三、类设计者的工具

13. 拷贝控制

类可以定义构造函数，用来控制在创建此类对象时做什么。本章还将学习类如何控制该类型对象拷贝、赋值、移动或销毁时做什么。类通过一些特殊的成员函数控制这些操作，包括：拷贝构造函数、移动构造函数、拷贝赋值运算符、移动赋值运算符以及析构函数。

拷贝和移动构造函数定义了当用同类型的另一个对象初始化本对象时做什么。拷贝和移动赋值运算符定义了将一个对象赋予同类型的另一个对象时做什么。析构函数定义了当此类型对象销毁时做什么。称这些操作作为拷贝控制操作（copy control）。

如果一个类没有定义所有这些拷贝控制成员，编译器会自动为它定义缺失的操作。因此，很多类会忽略这些拷贝控制操作。但是，对一些类来说，依赖这些操作的默认定义会导致灾难，通常，实现拷贝控制操作最困难的地方是认识到什么时候需要定义这些操作。

13.1 拷贝、赋值与销毁

13.1.1 拷贝构造函数

构造函数的第一个参数是自身类型的引用，且任何额外参数都有默认值，则此构造函数是拷贝构造函数。

拷贝构造函数在几种情况下都会被隐式地使用，因此它不应该是 explicit 的。

合成拷贝构造函数

如果没有为一个类定义拷贝构造函数，编译器会为我们定义一个，。与合成默认构造函数（7.1.4节）不同，即使我们定义了其他构造函数，编译器也会为我们合成一个拷贝构造函数。

13.1.6节，对于某些类来说，合成拷贝构造函数（synthesized copy consturctor）用来阻止我们拷贝该类类型的对象。一般情况下，合成的拷贝构造函数会将其参数的成员逐个拷贝到正在创建的对象中（7.1.5节）。编译器从给定对象中依次将每个非 static 成员拷贝到正在创建的对象中。

每个成员的类型决定了它如何拷贝：对类类型的成员，会使用其拷贝构造函数来拷贝；内置类型的成员则直接拷贝，虽然不能直接拷贝一个数组（3.5.1节），但合成拷贝构造函数会逐元素地拷贝一个数组类型的成员。如果数组元素是类类型，则使用元素的拷贝构造函数来进行拷贝。

举例，Sales_data 类的合成拷贝构造函数等价于：

```
1 class Sales_data {
```

```
public:
    // ...
    // 与合成拷贝构造函数等价的拷贝构造函数的声明
    Sales_data(const Sales_data&);
private:
    std::string bookNo;
    int units_sold = 0;
    double revenue = 0.0;
};
// 与 Sales_data 和合成拷贝构造函数等价
Sales_data::Sales_data(const Sales_data &orig):
    bookNo(orig.bookNo),        //使用string的拷贝构造函数
    unit_sold(orig.units_sold), //拷贝orig.units_sold
    revenue(orig.revenue)       //拷贝orig.revenue
    { }                         //空函数体
```

拷贝初始化

直接初始化和拷贝构造初始化的差异

```
string dots(10, '.');            //直接初始化
string s(dots);                  //直接初始化
stirng s2 = dots;                //拷贝初始化
string null_book = "9-999-9999-9"; //拷贝初始化
string nines = stirng(100,'9');    //拷贝初始化
```

使用直接初始化时，实际上是要求编译器使用普通的函数匹配（6.4节）来选择与我们提供的参数最匹配的构造函数。当我们使用拷贝初始化（copy initialization）时，要求编译器将右侧运算对象拷贝到正在创建的对象中，如果需要还要进行类型转换（7.5.4节）。

拷贝初始化通常使用拷贝构造函数来完成。但是，13.6.2节，如果一个类有一个移动构造函数，则拷贝初始化有时会使用移动构造函数而非拷贝构造函数来完成。现在只需了解拷贝初始化何时发生，以及拷贝初始化是依靠构造函数或移动构造函数来完成的就可以了。

拷贝初始化不仅在用 = 定义变量时会发生，下列情况下也会发生：

- 将一个对象作为实参传递给一个非引用类型的形参
- 从一个返回类型为非引用类型的函数返回一个对象
- 用花括号列表初始化一个数组中的元素或一个聚合类的成员（7.5.5节）

某些类类型还会对它们所分配的对象使用拷贝初始化。例如，初始化标准库容器或是调用其 insert 或 push 成员（9.3.1节）时，容器会对其元素进行拷贝初始化，用 emplace 成员创建的元素都进行直接初始化。

参数和返回值

函数调用过程中，具有非引用类型的参数要进行拷贝初始化（6.2.1节）。当一个函数具有非引用的返回类型时，返回值会被用来初始化调用方的结果（6.3.2节）。

拷贝构造函数被用来初始化非引用类类型参数，这一特性解释了为什么拷贝构造函数自己的参数必须是引用类型。

拷贝初始化的限制

编译器可以绕过拷贝构造函数

拷贝初始化过程中，编译器可以跳过拷贝/移动构造函数，直接创建对象。

```
// 编译器被允许将下面的代码
string null_book = "9-999-99999-9"; //拷贝初始化
// 改写为
string null_book("9-999-99999-9");  //编译器略过了拷贝构造函数
```

但是，拷贝/移动构造函数必须是存在且可访问的。

13.1.2 拷贝赋值运算符

与类控制其对象如何初始化一样，类也可以控制其对象如何赋值。如果类未定义自己的拷贝赋值运算符，编译器会为它合成一个。

重载赋值运算符

如果一个运算符是一个成员函数，其左侧运算对象就绑定到隐式的 this 参数。对于一个二元运算符，例如赋值运算符，其右侧运算对象作为显式参数传递。

> 赋值运算符通常应该返回一个车指向其左侧运算对象的引用。

合成拷贝赋值运算符

```
// 等价于合成拷贝赋值运算符
Sales_data&
Sales_data::operator=(const Sales_data &rhs)
```

```
{
    bookNo = rhs.bookNo;            // 调用 string::operator=
    units_sold = rhs.units_sold;    // 使用内置的 int 赋值
    revenue = rhs.revenue;          // 使用内置的 double 赋值
    return *this;                   // 返回一个此对象的引用
}
```

13.1.3 析构函数

析构函数释放对象使用的资源，并销毁对象的非 static 数据成员。

析构函数没有返回值，也不接受参数，因此它不能被重载。对于一个给定类，只会有唯一一个析构函数。

析构函数完成什么工作

如同构造函数有一个初始化部分和一个函数体（7.5.1节），析构函数也有一个函数体和一个析构部分。在一个构造函数中，成员的初始化是在函数体执行之前完成的，且按照它们在类中出现的顺序进行初始化。在一个析构函数中，首先执行函数体，然后销毁成员。成员按初始化顺序的逆序销毁。

在对象最后一次使用之后，析构函数的函数体可执行类设计者希望执行的任何收尾工作。通常，析构函数释放对象在生存期分配的所有资源。

在一个析构函数中，不存在类似构造函数中初始化列表的东西来控制成员如何销毁，析构部分是隐式的。成员销毁时发生什么完全依赖于成员的类型。销毁类类型的成员需要执行成员自己的析构函数。内置类型没有析构函数，因此销毁内置类型成员什么也不需要做。

> 隐式销毁一个内置指针类型的成员不会 delete 它所指向的对象.

与普通指针不同，智能指针是类类型，所以具有析构函数，智能指针成员在析构阶段会被自动销毁。

什么时候会调用析构函数

无论何时一个对象被销毁，就会自动调用其析构函数：

- 变量在离开其作用域时被销毁
- 当一个对象被销毁时，其成员被销毁
- 容器（无论是标准库容器还是数组）被销毁时，其元素被销毁
- 对于动态分配的对象，当对指向它的指针应用 delete 运算符时被销毁
- 对于临时对象，当创建它的完整表达式结束时被销毁

由于析构函数自动运行，程序可以按需分配资源，而无需担心何时释放这些资源。

下面代码定义了四个 Sales_data 对象：

```
{   // 新作用域
    // p和p2指向动态分配的对象
    Sales_data *p = new Sales_data;        // p 是一个内置指针
    auto p2 = make_shared<Sales_data>();  // p2 是一个 shared_ptr
    Sales_data item(*p);     // 拷贝构造函数将 *p 拷贝到 item 中
    vector<Sales_data> vec; // 局部对象
    vec.push_back(*p2);      // 拷贝 p2 指向的对象
    delete p;                // 对 p 指向的对象执行析构函数
}    // 退出局部作用域；对 item、p2 和 vec 调用析构函数
     // 销毁 p2 会递减其引用计数；如果引用计数变为 0，对象被释放
     // 销毁 vec 会销毁它的元素
```

> 当指向一个对象的引用或指针离开作用域时，析构函数不会执行。

合成析构函数

13.1.4 三 / 五法则

需要析构函数的类也需要拷贝和赋值操作

需要拷贝操作的类也需要赋值操作，反之亦然

13.1.5 使用 = default

类内用 = default 修饰成员声明时，合成的函数将隐式地声明为内联的。

13.1.6 阻止拷贝

> 大多数类应该定义默认构造函数、拷贝构造函数和拷贝赋值运算符，无论是显式地或是隐式地。

对某些类来说，拷贝构造函数和拷贝赋值运算符没有合理的意义。在此情况下，定义类时必须采用某种机制阻止拷贝或赋值。例如，iostream 类阻止了拷贝，以避免多个对象写入或读取相同的 IO 缓冲。

定义删除的函数

可以通过将拷贝构造函数和拷贝赋值运算符定义为删除的函数（deleted function）来阻止拷贝。

析构函数不能是删除的成员

合成的拷贝控制成员可能是删除的

private 拷贝控制

13.2 拷贝控制和资源管理

13.2.1 行为像值的类

类值拷贝赋值运算符

13.2.2 定义行为像指针的类

引用计数

定义一个使用引用计数的类

类指针的拷贝成员 “篡改” 引用计数

13.3 交换操作

编写自己的 swap 函数

swap 函数应该调用 swap，而不是 std::swap

在赋值运算符中使用 swap

13.4 拷贝控制示例

Message 类

save 和 remove 成员

Message类的拷贝控制成员

Message 的析构函数

Message 的拷贝赋值运算符

Message 的 swap 函数

13.5 动态内存管理类

StrVec 类的设计

StrVec 类定义

使用 construct

alloc_n_copy 成员

free 成员

拷贝控制成员

在重新分配内存的过程中移动而不是拷贝元素

移动构造函数和 std::move

reallocate 成员

13.6 对象移动

新标准的一个最主要的特性是可以移动而非拷贝对象的能力。13.1.1节所见，很多情况下都会发生对象拷贝。某些情况下，对象拷贝后就立即被销毁了。这些情况下，移动而非拷贝对象会大幅提升性能。

StrVec 类是这种不必要的拷贝的例子。在重新分配内存的过程中，从旧内存将元素拷贝到新内存是不必要的，更好的方式是移动元素。使用移动而不是拷贝的另一个原因源于 IO 类或 unique_ptr 这样的类。这些类都包含不能被共享的资源（如指针或IO缓冲）。因此，这些类型的对象不能拷贝但可以移动。新标准中，可以用容器保存不可拷贝的类型，只要它们能被移动即可。

> 标准库容器、string 和 shared_ptr 类既支持移动也支持拷贝。IO 类和 unique_ptr 类可以移动但不能拷贝。

13.6.1 右值引用

为了支持移动操作，引入了一种新的引用类型——右值引用（rvalue reference）。右值引用就是必须绑定到右值的引用。通过 && 而不是 & 来获得右值引用。右值引用有一个重要的性质——只能绑定到一个将要销毁的对象。因此，可以自由地将一个右值引用的资源 “移动” 到另一个对象中。

左值和右值是表达式的属性（4.1.1节）。一些表达式生成或要求左值，而另一些则生成或要求右值。一般而言，一个左值表达式表示的是一个对象的身份，而一个右值表达式表示的是对象的值。

类似任何引用，一个右值引用是某个对象的另一个名字。对于常规引用（为了与右值引用区分开来，可以称之为左值引用），不能将其绑定到要求转换的表达式、字面常量或者是返回右值的表达式（2.3.1节）。右值引用有着完全相反的绑定特性：可以将一个右值引用绑定到这类表达式上，但不能将一个右值引用直接绑定到一个左值上：

```
1 int i = 42;
2 int &r = i;        //正确：r 引用 i
```

```
3 int &&rr = i;       //错误：不能将一个右值引用绑定到一个左值上
4 int &r2 = i * 42;   //错误：i*42 是一个右值
5 const int &r3 = i * 42;//正确：可以将一个const的引用绑定到一个右值上
6 int &&rr2 = i * 42;//正确：将rr2绑定到乘法结果上
```

返回左值引用的函数，连同赋值、下标、解引用和前置递增/递减运算符，都是返回左值的表达式的例子。可以将一个左值引用绑定到这类表达式的结果上。

返回非引用类型的函数，连同算术、关系、位以及后置递增/递减运算符，都生成右值。不能将一个左值引用绑定到这类表达式的结果上。但可以将一个 const 的左值引用或者一个右值引用绑定到这类表达式上。

左值持久，右值短暂

左值和右值表达式的列表，两者相互区别很明显：左值有持久的状态，而右值要么是字面常量，要么是在表达式求值过程中创建的临时对象。

由于右值引用只能绑定到临时对象，可以得知

- 所引用的对象将要被销毁
- 该对象没有其他用户

这两个特性意味着：使用右值引用的代码可以自由地接管所引用的对象的资源。

> 右值引用指向将要被销毁的对象，因此可以从绑定到右值引用的对象“窃取”状态

变量是左值

变量可以看做只有一个运算对象而没有运算符的表达式。变量表达式都是左值。

```
1 int &&rr1 = 42;    //正确：字面常量是右值
2 int &&rr2 = rr1;   //错误：表达式 rr1 是左值
```

变量是持久的，直至离开作用域时才被销毁。

标准库 move 函数

不能将一个右值引用直接绑定到一个左值上，但可以显式地将一个左值转换为对应的右值类型。还可以通过调用一个名为 move 的新标准库函数来获得绑定到左值上的右值引用。move 函数使用了在16.2.6节描述的机制来返回给定对象的右值引用。

```
1 int &&rr3 = std::move(rr1);    //ok
```

move 调用告诉编译器：我们有一个左值，但希望像一个右值一样处理它。调用move就意味着承诺：除了对 rr1 赋值或销毁它外，将不再使用它。调用 move 之后，不能对移后源对象的值做任何假设。

> 可以销毁一个移后源对象，也可以赋予它新值，但不能使用一个移后源对象的值。

与大多数标准库名字的使用不同，对 move 不提供 using 声明。直接调用 std::move，原因 18.2.3节。避免潜在的名字冲突。

13.6.2 移动构造函数和移动赋值运算符

移动操作、标准库容器和异常

移动赋值运算符

移动源对象必须可析构

合成的移动操作

移动右值，拷贝左值...

...但如果没有移动构造函数，右值也被拷贝

拷贝并交换赋值运算符和移动操作

Message 类的移动操作

移动迭代器

13.6.3 右值引用和成员函数

> 区分移动和拷贝的重载函数通常有一个版本接受一个 const T&，而另一个版本接受一个 T&&。

右值和左值引用成员函数

重载和引用函数

> 如果一个成员函数有引用限定符，则具有相同参数列表的所有版本都必须有引用限定符。

小结

术语

14. 操作重载与类型转换

14.1 基本概念

> 当一个重载运算符是成员函数时，this 绑定到左侧运算对象。成员运算符函数的（显式）参数数量比运算对象的数量少一个。

对于运算符函数来说，它或者是类的成员，或者至少含有一个类类型的参数。

```
1 // 错误：不能为 int 重定义内置的运算符
2 int operator+(int, int);
```

这一约定意味着当运算符作用于内置类型的运算对象时，我们无法改变该运算符的含义。

表14.1：运算符

可以被重载的运算符					
+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-+	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]
不能被重载的运算符					
::	.*	.	?:		

直接调用一个重载的运算符函数

某些运算符不应该被重载

使用与内置类型一致的含义

赋值和复合赋值运算符

选择作为成员或者非成员

14.2 输入和输出运算符

14.2.1 重载输出运算符 <<

Sales_data 的输出运算符

输出运算符尽量减少格式化操作

输入输出运算符必须是非成员函数

14.2.2 重载输入运算符 >>

Sales_data 的输入运算符

输入时的错误

标示错误

14.3 算术和关系运算符

14.3.1 相等运算符

14.3.2 关系运算符

14.4 赋值运算符

复合赋值运算符

14.5 下标运算符

14.6 递增和递减运算符

定义前置递增/递减运算符

区分前置和后置运算符

显式调用后置运算符

14.7 成员访问运算符

对箭头运算符返回值的限定

14.8 函数调用运算符

含有状态的函数对象类

14.8.1 lambda是函数对象

表示 lambda 及相应捕获行为的类

14.8.2 标准库定义的函数对象

标准库定义了一组表示算术运算符、关系运算符和逻辑运算符的类，每个类分别定义了一个执行明明操作的调用运算符。

```
plus<int> intAdd;         //可执行 int 加法的函数对
negate<int> intNegate;    //可对 int 值取反的函数对象
// 使用 intAdd::operator(int, int)
int sum = intAdd(10, 20);
sum = negate(intAdd(10, 20));
```

表14.2：标准库函数对象

算术	关系	逻辑
plus<Type>	equal_to<Type>	logical_and<Type>
minus<Type>	not_equal_to<Type>	logical_or<Type>
multiplies<Type>	greater<Type>	logical_not<Type>
divides<Type>	greater_equal<Type>	

modulus<Type>	less<Type>	
negate<Type>	less_equal<Type>	

算法中使用标准库函数对象

```
// 传入一个临时函数对象用于执行两个 string 对象 > 比较运算
sort(svec.begin(), svec.end(), greater<string>());
```

注意，标准库规定其函数对象对于指针同样适用。比较两个无关指针将产生未定义的行为，然而我们可能希望通过比较指针的内存地址来 sort 指针的 vector。直接这么做产生 undefined 行为，可以使用一个标准库函数对象来实现该目的：

```
vector<string *> nameTable;    // 指针的vector
// 错误：nameTable 中的指针彼此间没有关系，所以 < 将产生未定义行为
sort(nameTable.begin(), nameTable.end(),
    [](string *a, string *b) { return a < b; });
//正确：标准库规定的指针的 less 是定义良好的
sort(nameTable.begin(), nameTable.end(), less<string*>());
```

关联容器使用 less<key_type> 对元素排序，因此可以定一个指针的 set 或者在 map 中使用指针作为关键值而无需直接声明 less。

14.8.3 可调用对象与function

C++中有几种可调用的对象：函数、函数指针、lambda表达式（10.3.2节）、bind创建的对象（10.3.4节）以及重载了函数调用运算符的类。

和其他对象一样，可调用的对象也有类型。例如，每个 lambda 有它自己唯一的（未命名）类类型；函数及函数指针的类型则由其返回值类型和实参类型决定。

不同类型可能具有相同的调用形式

标准库 function 类型

重载的函数与 function

14.9 重载、类型转换与运算符

14.9.1 类型转换运算符

定义含有类型转换运算符的类

类型转换运算符可能产生意外结果

显式的类型转换运算符

转换为 bool

14.9.2 避免有二义性的类型转换

实参匹配和相同的类型转换

二义性与转换目标为内置类型的多重类型转换

重载函数与转换构造函数

重载函数与用户定义的类型转换

14.9.3 函数匹配与重载运算符

小结

术语

15. 面向对象程序设计

面向对象程序设计基于三个基本概念：数据抽象、继承和动态绑定。第 7 章介绍了数据抽象，本章介绍继承和动态绑定。

继承和动态绑定对程序的编写有两方面影响：一是可以更容易定义与其他类相似但不完全相同的新类；二是在使用这些彼此相似的类编写程序时，可以在一定程度上忽略掉它们的区别。

15.1 OOP：概述

面向对象程序设计（object-oriented programming）的核心思想是数据抽象、继承和动态绑定。通过使用数据抽象，可以将类的接口与实现分离；使用继承，可以定义相似的类型并对其相似关系建模；使用动态绑定，可以在一定程度上忽略相似类型的区别，而以统一的方式使用它们的对象。

继承

C++ 中，基类将类型相关的函数与派生类不做改变直接继承的函数区分对待。对于某些函数，基类希望它的派生类各自定义自身的版本，此时基类就将这些函数声明成虚函数。

动态绑定

函数的运行版本由实参决定，即在运行时选择函数的版本，所以动态绑定又被称为运行时绑定。

> C++中，当我们使用基类的引用（或指针）调用一个虚函数时将发生动态绑定。

15.2 定义基类和派生类

15.2.1 定义基类

> 基类通常都应该定义一个虚析构函数，即使该函数不执行任何实际操作也是如此。

成员函数与继承

访问控制与继承

15.2.2 定义派生类

派生类中的虚函数

派生类对象及派生类向基类的类型转换

派生类构造函数

派生类使用基类的成员

继承与静态成员

派生类的声明

被用作基类的类

防止继承的发生

15.2.3 类型转换与继承

> 理解基类和派生类之间的类型转换是理解C++面向对象编程的关键

通常，如果我们想把引用或指针绑定到一个对象上，则引用或指针的类型应与对象的类型一致，或者对象的类型含有一个可接受的 const 类型转换规则（4.11.2节）。存在继承关系的类是一个重要的例外：我们可以将基类的指针或引用绑定到派生类对象上。例如，可以用 Quote& 指向一个 Bulk_quote 对象，也可以把一个 Bulk_quote 对象的地址赋给一个 Quote*。

可以将基类的指针或引用绑定到派生类对象上有一层重要含义：当使用基类的引用（或指针）时，实际上我们并不清楚该引用（或指针）所绑定对象的真实类型。该对象可能是基类的对象，也可能是派生类的对象。

> 和内置指针一样，智能指针类（12.1节）也支持派生类向基类的类型转换，意味着我们可以将一个派生类对象的指针存储在一个基类的智能指针内。

静态类型与动态类型

使用存在继承关系的类型时，必须将一个变量或其他表达式的静态类型与该表达式表示对象的动态类型区分开来。表达式的静态类型在编译时总是已知的，它是变量声明时的类型或表达式生成的类型；动态类型则是变量或表达式表示的内存中的对象的类型。动态类型直至运行时才可知。

```
1 // 当 print_total 调用 net_price 时（15.1节）
2 double ret = item.net_price(n);
```

item 的静态类型是 Quote&，它的动态类型则依赖于 item 绑定的实参，动态类型直到在运行时调用该函数时才会知道。如果传递一个 Bulk_quote 对象给 print_total，则 item 的 静态类型将与它的动态类型不一致。item 的静态类型是 Quote&，而在此例中它的动态类型则是 Bulk_quote。

如果表达式既不是引用也不是指针，则它的动态类型永远与静态类型一致。例如，Quote 类型的变量永远是一个 Quote 对象，无论如何都不能改变该变量对应的对象的类型。

> 基类的指针或引用的静态类型可能与其动态类型不一致。

不存在从基类向派生类的隐式类型转换...

之所以存在派生类向基类的类型转换是因为每个派生类对象都包含一个基类部分，而基类的引用或指针可以绑定到该基类部分上。一个基类的对象既可以以独立的形式存在，也可以作为派生类对象的一部分存在。

因为一个基类的对象可能是派生类对象的一部分，也可能不是，所以不存在基类向派生类的自动类型转换：

```
1 Quote base;
2 Bulk_quote* bulkP = &base;    //错误：不能将基类转换成派生类
3 Bulk_quote& bulkRef = base;   //错误：不能将基类转换成派生类
```

如果上述赋值是合法的，则我们有可能会使用 bulkP 或 bulkRef 访问 base 中本不存在的成员。

还有一种情况，即使一个基类指针或引用绑定在一个派生类对象上，也不能执行从基类向派生类的转换：

```
Bulk_quote bulk;
Quote *itemp = &bulk;          //正确：动态类型是 Bulk_quote
Bulk_quote *bulkP = itemP;     //错误：不能将基类转换成派生类
```

编译器在编译时无法确定某个特定的转换在运行时是否安全，因为编译器只能通过检查指针或引用的静态类型来推断该转换是否合法。如果基类中含有一个或多个虚函数，我们可以使用 dynamic_cast（19.2.1节）请求一个类型转换，该转换的安全检查将在运行时执行。同样，如果已知某个基类向派生类的转换是安全的，则可以使用 static_cast（4.11.3节）来强制覆盖掉编译器的检查工作。

...在对象之间不存在类型转换

派生类向基类的自动类型转换只对指针或引用类型有效，在派生类类型和基类类型之间不存在这样的转换。

> 当我们用一个派生类对象为一个基类对象初始化或赋值时，只有该派生类对象中的基类部分会被拷贝、移动或赋值，它的派生类部分将被忽略掉。

> 理解存在继承关系的类型之间的转换规则，有三点非常重要
>
> - 从派生类向基类的类型转换只对指针或引用类型有效
> - 基类向派生类不存在隐式类型转换
> - 和任何其他成员一样，派生类向基类的类型转换也可能会由于访问受限而变得不可行。15.5节
>
> 自动类型转换只对指针或引用类型有效，但是继承体系中的大多数类仍然（显式或隐式地）定义了拷贝控制成员。因此，通常能够将一个派生类对象拷贝、移动或赋值给一个基类对象。不过，这种操作只处理派生类对象的基类部分。

15.3 虚函数

对虚函数的调用可能在运行时才被解析

> C++ 的多态性
>
> OOP 的核心思想是多态性（polymorphism）。多态性，含义是 “多种形式”。我们把具有继承关系的多个类型称为多态类型，因为我们能使用这些类型的 “多种形式” 而无须在意它们的差异。引用或指针的静态类型与动态类型不同这一事实正是 C++ 支持多态性的根本所在。

派生类中的虚函数

final 和 override 说明符

虚函数与默认实参

回避虚函数的机制

15.4 抽象基类

纯虚函数

含有纯虚函数的类是抽象基类

> 不能创建抽象基类的对象

派生类构造函数只初始化它的直接基类

> 重构

15.5 访问控制与继承

每个类分别控制自己的成员初始化过程（15.2.2节），与之类似，每个类还分别控制着其成员对于派生类来说是否可访问（accessible）。

受保护的成员

一个类使用 protected 关键字来声明那些它希望与派生类分享但是不想被其他公共访问使用的成员。protected 说明符可以看作是 public 和 private 中和后的产物：

- 和私有成员类似，受保护的成员对于类的用户来说是不可访问的
- 和共有成员类似，受保护的成员对于派生类的成员和友元来说是可访问的
- 派生类的成员或友元只能通过派生类对象来访问基类的受保护成员。派生类对于一个基类对象中的受保护成员没有任何访问特权

公有、私有和受保护继承

某个类对其继承而来的成员的访问权限收到两个因素影响：一是在基类中该成员的访问说明符，二是在派生类的派生列表中的访问说明符。考虑如下的继承关系：

```
1 class Base {
```

```
public:
    void pub_mem();     // public 成员
protected:
    int prot_mem;       // protected 成员
private:
    char priv_mem;      // private 成员
};

struct Pub_Derv : public Base {
    // 正确：派生类能访问 proctected 成员
    int f() { return prot_mem; }
    // 错误：private 成员对于派生类来说是不可访问的
    char g() { return priv_mem; }
};

struct Priv_Derv : private Base {
    // private 不影响派生类的访问权限
    int f1() const { return prot_mem; }
};
```

派生访问说明符对于派生类的成员（及友元）能否访问其直接基类的成员没什么影响。对基类成员的访问权限只与基类中的访问说明符有关。

派生访问说明符的目的是控制派生类用户（包括派生类的派生类在内）对于基类成员的访问权限：

派生类向基类转换的可访问性

> 对于代码中的某个给定节点来说，如果基类的公有成员是可访问的，则派生类向基类的类型转换也是可访问的；反之不行

> **关键概念：类的设计与受保护的成员**
>
> 不考虑继承的话，可以认为一个类有两种不同的用户：普通用户和类的实现者。其中，普通用户编写的代码使用类的对象，这部分代码只能访问类的公有（接口）成员；实现者则负责编写类的成员和友元的代码，成员和友元既能访问类的公有部分，也能访问类的私有（实现）部分。
>
> 进一步考虑继承的话就会出现第三种用户，即派生类。基类把它希望派生类能够使用的部分声明成受保护的。普通用户不能访问受保护的成员，而派生类及其友元仍旧不能访问私有成员。
>
> 和其他类一样，基类应该将其接口成员声明为公有的；同时将属于其实现的部分分成两组：一组可供派生类访问，另一组只能由基类及基类的友元访问。对于前者应该声明为受保护的，这样派生类就能在实现自己的功能时使用基类的这些操作和数据；对于后者应该声明为私有的。

友元与继承

> 不能继承友元关系；每个类负责控制各自成员的访问权限

改变个别成员的可访问性

> 派生类只能为那些它可以访问的名字提供 using 声明

默认的继承保护级别

struct 和 class 关键字定义的类具有不同的默认访问说明符。类似，默认派生运算符也由定义派生类所用的关键字来决定。默认情况下，使用 class 关键字定义的派生类是私有继承的；而使用 struct 关键字定义的派生类是公有继承的。它们唯一的差别就是默认成员访问说明符及默认派生访问说明符。

> 一个私有派生的类最好显式地将 private 声明出来，而不要仅仅依赖于默认的设置。令私有继承关系清晰明了。

15.6 继承中的类作用域

在编译时进行名字查找

名字冲突与继承

通过作用域运算符来使用隐藏的成员

名字查找先于类型检查

虚函数与作用域

通过基类调用隐藏的虚函数

覆盖重载的函数

15.7 构造函数与拷贝控制

15.7.1 虚析构函数

虚析构函数将阻止合成移动操作

15.7.2 合成拷贝控制与继承

派生类中删除的拷贝控制与基类的关系

移动操作与继承

15.7.3 派生类的拷贝控制成员

定义派生类的拷贝或移动构造函数

派生类赋值运算符

派生类析构函数

在构造函数和析构函数中调用虚函数

15.7.4 继承的构造函数

继承的构造函数的特点

15.8 容器与继承

在容器中放置（智能）指针而非对象

15.8.1 编写 Basket 类

定义 Basket 的成员

隐藏指针

模拟虚拷贝

15.9 文本查询程序再探

15.9.1 面向对象的解决方案

抽象基类

将层次关系隐藏于接口类中

理解这些类的工作原理

15.9.2 Query_base类和Query类

Query 类

Query 的输出运算符

15.9.3 派生类

WordQuery 类

NotQuery 类及 ~运算符

BinaryQuery 类

AndQuery 类、OrQuery 类及相应的运算符

15.9.4 eval 函数

OrQuery::eval

AndQuery::eval

NotQuery::eval

小结

继承使得我们可以编写一些新类，这些新类既能共享其基类的行为，又能根据需要覆盖或添加行为。动态绑定使得我们可以忽略类型之间的差异，其机理是在运行时根据对象的动态类型来选择运行函数的哪个版本。继承和动态绑定的结合使得我们能够编写具有特定类型行为但又独立于类型的程序。

C++ 中，动态绑定只作用于虚函数，并且需要通过指针或引用调用。

在派生类对象中包含有与它的每个基类对应的子对象。因为所有派生类对象都含有基类部分，所以我们能将派生类的引用或指针转换为一个可访问的基类引用或指针。

当执行派生类的构造、拷贝、移动和赋值操作时，首先构造、拷贝、移动和赋值其中的基类部分，然后才轮到派生类部分。析构函数的执行顺序正好相反，首先销毁派生类，然后执行基类子对象的析构函数。基类通常定义一个虚析构函数，即使基类根本不需要析构函数也最好这么做。将基类的析构函数定义成虚函数的原因是为了确保当我们删除一个基类指针，而该指针实际指向一个派生类对象时，程序也能正确运行。

派生类为它的每个基类提供一个保护级别。public 基类的成员也是派生类接口的一部分；private 基类的成员是不可访问的；protected 基类的成员对于派生类的派生类是可访问的，但是对于派生类的用户是不可访问的。

术语表

16. 模版与泛型编程

面向对象编程（OOP）和泛型编程都能处理在编写程序时不知道类型的情况。不同之处在于：OOP 能处理类型在程序运行之前都未知的情况；而在泛型编程中，在编译时就能获知类型了。

容器、迭代器和算法都是泛型编程的例子。编写一个泛型程序时，是独立于任何特定类型来编写代码的。当使用一个泛型程序时，我们提供类型或值，程序实例可在其上运行。

标准库为每个容器提供了单一的、泛型的定义，如 vector。可以使用这个泛型定义很多类型的 vector，它们的差异在于包含的元素类型不同。

模板是 C++ 中泛型编程的基础。一个模板就是一个创建类或函数的蓝图或者公式。当使用一个 vector 这样的泛型类型，或者 find 这样的泛型函数时，提供足够的信息，将蓝图转换成特定的类。这种转换发生在编译时。

16.1 定义模板

写一个函数来比较两个值，初次尝试可能定义多个重载函数。

16.1.1 函数模板

可以定义一个通用的函数模板（function template），而不是为每个类型都定一个新函数。一个函数模板就是一个公式，可用来生成针对特定类型的函数版本。compare 的模板版本可能像下面这样：

```
1 template <typename T>
2 int compare(const T &v1, const T &v2)
3 {
4     if (v1 < v2) return -1;
5     if (v2 < v1) return 1;
6     return 0;
7 }
```

模板定义以关键字 template 开始，后跟一个模板参数列表（template parameter list），这是一个逗号分隔的一个或多个模板参数（template parameter）的列表，<>包起来。

模板参数列表的作用很像函数参数列表。函数参数列表定义了若干特定类型的局部变量，但并未指出如何初始化它们。在运行时，调用者提供实参来初始化形参。

类似，模板参数表示在类或函数定义中用到的类型或值。当使用模板时，我们（隐式或显式地）指定模板实参（template argument），将其绑定到模板参数上。

compare 函数声明一个名为 T 的类型参数，T 表示一个类型，实际类型则在编译时根据 compare 的使用情况来确定。

实例化函数模版

编译器用推断出的模板参数来为我们实例化（instantiate）一个特定版本的函数。当编译器实例化一个模板时，它使用实际的模板实参代替对应的模板参数来创建出模板的一个新实例。

模版类型参数

可以将类型参数看做类型说明符，就像内置类型或类类型说明符一样使用。

类型参数前必须使用关键字 class 或 typename。在模板参数列表中，这两个关键字含义相同，可以互换使用。一个模板参数列表中可以同时使用这两个关键字：

```
// 在模板参数列表中，typename 和 class 没有什么不同
template <typename T, class U> calc (const T&, const U&);
```

非类型模版参数

除了定义类型参数，还可以在模板中定义非类型参数（nontype parameter）。一个非类型参数表示一个值而非一个类型。

当一个模板被实例化时，非类型参数被一个用户提供的或编译器推断出的值所代替。这些值必须是常量表达式（2.4.4节），从而允许编译器在编译时实例化模板。

编写一个compare版本处理字符串字面常量。这种字面常量是 const char 的数组。由于不能拷贝一个数组，所以将自己的参数定义为数组的引用。

```
template<unsigned N, unsigned M>
int compare(const char (&p1)[N], const char (&p2)[M])
{
    return strcmp(p1, p2)
}
```

在模板定义内，模板非类型参数是一个常量值。在需要常量表达式的地方，可以使用非类型参数，例如，指定数组大小。

> 非类型模板参数的模板实参必须是常量表达式。

inline 和 constexpr 的函数模版

函数模板可以声明为 inline 或 constexpr 的，说明符放在模板参数列表之后，返回类型之前。

编写类型无关的代码

compare 函数虽然简单，但它说明了编写泛型代码的两个重要原则：

- 模板中的函数参数是 const 的引用
- 函数体中的条件判断仅使用 < 比较运算

通过将函数参数设定为 const 的引用，保证了函数可以用于不能拷贝的类型。大多数类型，包括内置类型和我们已经用过的标准库类型（除 unique_ptr 和 IO 类型之外）都是允许拷贝的。但是，不允许拷贝的类类型也是存在的。通过将参数设定为 const 的引用，保证了这些类型可以用我们的 compare 函数来处理。而且，如果 compare 用于处理大对象，这种设计策略还能是函数运行更快。

如果真的关心类型无关和可移植性，可能需要用 less（14.8.2节）来定义函数：

```
1 // 即使用于指针也正确的 compare 版本
2 template <typename T> int compare(const T &v1 , const T &v2)
3 {
4     if (less<T>()(v1, v2)) return -1;
5     if (less<T>()(v1, v2)) return 1;
6     return 0;
7 }
```

之前版本存在的问题是，如果用户当调用它来比较两个指针，且两个指针未指向相同的数组，则代码的行为是未定义的。

> 模板程序应该尽量减少对实际类型的要求

模版编译

当编译器遇到一个模板定义时，它并不生成代码。只有当实例化除模板的一个特定版本时，编译器才会生成代码。使用（而不是定义）模板时，编译器才生成代码。这一特性影响了如何组织代码以及错误何时被检测到。

通常，当我们调用一个函数时，编译器只需要掌握函数的声明。类似，当使用一个类类型的对象时，类定义必须是可用的，但成员函数的定义不必已经出现。因此，将类定义和函数声明放在头文件中，而普通函数和类的成员函数的定义放在源文件中。

模板不同：为了生成一个实例化版本，编译器需要掌握函数模板或类模板成员函数的定义。因此，模板的头文件通常既包括声明也包括定义。

> 函数模板和类模板成员函数的定义通常放在头文件中。

大多数编译错误在实例化期间报告

模板知道实例化时才会生成代码，这一特性影响了我们何时才会获知模板内代码的编译错误。通常，编译器会在三个阶段报告错误：

第一个阶段是编译模板本身时。这个阶段，编译器通常不会发现很多错误，可以检查语法错误。

第二个阶段是编译器遇到模板使用时。此阶段，仍然没有很多可检查的。对于函数模板调用，编译器通常会检查实参数目是否正确、检查参数类型是否匹配。

第三个阶段是模板实例化时，只有这个阶段才能发现类型相关的错误。依赖于编译器如何管理实例化，这类错误可能在链接时才报告。

16.1.2 类模板

与函数模板不同，编译器不能为类模板推断模板参数类型。

定义类模板

实例化类模板

在模板作用域中引用模板类型

类模板的成员函数

check 和元素访问成员

Blob 构造函数

类模板成员函数的实例化

在类代码内简化模板类名的使用

在类模板外使用类模板名

类模板和友元

一对一友好关系

通用和特定的模板友好关系

令模板自己的类型参数成为友元

模板类型别名

类模板的 static 成员

16.1.3 模板参数

模板参数与作用域

模板声明

使用类的类型成员

默认模板实参

模板默认实参与类模板

16.1.4 成员模板

普通（非模板）类的成员模板

类模板的成员模板

实例化与成员模板

16.1.5 控制实例化

实例化定义会实例化所有成员

16.1.6 效率与灵活性

在运行时绑定删除器

在编译时绑定删除器

通过在编译时绑定删除器，unique_ptr 避免了间接调用删除器的运行时开销。通过在运行时绑定删除器，shared_ptr 使用户重载删除器更为方便。

16.2 模板实参推断

16.2.1 类型转换与模版类型参数

使用相同模板参数类型的函数形参

正常类型转换应用于普通函数实参

16.2.2 函数模版显式实参

指定显式模板实参

正常类型转换应用于显式指定的实参

16.2.3 尾置返回类型与类型转换

进行类型转换的标准库模板类

16.2.4 函数指针和实参推断

16.2.5 模板实参推断和引用

从左值引用函数参数推断类型

从右值引用函数参数推断类型

引用折叠和右值引用参数

编写接受右值引用参数的模板函数

16.2.6 理解 std::move

std::move 是如何定义的

std::move 是如何工作的

从一个左值 static_cast 到一个右值引用是允许的

16.2.7 转发

定义能保持类型信息的函数参数

在调用中使用 std::forward 保持类型信息

16.3 重载与模板

编写重载模板

多个可行模板

非模板和模板重载

重载模板和类型转换

缺少声明可能导致程序行为异常

16.4 可变参数模板

sizeof... 运算符

16.4.1 编写可变参数函数模板

16.4.2 包扩展

理解包扩展

16.4.3 转发参数包

16.5 模板特例化

编写单一模板，使之对任何可能的模板实参都是合适的，都能实例化，不总是能办到的。某些情况下，通用模板的定义对特定类型是不适合的：通用定义可能编译失败或做的不正确。其他时候，也可以利用某些特定知识来编写更高效的代码，而不是从通用模板实例化。当不能使用模板版本时，可以定义类或函数模板的一个特例化版本。

compare 函数例子，它展示了函数模板的通用定义不适合一个特定类型（字符指针）的情况。

```
1 // 第一个版本：可以比较任意两个类型
2 template <typename T> int compare(const T&, const T&);
3
4 // 第二个版本：处理字符串字面常量
5 template<size_t N, size_t M>
6 int compare(const char (&)[N], const char (&)[M]);
```

只有当传递给 compare 一个字符串字面量或者一个数组时，编译器才会调用接受两个非类型模板参数的版本。如果传递给它字符指针，就会调用第一个版本：

```
const char *p1 = "hi", *p2 = "mom";
compare(p1, p2);
compaer("h1", "mom");
```

为了处理字符指针（而不是数组），可以为第一个版本的 compare 定义一个模板特例化版本。一个特例化版本就是模板的一个独立的定义，在其中一个或多个模板参数被指定为特定的类型。

定义函数模板特例化

```
// compare 的特殊版本，处理字符数组的指针
template <>
int compare(const char* const &p1, const char* const &p2)
{
    return strcmp(p1, p2);
}
```

理解此特例化版本的困难是函数参数类型。定义一个特例化版本时，函数参数类型必须与一个先前声明的模板中对应的类型匹配。本例中特例化：

```
template <typename T> int compare(const T&, const T&);
```

希望定义函数的一个特例化版本，其中 T 为 const char*。我们的函数要求一个指向此类型 const 版本的引用。一个指针类型的 const 版本是一个常量指针而不是指向 const 类型的指针。需要在特例化版本中使用的类型是 const char* const &，即一个指向 const char 的 const 指针的引用。

函数重载与模板特例化

定义函数模板的特例化版本时，本质上接管了编译器的工作。即，为原模板的一个特殊实例提供了定义。特例化的本质是实例化一个模板，而非重载它。因此，特例化不影响函数匹配。

> 普通作用域规则应用于特例化
>
> 如果一个程序使用一个特例化版本，而同时原模板的一个实例具有相同的模板实参集合，就会产生错误。但是，这种错误编译器无法发现。
>
> 最佳实践：模板及其特例化版本应该声明在同一个头文件中。所有同名模板的声明应该放在前面，然后是这些模板的特例化版本。

类模板特例化

除了特例化函数模板，还可以特例化类模板。例子，将为标准库 hash 模板定义一个特例化版本，可以用它来将 Sales_data 对象保存在无序容器中。默认下，无序容器使用 hash<key_type>（11.4节）来组织其元素。为了让自己的数据类型也能使用这种默认组织方式，必须定义 hash 模板的一个特例化版本。一个特例化 hash 类必须定义：

- 一个重载的调用运算符（14.8节），它接受一个容器关键字类型的对象，返回一个 size_t
- 两个类型成员，result_type 和 argument_type，分别调用运算符的返回类型和参数类型
- 默认构造函数和拷贝赋值运算符（可隐式定义，13.1.2节）

定义此特例化版本的 hash 时，唯一复杂的地方是：必须在原模板定义所在的命名空间中特例化它。18.2节命名空间有更多相关内容。现在只需知道，可以向命名空间添加成员。为达目的，首先必须打开命名空间：

```
// 打开 std 命名空间，以便特例化 std::hash
namespace std {
}  // 关闭 std 命名空间，最后没有分号
```

定义一个能处理 Sales_data 的特例化 hash 版本

```
// 打开 std 命名空间，以便特例化 std::hash
namespace std {
template <>    // 正在定义一个特例化版本，模板参数为 Sales_data
struct hash<Sales_data>
{
    // 用来散列一个无序容器的类型必须要定义下列类型
    typedef size_t reulst_type;
    typedef Sales_data argument_type;    // 默认此类型需要==
    size_t operator() (const Sales_data& s) const;
    // 我们的类使用合成的拷贝控制成员和默认构造函数
};
size_t
hash<Sales_data>::operator()(const Sales_data& s) const
{
    return hash<string>() (s.bookNo) ^
           hash<unsigned>() (s.units_sold) ^
           hash<double>() (s.revenue);
}
}
```

hash<Sales_data>定义以 template<> 开始，指出正在定义一个全特例化的模板。正在特例化的模板名为 hash，而特例化版本为 hash<Sales_data>。接下来的类成员是按照特例化 hash 的要求定义的。

类似其他任何类，可以在类内或类外定义特例化版本的成员，本例中就是在类外定义的。重载的调用运算符必须为给定类型的值定义一个哈希函数。对于一个给定值，任何时候调用此函数都应该返回相同的结果。一个好的哈希函数对不相等的对象应该产生不同的结果。

本例中，将定义一个好的哈希函数的复杂任务交给了标准库。标准库为内置类型和很多标准库类型定义了 hash 类的特例化版本。我们使用一个（未命名的）hash<string>对象来生成 bookNo 的哈希值，用一个 hash<unsinged>对象生成 units_sold 的哈希值，hash<double>对象生成 revenue 的哈希值。将这些结果进行异或运算，形成给定 Sales_data 对象的完整的哈希值。

我们的 hash 函数计算所有三个数据成员的哈希值，从而与我们为 Sales_data 定义的 operator==（14.3.1节）是兼容的。默认下，为了处理特定关键字类型，无序容器会组合使用 key_type 对应的特例化 hash 版本和 key_type 上的相等运算符。

假定特例化版本在作用域中，将 Sales_data 作为容器的关键字类型时，编译器就会自动使用此特例化版本：

```
// 使用 hash<Sales_data> 和 14.3.1节中 Sales_data 的 operator==
unordered_multiset<Sales_data> SDset;
```

由于 hash<Sales_data> 使用 Sales_data 的私有成员，必须将它声明为 Sales_data 的友元：

```
template <class T> class std::hash;     // 友元声明所需要的
class Sales_data {
friend class std::hash<Sales_data>;
    // 其他成员定义，如前
};
```

这段代码指出特殊实例 hash<Sales_data> 是 Sales_data 的友元。由于此实例定义在 std 命名空间中，必须记得在 friend 声明中应使用 std::hash。

> 为了让 Sales_data 的用户能使用 hash 的特例化版本，我们应该在 Sales_data 的头文件中定义该特例化版本。

类模板部分特例化

与函数模板不同，类模板的特例化不必为所有模板参数提供实参。可以只指定一部分而非所有模板参数，或是参数的一部分而非全部特性。一个类模板的部分特例化本身是一个模板，使用它时用户还要为那些在特例化版本中未指定的模板参数提供实参。

> 只能部分特例化类模板，而不能部分特例化函数模板。

特例化成员而不是类

可只特例化成员函数而不特例化整个模板。

小结

模板是 C++ 的特性，也是标准库的基础。一个模板就是一个编译器用来生成特定类型或函数的蓝图。生成特定类或函数的过程称为实例化。只编写一次模板，就可以将其用于多种类型和值，编译器会为每种类型和值进行模板实例化。

既可以定义函数模板，也可以定义类模板。标准库算法都是函数模板，标准库容器都是类模板。

显式模板实参允许固定一个或多个模板参数的类型或值。对于指定了显式模板实参的模板参数，可以应用正常的类型转换。

一个模板特例化就是一个用户提供的模板实例，它将一个或多个模板参数绑定到特定类型或值上。当我们不能（或不希望）将模板定义用于某些特定类型时，特例化非常有用。

一个可变参数模板可以接受数目和类型可变的参数。可变参数模板允许我们编写像容器的 emplace 成员和标准库 make_shared 函数这样的的函数，实现将实参传递给对象的构造函数。

术语