0 转换为 SmallInt 并使用+的 SmallInt 版本，也可以将 s3 转换为 int 值并使用 int 值上的内置加操作符。



既为算术类型提供转换函数，又为同一类类型提供重载操作符，可能会导致重载操作符和内置操作符之间的二义性。

4. 可行的操作符函数和转换

通过为每个调用列出可行函数，可以理解这两个调用的行为。在第一个调用中，有两个可行的加操作符：

- `operator+(const SmallInt&, const SmallInt&)`。
- 内置的 `operator+(int, int)`。

第一个加不需要实参转换——`s1` 和 `s2` 与形参的类型完全匹配。使用内置加操作符对两个实参都需要转换，因此，重载操作符与两个实参匹配得较好，所以将调用它。对于第二个加运算：

550 `int i = s3 + 0; // error: ambiguous`

两个函数同样可行。在这种情况下，重载的`+`版本与第一个实参完全匹配，而内置版本与第二个实参完全匹配。第一个可行函数对左操作数而言较好，而第二个可行函数对右操作数而言较好。因为找不到最佳可行函数，所以将该调用标记为有二义性的。

习题

习题 14.46 对于 `main` 中的加操作，哪个 `operator+` 是最佳可行函数？列出候选函数、可行函数以及对每个可行函数中实参的类型转换。

```
class Complex {
    Complex(double);
    // ...
};

class LongDouble {
    friend LongDouble operator+(LongDouble&, int);
public:
    LongDouble(int);
    operator double();
    LongDouble operator+(const complex &);
    // ...
};

LongDouble operator+(const LongDouble &, double);

LongDouble ld(16.08);
double res = ld + 15.05; // which operator+?
```

551

小结

第 5 章介绍了 C++ 为内置类型所定义的丰富的操作符集合，该章也涵盖了标准转换，标准转换自动将操作数从一个类型转换为另一类型。

通过定义内置操作符的重载版本，我们可以为自己的类型（即，类类型或枚举类型）的对象定义同样丰富的表达式集合。重载操作符必须具有至少一个类类型或枚举类型的操作数。应用于内置类型时，重载操作符与对应操作符具有同样数目的操作数、同样的结合性和优先级。

大多数重载操作符可以定义为类成员或普通非成员函数，赋值操作符、下标操作符、调用操作符和箭头操作符必须为类成员。操作符定义为成员时，它是普通成员函数。具体而言，成员操作符有一个隐式 `this` 指针，该指针一定是第一个操作数，即，一元操作符唯一的操作数，二元操作符的左操作数。

重载了 `operator()`（即，函数调用操作符）的类的对象，称为“函数对象”。这种对象通

常用于定义与标准算法结合使用的谓词函数。

类可以定义转换，当一个类型的对象用在需要另一不同类型对象的地方时，自动应用这些转换。接受单个形参且未指定为 `explicit` (12.4.4 节) 的构造函数定义了从其他类型到类类型¹ 的转换，重载操作符转换函数则定义了从类类型到其他类型² 的转换。转换操作符必须为所转换类的成员，没有形参并且不定义返回值，转换操作符返回操作符所具有类型的值，例如，`operator int` 返回 `int`。

重载操作符和类类型转换都有助于更容易、更自然地使用类型，但是，必须注意避免设计对用户而言不明显的操作符和转换，而且应避免定义一个类型与另一类型之间的多个转换。

术语

binary function object (二元函数对象) 具有函数调用操作符且表示一个二元操作符（例如一个算术操作符或关系操作符）的类。

binder (绑定器) 绑定指定函数对象的一个操作数的适配器。例如，`bind2nd(minus<int>(), 2)` 产生一个一元函数对象，从操作数中减去 2。

class-type conversion (类类型转换) 到类类型或从类类型的转换。接受一个形参的非显式构造函数定义从形参类型到类类型的转换。转换操作符定义从类类型到操作符所指定类型的转换。

conversion operators (转换操作符) 转换操作符是定义从类类型到另一类型的转换的成员函数。转换操作符必须是类的成员，而且不能指定返回类型不能接受形参。转换操作符返回转换操作符类型的值，即，`operator int` 返回 `int`，`operator Sales_item` 返回 `Sales_item`，依此类推。

function adaptor (函数适配器) 为函数对象提供新接口的标准库的类型。

function object (函数对象) 定义了重载调用操作符的类的对象。函数对象可以用在需要函数的地方。

negator (求反器) 将指定函数对象的返回值求反的适配器。例如，`not2(equal_to<int>())` 产生与 `not_equal_to<int>` 等价的函数对象。

smart pointer (智能指针) 一个类，定义了指针式行为和其他功能，如，引用计数、内存管理、更全面的检查等。这种类通常定义了解引用操作符 (`operator*`) 和成员访问操作符 (`operator->`) 的重载版本。

unary function object (一元函数对象) 具有函数调用操作符且表示一个一元操作符的类，如一元减或逻辑非。

1. 此处原文弄反了。——译者注

2. 此处原文弄反了。——译者注

面向对象编程与泛型编程

目录

第 15 章 面向对象编程	471
第 16 章 模板与泛型编程	525

第四部分继续第三部分的讨论，涵盖 C++ 如何支持面向对象编程和泛型编程。

第 15 章讨论继承和动态绑定。继承和动态绑定与数据抽象一起成为面向对象编程的基础。

第 16 章讨论函数模板和类模板。模板使我们能够编写独立于具体类型的泛型类和泛型函数。

编写自己的面向对象类型或泛型类型需要对 C++ 的充分理解，幸运的是，我们可以使用面向对象和泛型类型而无需了解它们的构建细节。事实上，标准库广泛使用了将在第 15 章和第 16 章中介绍的设施，而且我们已经在不了解实现细节的情况下使用了标准库中的类型和算法。因此，读者应该理解第四部分涵盖的是一些高级主题。编写模板或面向对象的类，需要充分理解 C++ 的基本原理并且很好地掌握怎样定义更基本的类。

第 15 章

面向对象编程

目录

15.1 面向对象编程：概述.....	472
15.2 定义基类和派生类.....	473
15.3 转换与继承.....	487
15.4 构造函数和复制控制.....	490
15.5 继承情况下的类作用域.....	497
15.6 纯虚函数.....	502
15.7 容器与继承.....	503
15.8 句柄类与继承.....	504
15.9 再谈文本查询示例.....	511
小结.....	522
术语.....	523

面向对象编程基于三个基本概念：数据抽象、继承和动态绑定。在 C++ 中，用类进行数据抽象，用类派生从一个类继承另一个类：派生类继承基类的成员。动态绑定使编译器能够在运行时决定是使用基类中定义的函数还是派生类中定义的函数。

继承和动态绑定在两个方面简化了我们的程序：能够容易地定义与其他类相似但又不相同的新类，能够更容易地编写忽略这些相似类型之间区别的程序。

许多应用程序的特性可以用一些相关但略有不同的概念来描述。例如，书店可以为不同的书提供不同的定价策略，有些书可以只按给定价格出售，另一些书可以根据不同的折扣策略出售。可以给购买某书一定数量的顾客打折，或者，购买一定数量以内可以打折而超过给定限制就付全价。

面向对象编程（Object-oriented programming, OOP）与这种应用非常匹配。通过继承可以定义一些类型，以模拟不同种类的书，通过动态绑定可以编写程序，使用这些类型而又忽略与具体类型相关的差异。

继承和动态绑定的思想在概念上非常简单，但对于如何创建应用程序以及对于程序设计语言必须支持哪些特性，它们的含义深远。在讨论 C++ 如何支持面向对象编程之前，我们将介绍这种编程风格的一些基本概念。

15.1 面向对象编程：概述

面向对象编程的关键思想是**多态性**（polymorphism）。多态性派生于一个希腊单词，意思是“许多形态”。之所以称通过继承而相关联的类型为多态类型，是因为在许多情况下可以互换地使用派生类型或基类型的“许多形态”。正如我们将看到的，在 C++ 中，多态性仅用于通过继承而相关联的类型的引用或指针。

1. 继承

通过继承我们能够定义这样的类，它们对类型之间的关系建模，共享公共的东西，仅仅特化本质上不同的东西。**派生类**（derived class）能够继承**基类**（base class）定义的成员，派生类可以无须改变而使用那些与派生类型具体特性不相关的操作，派生类可以重定义那些与派生类型相关的成员函数，将函数特化，考虑派生类型的特性。最后，除了从基类继承的成员之外，派生类还可以定义更多的成员。

我们经常称因继承而相关联的类为构成了一个**继承层次**（inheritance hierarchy）。其中有一个类称为根，所有其他类直接或间接继承根类。在书店例子中，我们将定义一个基类，命名为 `Item_base`，表示未打折的书，第二个类继承 `Item_base`，命名为 `Bulk_item`，表示带数量折扣销售的书。

这些类至少定义如下操作：

- 名为 `book` 的操作，返回 ISBN。
- 名为 `net_price` 的操作，返回购买指定数量的书的价格。

`Item_base` 的派生类将无须改变地继承 `book` 函数：派生类不需要重新定义获取 ISBN 的含义。另一方面，每个派生类需要定义自己的 `net_price` 函数版本，以实现适当的折扣价格策略。

在 C++ 中，基类必须指出希望派生类重定义哪些函数，定义为 **virtual** 的函数是基类期待派生类重新定义的，基类希望派生类继承的函数不能定义为虚函数。

讨论过这些之后，可以看到我们的类将定义三个（const）成员函数：

- 非虚函数 `std::string book()`，返回 ISBN。由 `Item_base` 定义，`Bulk_item` 继承。
- 虚函数 `double net_price(size_t)` 的两个版本，返回给定数目的某书的总价。`Item_base`

类和 Bulk_item 类将定义该函数自己的版本。

2. 动态绑定

通过动态绑定 (dynamic binding) 我们能够编写程序使用继承层次中任意类型的对象, 无须关心对象的具体类型。使用这些类的程序无须区分函数是在基类还是在派生类中定义的。

例如, 书店应用程序可以允许顾客在一次交易中选择几本书, 当顾客购书时, 应用程序可以计算总的应付款, 指出最终账单的一个部分将是为每本书打印一行, 以显示总数和售价。

可以定义一个名为 print_total 的函数管理应用程序的这个部分。给定一个项目和数量, 函数应打印 ISBN 以及购买给定数量的某书的总价。这个函数的输出应该像这样:

```
ISBN: 0-201-54848-8 number sold: 3 total price: 98
ISBN: 0-201-82470-1 number sold: 5 total price: 202.5
```

可以这样编写 print_total 函数:

```
// calculate and print price for given number of copies, applying any discounts
void print_total(ostream &os,
                 const Item_base &item, size_t n)
{
    os << "ISBN: " << item.book() // calls Item_base::book
      << "\tnumber sold: " << n << "\ttotal price: "
      // virtual call: which version of net_price to call is resolved at run time
      << item.net_price(n) << endl;
}
```

该函数的工作很普通: 调用其 item 形参的 book 和 net_price 函数, 打印结果。关于这个函数, 有两点值得注意。

第一, 虽然这个函数的第二个形参是 Item_base 的引用, 但可以将 Item_base 对象或 Bulk_item 对象传给它。

第二, 因为形参是引用且 net_price 是虚函数, 所以对 net_price 的调用将在运行时确定。调用哪个版本的 net_price 将依赖于传给 print_total 的实参。如果传给 print_total 的实参是一个 Bulk_item 对象, 将运行 Bulk_item 中定义的应用折扣的 net_price; 如果实参是一个 Item_base 对象, 则调用由 Item_base 定义的版本。



在 C++ 中, 通过基类的引用 (或指针) 调用虚函数时, 发生动态绑定。引用 (或指针) 既可以指向基类对象也可以指向派生类对象, 这一事实是动态绑定的关键。用引用 (或指针) 调用的虚函数在运行时确定, 被调用的函数是引用 (或指针) 所指对象的实际类型所定义的。

15.2 定义基类和派生类

基类和派生类的定义在许多方面像我们已见过的其他类一样。但是, 在继承层次中定义类还需要另外一些特性, 本节将介绍这些特性, 后续的节将介绍这些特性的使用对类以及使用继承类编写的程序有何影响。

15.2.1 定义基类

像任意其他类一样，基类也有定义其接口和实现的数据和函数成员。在（非常简化的）书店定价应用程序的例子中，Item_base 类定义了 book 和 net_price 函数并且需要存储每本书的 ISBN 和标准价格：

```
// Item sold at an undiscounted price
// derived classes will define various discount strategies
class Item_base {
public:
    Item_base(const std::string &book = "",
              double sales_price = 0.0):
        isbn(book), price(sales_price) { }
    std::string book() const { return isbn; }
    // returns total sales price for a specified number of items
    // derived classes will override and apply different discount algorithms
    virtual double net_price(std::size_t n) const
        { return n * price; }
    virtual ~Item_base() { }
private:
    std::string isbn;    // identifier for the item
protected:
    double price;       // normal, undiscounted price
};
```

560

这个类的大部分看起来像我们已见过的其他类一样。它定义了一个构造函数以及我们已描述过的函数，该构造函数使用默认实参（7.4.1 节），允许用 0 个、1 个或两个实参进行调用，它用这些实参初始化数据成员。

新的部分是 protected 访问标号以及对析构函数和 net_price 函数所使用的保留字 virtual。我们将在 15.4.4 节解释虚析构函数，现在只需注意到继承层次的根类一般都要定义虚析构函数即可。

1. 基类成员函数

Item_base 类定义了两个函数，其中一个前面带有保留字 virtual。保留字 virtual 的目的是启用动态绑定。成员默认为非虚函数，对非虚函数的调用在编译时确定。为了指明函数为虚函数，在其返回类型前面加上保留字 virtual。除了构造函数之外，任意非 static 成员函数都可以是虚函数。保留字只在类内部的成员函数声明中出现，不能用在类定义体外部出现的函数定义上。

15.2.4 节将进一步介绍虚函数。



最佳
实践

基类通常应将派生类需要重定义的任何函数定义为虚函数。

2. 访问控制和继承

在基类中，public 和 private 标号具有普通含义：用户代码可以访问类的 public 成员而不能访问 private 成员，private 成员只能由基类的成员和友元访问。派生类对基类的 public 和 private 成员的访问权限与程序中任意其他部分一样：它可以访问 public 成员而不能访问

private 成员。

有时作为基类的类具有一些成员，它希望允许派生类访问但仍禁止其他用户访问这些成员。对于这样的成员应使用受保护的访问标号（protected access label）。protected 成员可以被派生类对象访问但不能被该类型的普通用户访问。

我们的 Item_base 类希望它的派生类重定义 net_price 函数，为了重定义 net_price 函数，这些类将需要访问 price 成员。希望派生类用与普通用户一样通过 book 访问函数访问 isbn，因此，isbn 成员为 private，不能被 Item_base 的继承类所访问。

561

习题

习题 15.1 什么是虚成员？

习题 15.2 给出 protected 访问标号的定义。它与 private 有何不同？

习题 15.3 定义自己的 Item_base 类版本。

习题 15.4 图书馆可以借阅不同种类的资料——书、CD、DVD 等等。不同种类的借阅资料有不同的登记、检查和过期规则。下面的类定义了这个应用程序可以使用的基类。指出在所有借阅资料中，哪些函数可能定义为虚函数，如果有，哪些函数可能是公共的。（注：假定 LibMember 是表示图书馆读者的类，Date 是表示特定年份的日历日期的类。）

```
class Library {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    bool is_late(const Date& today);
    double apply_fine();
    ostream& print(ostream& = cout);
    Date due_date() const;
    Date date_borrowed() const;
    string title() const;
    const LibMember& member() const;
};
```

15.2.2 protected 成员

可以认为 protected 访问标号是 private 和 public 的混合：

- 像 private 成员一样，protected 成员不能被类的用户访问。
- 像 public 成员一样，protected 成员可被该类的派生类访问。

此外，protected 还有另一重要性质：

- 派生类¹只能通过派生类对象访问其基类的 protected 成员，派生类对其基类类型对象的 protected 成员没有特殊访问权限。

例如，假定 Bulk_item 定义了一个成员函数，接受一个 Bulk_item 对象的引用和一个 Item_base 对象的引用，该函数可以访问自己对象的 protected 成员以及 Bulk_item 形参的

1. 原文此处误为“派生类对象”。——译者注

562 protected 成员，但是，它不能访问 Item_base 形参的 protected 成员。

```
void Bulk_item::memfcn(const Bulk_item &d, const Item_base &b)
{
    // attempt to use protected member
    double ret = price; // ok: uses this->price
    ret = d.price; // ok: uses price from a Bulk_item object
    ret = b.price; // error: no access to price from an Item_base
}
```

d.price 的使用正确，因为是通过 Bulk_item 类型对象引用 price；b.price 的使用非法，因为对 Base_item 类型的对象没有特殊访问权限。

关键概念：类设计与受保护成员

如果没有继承，类只有两种用户：类本身的成员和该类的用户。将类划分为 private 和 public 访问级别反映了用户种类的这一分隔：用户只能访问 public 接口，类成员和友元既可以访问 public 成员也可以访问 private 成员。

有了继承，就有了类的第三种用户：从类派生定义新类的程序员。派生类的提供者通常（但并不总是）需要访问（一般为 private 的）基类实现，为了允许这种访问而仍然禁止对实现的一般访问，提供了附加的 protected 访问标号。类的 protected 部分仍然不能被一般程序访问，但可以被派生类访问。只有类本身和友元可以访问基类的 private 部分，派生类不能访问基类的 private 成员。

定义类充当基类时，将成员设计为 public 的标准并没有改变：仍然是接口函数应该为 public 而数据一般不应为 public。被继承的类必须决定实现的哪些部分声明为 protected 而哪些部分声明为 private。希望禁止派生类访问的成员应该设为 private，提供派生类实现所需操作或数据的成员应设为 protected。换句话说，提供给派生类型的接口是 protected 成员和 public 成员的组合。

15.2.3 派生类

为了定义派生类，使用类派生列表（class derivation list）指定基类。类派生列表指定了一个或多个基类，具有如下形式：

```
class classname: access-label base-class
```

这里 access-label 是 public、protected 或 private，base-class 是已定义的类的名字。类派生列表可以指定多个基类。继承单个基类最为常见，也是本章的主题。17.3 节讨论多个基类的使用。

563

15.2.5 节将进一步介绍派生列表中使用的访问标号，现在，只需要了解访问标号决定了对继承成员的访问权限。如果想要继承基类的接口，则应该进行 public 派生。

派生类继承基类的成员并且可以定义自己的附加成员。每个派生类对象包含两个部分：从基类继承的成员和自己定义的成员。一般而言，派生类只（重）定义那些与基类不同或扩展基类行为的方面。

1. 定义派生类

在书店应用程序中, 将从 `Item_base` 类派生 `Bulk_item` 类, 因此 `Bulk_item` 类将继承 `book`、`isbn` 和 `price` 成员。`Bulk_item` 类必须重定义 `net_price` 函数并定义该操作所需要的数据成员:

```
// discount kicks in when a specified number of copies of same book are sold
// the discount is expressed as a fraction used to reduce the normal price
class Bulk_item : public Item_base {
public:
    // redefines base version so as to implement bulk purchase discount policy
    double net_price(std::size_t) const;
private:
    std::size_t min_qty; // minimum purchase for discount to apply
    double discount;     // fractional discount to apply
};
```

每个 `Bulk_item` 对象包含四个数据成员: 从 `Item_base` 继承的 `isbn` 和 `price`, 自己定义的 `min_qty` 和 `discount`, 后两个成员指定最小数量以及购买超过该数量时给的折扣。`Bulk_item` 类还需要定义一个构造函数, 我们将在 15.4 节定义它。

2. 派生类和虚函数

尽管不是必须这样做, 派生类一般会重定义所继承的虚函数。如果派生类没有重定义某个虚函数, 则使用基类中定义的版本。

派生类型必须对想要重定义的几个继承成员进行声明。`Bulk_item` 类指出, 它将重定义 `net_price` 函数但将使用 `book` 的继承版本。

派生类中虚函数的声明 (7.4 节) 必须与基类中的定义方式完全匹配, 但有一个例外: 返回对基类型的引用 (或指针) 的虚函数。派生类中的虚函数可以返回基类函数所返回类型的派生类的引用 (或指针)。

例如, `Item_base` 类可以定义返回 `Item_base*` 的虚函数, 如果这样, `Bulk_item` 类中定义的实例可以定义为返回 `Item_base*` 或 `Bulk_item*`。15.9 节将介绍这种虚函数的一个例子。

564

注解 一旦函数在基类中声明为虚函数, 它就一直为虚函数, 派生类无法改变该函数为虚函数这一事实。派生类重定义虚函数时, 可以使用 `virtual` 保留字, 但不是必须这样做。

3. 派生类对象包含基类对象作为子对象

派生类对象由多个部分组成: 派生类本身定义的 (非 `static`) 成员加上由基类 (非 `static`) 成员组成的子对象。可以认为 `Bulk_item` 对象由图 15-1 表示的两个部分组成。



图 15-1 `Bulk_item` 对象的概念结构



C++语言不要求编译器将对象的基类部分和派生部分连续排列，因此，图 15-1 是关于类如何工作的概念表示而不是物理表示。

4. 派生类中的函数可以使用基类的成员

像任意成员函数一样，派生类函数可以在类的内部或外部定义，正如这里的 `net_price` 函数一样：

```
// if specified number of items are purchased, use discounted price
double Bulk_item::net_price(size_t cnt) const
{
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

该函数产生折扣价格：如果给定数量多于 `min_qty`，就对 `price` 应用 `discount` (`discount` 存储为分数)。

565



因为每个派生类对象都有基类部分，类可以访问其基类的 `public` 和 `protected` 成员，就好像那些成员是派生类自己的成员一样。

5. 用作基类的类必须是已定义的

已定义的类才可以用作基类。如果已经声明了 `Item_base` 类，但没有定义它，则不能用 `Item_base` 作基类：

```
class Item_base; // declared but not defined

// error: Item_base must be defined
class Bulk_item : public Item_base { ... };
```

这一限制的原因应该很容易明白：每个派生类包含并且可以访问其基类的成员，为了使用这些成员，派生类必须知道它们是什么。这一规则暗示着不可能从类自身派生出一个类。

6. 用派生类作基类

基类本身可以是一个派生类：

```
class Base { /* ... */ };
class D1: public Base { /* ... */ };
class D2: public D1 { /* ... */ };
```

每个类继承其基类的所有成员。最底层的派生类继承其基类的成员，基类又继承自己的基类的成员，如此沿着继承链依次向上。从效果来说，最底层的派生类对象包含其每个直接基类 (`immediate-base`) 和间接基类 (`indirect-base`) 的子对象。

7. 派生类的声明

如果需要声明 (但并不实现) 一个派生类，则声明包含类名但不包含派生列表。例如，下面的前向声明会导致编译时错误：

```
// error: a forward declaration must not include the derivation list
```

```
class Bulk_item : public Item_base;
```

正确的前向声明为:

```
// forward declarations of both derived and nonderived class
class Bulk_item;
class Item_base;
```

习题

习题 15.5 如果有, 下面声明中哪些是错误的?

```
class Base { ... };
```

- (a) class Derived : public Derived { ... };
- (b) class Derived : Base { ... };
- (c) class Derived : private Base { ... };
- (d) class Derived : public Base;
- (e) class Derived inherits Base { ... };

习题 15.6 编写自己的 Bulk_item 类版本。

习题 15.7 可以定义一个类型实现有限折扣策略。这个类可以给低于某个上限的购书量一个折扣, 如果购买的数量超过该上限, 则超出部分的书应按正常价格购买。定义一个类实现这种策略。

15.2.4 virtual 与其他成员函数

C++中的函数调用默认不使用动态绑定。要触发动态绑定, 必须满足两个条件: 第一, 只有指定为虚函数的成员函数才能进行动态绑定, 成员函数默认为非虚函数, 非虚函数不进行动态绑定; 第二, 必须通过基类类型的引用或指针进行函数调用。要理解这一要求, 需要理解在使用继承层次中某一类型的对象的引用或指针时会发生什么。

566

1. 从派生类到基类的转换

因为每个派生类对象都包含基类部分, 所以可将基类类型的引用绑定到派生类对象的基类部分, 也可以用指向基类的指针指向派生类对象:

```
// function with an Item_base reference parameter
double print_total(const Item_base&, size_t);
Item_base item; // object of base type
// ok: use pointer or reference to Item_base to refer to an Item_base object
print_total(item, 10); // passes reference to an Item_base object
Item_base *p = &item; // p points to an Item_base object
Bulk_item bulk; // object of derived type
// ok: can bind a pointer or reference to Item_base to a Bulk_item object
print_total(bulk, 10); // passes reference to the Item_base part of bulk
p = &bulk; // p points to the Item_base part of bulk
```

这段代码使用同一基类类型指针指向基类类型的对象和派生类型的对象, 该代码还传递基类类型和派生类型的对象来调用需要基类类型引用的函数, 两种使用都是正确的, 因为每个派生类对象都拥有基类部分。

因为可以使用基类类型的指针或引用来引用派生类型对象, 所以, 使用基类类型的引用或指针时, 不知道指针或引用所绑定的对象的类型: 基类类型的引用或指针可以引用基类类型对象,

567

也可以引用派生类型对象。无论实际对象具有哪种类型，编译器都将它当作基类类型对象。将派生类对象当作基类对象是安全的，因为每个派生类对象都拥有基类子对象。而且，派生类继承基类的操作，即，任何可以在基类对象上执行的操作也可以通过派生类对象使用。



基类类型引用和指针的关键点在于静态类型 (static type, 在编译时可知的引用类型或指针类型) 和动态类型 (dynamic type, 指针或引用所绑定的对象的类型, 这是仅在运行时可知的) 可能不同。

2. 可以在运行时确定 virtual 函数的调用

将基类类型的引用或指针绑定到派生类对象对基对象没有影响，对象本身不会改变，仍为派生类对象。对象的实际类型可能不同于该对象引用或指针的静态类型，这是 C++ 中动态绑定的关键。

通过引用或指针调用虚函数时，编译器将生成代码，在运行时确定调用哪个函数，被调用的是与动态类型相对应的函数。例如，我们再来看 print_total 函数：

```
// calculate and print price for given number of copies, applying any discounts
void print_total(ostream &os,
                 const Item_base &item, size_t n)
{
    os << "ISBN: " << item.book() // calls Item_base::book
    << "\tnumber sold: " << n << "\ttotal price: "
    // virtual call: which version of net_price to call is resolved at run time
    << item.net_price(n) << endl;
}
```

因为 item 形参是一个引用且 net_price 是虚函数，item.net_price(n) 所调用的 net_price 版本取决于在运行时绑定到 item 形参的实参类型：

```
Item_base base;
Bulk_item derived;
// print_total makes a virtual call to net_price
print_total(cout, base, 10); // calls Item_base::net_price
print_total(cout, derived, 10); // calls Bulk_item::net_price
```

在第一个调用中，item 形参在运行时绑定到 Item_base 类型的对象，因此，print_total 内部调用 Item_base 中定义的 net_price 版本。在第二个调用中，item 形参绑定到 Bulk_item 类型的对象，从 print_total 调用的是 Bulk_item 类定义的 net_price 版本。

568

关键概念：C++ 中的多态性

引用和指针的静态类型与动态类型可以不同，这是 C++ 用以支持多态性的基石。

通过基类引用或指针调用基类中定义的函数时，我们并不知道执行函数的对象的确切类型，执行函数的对象可能是基类类型的，也可能是派生类型的。

如果调用非虚函数，则无论实际对象是什么类型，都执行基类类型所定义的函数。如果调用虚函数，则直到运行时才能确定调用哪个函数，运行的虚函数是引用所绑定的或指针所指向的对象所属类型定义的版本。

从编写代码的角度看我们无需担心。只要正确地设计和实现了类，不管实际对象是基类类型或派生类型，操作都将完成正确的工作。

另一方面，对象是非多态的——对象类型已知且不变。对象的动态类型总是与静态类型相同，这一点与引用或指针相反。运行的函数（虚函数或非虚函数）是由对象的类型定义的。



只有通过引用或指针调用，虚函数才在运行时确定。只有在这些情况下，直到运行时才知道对象的动态类型。

3. 在编译时确定非 **virtual** 调用

不管传给 `print_total` 的实参的实际类型是什么，对 `book` 的调用在编译时确定为调用 `Item_base::book`。



即使 `Bulk_item` 定义了自己的 `book` 函数版本，这个调用也会调用基类中的版本。

非虚函数总是在编译时根据调用该函数的对象、引用或指针的类型而确定。`item` 的类型是 `const Item_base` 的引用，所以，无论在运行时 `item` 引用的实际对象是什么类型，调用该对象的非虚函数都将会调用 `Item_base` 中定义的版本。

4. 覆盖虚函数机制

在某些情况下，希望覆盖虚函数机制并强制函数调用使用虚函数的特定版本，这时可以使用作用域操作符：

```
Item_base *baseP = &derived;
// calls version from the base class regardless of the dynamic type of baseP
double d = baseP->Item_base::net_price(42);
```

这段代码强制将 `net_price` 调用确定为 `Item_base` 中定义的版本，该调用将在编译时确定。

569



只有成员函数中的代码才应该使用作用域操作符覆盖虚函数机制。

为什么会希望覆盖虚函数机制？最常见的理由是为了派生类虚函数调用基类中的版本。在这种情况下，基类版本可以完成继承层次中所有类型的公共任务，而每个派生类型只添加自己的特殊工作。

例如，可以定义一个具有虚操作的 `Camera` 类层次。`Camera` 类中的 `display` 函数可以显示所有的公共信息，派生类（如 `PerspectiveCamera`）可能既需要显示公共信息又需要显示自己的独特信息。可以显式调用 `Camera` 版本以显示公共信息，而不是在 `PerspectiveCamera` 的 `display` 实现中复制 `Camera` 的操作。在这种情况下，已经确切知道调用哪个实例，因此，不需要通过虚函数机制。



派生类虚函数调用基类版本时，必须显式使用作用域操作符。如果派生类函数忽略了这样做，则函数调用会在运行时确定并且将是一个自身调用，从而导致无穷递归。

5. 虚函数与默认实参

像其他任何函数一样，虚函数也可以有默认实参。通常，如果有用在给定调用中的默认实参值，该值将在编译时确定。如果一个调用省略了具有默认值的实参，则所用的值由调用该函数的类型定义，与对象的动态类型无关。通过基类的引用或指针调用虚函数时，默认实参为在基类虚函数声明中指定的值，如果通过派生类的指针或引用调用虚函数，则默认实参是在派生类的版本中声明的值。

在同一虚函数的基类版本和派生类版本中使用不同的默认实参几乎一定会引起麻烦。如果通过基类的引用或指针调用虚函数，但实际执行的是派生类中定义的版本，这时就可能会出现问題。在这种情况下，为虚函数的基类版本定义的默认实参将传给派生类定义的版本，而派生类版本是用不同的默认实参定义的。

习题

习题 15.8 对于下面的类，解释每个函数：

```
struct base {
    string name() { return basename; }
    virtual void print(ostream &os) { os << basename; }
private:
    string basename;
};

struct derived {
    void print() { print(ostream &os); os << " " << mem; }
private:
    int mem;
};
```

如果该代码有问题，如何修正？

习题 15.9 给定上题中的类和如下对象，确定在运行时调用哪个函数：

```
base bobj;    base *bp1 = &bobj; base &br1 = bobj;
derived dobj; base *bp2 = &dobj; base &br2 = dobj;

(a) bobj.print();    (b) dobj.print();    (c) bp1->name();
(d) bp2->name();    (e) br1.print();    (f) br2.print();
```

15.2.5 公用、私有和受保护的继承

派生类中定义的成员访问控制的处理与任意其他类中完全一样（12.1.2 节）。派生类可以定义零个或多个访问标号，指定跟随其后的成员的访问级别。对类所继承的成员的访问由基类中的成员访问级别和派生类派生列表中使用的访问标号共同控制。



每个类控制它所定义的成员的访问。派生类可以进一步限制但不能放松对所继承的成员的访问。

基类本身指定对自身成员的最小访问控制。如果成员在基类中为 `private`，则只有基类和基类的友元可以访问该成员。派生类不能访问基类的 `private` 成员，也不能使自己的用户能够访

问那些成员。如果基类成员为 `public` 或 `protected`，则派生列表中使用的访问标号决定该成员在派生类中的访问级别：

- 如果是**公用继承** (`public inheritance`)，基类成员保持自己的访问级别：基类的 `public` 成员为派生类的 `public` 成员，基类的 `protected` 成员为派生类的 `protected` 成员。
- 如果是**受保护继承** (`protected inheritance`)，基类的 `public` 和 `protected` 成员在派生类中为 `protected` 成员。
- 如果是**私有继承** (`private inheritance`)，基类的所有成员在派生类中为 `private` 成员。

例如，考虑下面的继承层次：

571

```
class Base {
public:
    void basemem(); // public member
protected:
    int i;           // protected member
    // ...
};
struct Public_derived : public Base {
    int use_base() { return i; } // ok: derived classes can access i
    // ...
};
struct Private_derived : private Base {
    int use_base() { return i; } // ok: derived classes can access i
};
```

无论派生列表中是什么访问标号，所有继承 `Base` 的类对 `Base` 中的成员具有相同的访问。派生访问标号将控制派生类的用户对从 `Base` 继承而来的成员的访问：

```
Base b;
Public_derived d1;
Private_derived d2;
b.basemem(); // ok: basemem is public
d1.basemem(); // ok: basemem is public in the derived class
d2.basemem(); // error: basemem is private in the derived class
```

`Public_derived` 和 `Private_derived` 都继承了 `basemem` 函数。当进行 `public` 继承时，该成员保持其访问标号，所以，`d1` 可以调用 `basemem`。在 `Private_derived` 中，`Base` 的成员为 `private`，`Private_derived` 的用户不能调用 `basemem`。

派生访问标号还控制来自非直接派生类的访问：

```
struct Derived_from_Private : public Private_derived {
    // error: Base::i is private in Private_derived
    int use_base() { return i; }
};
struct Derived_from_Public : public Public_derived {
    // ok: Base::i remains protected in Public_derived
    int use_base() { return i; }
};
```

从 `Public_derived` 派生的类可以访问来自 `Base` 类的 `i`，是因为该成员在 `Public_derived` 中仍为 `protected` 成员。从 `Private_derived` 派生的类没有这样的访问，对它们而言，`Private_`

572 derived 从 Base 继承的所有成员均为 private。

1. 接口继承与实现继承

public 派生类继承基类的接口，它具有与基类相同的接口。设计良好的类层次中，public 派生类的对象可以用在任何需要基类对象的地方。

使用 private 或 protected 派生的类不继承基类的接口，相反，这些派生通常被称为实现继承。派生类在实现中使用被继承类但继承基类的部分并未成为其接口的一部分。

如 15.3 节所介绍的，类是使用接口继承还是实现继承对派生类的用户具有重要含义。

注解

迄今为止，最常见的继承形式是 public。

关键概念：继承与组合

继承层次的设计本身是个复杂的主题，已超出本书的范围。但是，有一个重要的设计指南非常基础，每个程序员都应该熟悉它。

定义一个类作为另一个类的公用派生类时，派生类应反映与基类的“是一种 (Is A)”关系。在书店例子中，基类表示按规定价格销售的书的概念，Bulk_item 是一种书，但具有不同的定价策略。

类型之间另一种常见的关系是称为“有一个 (Has A)”的关系。书店例子中的类具有价格和 ISBN。通过“有一个”关系而相关的类型暗含有成员关系，因此，书店例子中的类由表示价格和 ISBN 的成员组成。

2. 去除个别成员

如果进行 private 或 protected 继承，则基类成员的访问级别在派生类中比在基类中更受限：

```
class Base {
public:
    std::size_t size() const { return n; }
protected:
    std::size_t n;
};
class Derived : private Base { . . . };
```

注解

派生类可以恢复继承成员的访问级别，但不能使访问级别比基类中原来指定的更严格或更宽松。

573 在这一继承层次中，size 在 Base 中为 public，但在 Derived 中为 private。为了使 size 在 Derived 中成为 public，可以在 Derived 的 public 部分增加一个 using 声明。如下这样改变 Derived 的定义，可以使 size 成员能够被用户访问，并使 n 能够被从 Derived 派生的类访问：

```
class Derived : private Base {
public:
    // maintain access levels for members related to the size of the object
    using Base::size;
```