

# C++ 笔记

## C++ 笔记

### 第 0 部分 开篇！

#### 第一章 开始

1.1 编写一个简单的C++程序

1.2 初始输入输出

1.3 注释简介

1.4 控制流

    1.4.1 while

    1.4.2 for

    1.4.4 if 语句

1.5 类简介

1.6 书店程序

### 第 1 部分 C++ 基础

#### 第二章 变量和基本类型

2.1 基本内置类型

    2.1.1 算数类型

    2.1.2 类型转换

    2.1.3 字面值常量

2.2 变量

    2.2.1 变量定义

        列表初始化

        默认初始化

    2.2.2 变量声明和定义的关系

    2.2.3 标识符

        变量命名规范

    2.2.4 名字的作用域

2.3 复合类型

    2.3.1 引用

    2.3.2 指针

取地址符

指针值

解引用符

空指针

void\* 指针

## 2.4 const 限定符

2.4.1 const 的引用

2.4.2 指针和const

const指针

2.4.3 顶层const

2.4.4 `constexpr` 和常量表达式

`constexpr` 变量

## 2.5 处理类型

2.5.1 类型别名

2.5.2 auto类型说明符

2.5.3 decltype类型说明符

## 2.6 自定义数据结构

预处理器概述

# 第三章 字符串、向量和数组

## 3.1 命名空间的using声明

## 3.2 标准库类型 string

3.2.1 定义和初始化 string 对象

3.2.2 string 对象上的操作

## 3.3 标准库类型 vector

3.3.2 向vector对象中添加元素

## 3.4 迭代器介绍

3.4.1 使用迭代器

3.4.2 迭代器运算

## 3.5 数组

3.5.1 定义和初始化内置数组

字符数组的特殊性

3.5.3 指针和数组

标准库函数begin和end

### 3.5.4 C风格字符串

## 第四章 表达式

### 4.1 基础

#### 4.1.1 基本概念

左值和右值

#### 4.1.3 求值顺序

规定了运算对象求值顺序的运算符

### 4.2 算数运算符

### 4.3 逻辑和关系运算符

### 4.4 赋值运算符

### 4.5 递增递减运算符

### 4.6 成员访问运算符

### 4.7 条件运算符

### 4.8 位运算符

### 4.11 类型转换

#### 4.11.1 算术转换

整型提升

#### 4.11.3 显式转换

命名的强制类型转换

`static_cast`

`const_cast`

`reinterpret_cast`

## 第五章 语句

### 5.1 简单语句

空语句

复合语句

### 5.2 语句作用域

### 5.3 条件语句

#### 5.3.1 if语句

#### 5.3.2 switch 语句

switch内部变量定义

### 5.4 迭代语句

#### 5.4.1 while语句

5.4.2 for语句

5.4.3 do while 语句

5.5 跳转语句

5.5.1 break

5.5.2 continue

5.5.3 goto

5.6 try语句块和异常处理

5.6.1 throw表达式

5.6.2 try语句块

5.6.3 标准异常

第六章 函数

6.1 函数基础

形参和实参

6.1.1 局部对象

6.1.2 函数声明

6.2 参数传递

6.2.1 传值参数

6.2.2 传引用参数

6.2.3 const形参和实参

6.2.4 数组形参

数组引用形参

6.2.5 main处理命令行选项

6.2.6 含有可变形参的函数

initializer\_list

省略符形参

6.3 返回类型和 return 语句

6.3.1 无返回值函数

6.3.2 有返回值函数

不要返回局部对象的引用或指针

引用返回左值

列表初始化返回值

主函数main的返回值

6.3.3 返回数组指针

声明一个返回数组指针的函数

使用尾置返回类型

使用decltype

## 6.4 函数重载

重载和const形参

const\_cast 和重载

调用重载的函数

### 6.4.1 重载与作用域

## 6.5 特殊用途语言特性

### 6.5.1 默认实参

默认实参声明

默认实参初始值

### 6.5.2 内联函数和constexpr函数

内敛函数

constexpr函数

### 6.5.3 调试帮助

assert预处理宏

NDEBUG预处理变量

## 6.6 函数匹配

### 6.6.1 实参类型转换

类型提升和算术类型转换

函数匹配和const实参

## 6.7 函数指针

使用函数指针

重载函数指针

函数指针形参

返回指向函数的指针

# 第七章 类

## 7.1 定义抽象数据类型

定义成员函数

引入const成员函数

类作用域和成员函数

在类的外部定义成员函数

## 7.1.4 构造函数

默认构造函数

构造函数初始值列表

## 7.1.5 拷贝、赋值和析构

### 7.2 访问控制与封装

class 和 struct

#### 7.2.1 友元

### 7.3 类的其他特性

#### 7.3.1 类成员再探

可变数据成员

类的声明

#### 7.3.4 友元再探

类之间的友元

### 7.4 类的作用域

作用域和定义在类外部的成员

#### 7.4.1 名字查找与类的作用域

用于类成员声明的名字查找

### 7.5 构造函数再探

#### 7.5.1 构造函数初始值列表

构造函数的初始值有时必不可少 

成员初始化的顺序

#### 7.5.2 委托构造函数

#### 7.5.3 默认构造函数的作用

#### 7.5.4 隐式的类类型转换

只允许一步类类型转换

抑制构造函数定义的隐式转换

#### 7.5.5 聚合类

#### 7.5.6 字面值常量类

constexpr构造函数

### 7.6 类的静态成员

## 第 II 部分 C++ 标准库

### 第八章 IO 库

#### 8.1 IO 类

## IO类型间的关系

### 8.1.1 IO对象无拷贝或赋值

### 8.1.2 条件状态

查询流的状态

管理条件状态

### 8.1.3 管理输出缓冲

刷新输出缓冲区

关联输入和输出流

## 8.2 文件输入输出

### 8.2.1 使用文件流对象

成员函数open和close

### 8.2.2 文件模式

## 8.3 string 流

### 8.3.1 使用istringstream

### 8.3.2 使用ostringstream

## 第九章 顺序容器

### 9.1 顺序容器概述

### 9.2 容器库概览

#### 9.2.1 迭代器

#### 9.2.2 容器类型成员

#### 9.2.3 `begin` 和 `end` 成员

#### 9.2.4 容器定义和初始化

将一个容器初始化为另一个容器的拷贝

标准库array具有固定大小

#### 9.2.5 赋值和swap

swap

assign

#### 9.2.6 容器大小操作

#### 9.2.7 关系运算符

### 9.3 顺序容器操作

#### 9.3.1 向顺序容器添加元素

#### 9.3.2 访问元素

#### 9.3.3 删除元素

### 9.3.4 特殊的forward\_list操作

### 9.3.5 改变容器大小

### 9.3.6 容器操作可能使迭代器失效

## 9.4 vector对象是如何增长的

## 9.5 额外的string操作

### 9.5.1 构造string的其他方法

#### substr操作

### 9.5.2 改变string的其他方法

#### 9.5.3 string搜索操作

#### 9.5.4 compare函数

#### 9.5.5 数值转换

## 9.6 容器适配器

# 第十章 泛型算法

## 10.1 概述

### 算法永远不会执行容器的操作

## 10.2 初始泛型算法

### 10.2.1 只读算法

#### 1) find算法

#### 2) accumulate

#### 3) equal

### 10.2.2 写容器元素算法

#### 1) fill

#### 2) copy

#### 3) replace

### 10.2.3 重排容器元素的算法

#### 1) sort

#### 1) stable\_sort

#### 2) unique

## 10.3 定制操作

### 10.3.1 向算法传递函数

#### 谓词

### 10.3.2 lambda表达式

### 10.3.3 lambda捕获和返回

值捕获

引用捕获

隐式捕获

可变lambda

指定lambda返回类型

10.3.4 参数绑定

标准库bind函数

bind的参数

绑定引用参数

10.4 再探迭代器

10.4.1 插入迭代器

10.4.2 iostream迭代器

istream\_iterator操作

ostream\_iterator操作

10.4.3 反向迭代器

10.5 泛型算法结构

10.5.1 5类迭代器

10.5.2 算法形参模式

10.5.3 算法命名规范

10.6 特定容器算法

list和forward\_list成员函数版本算法

splice成员

第十一章 关联容器

11.1 使用关联容器

11.2 关联容器概述

11.2.1 定义关联容器

初始化map和set

初始化multimap或multiset

11.2.2 关键字类型要求

有序容器的关键字类型

使用关键字类型的比较函数

11.2.3 pair类型

11.3 关联容器操作

### 11.3.1 关联容器迭代器

### 11.3.2 添加元素

检测insert的返回值

向multiset或multimap添加元素

### 11.3.3 删除元素

### 11.3.4 map的下标操作

### 11.3.5 访问元素

## 11.4 无序容器

管理桶

无序容器对关键字类型的要求

## 第十二章 动态内存

### 12.1 动态内存与智能指针

#### 12.1.1 shared\_ptr类

make\_shared函数

shared\_ptr的拷贝和赋值

shared\_ptr自动销毁所管理的对象

使用了动态生存期的资源的类

#### 12.1.2 直接管理内存

使用new动态分配和初始化对象

动态分配的const对象

内存耗尽

释放内存delete

指针值和delete

动态对象的生命周期直到被释放

#### 12.1.3 shared\_ptr和new结合使用

不要混合使用普通指针和智能指针

不要使用get初始化另一个智能指针或为智能指针赋值

#### 12.1.4 智能指针和异常

#### 12.1.5 unique\_ptr

unique\_str操作

传递unique\_str参数和返回unique\_str

向unique\_str传递删除器

#### 12.1.6 weak\_ptr

## 12.2 动态数组

### 12.2.1 new和数组

初始化动态分配对象的数组

动态分配一个空数组是合法的

释放动态数组

智能指针和动态数组

### 12.2.2 allocator类

allocator类

allocator分配未构造的内存

拷贝和填充未初始化内存的算法

## 12.3 使用标准库：文本查询程序

# 第 III 部分 类设计者的工具

## 第十三章 拷贝控制

### 13.1 拷贝、赋值与销毁

#### 13.1.1 拷贝构造函数

合成拷贝构造函数

拷贝初始化

拷贝初始化参数为什么必须是引用类型？

编译器可以绕过拷贝构造函数

#### 13.1.2 拷贝赋值运算符

重载赋值运算符

合成拷贝赋值运算符

#### 13.1.3 析构函数

什么时候会调用析构函数

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

#### 13.1.6 阻止拷贝

定义删除的函数

析构函数不能是删除的

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

### 13.3 交换操作

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

### 13.5 动态内存管理类

移动构造函数和std::move

### 13.6 对象移动

#### 13.6.1 右值引用

左值持久；右值短暂

变量是左值

标准库move函数

#### 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

移后源对象必须可析构

合成的移动操作

移动右值，拷贝左值

如果没有移动构造函数，右值也会被拷贝

拷贝并交换赋值运算符和移动操作

移动迭代器

#### 13.6.3 右值引用和成员函数

右值和左值引用成员函数(引用限定符)

重载和引用函数

## 第十四章 操作重载与类型转换

### 14.1 基本概念

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

赋值和复合赋值运算符

## 选择作为成员或者非成员

### 14.2 输入输出运算符

#### 14.2.1 重载输出运算符<<

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>>

输入时的错误

### 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

显式地调用后置运算符

### 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

#### 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

在算法中使用标准库函数对象

#### 14.8.3 可调用对象与function

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

重载函数和转换构造函数

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

# 第十五章 面向对象程序设计 (OOP)

## 15.1 OOP概述

继承

动态绑定

## 15.2 定义基类和派生类

### 15.2.1 定义基类

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

静态类型与动态类型

不存在基类向派生类的隐式类型转换

在对象之间不存在类型转换

## 15.3 虚函数

对虚函数的调用可能在运行时才被解析

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

派生类构造函数只初始化它的直接基类

## 15.5 访问控制与继承

受保护的成员

公有、私有和受保护继承

派生类向基类转换的可访问性

友元与继承

改变个别成员的可访问性

默认的继承保护级别

## 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

### 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

## 第十六章 模板与泛型编程

### 16.1 定义模板

#### 16.1.1 函数模板

实例化函数模板

模板类型参数

非类型模板参数

inline和constexpr的函数模板

编写类型无关的代码

模板编译

大多数编译错误在实例化期间报告

#### 16.1.2 类模板

实例化类模板

类模板的成员函数

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

## 16.2 模板实参推断

### 16.2.1 类型转换与模板类型参数

使用相同模板参数类型的函数形参

正常类型转换应用与普通函数实参

### 16.2.2 函数模板显式实参

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 函数指针和实参判断

### 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

### 16.2.6 理解std::move

std::move是如何定义的

std::move是如何工作的

### 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

## 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

定义函数模板特例化

函数重载与模板特例化  
类模板特例化  
类模板部分特例化  
特例化成员而不是类

## 第 IV 部分 高级主题

### 第十七章 标准库特殊设施

#### 17.1 tuple 类型

##### 17.1.1 定义和初始化 tuple

访问 tuple 成员  
关系和相等运算符

#### 17.2 bitset 类型

##### 17.2.1 定义和初始化 bitset

用 unsigned 值初始化 bitset  
从一个 string 初始化 bitset

##### 17.2.2 bitset 操作

#### 17.3 正则表达式

##### 17.3.1 使用正则表达式库

指定regex对象的选项  
指定或使用正则表达式时的错误  
正则表达式类和输入序列类型

##### 17.3.2 匹配与Regex迭代器类型

使用匹配数据

##### 17.3.3 使用子表达式

##### 17.3.4 使用regex\_replace

#### 17.4 随机数

##### 17.4.1 随机数引擎和分布

分布类型和引擎  
比较随机数引擎和rand函数  
引擎生成一个数值序列  
设置随机数发生器种子

##### 17.4.2 其他随机数分布

生成随机实数  
使用分布的默认结果类型

## 生成非均匀分布的随机数

### 17.5 IO库再探

#### 17.5.1 格式化输入输出

很多操纵符改变格式状态

控制布尔值的格式

指定整型值的进制

在输出中指出进制

控制浮点数格式

指定打印精度

指定浮点数计数法

#### 17.5.2 未格式化的输入/输出操作

单字节操作

多字节操作

#### 17.5.3 流随机访问

seek和tell函数

## 第十八章 用于大型程序的工具

### 18.1 异常处理

#### 18.1.1 抛出异常

栈展开

栈展开过程中对象被自动销毁

析构函数与异常

异常对象

#### 18.1.2 捕获异常

查找匹配的处理代码

重新抛出

捕获所有异常的处理代码

#### 18.1.3 函数 `try` 语句块与构造函数

#### 18.1.4 noexcept异常说明

违反异常说明

异常说明的实参

`noexcept`运算符

异常说明与指针、虚函数和拷贝控制

#### 18.1.5 异常类层次

## 18.2 命名空间

### 18.2.1 命名空间定义

每个命名空间都是一个作用域

命名空间可以是不连续的

全局命名空间

嵌套的命名空间

内联命名空间

未命名的命名空间

### 18.2.2 使用命名空间成员

命名空间的别名

using声明：扼要概述

using指示

using指示与作用域

头文件与using声明或指示

### 18.2.3 类、命名空间与作用域

实参相关的查找与类类型形参

查找与std::move和std::forward

友元声明与实参相关的查找（与H5标题联合起来看）

### 18.2.4 重载与命名空间

与实参相关的查找与重载

重载与using声明

重载与using指示

跨越多个using指示的重载

## 18.3 多重继承与虚继承

### 18.3.1 多重继承

多重继承的派生类从每个基类中继承状态

派生类构造函数初始化所有基类

继承的构造函数与多重继承

析构函数与多重继承

多重继承的派生类的拷贝与移动操作

### 18.3.2 类型转换与多个基类

基于指针或引用类型的查找

### 18.3.3 多重继承下的类作用域

### 18.3.4 虚继承

使用虚基类

支持向基类的常规类型转换

虚基类成员的可见性

### 18.3.5 构造函数与虚继承

虚继承的对象的构造方式

构造函数与析构函数的次序

## 第十九章 特殊工具与技术

### 19.1 控制内存分配

#### 19.1.1 重载new和delete

operator new接口和operator delete接口

malloc函数与free函数

#### 19.1.2 定位new表达式

显式地析构函数调用

### 19.2 运行时类型识别

#### 19.2.1 dynamic\_cast运算符

指针类型的dynamic\_cast

引用类型的dynamic\_cast

#### 19.2.2 typeid运算符

使用typeid运算符

#### 19.2.3 使用RTTI

#### 19.2.4 type\_info类

### 19.3 枚举类型

枚举成员

枚举定义新的类型

指定enum的大小

枚举类型的前置说明

形参匹配与枚举类型

### 19.4 类成员指针

#### 19.4.1 数据成员指针

使用数据成员指针

返回数据成员指针的函数

#### 19.4.2 成员函数指针

使用成员函数指针

使用成员指针的类型别名

成员指针函数表

### 19.4.3 将成员函数用作可调用对象

使用function生成一个可调用对象

使用mem\_fn生成一个可调用对象

使用bind生成一个可调用对象

## 19.5 嵌套类

声明一个嵌套类

在外层类之外定义一个嵌套类

定义嵌套类的成员

嵌套类的静态成员定义

嵌套类作用域中的名字查找

嵌套类和外层类是相互独立的

## 19.6 union一种节省空间的类

定义union

使用union

匿名union

含有类类型成员的union

使用类管理union成员

管理判别式并销毁string

管理需要拷贝控制的联合成员

## 19.7 局部类

局部类不能使用函数作用域中的变量

常规的访问保护规则对局部类同样适用

局部类中的名字查找

嵌套的局部类

## 19.8 固有的不可移植的特性

### 19.8.1 位域

使用位域

### 19.8.2 volatile限定符

合成的拷贝对volatile对象无效

### 19.8.3 链接指示：extern "C"

声明一个非C++函数

链接指示与头文件

指向extern "C" 函数的指针

链接指示对整个声明都有效

导出C++函数到其他语言

重载函数与链接指示

# 第 0 部分 开篇！

## 第一章 开始

### 1.1 编写一个简单的C++程序

一个函数的定义包含4部分：

- 返回类型
- 函数名
- 形参列表（允许为空）
- 函数体

### 1.2 初始输入输出

### 1.3 注释简介

```
1 /*  
2 *  
3 */  
4  
5 //
```

### 1.4 控制流

#### 1.4.1 while

#### 1.4.2 for

## 1.4.4 if 语句

## 1.5 类简介

成员函数是定义为类的一部分的函数，有时也被称为方法

## 1.6 书店程序

---

# 第 I 部分 C++基础

- 整型、字符型等内置类型
- 变量，用来为对象命名
- 表达式和语句，用于操纵上述数据类型的具体值
- if或while等控制结构，这些结构允许我们有选择地执行一些语句或者重复地执行一些语句
- 函数，用于定义可供随时调用的计算单元

## 第二章 变量和基本类型

### 2.1 基本内置类型

#### 算数类型和空类型

##### 2.1.1 算数类型

类型		长度
bool	布尔类型	未定义
char	字符	8位
wchar_t	宽字符	16位
char16_t	Unicode 字符	16位
char32_t	Unicode 字符	32位
short	短整型	16位
int	整型	16位
long	长整型	32位

类型		长度
long long	长整型	64位
float	单精度浮点数	6位有效数字
double	双精度浮点数	10位有效数字
long double	扩展精度浮点数	10位有效数字

## 2.1.2 类型转换

类型所能表示的值的范围决定了转换的过程：

- 当我们把一个非布尔类型的算术值赋值给布尔类型时，初始值为0则结果为false，否则结果为true
- 当我们把一个布尔值赋给非布尔类型时，初始值为false则结果为0，初始值为true则结果为1
- 当我们把一个浮点数赋给整数类型时，进行了近似处理，结果值仅保留整数部分（直接截断）
- 当我们赋给无符号类型一个超出他表示范围的值时，结果是初始值对无符号类型表示数值总数取模后的余数
- 当我们赋给带符号类型一个超出它表示范围的值时，结果是未定义的（undefined）

## 2.1.3 字面值常量

字面值常量的形式和值决定了它的数据类型

```
1 'a' //字符字面值
2 "aaa" //字符串字面值
```

名称	符号
换行符	\n

名称	符号
横向制表符	\t
报警（响铃）符	\a
纵向制表符	\v
退格符	\b
双引号	\"
反斜线	\\'
问号	\?
单引号	\'
回车符	\r
进纸符	\f

## 指定字面值的类型

### 字符和字符串字面值

前缀	含义	类型
u	Unicode 16 字符	char16_t
U	Unicode 32 字符	char32_t
L	宽字符	wchar_t
u8	UTF-8 (仅用于字符串字面值常量)	char

### 整型字面值

后缀	最小匹配类型
u or U	unsigned
l or L	long
ll or LL	long long

### 浮点型字面值

后缀	类型
f或F	float
l或L	long double

## 2.2 变量

### 2.2.1 变量定义

**变量定义的基本形式是：**首先是类型说明符，随后紧跟由一个或多个变量名组成的列表，变量名以逗号分隔，最后以分号结束。

**在同一条定义语句中，可以用先定义的变量值去初始化后定义的其他变量**

### 列表初始化

```

1 int units_sold = 0;
2 int units_sold = {0};
3 int units_sold{0};
4 int units_sold(0);

```

用花括号初始化的方式称为**列表初始化**。当用于内置类型的变量时，这种初始化形式用一个重要的特点：如果我们使用列表初始化且初始值存在丢失信息的风险，编译器将报错

```
1 long double ld = 3.13231515151241;
2 int a{ld}, b = {ld}; //错误，会丢失信息
```

## 默认初始化

如果定义变量时没有指定初值，则变量被**默认初始化**。默认值由变量类型和定义变量的位置决定。

- 定义在任何函数之外的变量被初始化为0
- 定义在函数体内部的内置类型将不被初始化，一个未被初始化的内置类型变量的值是未定义的，此时访问会引发错误
- 每个类各自决定其初始化对象的方式。
- 一些类要求每个对象都显式初始化，若未初始化将引发错误。

## 2.2.2 变量声明和定义的关系

为了允许把程序拆分成多个逻辑部分来编写，C++语言支持**分离式编译**机制，允许程序分隔为若干个文件，每个文件可独立编译。

为了支持分离式编译，C++将声明和定义区分开来。

**声明**使得名字为程序所知，一个文件如果想使用别处定义的名字则必须包含对那个名字的声明

**定义**负责创建与名字关联的实体

如果想声明一个变量而非定义，则可以添加 `extern` 关键字，不要显式的初始化变量

```
1 extern int i; //声明 i 而非定义 i
2 int j; //声明并定义 j
```

任何包含了显式初始化的声明即成为定义，即使添加了 `extern` 关键字

## 2.2.3 标识符

C++的标识符由 **字母、数字、下画线**组成，必须以字母或下画线开头。标识符的长度没有限制，但是对大小写字母敏感。

### 变量命名规范

- 标识符要能体现实际含义
- 变量名一般用小写字符，如`index`
- 用户自定义的类名一般以大写字母开头
- 如果标识符由多个单词组成，单词间最好有明显区分

## 2.2.4 名字的作用域

作用域是程序的一部分，在其中名字有其特定的含义，C++中大多数作用域以花括号分隔

## 2.3 复合类型

复合类型是指基于其他类型定义的类型。

### 2.3.1 引用

引用为对象起了另外一个名字，引用类型引用另外一种类型，通常使用 `&` 符号。

无法将引用重新绑定到另一个对象上，因此引用必须初始化

### 2.3.2 指针

指针是指向另外一种类型的复合类型

- 指针本身就是一个对象（这一点和引用不同），允许对指针赋值和拷贝，而且指针的生命周期内可以先后指向几个不同的对象

- 指针无需在定义时赋初值（引用必须初始化），和其他类型一样，如果没有赋初值，将拥有一个不确定的值（非常的危险！）

定义指针类型的方法将声明符写成 `*d` 的形式

## 取地址符

获取对象的地址需要使用取地址符 `&`

## 指针值

指针的值应该处于下列4中状态之一：

- 指向一个对象
- 指向紧邻对象所占空间的下一位置
- 空指针，意味着不指向任何对象
- 无效指针，除上述情况之外的其他值

## 解引用符

允许使用解引用符 `*` 来访问指针指向的对象

## 空指针

空指针不指向任何对象，在试图使用一个指针之前代码可以首先检查它是否为空

```
1 int *p1 = nullptr;
2 int *p2 = 0;
3 int *p3 = NULL; //需要先 #include <cstdlib>
```

## void\* 指针

`void*` 指针是一种特殊的指针，可以存放任意对象的地址，但是我们不知道指向的地址中的数据是什么类型的

能做的操作有：

- 拿它和别的指针比较
- 作为函数的输入输出
- 赋值给另外一个void\*指针

## 2.4 const 限定符

默认状态下，const 对象仅在文件内有效

编译器编译过程中将const变量替换成对应的值，所以为了执行这一替换编译器必须知道变量的初始值，如果有多个文件，每个使用了const对象的文件都必须访问到它的初始值，所以需要在每一个用到变量的文件中都有对他的定义，所以默认情况下，const对象被设定为仅在文件内有效，避免了对同一变量的重复定义的冲突。

### 2.4.1 const 的引用

把引用绑定到 const 对象上（对象是 const 的），称之为对常量的引用。对常量的引用不能修改它绑定的对象。

对const的引用可以引用一个非const的对象，只是我们通过该引用认为这个值是const的

### 2.4.2 指针和const

类似于常量引用，指向常量的指针不能用于改变其所指对象的值。要想存放常量对象的地址，只能使用指向常量的指针。

#### const指针

指针是对象而引用不是，所以允许把指针定义为常量。常量指针必须被初始化，而且它的值不能改变。把\*放在const关键字之前用以说明指针是一个常量。

```
1 const double *const pip = &pi; //pip是一个指向常量对象的常量指针
```

## 2.4.3 顶层const

顶层const表示指针本身是个常量，底层const表明指针所指的对象是一个常量（这两个概念也可以泛指任意数据类型，例如const int ci = 42，此处的const就是顶层const，表明变量本身是个const）。指针类型既可以是顶层const也可以是底层const。

## 2.4.4 constexpr 和常量表达式

常量表达式是指值不会改变并且在编译过程就能得到计算结果的表达式

字面值属于常量表达式，用常量表达式初始化的const对象也是常量表达式

### constexpr 变量

C++11中，允许将变量声明为constexpr类型以便由编译器来验证变量的值是否是一个常量表达式。声明为constexpr的变量一定是一个常量，而且必须用常量表达式初始化

如果constexpr声明了一个指针，限定符constexpr只对指针有效，与指针所指对象无关

```
1 const int *p = nullptr;
2 constexpr int *q = nullptr;
3 二者一个是指向const int 的指针，一个是指向 int 的
  const 指针
```

## 2.5 处理类型

### 2.5.1 类型别名

类型别名是一个名字，它是某种类型的同义词。

两种方法可以定义类型别名：

- `typedef` 关键字

```
1 typedef double wages; //wages是double的同
   义词
2 typedef wages base,*p; //base是double的同
   义词, p是double*的同义词
```

- 别名声明

```
1 using SI = Sales_item; //SI是Sales_item的
   别名
```

## 2.5.2 `auto`类型说明符

`auto`类型说明符，编译器会帮助我们推断表达式所属的类型

## 2.5.3 `decltype`类型说明符

`decltype`的作用是选择并返回操作数的数据类型。编译器分析表达式并得到它的类型，却不实际计算表达式的值。

```
1 decltype(f()) sum = x; //sum的类型就是函数f的返回类
   型
```

## 2.6 自定义数据结构

### 预处理器概述

确保头文件多次包含仍能安全工作的常用技术是预处理器。预处理器是在编译之前执行的一段程序，可以部分的改变我们所写的程序。

`#include`：当预处理器看到 `#include` 标记时就会使用指定的头文件的内容代替`#include`

头文件保护符：头文件保护符依赖于预处理变量。预处理变量有两种状态：

- 已定义
- 未定义

#define： #define 把一个名字设定为预处理变量，另外两个指令则分别检查某个指定的预处理变量是否已经定义

#ifdef： 当且仅当变量已定义时为真

#ifndef： 当且仅当变量未定义时为真。一旦检查结果为真，则执行后续操作直到遇到#endif指令

```

1 #ifndef SALES_DATA_H
2 #define SALES_DATA_H
3 #include <string>
4 struct Sales_data{
5     std::string bookNo;
6     unsigned units_sold = 0;
7     double revenue = 0.0;
8 };
9 #endif

```

整个程序中的预处理变量必须唯一，一般是基于头文件中类的名字来构建保护符的名字，以确保其唯一性。并且建议全部大写。

## 第三章 字符串、向量和数组

### 3.1 命名空间的using声明

位于头文件的代码一般来说不应该使用using声明

## 3.2 标准库类型 `string`

包含`string`头文件

### 3.2.1 定义和初始化 `string` 对象

#### 初始化方式

<code>string s1</code>	默认初始化， <code>s1</code> 是一个空串
<code>string s2(s1)</code>	<code>s2</code> 是 <code>s1</code> 的副本
<code>string s2 = s1</code>	等价于 <code>s2(s1)</code>
<code>string s3("value")</code>	除了字面值最后的那个空字符外的副本
<code>string s3 = "value"</code>	同上
<code>string s4(n, 'c')</code>	<code>n</code> 个字符'c'组成的字符串

使用等号 (=) 初始化一个变量，采用的是**拷贝初始化**；如果不使用等号，则执行的是**直接初始化**

```
1 string s5 = "hi"; //拷贝初始化
2 string s6("hi"); //直接初始化
```

### 3.2.2 `string` 对象上的操作

#### 操作

<code>os &lt;&lt; s</code>	将 <code>s</code> 写入到输出流 <code>os</code> 当中，返回 <code>os</code>
<code>is &gt;&gt; s</code>	从 <code>is</code> 中读取字符串赋给 <code>s</code> ，字符串以空白分隔，返回 <code>is</code>
<code>getline(is,s)</code>	从 <code>is</code> 中读取一行赋给 <code>s</code> ，返回 <code>is</code>

操作	
s.empty()	判断s是否为空
s.size()	返回s中的字符个数

## 3.3 标准库类型 `vector`

头文件 `vector`

### 3.3.2 向`vector`对象中添加元素

`push_back()` 能够向`vector`尾部添加一个元素，不能以下标形式添加元素

## 3.4 迭代器介绍

### 3.4.1 使用迭代器

获取迭代器不是使用取地址符，有迭代器的类型同时拥有返回迭代器的成员。

- `begin` 成员返回指向第一个元素的迭代器
- `end` 成员返回指向尾元素下一位置的迭代器（尾后迭代器）

### 3.4.2 迭代器运算

简单的移动运算例如 `+n, -n`

迭代器也可以相减，得到的是它们之间的距离

## 3.5 数组

数组的大小固定，不能随意向数组中增加元素

## 3.5.1 定义和初始化内置数组

数组的声明形式如 `a[d]`, 其中 `a` 是数组的名字, `d` 是数组的维度。 `d` 必须是一个常量表达式

默认情况下, 数组的元素被默认初始化

## 字符数组的特殊性

字符数组可以用字符串字面值初始化, 一定要注意字符串字面值尾部还有一个空字符, 这个空字符也会像字符串其他的字符一样被拷贝到字符数组中去

## 3.5.3 指针和数组

使用数组的时候编译器一般将其转化为指针

指针也是迭代器

## 标准库函数begin和end

这两个函数不是成员函数, 正确的使用形式如下:

相对于成员函数的概念是相同的

```
1 int ia[] = {0,1,2,3,4};
2 int *beg = begin(ia);
3 int *last = end(ia);
```

## 3.5.4 C风格字符串

C风格字符串不是一种类型, 而是一种约定成俗的写法。按此习惯书写的字符串存放在字符数组中并以空字符结束

- 允许使用以空字符结束的字符数组来初始化`string`对象或为`string`对象赋值。

- 在string对象的加法运算中允许使用以空字符结束的字符数组作为其中一个运算对象（不能两个运算对象都是）；在string对象的复合赋值运算中允许使用以空字符结束的字符数组作为右侧的运算对象

上述性质反过来不成立，不能使用string对象直接初始化指向字符的指针

```
1 char *str = s; //错误不能直接初始化
2 const char *str = s.c_str(); //正确 结果的类型时
    const char * 防止我们更改
```

## 第四章 表达式

### 4.1 基础

#### 4.1.1 基本概念

##### 左值和右值

C++的表达式要么是右值，要不然左值。这两个名词是从C语言继承而来的，原本是为了帮助记忆；左值可以位于赋值语句的左侧，右值则不能；

在C++中：一个左值表达式的求值结果是一个对象或者一个函数，然而以常量对象为代表的某些左值实际上不能作为赋值语句的左侧运算对象（这就跟之前表达的 左值可以位于赋值语句的左侧 有所不同）。此外虽然某些表达式的求值结果是对象，但它们是右值而非左值。

当一个对象被用作右值的时候，用的是对象的值（内容）；

当对象被用作左值的时候用的是对象的身份（在内存中的位置）

一个重要的原则是在需要右值的地方可以使用左值代替，但是不能把右值当做左值使用。当一个左值被当成右值使用时，实际使用的是他的内容。目前有几种熟悉的运算需要左值：

- 赋值运算符需要一个（非常量）左值作为其左侧运算对象，得到的结果也仍然是一个左值
- 取地址符作用于一个左值运算对象，返回一个指向该运算对象的指针，这个指针是一个右值（要赋给一个指针）
- 内置解引用运算符、下标运算符、迭代器解引用运算符、string 和vector的下标运算符的求值结果都是左值
- 内置类型和迭代器的递增递减运算符作用于左值运算对象，其前置版本所得的结果也是左值

使用关键字 `decltype` 的时候，左值和右值也有所不同。表达式的求值结果是左值，`decltype`作用于该表达式得到一个引用类型。i.e. 若p的类型为`int*`，如果`decltype(*p)`则结果为`int&`，`decltype(&p)`则结果为`int**`。

### 4.1.3 求值顺序

优先级规定了运算对象的组合方式，但是不会指明运算对象按什么顺序求值

```
1 int i = f1() * f2();
```

对于这个式子，我们只知道f1和f2一定会在乘法之前被调用，但是无法知道到底是f1先被调用还是f2先被调用

```
1 int i = 0;
2 cout << i << " " << ++i << endl; //未定义的
```

该表达式不知道先执行`i`还是`++i`，所以此表达式的行为不可预知，因此不论编译器生成什么样的代码程序都是错误的

## 规定了运算对象求值顺序的运算符

- 逻辑与
- 逻辑或
- 条件运算符
- 逗号运算符

以下两条经验准则对书写复合表达式有益：

1. 拿不准的时候用括号强制优先级
2. 如果改变了某个运算对象的值，在表达式的其他地方不要再使用这个运算对象

## 4.2 算数运算符

## 4.3 逻辑和关系运算符

逻辑与和逻辑或运算符都是先求左侧运算对象的值再求右侧运算对象的值，当且仅当左侧运算对象无法确定表达式的结果时才会计算右侧运算对象的值。这种策略称为短路求值

## 4.4 赋值运算符

赋值运算符的左侧运算对象必须是一个可修改的左值

## 4.5 递增递减运算符

这两种运算符必须作用于左值运算对象

前置版本将对象本身作为左值返回

后置版本则将对象原始值的副本作为右值返回



除非必须，否则不用递增递减运算符的后置版本

前置版本的递增运算符避免了不必要的工作，它将值加1后直接返回改变了的运算对象。

后置版本需要将原始值存储下来以便返回这个未修改的内容，如果我们不需要修改前的值，那么后置版本的操作就是一种浪费

后置递增运算符的优先级高于解引用运算符，因此`*pbeg++`等价于`(pbeg++)`，因为后置`++`是返回`pbeg`的初始值，所以解引用运算符的运算对象是`pbeg`未增加之前的值

## 4.6 成员访问运算符

点运算符 `.` 和 箭头运算符 `->`

## 4.7 条件运算符

`?:` 运算符

## 4.8 位运算符

运算符	功能	用法
<code>~</code>	位求反	<code>~expr</code>
<code>&lt;&lt;</code>	左移	<code>expr1 &lt;&lt; expr2</code>
<code>&gt;&gt;</code>	右移	<code>expr1 &gt;&gt; expr2</code>
<code>&amp;</code>	位与	<code>expr &amp; expr</code>
<code>^</code>	位异或	<code>expr ^ expr</code>
<code> </code>	位或	<code>expr   expr</code>

## 4.11 类型转换

**隐式转换：**不需要程序员的介入和了解，自动执行的类型转换

下面的情况，编译器会自动转换运算对象的类型

- 在大多数表达式中，比int类型小的整型值首先提升为较大的整数类型
- 在条件中，非布尔值转换成布尔类型
- 初始化过程中，初始值转换成变量的类型；在赋值语句中，右侧运算对象转换成左侧运算对象的类型
- 如果算术运算或关系运算的运算对象有多种类型，需要转换成同一种类型
- 函数调用时也会发生类型转换

### 4.11.1 算术转换

**算术转换**的含义是把一种算术类型转换成另一种算数类型

#### 整型提升

**整型提升**负责把小整数类型转换成较大的整数类型

### 4.11.3 显式转换

#### 命名的强制类型转换

1 `cast-name<type>(expression):`

`type`是转换的目标类型，如果`type`是引用类型，则结果是左值

`expression`是要转换的值。

`cast-name`是 `static_cast`、`dynamic_cast`、`const_cast` 和 `reinterpret_cast`

## static\_cast

任何具有明确定义的类型转换，只要不包含底层const，都可以使用static\_cast

当需要把一个较大的算数类型赋值给较小的类型时，static\_cast非常有用。此时强制类型转换告诉读者和编译器：我们知道并且不在乎精度的损失

static\_cast对于编译器无法自动执行的类型转换也非常有用。例如我们可以使用static\_cast找回存在与void\*的指针中的值：

```
1 void *p = &d;
2 double *dp = static_cast<double*>(p);
```

## const\_cast

const\_cast只能改变运算对象的底层const

```
1 const char *pc;
2 char *p = const_cast<char*>(pc); //正确，但是通过p
写值是未定义的行为
```

## reinterpret\_cast

reinterpret\_cast通常为运算对象的位模式提供较低层次上的重新解释

```
1 int *ip;
2 char *pc = reinterpret_cast<char*>(ip); //这种行为非常危险
```

# 第五章 语句

## 5.1 简单语句

### 空语句

最简单的语句就是一个；

### 复合语句

复合语句是指用花括号括起来的语句和声明的序列，符合语句也被称作块。

## 5.2 语句作用域

pass;

## 5.3 条件语句

### 5.3.1 if语句

pass;

### 5.3.2 switch 语句

switch语句首先对括号里的表达式求值，该表达式紧跟在关键字switch的后面，可以是一个初始化的变量声明。**表达式的值转换成整数类型，然后与每个case标签的值比较（表达式的值必须可以转换为整数类型）**；case标签必须是整型常量表达式

### switch内部变量定义

switch的执行流程有可能会跨过某些case标签。那么如果被跨过的标签中有变量的定义怎么办？

答案是：如果在某处一个带有初值的变量位于作用域之外，在另一处该变量位于作用域之内，则从前一处跳转到后一处的行为是非法的。

```

1 case true:
2     string file_name; //错误：控制流绕过一个隐式
3         int ival = 0; //错误：控制流绕过一个显式初始化的
4             int jval; //正确：因为jval没有初始化
5             break;
6 case false:
7     //正确：jval虽然在作用域内，但是它没有被初始化
8     jval = next_num(); //正确，给jval赋一个值
9     if(file_name.empty()) //错误 file_name 在作
10        用域内，但是没有被初始化

```

如果需要为某个case分支定义并初始化一个变量，那么应该把变量定义在块内，从而确保后面的所有case标签都在作用域之外。

```

1 case true:
2 {
3     string file_name = getname(); //正确
4 }
5 case false:
6

```

## 5.4 迭代语句

### 5.4.1 while语句

## 5.4.2 for语句

传统for

范围for

## 5.4.3 do while 语句

## 5.5 跳转语句

### 5.5.1 break

### 5.5.2 continue

### 5.5.3 goto

## 5.6 try语句块和异常处理

异常处理机制为程序中异常检测和异常处理这两部分的协作提供支持

异常处理包括

- throw表达式，异常检测部分使用throw表达式来表示它遇到了无法处理的问题。我们说throw引发了异常
- try语句块，异常处理部分使用try语句块处理异常。try语句块以关键字try开始，并以一个或多个catch子句结束。try语句块以关键字try开始，并以一个或多个catch子句结束。try语句块中代码抛出的异常通常会被某个catch子句处理。因为catch子句“处理”异常，所以它们也被称为异常处理代码
- 一套异常类，用于在throw表达式和相关的catch子句之间传递异常的具体信息

### 5.6.1 throw表达式

程序的异常检测部分使用throw表达式引发一个异常。throw表达式包含关键字throw和紧随其后的一个表达式，其中表达式的类型就是抛出的异常类型。throw表达式后面通常紧跟一个分号，从而构成一条表达式语句

```
1 if(i > j){  
2     throw runtime_error("message");  
3 }
```

抛出异常将终止当前的函数，并把控制权转移给能处理该异常的代码

## 5.6.2 try语句块

```
1 try{  
2     program-statements  
3 }catch(exception-declaration){ //exception-  
declaration 是异常声明  
4     handler-statements  
5 }catch(exception-declaration){  
6     handler-statements  
7 } // ...
```

## 5.6.3 标准异常

- exception头文件定义了最通用的异常类exception。它只报告异常的发生，不提供任何额外信息
- stdexcept头文件定义了几种常用的异常类。
- new头文件定义了bad\_alloc异常类型
- type\_info头文件定义了bad\_cast异常类型

# 第六章 函数

## 6.1 函数基础

一个典型的函数包括以下几个部分：

- 返回类型
- 函数名字
- 形参列表
- 函数体

通过调用运算符 `()` 来执行函数

函数的调用完成两项工作：

1. 用实参初始化函数对应的形参
2. 将控制权转移给被调用函数

`return`语句完成两项工作：

1. 返回`return`语句中的值
2. 将控制权转移回主调函数

### 形参和实参

实参和形参一一对应，但是并没有规定实参的求值顺序，编译器能以任意可行的顺序对实参数求值。

形参一定会被初始化

#### 6.1.1 局部对象

形参和函数体内部定义的变量统称为局部变量

局部静态对象，在程序的执行路径第一次经过对象定义语句时初始化，并且直到程序终止才被销毁

## 6.1.2 函数声明

函数声明也称作函数原型

函数的声明不包含函数体，所以无须参数的名字。

## 6.2 参数传递

### 6.2.1 传值参数

### 6.2.2 传引用参数

使用引用避免拷贝，对于大类型对象或容器对象比较有用

### 6.2.3 const形参和实参

```
1 void fcn(const int i){}
2 void fcn(int i){}
```

用实参初始化形参时会忽略掉[顶层const](#)，传给该形参的值是不是const的都可以，但是像上述的代码就会出现歧义，所以这样定义是错的

### 6.2.4 数组形参

不允许拷贝数组以及使用数组时通常会将其转换成指针。

因为数组是以指针形式传递给函数的，所以还需要额外提供数组的尺寸等信息

管理指针形参有三种常用的技术：

1. 使用标记指定数组长度

要求数组本身包含一个结束标记

2. 使用标准库规范

传递指向数组首元素和尾后元素的指针

### 3. 显式传递一个表示数组大小的值

## 数组引用形参

C++允许将变量定义成数组的引用，所以形参也可以是数组的引用。  
引用形参绑定到对应的实参上，也就是绑定到数组上

```
1 void print(int (&arr)[10]){
2     for (auto elem : arr){
3         cout << elem << endl;
4     }
5 }
```



&arr 两端的括号必不可少

```
1 f(int &arr[10]) //错误，将arr声明成了引用的数组
2 f(int (&arr)[10]) //正确 arr是具有10个int型数
据的数组的引用
```

## 6.2.5 main处理命令行选项

```
1 prog -d -o ofile data0
```

```
1 int main(int argc, char *argv[]){ ... }
2 int main(int argc, char **argv){ }
```

第二个形参argv是一个数组，他的元素是指向C风格字符串的指针；  
第一个形参argc表示数组中字符的个数

## 6.2.6 含有可变形参的函数

C++11 提供了两种主要的方法：

- 如果所有的实参类型相同，可以传递一个名为 `initializer_list` 的标准库类型
- 如果实参的类型不同，可以编写一种特殊的函数，即可变参数模板（16.4 节介绍）

C++还有一种特殊的形参类型（省略符），可以用它传递可变数量的实参，这种功能一般只用于与C函数交互的接口程序

### initializer\_list

操作	含义
<code>initializer_list&lt;T&gt; list</code>	默认初始化：T类型的元素空列表
<code>initializer_list&lt;T&gt; list{a,b,c,...}</code>	列表中的元素是const
<code>lst2(lst)</code>	
<code>lst2 = lst</code>	
<code>lst.begin()</code>	
<code>lst.end()</code>	

`initializer_list` 对象中的元素永远是常量值，我们无法改变。

### 省略符形参

省略符形参只能出现在形参列表的最后一个位置

```
1 void foo(parm_list, ...);
2 void foo(...);
```

## 6.3 返回类型和 `return` 语句

### 6.3.1 无返回值函数

### 6.3.2 有返回值函数

返回一个值的方式和初始化一个变量或形参的方式完全一样：返回的值用于初始化调用点的一个临时两，该临时量就是函数调用的结果

#### 不要返回局部对象的引用或指针

#### 引用返回左值

调用一个返回引用（不能是局部变量的引用）的函数得到左值，其他返回类型得到右值。我们能为返回类型是非常量引用的函数的结果赋值

#### 列表初始化返回值

C++11新标准规定，函数可以返回花括号包围的值的列表

如果函数返回的是内置类型，则花括号包围的列表最多包含一个值，而且该值所占空间不应该大于目标类型的空间（这个设定很奇怪）。如果函数返回的是类类型，由类本身定义初始值如何使用。

#### 主函数main的返回值

我们允许main函数没有return语句直接结束。如果控制到达了main函数的结尾处而且没有return语句，则隐式插入一条return 0;

### 6.3.3 返回数组指针

无法返回数组（因为数组无法拷贝），只能返回数组指针

从语法上讲，定义一个返回数组的指针或引用的函数比较繁琐，但是使用类型别名等方法可以简化这一操作

```

1 typedef int arrT[10]; //arrT是一个类型别名，表示的类
  型是含有10个整数的数组
2 using arrT = int[10]; //arrT的等价声明
3 arrT* func(int i); //func返回一个指向含有10个整数的
  数组的指针

```

## 声明一个返回数组指针的函数

要想在声明func时不使用类型别名，必须牢记被定义的名字后面数组的维度

```

1 int arr[10]; //arr是含有10个指针的数组
2 int *p1[10]; //p1是含有10个指针的数组
3 int (*p2)[10] = &arr; // p2是一个指针，指向含有10个
  整数的数组

```

与这些声明一样，如果我们想定义一个返回数组指针的函数，则数组的维度必须跟在函数名字之后（实际上应该在形参列表之后）

```

1 Type (*function(parameter_list))[dimension]
2 int (*func(int i))[10]

```

- func(int i) 表明调用func函数需要一个整型值
- (\*func(int i)) 意味着我们可以对函数调用的结果执行解引用操作
- (\*func(int i))[10] 表示解引用操作将得到一个大小是10的数组
- int (\*func(int i))[10] 表示数组中的元素类型是int

## 使用尾置返回类型

为了表示函数真正的返回类型跟在形参列表之后，我们在返回类型的位置上放一个 auto

```

1 auto func(int i) -> int(*)[10]; //接受int类型的实
  参，返回一个指针，指向含有10个整型的数组

```

## 使用decltype

```

1 int odd[] = {1,3,5,7,9};
2 int even[] = {0,2,4,6,8};
3 decltype(odd) *arrPtr(int i){
4     return (i % 2) ? &odd : &even;
5 }
```

arrPtr使用关键字decltype表示它返回类型是个指针，并且该指针所指对象与odd的类型一致。因为odd是数组，所以arrPtr返回指向含有5个整数的数组的指针。decltype并不负责将数组类型转换为相应的指针，所以decltype的结果是个数组，要想表示arrPtr返回指针还必须在函数声明时加一个\*符号

## 6.4 函数重载

同一作用域几个函数名字相同但是形参列表不同，称之为重载

不允许两个函数除了返回类型外其他所有的要素都相同

### 重载和const形参

顶层const无法实现重载，底层const可以实现重载，我们只能把const对象传递给const形参，当我们传一个非常量对象或者指向非常量对象的指针时，编译器会优先选择非常量版本的函数。

### const\_cast 和重载

下面这个方法返回const string shorterString

```

1 const string &shorterString(const string &s1,
2                             const string &s2){
3     return s1.size() < s2.size() ? s1 : s2;
4 }
```

这个函数的参数和返回类型都是`const string`的引用。我们可以对非常量的`string`实参调用这个方法，但是它返回的仍旧是`const string`的引用，这导致我们可能无法修改返回的`string`。

因此我们需要新的`shorterString`方法，当它的实参不是常量时，得到的结果是一个普通的引用，使用 `const_cast` 可以做到

```

1 string &shorterString(string &s1, string &s2) {
2     auto &r = shorterString(const_cast<const
3         string&>(s1), const_cast<const string&>(s2));
4     return const_cast<string&>(r);
5 }
```

能够得到如下的效果，如果实参是`const string`引用，那么直接调用`const`版本，如果不是，那么调用非`const`版本，此时将一个本来就是非常量的数据（虽然返回的时候是按常量返回的）强制转换成非`const`是可以接受的。

```

1 const string& shorterString(const string& s1,
2     const string& s2)
3 {
4     cout << "这是 const 版本" << endl;
5     return s1.size() < s2.size() ? s1 : s2;
6 }
7 string& shorterString(string& s1, string& s2)
8 {
9     cout << "这是非 const 版本" << endl;
10    auto& r = shorterString(const_cast<const
11        string&>(s1), const_cast<const string&>(s2));
12    return const_cast<string&>(r);
13 }
14 int main(int argc, char const* argv[])
15 {
16     const string s1 = "aaa", s2 = "bbb";
17     string s3 = "ccc", s4 = "ddd";
18     string s5 = shorterString(s1, s2);
```

```

17     string s6 = shorterString(s3, s4);
18     return 0;
19 }
20 //输出结果为
21 //这是 const 版本
22 //这是非 const 版本
23 //这是 const 版本

```

## 调用重载的函数

**函数匹配（重载确定）**：在这个过程中，我们把函数调用与一组重载函数中的某一个关联起来，函数匹配也叫做重载确定。

我们将在后续[6.6节](#)介绍当重载函数参数数量相同且参数类型可以互相转换时编译器的处理方法（因为其余情况都非常好确定）

当调用重载函数时有三种可能的结果：

- 编译器找到一个与实参最佳匹配的函数，并生成调用该函数的代码
- 找不到任何一个函数与调用的实参匹配，此时编译器发出无匹配的错误信息
- 有多于一个函数可以匹配，但都不是最佳选择。此时也将发生错误，称为二义性调用

### 6.4.1 重载与作用域

如果我们在内层作用域中声明名字，它将隐藏外层作用域中声明的同名实体（而不是重载它）。所以在不同的作用域中无法重载函数名。

## 6.5 特殊用途语言特性

### 6.5.1 默认实参

一旦某一个形参被赋予了默认值，它后面的所有形参都必须有默认值

## 默认实参声明

通常将函数声明放在头文件中，并且一个函数只声明一次。但是多次声明函数也是合法的。需要注意的是在给定的作用域中一个形参只能被赋予一次默认实参。函数的后续声明只能为之前没有默认值的形参添加默认实参，而且该形参右侧的所有形参都必须有默认值。

```

1 string screen(sz, sz, char = ' ');
2 string screen(sz, sz, char = '*');// error 无法修改默认值
3 string screen(sz, sz = 80, char = ' ')// 正确：添加默认实参

```

## 默认实参初始值

局部变量不能作为默认实参

## 6.5.2 内联函数和constexpr函数

### 内敛函数

对于shorterString方法，若定义为内联，如下调用

```

1 cout << shorterString(s1,s2) << endl;
2 //将展开为
3 cout << (s1.size() < s2.size() ? s1 : s2) <<
    endl;

```

### constexpr函数

constexpr函数是指能用于常量表达式的函数。

- 函数的返回类型及所有形参的类型都得是字面值类型
- 函数体中必须有且只有一条return语句

```

1 constexpr int new_sz(){return 42;}
2 constexpr int foo = new_sz(); //正确: foo是一个常量表达式

```

constexpr被隐式的指定为内联函数

constexpr函数体内可以包含其他语句，只要这些语句在运行时不执行任何操作就行，例如空语句、类型别名、using声明

允许constexpr函数的返回值为非常量

```

1 constexpr size_t scale(size_t cnt){return
    new_sz() * cnt;}
2 int arr[scale(2)]; //正确scale(2)是常量表达式
3 int i = 2;
4 int a2[scale(i)]; //错误: scale(i)不是常量表达式

```

当scale的实参是常量表达式时，返回值也是常量表达式；反之则不然

内联函数和constexpr函数的多个定义必须完全一致，所以一般定义在头文件中

### 6.5.3 调试帮助

#### assert预处理宏

assert是一种预处理宏，是一个预处理变量。assert宏使用一个表达式作为它的条件：

```

1 assert(expr);
2 //首先对expr求值，如果表达式为false，assert输出信息并
  终止程序的执行，如果表达式为true，assert什么也不做

```

宏名字在程序内必须唯一

## NDEBUG预处理变量

assert的行为依赖于一个名为NDEBUG的预处理变量。如果定义了NDEBUG，则assert什么也不做。默认状态下没有定义NDEBUG，此时assert将执行运行时检查

我们可以使用一个#define语句定义NDEBUG，从而关闭调试状态。

## 6.6 函数匹配

函数匹配的第一步是选定本次调用对应的重载函数集，集合中的函数称为候选函数。候选函数具备两个特征：

- 与被调用函数同名
- 声明在调用点可见

第二步考察本次调用提供的实参，然后从候选函数中选出能被这组实参调用的函数，这些新选出的函数称为可行函数。可行函数也具有两个特征

- 形参数量与本次调用提供的实参数量相等
- 每个实参的类型和对应的形参类型相同，或者可以转换为形参的类型

第三步从可行函数中选择与本次调用最匹配的函数。逐一检查函数调用提供的实参，寻找形参类型与实参类型最匹配的那个可行函数。

- 该函数每个实参的匹配都不劣于其他可行函数需要的匹配
- 至少有一个实参的匹配优于其他可行函数提供的匹配

如果没有函数脱颖而出则会报错二义性调用

### 6.6.1 实参类型转换

为了确定最佳匹配，编译器将实参类型到形参类型的转换划分为几个等级，具体排序如下：

1. 精确匹配

- 实参类型与形参类型相同
- 实参从数组类型或函数类型转换成对应的指针类型参见 [6.7 函数指针](#)
- 向实参添加顶层const或者从实参中删除顶层const (这里应该要保证实参本质上是一个非常量? )

2. 通过const转换实现的匹配
3. 通过类型提升实现的匹配
4. 通过[算术类型](#)的转换实现的匹配
5. 通过类类型转换实现的匹配 (参见[14.9节](#))

## 类型提升和算术类型转换

所有算术类型转换的级别都一样。例如从int转换为unsigned int 与 int 向 double 的转换时相同的级别。具体例子

```
1 void manip(long);
2 void manip(float);
3 manip(3.14);
```

3.14的类型是double，它既能转换为long也能转换为float，所以此时会发生二义性调用错误

## 函数匹配和const实参

如果重载函数的区别在于它们引用类型的形参是否引用了const，或者指针类型的形参是否指向const，当调用发生时编译器通过实参是否是常量来决定选择哪个函数

## 6.7 函数指针

函数指针指向的是函数而非对象。函数指针也指向特定的类型。函数的类型由它的返回类型和形参类型共同决定与函数名无关

```
1 bool lengthCompare(const string &, const string &);
```

该函数的类型是bool (const string &, const string &)。

函数指针声明如下：

```
1 bool (*pf)(const string &, const string &);  
2 //pf指向一个函数，该函数的参数是两个const string 的引用，返回值是bool类型。 pf 两端的括号不能少
```

## 使用函数指针

将函数名字作为一个值使用，该函数自动地转换成指针，

```
1 pf = lengthCompare;  
2 pf = &lengthCompare; //等价操作  
3  
4 bool b1 = pf("hello", "good");  
5 bool b2 = (*pf)("hello", "good");  
6 bool b3 = lengthCompare("hello", "good");
```

指向不同的函数类型的指针之间不存在转换关系

函数指针也可以赋一个nullptr或者0表示未指向任何函数

## 重载函数指针

使用重载函数的时候，上下文必须清晰地界定到底该选用哪个函数。

```

1 void ff(int* );
2 void ff(unsigned int);
3 void (*pf1)(unsigned int) = ff; //指向ff(unsigned int)
4
5 void (*pf2)(int) = ff; //错误，没有ff与形参列表匹配
6 double (*pf3)(int*) = ff; //错误，没有返回类型与此匹配

```

## 函数指针形参

与数组类型，虽然不能定义函数类型的形参，但是可以定义指向函数的指针作为形参

```

1 //第三个参数是函数类型，会自动转换成函数指针
2 void useBigger(const string &s1, const string
&s2,
3                 bool pf(const string &,const
string&));
4 void useBigger(const string &s1, const string
&s2,
5                 bool (*pf)(const string &,const
string&));
6 useBigger(s1,s2,lengthCompare);

```

## 返回指向函数的指针

虽然无法返回一个函数，但是可以返回指向函数的指针。必须将返回类型写成指针形式，编译器不会自动地将函数返回类型当成对应的指针类型处理。

```

1 using F = int(int*, int); //F是函数类型, 不是指针
2 using PF = int(*)(int*, int); // PF 是指针类型
3
4 PF f1(int); //正确, 返回指向函数的指针
5 F f1(int); //错误, 不能返回一个函数
6 F *f1(int); //正确, 显式的声明返回类型是指向函数的指针

```

## 第七章 类

类的基本思想是**数据抽象和封装**。

数据抽象是一种依赖于接口和实现的分离式编程技术

### 7.1 定义抽象数据类型

定义在类内部的函数是**隐式**的**inline**函数

#### 定义成员函数

成员都必须在类内说明, 但是成员函数体可以在类外定义

成员函数通过一个 `this` 的额外**隐式**参数来访问调用它的那个对象

#### 引入**const**成员函数

默认的情况下`this`的类型是指向类类型非常量版本的常量指针。`this`同样要遵守初始化规则, 这意味着默认情况下我们不能把`this`绑定到一个常量对象上。这也使得我们无法在常量对象上调用普通成员函数。

**紧跟在参数列表之后的**const**表示**this**是一个指向常量的指针**。像这样使用**const**的成员函数被称作**常量成员函数**。

```

1 class A{
2     public:
3         int func() const;
4 }
```

## 类作用域和成员函数

编译器分两步处理类：

1. 编译成员的声明
2. 然后才轮到成员函数体

因此成员函数体可以随意使用类中的其他成员而无须在意这些成员的出现顺序

## 在类的外部定义成员函数

在类的外部定义成员函数时，成员函数的定义必须与它的声明匹配。也就是说，返回类型、参数列表和函数名都需保持一致。如果成员被声明成常量成员函数，那么它定义也必须在参数列表后明确指定const属性

### 7.1.4 构造函数

构造函数的任务是初始化类对象的数据成员，无论何时只要类被创建就会执行构造函数

- 构造函数的名字和类名相同
- 构造函数没有返回类型
- 构造函数不能被声明成const。当我们创建类的一个const对象时，知道构造函数完成初始化过程，对象才能真正取得其常量属性，因此构造函数在const对象的构造过程中可以向其写值。

## 默认构造函数

没有显式定义构造函数的时候，编译器会隐式的定义一个默认构造函数。编译器创建的构造函数又称为合成的默认构造函数。

- 如果存在类内的初始值，用它来初始化成员
- 否则，默认初始化该成员

对于一个普通的类来说，必须定义它自己的默认构造函数。

- 只有当类没有声明任何构造函数的时候，编译器才会自动地生成默认构造函数
- 对于某些类来说，合成的默认构造函数可能执行错误的操作
- 有时候编译器不能为某些类合成默认的构造函数，比如类中成员是一个类，且该类没有默认构造函数

我们可以使用 =default的方式定义默认构造函数

```
1 Slaes_data() = default; //要求编译器生成默认构造函数
```

## 构造函数初始值列表

```
1 Sales_data(const std::string &s) : bookNo(s){}
```

冒号和花括号之间的代码负责为新创建的对象的一个或几个数据成员赋初值

## 7.1.5 拷贝、赋值和析构

如果我们不主动定义这些操作，编译器将会替我们合成

## 7.2 访问控制与封装

访问说明符可以加强类的封装性：

- 定义在public说明符之后的成员在整个程序内可以被访问
- 定义在private说明符之后的成员可以被类的成员函数访问

# class 和 struct

struct的默认访问权限是public的，class的默认访问权限是private的

## 7.2.1 友元

类可以允许其他类或者函数访问它的非公有成员，增加一条以friend关键字开始的函数声明

友元声明只能出现在类定义的内部，但是在类内出现的具体位置不限

```
1 friend Sales_data add(const Sales_data&, const
Sales_data&);
```

## 7.3 类的其他特性

### 7.3.1 类成员再探

除了定义数据和函数成员之外，类还可以自定义某种类型在类内的别名

```
1 class Screen{
2     public:
3         typedef std::string::size_type pos;
4     private:
5         pos cursor = 0;
6         pos height = 0;
7 };
```

## 可变数据成员

如果我们希望改变类的某个数据成员，即使是在const成员函数内，我们可以在变量声明的时候加入 `mutable` 关键字

一个可变数据成员永远不会是const的，即使它是const对象的成员

## 类的声明

能够仅仅声明类而不定义它：

```
1 class Screen; //只声明
```

对于类型 Screen 来说，在它的声明之后定义之前是一个**不完全类型**

不完全类型只能在有限的情况下使用：

- 可以定义指向这种类型的指针或引用，静态数据类型也可以
- 可以声明(但是不能定义)以不完全类型作为参数或者返回类型的函数

### 7.3.4 友元再探

#### 类之间的友元

如果一个类指定了友元类，则友元类的成员函数可以访问此类包括非公有成员在内的所有成员

除了令整个类作为友元之外也可以只为成员函数声明友元

```
1 class Screen{
2     friend class Window_mgr;
3 }; //Window_mgr 的成员可以访问Screen的私有部分
4 class Screen{
5     friend void Window_mgr::clear(ScreenIndex);
6 }; //这个成员函数可以访问Screen的私有部分
```

我们必须按照如下方式设计程序：

- 首先定义Window\_mgr类，其中声明clear函数，但是不能定义它。在clear使用Screen的成员之前必须先声明Screen
- 接下来定义Screen，包括对于clear的友元声明
- 最后定义clear，此时它才可以使用Screen的成员

如果一个类想把一组重载函数声明成它的友元，必须对每一个函数分别声明

## 7.4 类的作用域

### 作用域和定义在类外部的成员

一个类就是一个定义域能够解释为何在类的外部定义成员函数的时候要指明它所属的类 (A::func())。一旦遇到了类名，就知道剩余部分在类的作用域之内了。

函数的返回类型通常出现在函数名之前，因此当成员函数定义在类的外部时，返回类型中使用的名字都位于类的作用域之外。这时返回类型必须指明它是那个类的成员

#### 7.4.1 名字查找与类的作用域

名字查找的过程比较直接：

- 首先在名字所在的块中查找声明，只考虑在名字之前出现的声明。
- 如果没找到，继续查找外层作用域
- 如果最终没找到，则报错

但是在类中有所区别：

- 编译成员的声明
- 直到类全部可见后才编译函数体

### 用于类成员声明的名字查找

这种两阶段的处理方式只适用于成员函数中使用的名字，声明中使用的名字，包括返回类型或者参数列表中使用的名字，都必须在使用前确保可见。

## 7.5 构造函数再探

### 7.5.1 构造函数初始值列表

构造函数初始值列表和构造函数体内的赋值是有区别的，如果没有在构造函数的初始值列表中显式地初始化成员，则该成员将在构造函数体之前执行默认初始化。

#### 构造函数的初始值有时必不可少

如果成员是const或者引用的话，必须将其初始化。当成员属于某种类类型且该类没有定义默认构造函数时，也必须将这个成员初始化。随着构造函数体一开始执行，初始化就完成了。我们初始化const或者引用类型的唯一机会就是通过构造函数初始值列表 

#### 成员初始化的顺序

成员初始化的顺序与它们在类定义中的顺序一致。构造函数初始值列表中的顺序不会影响实际的初始化顺序。不过最好令构造函数初始值的顺序与成员声明的顺序保持一致 

### 7.5.2 委托构造函数

一个委托构造函数使用它所属类的其他构造函数执行它自己的初始化过程。

委托构造函数也有一个成员初始值的列表和一个函数体。但是成员初始值列表只有一个唯一的入口，就是类名本身

```

1 class Sales_data{
2     public:
3         Sales_data(std::string s,unsigned
4             cnt,double
5             price):bookNo(s),units_sold(cnt),revenue(cnt *
6             price){}
7             Sales_data():Sales_data("",0,0){} //委托
8             构造函数
9             Sales_data(std::string
10             s):Sales_data(s,0,0){} //委托构造函数
11         };

```

执行顺序：

1. 委托这的构造函数初始值列表
2. 被委托者的构造函数初始值列表
3. 被委托者的构造函数体
4. 委托者的构造函数体

### 7.5.3 默认构造函数的作用

对象被默认初始化或值初始化时自动执行默认构造函数。

默认初始化在以下情况下发生：

- 当我们在块作用域内不使用任何初始值定义一个非静态变量或者数组时
- 当一个类本身含有类类型的成员且使用合成默认构造函数时
- 当类类型的成员没有在构造函数初始值列表中显式地初始化时

值初始化在以下情况下发生：

- 在数组初始化的过程中如果我们提供的初始值数量少于数组的大小时
- 当我们不使用初始值定义一个局部静态变量时
- 当我们通过书写形如T()的表达式显式地请求值初始化时，其中T是类型名，它就是使用一个这种形式的实参来对它的元素初始化

## 7.5.4 隐式的类类型转换

```

1 class Sales_data{
2     public:
3         Sales_data(std::string s, unsigned cnt,
4             double price): bookNo(s),
5             units_sold(cnt), revenue(cnt * price){}
6         Sales_data(std::string
7             s):Sales_data(s,0,0){} //这里实际上是叫委托构造函
8             数的东西，但是在这不是重点
9         Sales_data(std::istream& );
10    };
11 //看下面的代码
12 std::string null_book = "9-999-9999-9";
13 item.combine(null_book); //item 是一个
14 Sales_data 对象, combine是 Sales_data 的成员其中参
15 数列表是 Sales_data 类型的。
16 //但是这里能够用一个 string 类型作为实参, 是因为发生了
17 隐式类型转换(原因是 Sales_data 类中有使用一个 string
18 的构造函数)

```

## 只允许一步类类型转换

```

1 item.combine("9-999-9999-9"); //错误, 字符串字面值
2 不是 string 类型的, 所以此处需要两次转换
3 1. 把字符串字面量转换为 string 类型
4 2. 再将 string 类型转换为 Sales_data类型
5 这种操作是不允许的, 因为编译器只会自动执行一步类型转换

```

## 抑制构造函数定义的隐式转换

可以通过将构造函数声明为 `explicit` 阻止隐式转换

```
1 explicit Sales_data(std::string
s):Sales_data(s,0,0){}
```

1. 关键字 `explicit` 只对一个实参的构造函数有效。多个实参的构造函数不能隐式转换
2. 只能在类内声明处使用 `explicit` 关键字，在类外定义时不能重复】
3. 尽管编译器不会将 `explicit` 的构造函数用于隐式转换，但是仍旧能够使用它进行显示的强制转换

```
1 item.combine(static_cast<Sales_data>
(cin));
```

## 7.5.5 聚合类

聚合类具有特殊的初始化语法形式

满足以下条件的类是聚合类

- 所有成员都是 `public` 的
- 没有定义任何构造函数
- 没有类内初始值
- 没有基类，也没有 `virtual` 函数

```
1 struct{
2     int ival;
3     string s;
4 };
5 Data vall = {0, "aaa"}; //保持与声明顺序一致即可初始化
//如果初始值列表中的元素个数少于类的成员数量，则靠后的
//成员被值初始化
```

显式地初始化类的对象的成员存在三个明显的缺点：

- 要求类的所有成员都是`public`的

- 将正确初始化每个对象的每个成员的重任交给了类的用户（而非类的作者）。
- 添加或删除一个成员之后，所有的初始化语句都要更新

## 7.5.6 字面值常量类

数据成员都是字面值类型的聚合类是字面值常量类

如果一个类不是聚合类，但符合下述要求，也是一个字面值常量类：

- 数据成员都必须是字面值类型
- 类必须至少含有一个`constexpr`构造函数
- 如果一个数据成员含有类内初始值，则内置类型成员的初始值必须是一条常量表达式；或者如果成员属于某种类类型，则初始值必须使用成员自己的`constexpr`构造函数
- 类必须使用析构函数的默认定义，该成员负责销毁类的对象

### constexpr构造函数

构造函数不能是`const`的，但是字面值常量类的构造函数可以是`constexpr`函数。一个字面值常量类必须至少提供一个`constexpr`构造函数

`constexpr`构造函数可以声明成`=default`的形式或者是删除函数的形式。否则`constexpr`函数就必须符合构造函数的要求不能有返回值，又符合`constexpr`函数的要求即唯一的可执行语句是返回语句，综合这两点可知，`constexpr`构造函数体一般应该是空的。

```

1 class Debug{
2     public:
3         constexpr Debug(bool b =
4             true) : hw(b), io(b), other(b) {}
5 }
```

`constexpr`构造函数必须初始化所有数据成员，初始值或者使用`constexpr`构造函数或者是一条常量表达式

## 7.6 类的静态成员

- 在类的外部定义静态成员时，不能重复 `static` 关键字，该关键字只出现在类的内部的声明语句
  - 不能在类的内部初始化静态成员。必须在类的外部定义和初始化每个静态成员，一个静态数据成员只能定义一次(可以为静态成员提供 `const` 整数类型的类内初始值，不过要求静态数据成员必须是字面值常量类型的 `constexpr` )
  - 静态数据成员可以是**不完全类型**。特别的静态数据成员的类型可以是它所属的类型。
-

# 第 II 部分 C++ 标准库

## 第八章 IO库

### 8.1 IO类

IO库类分为以下几大类

头文件	类型
iostream	istream, wistream 从流读入数据 ostream, wostream 向流写入数据 iostream, wiostream 读写流
fstream	ifstream, wifstream 从文件读取数据 ofstream, wofstream 向文件写入数据 fstream, wfstream 读写文件
sstream	istringstream, wistringstream 从string读取数据 ostringstream, wostringstream 向string写入数据 stringstream, wstringstream 读写string

类型 `ifstream` 和 `istringstream` 都继承自 `istream`

### IO类型间的关系

类型 `ifstream` 和 `istringstream` 都继承自 `istream`

`ofstream` 和 `ostringstream` 都继承自 `ostream`

## 8.1.1 IO对象无拷贝或赋值

不能拷贝或对IO对象赋值，因此无法将形参或返回类型设置为流类型

进行IO操作的函数通常以引用方式传递和返回流。读写一个IO对象会改变其状态，因此传递和返回的引用不能是 `const`

## 8.1.2 条件状态

### IO库条件状态

属性和操作	含义
<code>strm::iostate</code>	<code>strm</code> 是一种IO类型，在 <a href="#">8.1 IO类</a> 中列出。 <code>iostate</code> 是一种机器相关的类型，提供了表达条件状态的完整功能
<code>strm::badbit</code>	<code>strm::badbit</code> 用来指出流已崩溃
<code>strm::failbit</code>	<code>strm::failbit</code> 用来指出一个IO操作失败了
<code>strm::eofbit</code>	<code>strm::eofbit</code> 用来指出流到达了文件结束
<code>strm::goodbit</code>	<code>strm::goodbit</code> 用来指出流未处于错误状态。此值保证为0
<code>s.eof()</code>	若流s的eofbit置位，则返回true
<code>s.fail()</code>	若流s的failbit或badbit置位，返回true
<code>s.bad()</code>	流s的badbit置位返回true
<code>s.good()</code>	流处于有效状态，返回true
<code>s.clear()</code>	将流中的所有条件状态复位，将流的状态设置为有效，返回void

属性和操作	含义
s.clear(flag)	根据给定的flag标志位，将流s中的对应条件状态位复位。flag的类型为strm::iostate。返回void
s.setstate(flags)	根据给定的标志位flags，将流中对应条件状态位置位。flags的类型strm::iostate。返回void
s.rdstate()	返回流s的当前条件状态，返回值类型为strm::iostate

## 查询流的状态

IO库定义了一个与机器无关的 `iostate` 类型，它提供了表达流状态的完整功能。这个类型作为一个位集合来使用

- `badbit` 表示系统级错误。通常被置位，流就无法再使用了。
- `failbit` 在发生可恢复错误后被置位，如期望读取数值却读入一个字符等错误。
- 如果到达文件结束位置 `eofbit` 和 `failbit` 都会被置位。
- `goodbit` 的值为0，表示流未发生错误。

如果 `badbit`、`failbit`、`eofbit` 任一个被置为，检测流状态会返回 `false`

标准库还定义了一组函数来查询这些标志位的状态。`good()`在所有错误位均为置位的情况下返回 `true`，而`bad()`、`fail()`和`eof()`则在对应错误被置位时返回`true`。使用流对象`.good()`的方式调用。

## 管理条件状态

流对象的`rdstate`成员返回一个`iostate`值，对应流的当前状态；`setstate`操作将给定条件位置位，表示发生了对应的错误。

`clear()`方法有两个版本：

- 无参版本清除所有的错误标志位

- 接受 iostate 值的版本表示流的状态更新为新的 iostate

### 8.1.3 管理输出缓冲

导致缓冲刷新得原因有很多：

- 程序正常结束，作为 main 函数得 return 操作的一部分，缓冲刷新被执行
- 缓冲区满时刷新
- 使用操纵符如 endl 显式刷新缓冲区
- 可以设置 unitbuf 设置流的内部状态来清空缓冲区，默认情况下 cerr 是设置 unitbuf 的

```
1 cout << unitbuf; //所有输出操作后都会立即刷新
  缓冲区
2 cout << nounitbuf; //回到正常的缓冲方式
```

- 一个输出流可以关联到另一个流

### 刷新输出缓冲区

endl:换行并刷新缓冲区

flush:刷新缓冲区，但不输出任何额外的字符

ends:向缓冲区插入一个空字符，然后刷新缓冲区

### 关联输入和输出流

一个输入流和输出流关联在一起时，任何试图从输入流读取数据的操作都会先刷新关联的输出流。标准库将 cout 和 cin 关联在一起

```
1 cin >> ival; //cout 的缓冲区被刷新
```

tie()有两个版本：

1. 无参版本，返回指向输出流的指针。如果对象关联了某个输出流，则返回指向该输出流的指针，如果未关联，则返回空指针。
2. 接受一个指向 ostream 的指针，将自己关联到此 ostream。

```

1 cin.tie(&cout); //仅仅是个展示，标准库将cin和cout关联在一起
2 ostream *old_tie = cin.tie(nullptr); // cin 不再与其他流关联，old_tie此时指向cout
3 cin.tie(old_tie); //将cin和cout重新关联起来

```

## 8.2 文件输入输出

头文件 `fstream` 定义了三个类型来支持文件IO：

- `ifstream` 从一个给定文件读取数据
- `ofstream` 向一个给定文件写入数据
- `fstream` 可以读写给定文件

除了继承自 `iostream` 类型的行为之外，`fstream` 中定义的类型还增加了一些新的成员来管理与流关联的文件。

### fstream 特有的操作

操作	
<code>fstream fstrm;</code>	创建一个未绑定的文件流。 <code>fstream</code> 是头文件 <code>fstream</code> 中定义的一个类型
<code>fstream fstrm(s);</code>	创建一个 <code>fstream</code> ，并打开名为s的文件。s可以是 <code>string</code> 或者C风格字符串的指针。默认的文件模式依赖于 <code>fstream</code>
<code>fstream fstrm(s,mode);</code>	按指定mode打开文件

## 操作

fstrm.open(s)	打开名为s的文件，并将文件与fstrm绑定。s可以是string或者C风格字符串指针。默认的mode依赖于fstream
fstrm.close()	关闭与fstrm绑定的文件
fstrm.is_open()	返回一个bool值，指出与fstrm关联的文件是否成功打开且尚未关闭

## 8.2.1 使用文件流对象

读写文件时，可以定义一个文件流对象，并将对象与文件关联起来。每个文件流对象都定义了一个名为open的成员函数，它完成一些系统相关操作，来定位给定的文件，并视情况打开为读或写模式。

如果提供了一个文件名，则open会被自动调用

```
1 ifstream in(ifile);
2 ofstream out;
```

### 成员函数open和close

如果定义了一个空文件流对象，可以随后调用open来将它与文件关联起来：如果调用open失败，failbit会被置位

调用close()之后还能够打开新的文件，close会关闭已经关联的文件

当一个fstream被销毁时，close会自动调用

## 8.2.2 文件模式

每个流都有一个关联的文件模式，用来指出如何使用文件

### 文件模式

模式	意义
in	以读方式打开
out	以写方式打开
app	每次写操作前均定位到文件末尾
ate	打开文件后立即定位到文件末尾
trunc	截断文件
binary	以二进制方式进行IO

指定文件模式有如下限制：

- 只可以对ofstream或fstream对象设定out模式
- 只可以对ifstream或fstream对象设定in模式
- 只有当out也被设定时才可以设定trunc模式
- 只要trunc没被设定，就可以设定app模式。在app模式下，即使没有显式指定out模式，文件也总是以输出方式被打开
- 默认情况下，即使我们没有指定trunc，以out模式打开的文件也会被截断。为了保留以out模式打开的文件的内容，我们必须同时指定app模式，这样只会将数据追加写到文件末尾；或者同时指定in模式，即打开文件同时进行读写操作
- ate和binary模式可以用于任何类型的文件流对象，且可以与其他任何文件模式组合使用。

每一个文件流类型都定义了一个默认的文件模式，当未指定文件模式时，以默认方式打开

- 与ifstream关联的文件默认in模式
- 与ofstream关联的文件默认out模式
- 与fstream关联的文件默认in和out模式

保留被ofstream打开的文件中已有数据的唯一方法是显式指定app或in模式

每次调用open时都会重新确定文件模式

## 8.3 string 流

- `istringstream`从`string`读取数据
- `ostringstream`向`string`写入数据
- `stringstream`既可以从`string`读数据也可以写入数据

`sstream`继承自`iostream`

### stringstream特有操作

操作	
<code>sstream</code> <code>strm;</code>	<code>strm</code> 是一个未绑定的 <code>stringstream</code> 对象， <code>sstream</code> 是头文件 <code>sstream</code> 中定义的一个类型
<code>sstream</code> <code>strm(s);</code>	<code>strm</code> 是一个 <code>sstream</code> 对象，保存 <code>string s</code> 的一个拷贝。此构造函数是 <code>explicit</code> 的
<code>strm.str()</code>	返回 <code>strm</code> 所保存的 <code>string</code> 的拷贝
<code>strm.str(s)</code>	将 <code>string s</code> 拷贝到 <code>strm</code> 中，返回 <code>void</code>

### 8.3.1 使用`istringstream`

### 8.3.2 使用`ostringstream`

## 第九章 顺序容器

### 9.1 顺序容器概述

顺序容器大全

#### 容器类型

## 容器类型

vector	可变大小数组。支持快速随机访问
deque	双端队列
list	双向链表
forward_list	单向链表
array	固定大小数组
string	与vector相似，但是专门保存字符

## 9.2 容器库概览

类型别名	
iterator	此容器类型的迭代器类型
const_iterator	可以读取元素，但是不能修改元素的迭代器类型
size_type	足够保存此种容器类型最大可能容器的大小，无符号整型
difference_type	足够保存两个迭代器之间的距离，有符号整型
value_type	元素类型
reference	元素的左值类型；与value_type&含义相同
const_reference	元素的const左值类型
构造函数	
C c;	默认构造函数，构造空容器

## 类型别名

`C c1(c2);`

构造c2的拷贝c1

`C c(b,e)`

将迭代器b和e指定的范围内的元素拷贝到c中

`C c{a,b,d,...}`

列表初始化

## 赋值与swap

`c1 = c2`

将c1中的元素替换为c2中的元素

`c1 = {a,b,c...}`

将c1中的元素替换为列表中的元素（不适用array）

`a.swap(b)`

交换a和b中的元素

`swap(a,b)`

等价

## 大小

`c.size()`

c中元素的数目

`c.max_size()`

c可保存的最大元素数目

`c.empty()`

c是否为空

## 添加/删除元素

`c.insert(args)`

将args中的元素拷贝进c

`c.emplace(inits)`

使用inits构造c中的一个元素

`c.erase(args)`

删除args指定的元素

`c.clear()`

删除c中所有的元素，返回void

## 关系运算符

## 类型别名

`==, !=` 所有容器都支持

`<, <=, >, >=` 关系运算符

### 获取迭代器

`c.begin(), c.end()` 返回指向c首元素和尾后元素的迭代器

`c.cbegin(), c.cend()` 返回const版本

### 反向迭代器的额外成员

`reverse_iterator` 按逆序寻址元素的迭代器

`const_reverse_iterator` const版本

`c.rbegin(), c.rend()` 返回指向c尾元素和首元素之前位置的迭代器

`c.crbegin(), c.crend()` const版本

## 9.2.1 迭代器

迭代器范围由一对迭代器表示，两个迭代器分别指向同一个容器中的元素或者是尾后位置

## 9.2.2 容器类型成员

### 9.2.3 `begin` 和 `end` 成员

以`begin()`为例：有四种

- `begin()`, 返回 `iterator`
- `rbegin()`, 返回 `reverse_iterator`
- `cbegin()`, 返回 `const_iterator`
- `crbegin()`, 返回 `const_reverse_iterator`

不以c开头的函数都是被重载过的，即

- 若v为非常量成员,v.begin()返回 iterator
- 若v为常量成员,v.begin()返回 const\_iterator

## 9.2.4 容器定义和初始化

### 将一个容器初始化为另一个容器的拷贝

- 直接拷贝整个容器
- 拷贝一个由迭代器对指定的元素范围

### 标准库array具有固定大小

标准库array的大小也是类型的一部分

使用array类型，我们必须同时指定元素类型和大小

array大小固定的特性影响了它所定义的构造函数的行为。与其他容器不同，一个默认构造的array是非空的：它包含了与其大小一样多的元素。这些元素都被默认初始化（就像内置数组）。如果对array进行列表初始化，那么列表元素数目必须小于等于array的大小，剩余的部分将进行值初始化，并且如果存储的类型是类类型，该类类型还必须有默认构造函数以便能够进行值初始化

## 9.2.5 赋值和swap

### swap

```
1 swap(c1,c2);
2 c1.swap(c2);
```

### assign

- 1 `seq.assign(b, e);` //将seq中的元素替换为迭代器b和e范围内的元素。b和e不能指向seq
- 2 `seq.assign(il);` //将seq中的元素替换为列表值
- 3 `seq.assign(n, t);` //将seq中的元素替换为n个t

## 9.2.6 容器大小操作

`size()`: 返回容器中元素的数目

`empty()`: `size`为0时返回true

`max_size()`: 返回一个大于或等于该类型容器所能容纳的最大元素数的值

## 9.2.7 关系运算符

## 9.3 顺序容器操作

### 9.3.1 向顺序容器添加元素

`forward_list`有自己专有版本的`insert` 和 `emplace` [9.3.4节](#)

`forward_list`不支持`push_back`和`emplace_back`

`vector`和`string`不支持`push_front`和`emplace_back`

操作	
<code>c.push_back()</code>	尾部添加一个值为t的元素
<code>c.emplace_back(args)</code>	尾部添加一个由args创建的元素
<code>c.push_front</code>	

## 操作

c.emplcae\_front()

c.insert(p,t)

插入元素的几种方式，都是在迭代器p之前  
插入

c.emplace(p,args)

c.insert(p,b,e)

c.insert(p,il)

c.insert(p,n,t)

## 9.3.2 访问元素

at和下标操作只适用于string、vector、deque和array

back不适用于forward\_list

### 操作

c.back()

返回c中尾元素的引用

c.front()

返回c中首元素的引用

c[n]

返回c中下标为n的元素的引用，n是一个无符号整数

c.at(n)

返回下标为n的元素的引用

访问成员函数返回的是引用，如果我们使用auto来保存这些函数的返回值，并且希望使用此变量来改变元素的值，必须将变量定义为引用类型。否则默认是拷贝

### 9.3.3 删 除 元 素

array无法删除元素

forward\_list 有特殊版本的erase [9.3.4节](#)

forward\_list 不支持pop\_back

vector和string不支持pop\_front

操作	
c.pop_back()	删除c中尾元素
c.pop_front()	删除c中首元素
c.erase(p)	删除迭代器p指定的元素，返回一个指向被删除元素之后元素的迭代器
c.erase(b,e)	删除迭代器b和e范围内的元素，返回一个指向最后一个被删除元素之后元素的迭代器
c.clear()	删除c中所有元素

### 9.3.4 特 殊 的 forward\_list 操 作

操作	
lst.before_begin()	返回指向链表首元素之前不存在的元素的迭代器，此迭代器不能解引用
lst.cbefore_begin()	返回const版本迭代器
lst.insert_after(p,t)	在迭代器p之后的位置插入元素（其他容器是在p之前插入）
lst.insert_after(p,n,t)	

## 操作

`lst.insert_after(p,b,e)`

`lst.insert_after(p,il)`

`emplace_after(p,args)`

使用args参数创建元素，插入到p之后

`lst.erase_after(p)`

删除p指向的位置之后的元素，或删除从b之后直到e之间的元素。返回被删除元素之后元素的迭代器

`lst.erase_after(b,e)`

## 9.3.5 改变容器大小

可以使用`resize`来增大或缩小容器。如果当前大小大于所要求的大小，容器后部的元素会被删除；如果当前大小小于新大小，会将新元素添加到容器后部

### 操作

`c.resize(n)`

调整c的大小为n个元素

`c.resize(n,t)`

调整c的大小为n个元素，任何新添加的元素都初始化为t

## 9.3.6 容器操作可能使迭代器失效

向容器添加元素后：

- 如果容器时`vector`或`string`，且存储空间被重新分配，则指向容器的迭代器、指针和引用都会失效。如果存储空间未重新分配，指向插入位置之前元素的迭代器、指针和引用仍旧有效，但指向插入位置之后元素的迭代器、指针和引用将会失效

- 对于deque，插入到除首尾位置之外的任何位置都会导致迭代器、指针和引用失效。如果在首尾位置添加元素，迭代器会失效，但指向存在元素的引用和指针不会失效
- 对于list和forward\_list，指向容器的迭代器、指针和引用仍然有效

删除一个元素后：

- 对于list、forward\_list，指向容器其他位置的迭代器、引用和指针仍旧有效
- 对于deque，如果在首尾之外的任何位置删除元素，那么指向被删除元素外其他元素的迭代器、引用和指针也会失效。如果是删除deque的尾元素，则尾后迭代器失效，但其他迭代器、引用和指针不受影响；如果是删除首元素，这些也不受影响
- 对于vector和string，指向被删除元素之前元素的迭代器、引用和指针仍有效

## 9.4 vector对象是如何增长的

shrink\_to\_fit只适用于vector、string和deque

capacity和reserve只适用于vector和string

方法	
shrink_to_fit()	请将capacity()减少为与size()相同大小
c.capacity()	不重新分配内存的话，c可以保存多少元素
c.reserve(n)	分配至少能容纳n个元素的内存空间

reserve并不改变容器中元素的数量，它仅影响vector预先分配多大的内存空间

## 9.5 额外的string操作

### 9.5.1 构造string的其他方法

构造string	
string s(cp,n)	s是cp指向的数组中前n个字符的拷贝。此数组应该至少包含n个字符
string s(s2, pos2)	s是string s2从下标pos2开始的字符的拷贝
string s(s2, pos2, len2)	s是string s2从下标pos2开始的len2个字符的拷贝 (最多也就拷到末尾)

### substr操作

substr返回一个string，它是原始string的一部分或全部的拷贝。可以传递给substr一个可选的开始位置和计数值：

```
1 s.substr(pos, n);
```

### 9.5.2 改变string的其他方法

除了之前接受迭代器的insert和erase版本，string还提供了接受下标的版本

操作	
s.insert(pos,args)	在pos之前插入args指定的字符 (pos可以是下标或者迭代器，接受下标的版本返回指向s的引用，接受迭代器的版本返回指向第一个插入字符的迭代器)

**操作****s.erase(pos,len)**

删除从pos开始的len个字符，返回一个指向s的引用

**s.assign(args)**

将s中的字符替换为args指定的字符

**s.append(args)**

将args追加到s

**s.replace(range,args)**

删除s中范围range内的字符，替换为args指定的字符

args可以是下列形式之一；append和assign可以使用所有形式

1 str 字符串str

2 str, pos, len str中从pos开始的至多len个字符

3 cp, len cp是字符数组

4 n, c n个字符c

5 b, e b, e之间的范围

6 初始化列表

### 9.5.3 string搜索操作

每个搜索操作都返回**string::size\_type**值，表示匹配发生位置的下标

**find**函数完成最简单的搜索。查找指定的字符串，若找到返回第一个匹配位置的下标，否则返回**npos**（标准库将**npos**定义为**const string::size\_type**并初始化为-1），搜索是大小写敏感的

**rfind**函数则查找s中指定字符串最后一次出现的位置**查找操作****s.find(args)**

查找s中args第一次出现的位置

**s.rfind(args)**

查找s中args最后一次出现的位置

## 查找操作

`s.find_first_of(args)`

在s中查找args中任何一个字符第一次出现的位置

`s.find_last_of(args)`

在s中查找args中任何一个字符最后一次出现的额位置

`s.find_first_not_of(args)`

在s中查找第一个不在args中的字符

`s.find_last_not_of(args)`

在s中查找最后一个不在args中的字符

可以传递给find操作一个可选的开始位置，指定从什么位置开始搜索， 默认为0

## 9.5.4 compare函数

根据s是等于、大于、 小于指定字符串分别返回0、 整数或负数

## 9.5.5 数值转换

### 数值转换

`to_string(val)`

一组重载函数， 返回数值val的string表示

`stoi(s,p,b)`

返回s的起始子串（表示整数内容）的数值， 返回类型分别为int、 long、 unsigned long、 long long、 unsigned long long。 b表示转换所用的基数， 默认为10。 p是size\_t指针， 用来保存s中第一个非数值字符的下标， 默认为0， 即函数不保存下标

`stol(s,p,b)`

`stoul(s,p,b)`

`stoll(s,p,b)`

## 数值转换

stoull(s,p,b)

stof(s,p)

浮点类型 float、double、long double

stod(s,p)

stold(s,p)

## 9.6 容器适配器

标准库定义了三个顺序容器适配器：stack、queue和priority\_queue

```

1 //栈 默认基于deque实现，也可以在list或vector上实现
2 stack<int> intStack;
3 intStack.pop();
4 intStack.push(item);
5 intStack.emplace(args);
6 intStack.top();
7 //队列 queue默认基于deque实现，priority_queue默认基于vector实现
8 q.pop(); //返回queue首元素或priority_queue最高优先级元素
9 q.front();
10 q.back();
11 q.top();
12 q.push(item);
13 q.emplace(args);

```

# 第十章 泛型算法

## 10.1 概述

大多数算法都定义在 `algorithm` 中，标准库还在头文件 `numeric` 中定义了一组数值泛型算法

一般情况下，这些算法并不直接操作容器，而是遍历由两个迭代器指定的一个元素范围

### 算法永远不会执行容器的操作



泛型算法本身不会执行容器操作，它们只会运行于迭代器智商，执行迭代器的操作。泛型算法运行于迭代器之上而不会执行容器操作的特性带来了一个编程假定：算法永远不会改变底层容器的大小。算法可能改变容器中保存的元素，也可能在容器中移动元素，但永远不会添加和删除元素

## 10.2 初始泛型算法

### 10.2.1 只读算法

#### 1) `find` 算法

`find` 的前两个参数是表示范围的迭代器，第三个参数是一个值。`find` 将范围中的元素逐个与 `val` 比较，返回第一个等于给定值的元素的迭代器，如果查找失败，返回第二个元素

```
1 find(iterator, iterator, val);
```

## 2) accumulate

accumulate定义在头文件numeric中。前两个参数是需要求和的元素范围，第三个参数是和的初值

```
1 accumulate(iterator, iterator, val);
```

## 3) equal

用于确定两个序列是否保存相同的值

接受三个迭代器，前两个表示第一个序列中的元素范围，第三个表示第二个序列的首元素。它假定第二个序列至少与第一个序列一样长

## 10.2.2 写容器元素算法

算法不会执行容器操作，因此它们自身不可能改变容器大小

### 1) fill

fill接受一对迭代器表示的范围，还接受一个值。fill将此值赋予序列中的每个元素

```
1 fill(vec.begin(), vec.end(), 0);
```

### 2) copy

拷贝算法是另一个向目的位置迭代器指向的输出序列中的元素写入数据的算法

前两个迭代器表示一个输入范围，第三个表示目的序列的起始位置

### 3) replace

replace算法读入一个序列，并将其中所有等于给定值的元素都修改为另一个元素

接受四个参数：

- 前两个是迭代器，表示范围
- 要搜索的值
- 替换的值

```
1 replace(list.begin(), list.end(), 0, 42);
```

### 10.2.3 重排容器元素的算法

the quick red fox jumps over the slow red turtle

最终的目标是生成如下的vector

fox	jumps	over	quick	red	slow	the	turtle
-----	-------	------	-------	-----	------	-----	--------

#### 1) sort

sort算法接收两个迭代器表示排序元素的范围，结果应如下（相同的单词放在了一起）

fox	jumps	over	quick	red	red	slow	the	the	turtle
-----	-------	------	-------	-----	-----	------	-----	-----	--------

#### 1) stable\_sort

稳定排序，能够维持相等元素的原有顺序

#### 2) unique

unique算法重排输入序列，能够将相邻的重复项“消除”，并返回一个指向不重复值范围末尾的迭代器

fox	jumps	over	quick	red	slow	the	turtle	???	???
-----	-------	------	-------	-----	------	-----	--------	-----	-----

此时vector的大小并未改变，得到指向不重复值范围末尾的迭代器之后再调用erase方法逐个删除，即可达到效果

## 10.3 定制操作

默认情况下，比较输入序列元素的方法使用元素类型的<，=运算符完成比较。我们也可以定制额外的版本

### 10.3.1 向算法传递函数

#### 谓词

**一元谓词：**只接受单一参数

**二元谓词：**它们有两个参数

```

1 bool isShorter(const string &s1,const string
&s2){
2     return s1.size() < s2.size();
3 }
4 sort(words.begin(),words,end(),isShorter);

```

### 10.3.2 lambda表达式

传递给算法的谓词必须严格接受一个或两个参数。我们可以使用lambda表达式完成更复杂的功能

一个lambda表达式表示一个可调用的代码单元。可以将其理解为未命名的内联函数

一个lambda具有一个返回类型、一个参数列表和一个函数体

```

1 [capture list](parameter list) -> return type
{function body}

```

其中capture list 是一个lambda所在函数中定义的局部变量的列表。

**lambda必须使用尾置返回**，我们可以忽略参数列表和返回类型，但必须永远包含捕获列表和函数体

```
1 auto f = []{return 42;};
2 cout << f() << endl;
```

lambda不能有默认参数

### 10.3.3 lambda捕获和返回

当定义一个lambda时，编译器生成一个与lambda对应的新的类类型，将在14.8.1节介绍如何生成。

默认情况下，从lambda生成的类都包含一个对应该lambda所捕获变量的数据成员

#### 值捕获

被捕获的变量的值是在lambda创建时拷贝，而不是调用时拷贝

此时修改该变量不会影响原来的值

#### 引用捕获

#### 隐式捕获

除了显式列出我们需要的变量之外，还可以让编译器根据lambda体中的代码来推断我们要使用的变量。为了指示编译器推断捕获列表，应在捕获列表中写一个&或=。&告诉编译器捕获采用引用方式，=告诉编译器捕获采用值方式

当我们混合使用隐式捕获或者显式捕获时，捕获列表中的第一个元素必须是一个&或=，指定默认捕获方式是什么

## 可变lambda

默认情况下对于值被拷贝的变量，lambda不会改变其值。如果我们希望能够改变一个被捕获变量的值，就必须在参数列表首加上关键字

mutable

```

1 void fun3(){
2     size_t v1 = 42;
3     auto f = [v1]() mutable {return ++v1;};
4     v1 = 0;
5     auto j = f(); //j为43
6 }
```

一个引用捕获的变量是否能够修改取决于此引用指向的是const类型还是非const

## 指定lambda返回类型

默认情况下，如果一个lambda体包含return之外的任何语句，则编译器假定此lambda返回void

### 10.3.4 参数绑定

#### 标准库bind函数

定义在头文件functional中，可以将bind看做通用的函数适配器，它接受一个可调用对象，生成一个新的可调用对象来适应原对象的参数列表

```
1 auto newCallable = bind(callable,arg_list);
```

当我们调用newCallable时,newCallable会调用callable，并传递给它arg\_list中的参数

arg\_list中的参数可能包含形如\_n的名字，其中n是一个整数。这些参数是占位符，表示newCallable的参数，它们占据了传递给newCallable的参数的位置。\_1表示第一个参数，\_2表示第二个参数

```

1 bool check_size(const string &s,
2     string::size_type sz){
3     return s.size() >= sz;
4 }
5 //只有一个占位符，表示check6只接受单一参数。占位符出现在
6 //arg_list的第一个位置，表示check6的此参数对应
7 //check_size的第一个参数
8 string s = "hello";
9 bool b1 = check6(s); //check6会调用
10 check_size(s,6)

```

## bind的参数

```

1 auto g = bind(f,a,b,_2,c,_1);
2 //生成一个新的可调用对象，它有两个参数，分别用占位符_2、
3 //_1表示。
4 //即第一个g的参数将会绑定到f的最后一个参数上，第二个g的
5 //参数将会绑定到f的第三个参数上

```

## 绑定引用参数

- `ref()`
- `cref()`

```

1 for_each(words.begin(),words.end(),bind(print,
2     ref(os), _1, ' '));
3 //因为bind拷贝其参数，但是对于ostream的对象os来说无法
4 //拷贝，可以使用ref函数返回一个引用，这个返回的对象是可以拷
5 //贝的
6 //还有一个cref()可以返回const引用

```

## 10.4 再探迭代器

标准库在头文件iterator中还定义了额外几种迭代器

- 插入迭代器：这些迭代器被绑定到一个容器上，可以用来向容器插入元素
- 流迭代器：这些迭代器被绑定到输入或输出流上，可用来遍历所关联的IO流
- 反向迭代器：这些迭代器向后而不是向前移动，除了forward\_list之外的标准库容器都有反向迭代器
- 移动迭代器：这些专用的迭代器不是拷贝其中的元素，而是移动它们 [13.6.2节]

### 10.4.1 插入迭代器

- back\_inserter创建一个使用push\_back的迭代器
- front\_inserter创建一个使用push\_front的迭代器
- inserter创建一个使用insert的迭代器。此函数接受第二个参数，这个参数必须是一个指向给定容器的迭代器。元素将被插入到给定迭代器所表示的元素之前

插入迭代器的工作过程：

1. 当调用inserter(c,iter)时，我们得到一个迭代器
2. 使用它时，会将元素插入到iter原来所指向的元素之前的位置

如果it是由inserter生成的迭代器，则下面的赋值语句

```
1 *it = val;
```

效果与下述代码相同

```
1 it = c.insert(it, val);
2 ++it; //递增it使它指向原来的元素
```

## 10.4.2 iostream迭代器

- `istream_iterator`读取输入流
- `ostream_iterator`向输出流写入数据

### istream\_iterator操作

当创建一个流迭代器时，必须指定迭代器将要读写的对象类型。使用`>>`来读取流

- 当创建一个`istream_iterator`时，我们可以将它绑定到一个流。
- 也可以默认初始化，这样得到一个可以当做尾后值使用的迭代器

```

1 istream_iterator<int> in_iter(cin);
2 istream_iterator<int> eof; //尾后迭代器
3 while(in_iter != eof)
4     vec.push_back(*in_iter++);

```

### ostream\_iterator操作

操作	
<code>ostream_iterator&lt;T&gt;</code> <code>out(os);</code>	<code>out</code> 将类型为T的值写入到os中
<code>outstream_iterator&lt;T&gt;</code> <code>out(os,d);</code>	<code>out</code> 将类型为T的值写到输出流os中，每个值后面都输出一个d。d指向一个空字符结尾的字符数组
<code>out = val</code>	将val写入到out绑定的ostream中

## 10.4.3 反向迭代器

反向迭代器就是容器中从尾元素向首元素反向移动的迭代器

## 10.5 泛型算法结构

迭代器类型	
输入迭代器	只读、不写；单遍扫描，只能递增
输出迭代器	只写、不读；单遍扫描，只能递增
前向迭代器	可读写；多遍扫描，只能递增
双向迭代器	可读写；多遍扫描，可递增递减
随机访问迭代器	可读写；多遍扫描，支持全部迭代器运算

### 10.5.1 5类迭代器

C++标准指明了泛型和数值算法的每个迭代器参数的最小类别。例如，`find`算法在一个序列上进行一遍扫描，对元素进行只读操作，因此至少需要输入迭代器

1. 输入迭代器，可以读取序列中的元素。一个输入迭代器必须支持

- 用于比较两个迭代器的相等和不相等运算符 (`==`、`!=`)
- 用于推进迭代器的前置和后置递增运算 (`++`)
- 用于读取元素的解引用运算符 (`*`)：解引用只会出现在赋值运算符的右侧（不能修改？）
- 箭头运算符(`->`)，等价于 `(*it).member`，即，解引用迭代器，并提取对象的成员

输入迭代器只能用于单遍扫描算法，如`find`、`accumulate`；  
`istream_iterator`是一种输入迭代器

2. 输出迭代器：可以看做输入迭代器功能上的补集 --- 只写而不读元素

- 用于推进迭代器的前置和后置递增运算符
- 解引用运算符，只支出现在赋值运算符的左侧

我们只能像一个输出迭代器赋值一次。类似输入迭代器，输出迭代器只能用于单遍扫描算法。用作目的位置的迭代器通常都是输出迭代器，如copy算法的第三个参数就是输出迭代器。

`ostream_iterator`是一种输出迭代器

3. 前向迭代器，可以读写元素，这类迭代器只能在序列中沿一个方向移动。前向迭代器支持所有输入和输出迭代器的操作，而且可以多次读写同一个元素
4. 双向迭代器：可以正向/反向读写序列中的元素。除了支持所有前向迭代器操作之外，双向迭代器还支持前置和后置递减运算符
5. 随机访问迭代器：提供在常量时间内访问序列中任意元素的能力。此迭代器支持双向迭代器的所有功能，此外还支持
  - 用于比较两个迭代器相对位置的关系运算符(`<`,`<=`,`>`,`>=`)
  - 迭代器和一个整数值的加减运算
  - 用于两个迭代器上的减法运算
  - 下表运算符

## 10.5.2 算法形参模式

## 10.5.3 算法命名规范

## 10.6 特定容器算法

### list和forward\_list成员函数版本算法

操作	
<code>lst.merge(lst2)</code>	将来自 <code>lst2</code> 的元素合并入 <code>lst</code> 。 <code>lst</code> 和 <code>lst2</code> 都必须是有序的。元素将从 <code>lst2</code> 中删除。在合并之后 <code>lst2</code> 变为空。第一个版本使用 <code>&lt;</code> 运算符；第二个版本使用给定的比较运算符

**操作**`lst.merge(lst2,comp)``lst.remove(val)`

调用erase删除掉与给定值相等或令一元谓词为真的每个元素

`lst.remove_if(pred)``lst.reverse()`

反转lst中元素的顺序

`lst.sort()`

使用<或给定比较操作排序元素

`list.sort(comp)``lst.unique()`

调用erase删除同一个值的连续拷贝。

`lst.unique(pred)`

二元谓词版

**splice成员**`lst.splice(args)或flst.splice_after(args)`**splice成员**`(p,lst2)`

p是一个指向lst中元素的迭代器，或一个指向flst首位位置的迭代器。函数将lst2的所有元素移动到lst中p之前的位置或是flst中p之后的位置，将lst2元素删除

`(p,lst2,p2)`

p2是一个指向lst2中位置的有效的迭代器。将p2指向的元素移动到lst中，或将p2之后的元素移动到flst中。lst2可以是与lst或flst相同的链表

`(p,lst2,b,e)`

b和e表示lst2中的合法范围。将给定范围中的元素从lst2移动到lst或flst

**链表的成员函数版本算法与模板算法的区别是能够改变容器**

# 第十一章 关联容器

关联容器中的元素是按关键字来保存和访问的。

两个主要的关联容器是map和set

有序容器	
map	关联数组，保存关键字-值对
set	关键字即值
multimap	关键字可重复出现的map
multiset	关键字可重复出现的set

无序容器	
unordered_map	用哈希函数组织的map
unordered_set	用哈希函数组织的set
unordered_multimap	哈希组织的map；关键字可重复出现
unordered_multiset	哈希组织的set；关键字可重复出现

## 11.1 使用关联容器

## 11.2 关联容器概述

关联容器的迭代器都是双向的

## 11.2.1 定义关联容器

### 初始化map和set

定义map时，必须指明关键字类型和值类型

定义set时，必须指明关键字类型

默认初始化是一个指定类型的空容器，我们可以将关联容器初始化为另一个同类型容器的拷贝，或者是从一个值范围来初始化关联容器

### 初始化multimap或multiset

如果初始化的初值中具有相同的元素（对于multimap来说是相同的key），multimap和multiset与map和set的不同是map和set只会保留一份内容，但是multimap和multiset可以全部保留

## 11.2.2 关键字类型要求

对于无序容器中关键字的要求将在11.4节介绍

对于有序容器——map、multimap、set以及multiset，**关键字类型必须定义元素比较的方法**。默认情况下使用关键字的<运算符来比较两个关键字

### 有序容器的关键字类型

可以自己定义操作来代替关键字上的<运算符。所提供的操作必须在关键字类型上定义一个**严格弱序**（小于等于）。这个比较函数必须具备如下基本性质：

- 两个关键字不能同时“小于等于”对方；如果k1“小于等于”k2，那么k2决不能“小于等于”k1
- 如果k1小于等于k2，k2小于等于k3，那么k1必须小于等于k3
- 如果存在两个关键字，任何一个都不小于等于另一个，那么称这两个关键字是等价的

## 使用关键字类型的比较函数

用来组织一个容器中元素的操作的类型也是该容器类型的一部分。为了指定使用自定义的操作，必须在定义关联容器类型时提供此操作的类型。

在尖括号中出现的每个类型，就仅仅是个类型而已。当我们创建一个容器时，才会以构造函数的形式提供真正的比较操作

```

1 bool compareIsbn(const Sales_data &lhs, const
2   Sales_data &rhs){
3     return lhs.isbn() < rhs.isbn();
4   }
5 multiset<Sales_data, decltype(compareIsbn)*>
6   bookstore(compareIsbn);

```

### 11.2.3 pair类型

定义在utility中，first是第一个元素，second是第二个元素

pair的数据成员是public的

map的元素是pair

make\_pair(first,second)能够返回一个pair

## 11.3 关联容器操作

操作	
key_type	此容器类型的关键字类型
mapped_type	每个关键字关联的类型；只适用于map

**操作****value\_type**

对于set，与key\_type相同  
 对于map，为pair<const  
 key\_type,mapped\_type>

### 11.3.1 关联容器迭代器

当解引用一个关联容器迭代器时，我们会得到一个类型为容器的value\_type的值的引用。对map而言，value\_type是pair类型，关键字是const的。对于set来说iterator和const\_iterator都不能改变set中的关键字（都是const的）

### 11.3.2 添加元素

关联容器的insert成员向容器添加一个元素或一个元素范围。插入一个已存在的元素（对关键字而言）对容器没有任何影响。

insert有两个版本，分别接受一对迭代器，或是一个初始化器列表。

#### 检测insert的返回值

insert返回的值依赖于容器类型和参数，对于不包含重复关键字的容器，添加单一元素的insert和emplace版本返回一个pair，告诉我们插入是否成功。pair的first是一个迭代器，指向具有给定关键字的元素，second是一个bool值，指出元素是插入成功还是已经存在在容器中（false元素已经在容器中，true插入成功）

#### 向multiset或multimap添加元素

接受单个元素的insert操作返回一个指向新元素的迭代器（无须返回bool值，因为总会成功）

## 11.3.3 删除元素

关联容器定义了三个版本的erase

- erase一个迭代器
- erase一个迭代器范围
- 接受一个key\_type参数，此版本删除所有匹配给定关键字的元素，返回实际删除的元素的数量。

## 11.3.4 map的下标操作

map下标运算接受一个索引，获取与此关键字相关联的值，如果关键字并不在map中，会为他创建一个元素并插入到map中，关联值将进行值初始化

```
1 map<string, size_t> word_count;
2 word_count["Anna"] = 1;
```

将会执行如下操作：

- 在word\_count中搜索关键字为Anna的元素
- 将一个新的关键字值对插入到word\_count中。关键字是一个const string，保存Anna。值进行值初始化，在上述代码中为0
- 提取出新插入的元素，并将值1赋予它

只能对非const的map执行下标操作

## 11.3.5 访问元素

### 操作

lower\_bound和  
upper\_bound不适用于  
无序容器

## 操作

下标和at操作只适用于  
非const的map和  
unordered\_map

c.find(k)	返回一个迭代器，指向第一个关键字为k的元素，若未找到，返回尾后迭代器
c.count(k)	返回关键字等于k的元素的数量
c.lower_bound(k)	返回一个迭代器，指向第一个关键字不小于k的元素
c.upper_bound(k)	返回一个迭代器，指向第一个关键字大于k的元素
c.equal_range(k)	返回一个迭代器pair，表示关键字等于k的元素的范围，若不存在，pair的两个成员均等于c.end()

## 11.4 无序容器

4个无序关联容器，这些容器不是使用比较运算符组织元素的，而是使用一个哈希函数和关键字类型的==运算符

### 管理桶

无序容器在存储上组织为一组桶，每个桶保存零个或多个元素。无序容器使用一个哈希函数将元素映射到桶。

为了访问一个元素，容器首先计算元素的哈希值，它指出应该搜索哪个桶。容器将一个具有特定哈希值的所有元素都保存在相同的桶中。如果容器允许重复关键字，所有具有相同关键字的元素也都会出现在同一个桶中。无序容器的性能依赖于哈希函数的质量和桶的数量和大小

## 桶接口

c.bucket_count()	正在使用的桶的数目
c.max_bucket_count()	容器能容纳的最多的桶的数量
c.bucket_size(n)	第n个桶中有多少个元素
c.bucket(k)	关键字为k的元素在那个桶中
桶迭代	
local_iterator	可以用来访问桶中元素的迭代器类型
const_local_iterator	桶迭代器的const版本
c.begin(n),c.end(n)	桶n的首元素迭代器和尾后迭代器
c.cbegin(n),c.cend(n)	与前两个函数类似，返回的是 cosnt_local_iterator
哈希策略	
c.load_factor()	每个桶的平均元素数量，返回float值
c.max_load_factor()	c试图维护的平均桶大小，返回float值。c 会在需要时添加新的桶，以使得 load_factor <= max_local_factor
c.rehash(n)	重组存储，使得bucket_count >= n 且 bucket_count > size/max_load_factor
c.reserve(n)	重组存储，使得c可以保存n个元素且不必 rehash



## 无序容器对关键字类型的要求

默认情况下，无序容器使用关键字类型`==`运算符来比较元素，使用一个`hash<key_type>`类型的对象来生成每个元素的hash值。内置类型、`string`、智能指针已经提供了hash模板

- 在16.5节介绍如何自定义hash模板
- 使用另一种方式，类似于为有序容器重载关键字类型的默认比较操作，我们提供函数替代`==`和哈希值计算

```

1 size_t hasher(const CSales_data &sd){
2     return hash<string>()(sd.isbn());
3 }
4 bool eqOp(const Sales_data &lhs, const
5 Sales_data &rhs){
6     return lhs.isbn() == rhs.isbn();
7 }
8 using SD_multiset =
9 unordered_multiset<Sales_data,
10 decltype(hasher)*, decltype(eqOp)*>;
11 SD_multiset bookstore(42,hasher,eqOp);

```

如果我们的类定义了`==`运算符，也可以只重载哈希函数

## 第十二章 动态内存

除了自动和`static`对象外，C++还支持动态分配对象。动态分配的对象的生存期与它们在哪里创建是无关的，只有当显式地被释放时，这些对象才会销毁

为了更安全地使用动态对象，标准库定义了两个只能指针类型来管理动态分配的对象。当一个对象应该被释放时，指向它的只能指针可以确保自动地释放它

- 静态内存：保存局部static对象，类static数据成员以及定义在任何函数之外的变量
- 栈内存：用来保存定义在函数内的非static对象

分配在静态内存和栈内存的对象由编译器创建和销毁。栈对象在程序块运行时创建，程序块结束时销毁；static对象在使用之前分配，在程序结束时销毁。

除了静态内存和栈内存，每个程序还拥有一个内存池。这部分被称作自由空间或堆。程序用堆来存储动态分配的对象。

## 12.1 动态内存与智能指针

- new 在动态内存中为对象分配空间并返回一个指向该对象的指针
- delete 接受一个动态对象的指针，销毁该对象，并释放与之关联的内存

智能指针的行为类似常规指针，区别是它负责自动释放所指向的对象。定义在memory头文件

- shared\_ptr 允许多个指针指向同一个对象
- unique\_ptr “独占” 所指向的对象
- weak\_ptr 是一种弱引用，指向shared\_ptr 所管理的对象

### 12.1.1 shared\_ptr类

```
1 shared_ptr<string> p1;
2 shared_ptr<list<int>> p2;
```

默认初始化的智能指针中保存着一个空指针

解引用一个智能指针返回它指向的对象

## shared\_ptr和unique\_ptr都支持的操作

操作	
shared_ptr<T> sp	空智能指针，可以指向类型为T的对象
unique_ptr<T> up	
p	将p作为条件判断，若p指向一个对象则为true
*p	对p解引用，获取指向的对象
p->mem	
p.get()	返回p中保存的指针（内置指针）。若智能指针释放了其对象，则返回的指针所指对象也消失了
swap(p,q) p.swap(q)	交换p和q中的指针

## shared\_ptr独有的操作

shared_ptr独有操作	
make_shared<T> (args)	返回一个shared_ptr，指向一个动态分配的类型为T的对象。使用args初始化此对象（调用构造器）
shared_ptr<T>p(q)	p是shared_ptr q 的拷贝；此操作会递增q中的计数器。q中的指针必须能转换为T*
p = q	p 和 q都是shared_ptr，所保存的指针必须能相互转换。此操作会递减p的引用计数，递增q的引用计数；若p的引用计数变为0，则将其管理的原内存释放

## shared\_ptr独有操作

p.unique()

若p.use\_count()为1，返回true；否则返回false

p.use\_count()

返回与p共享对象的智能指针数量；可能很慢，主要用于调试

## make\_shared函数

此函数在动态内存中分配一个对象并初始化它，返回指向此对象的shared\_ptr。定义在memory头文件中

```
1 shared_ptr<int> p3 = make_shared<int>(42);
2 shared_ptr<string> p4 = make_shared<string>
  ('10', '9');
```

使用括号内的参数构造对象（调用对应的构造器）。如果不传递参数，则进行值初始化。

通常使用auto配合make\_shared

## shared\_ptr的拷贝和赋值

每个shared\_ptr都会记录有多少个其他shared\_ptr指向相同的对象，通常称其为引用计数。

一旦一个shared\_ptr的计数器变为0，它就会自动释放自己所管理的对象

## shared\_ptr自动销毁所管理的对象

通过析构函数完成销毁工作的

`shared_ptr`的析构函数会递减它所指向的对象的引用计数。如果引用计数变为0, `shared_ptr`的析构函数就会销毁对象，并释放它所占用的内存

## 使用了动态生存期的资源的类

程序使用动态内存出于以下三种原因之一：

1. 程序不知道自己需要使用多少对象
2. 程序不知道所需对象的准确类型
3. 程序需要在多个对象间共享数据

### 12.1.2 直接管理内存

#### 使用new动态分配和初始化对象

在自由空间分配的内存是无名的，因此new无法为其分配的对象命名，而是返回一个指向该对象的指针

```
1 int *pi = new int;
```

动态分配的对象是默认初始化的，即内置类型或组合类型的对象的值将是未定义的，而类类型对象将用默认构造函数进行初始化

使用直接初始化方式初始化一个动态分配的对象，可以使用传统的构造方式，也可以使用列表初始化

也可以对动态分配的内存使用值初始化，只需要在类型名后面跟一个空括号即可，我们当然建议值初始化，因为这样对象才有良好的定义

```
1 string *ps1 = new string; //默认初始化为空string
2 string *ps = new string(); //值初始化为空string
```

如果我们提供了一个括号包围的初始化器，就可以使用auto从此初始化器来推断我们想要分配的对象的类型，只有当括号中仅有单一初始化器时才可以使用auto

```
1 auto p1 = new auto(obj); //p指向一个与obj同类型的对象，该对象用obj进行初始化
```

## 动态分配的const对象

用new分配const对象是合法的

```
1 const int *pci = new const int(1024);
```

一个动态分配的const对象必须进行初始化。对于一个定义了默认构造函数的类类型，其const动态对象可以隐式初始化，而其他类型的对象就必须显式初始化

## 内存耗尽

一旦一个程序用完了所有可使用的内存，new表达式就会失败。

默认情况下，如果new不能分配所要求的内存空间，它会抛出一个bad\_alloc的异常。可以通过改变使用new的方式组织其抛出异常。

```
1 int *p1 = new int; //分配失败时，抛出
                     std::bad_alloc异常
2 int *p2 = new (nothrow) int; //分配失败时，new返回
                     一个空指针
```

称这种形式的new为定位new，在19.1.2节解释。

定位new表达式允许我们向new传递额外的参数

## 释放内存delete

delete表达式接受一个指针，指向我们想要释放的对象

delete表达式执行两个动作：

- 销毁给定的指针指向的对象

- 释放对应的内存

## 指针值和delete

传递给delete的指针必须指向动态分配的内存，或者是一个空指针。释放并非new分配的内存，或者将相同的指针值释放多次，是未定义的。

编译器不能分辨一个指针指向的是静态还是动态分配的对象。编译器也不能分辨一个指针指向的内存是否已经被释放了。这些行为能够编译通过，但是是错误的

## 动态对象的生命周期直到被释放

由内置指针（不是智能指针）管理的动态内存被显示释放前一直都会存在

### 12.1.3 shared\_ptr和new结合使用

我们可以用new返回的指针来初始化智能指针

```
1 shared_ptr<int> p2(new int(42));
```

接受指针参数的智能指针构造函数是[explicit](#)的，所以我们不能将一个内置指针隐式转换为一个智能指针，必须使用直接初始化形式来初始化一个智能指针

```
1 shared_ptr<int> p1 = new int(1024); //错误，必须使用直接初始化形式
2 shared_ptr<int> p2(new int(1024)); //正确
```

默认情况下，一个用来初始化智能指针的普通指针必须指向动态内存，因为智能指针默认使用delete释放它所关联的对象。可以将智能指针绑定到一个指向其他类型的资源的指针上（非动态内存的指针），我们需要提供自己的操作代替delete（12.1.4节）

表12.3 定义和改变shared\_ptr的其他方法

## 定义和改变 shared\_ptr的其 他方法

`shared_ptr<T> p(q)` p管理内置指针q指向的对象；q必须指向new分配的内存，且能够转换为T\*

`shared_ptr<T> p(u)` p从unique\_ptr u那里接管了对象的所有权；将u置为空

`shared_ptr<T> p(q,d)` p接管了内置指针q，q需要能转换为T\*。p使用可调用对象d代替delete

`shared_ptr<T> p(p2,d)` p是shared\_ptr p2的拷贝（？计数是独立的还是？），唯一的区别是p将可调用对象d代替delete

`p.reset()` 若p是唯一指向其对象的shared\_ptr,reset会释放此对象。

`p.reset(q)` 若传递了可选的参数内置指针q，p将会指向q，否则p置为空（q应该会释放掉）

`p.reset(q,d)` 若还传递了可调用对象d，将会调用d代替delete来释放q

## 不要混合使用普通指针和智能指针

shared\_ptr可以协调对象的析构，但是仅限于其自身的拷贝（也是shared\_ptr）之间

```

1 void process(shared_ptr<int> ptr){
2
3 } // ptr离开作用域，被销毁
4 shared_ptr<int> p(new int(42)); //引用计数为1
5 process(p); //引用计数为2
6 int i = *p; //正确，引用计数为1
7 //虽然无法传递给process一个内置指针（因为构造函数是
8 //explicit的），但是可以传递给它一个临时的使用内置指针初
9 //始化的shared_ptr
10
11 int *x(new int(1024));
12 process(x); //错误
13 process(shared_ptr<int>(x)); //合法的，但是内存会被
14 //释放 process执行结束退出的时候引用计数-1，这条调用结
15 //束的时候引用计数又-1，此时引用计数为0，内存会被释放，但
16 //是x仍旧指向这块内存，称为悬空指针
17
18 int j = *x; //x此时是一个悬空指针

```

我们将shared\_ptr绑定普通指针后，就将内存的管理责任交给了shared\_ptr,之后我们就不应该再使用内置指针来访问shraed\_ptr指向的内存了！

## 不要使用get初始化另一个智能指针或为智能指针赋值

get()返回一个内置指针，是为了向无法使用智能指针的代码传递一个内置指针。使用get返回的指针的代码不能delete此指针

将另一个智能指针绑定到get返回的指针上是错误的行为

```

1 shared_ptr<int> p(new int(42));
2 int *q = p.get();
3 {
4     shared_ptr<int>(q);
5 } //q被释放
6 int foo = *p; //错误，p指向的内存被释放了

```

p和q指向了相同的内存，但是是独立的所以引用计数都是1，在退出程序块时，内存已经被释放

## 12.1.4 智能指针和异常

### 12.1.5 unique\_ptr

#### unique\_ptr操作

某个时刻只能有一个unique\_ptr指向一个给定对象。

没有类似make\_shared的函数，定义unique\_ptr时，需要将其绑定到一个new返回的指针上。初始化unique\_ptr也必须直接初始化

unique_ptr操作	
unique_ptr<T> u1	空unique_ptr，可以指向类型为T的对象。u1会使用delete释放指针；u2使用可调用对象D来释放它的指针
unique_ptr<T,D> u2	
unique_ptr<T,D> u(d)	空unique_ptr，指向类型为T的对象，用类型为D的对象d代替delete
u = nullptr	
u.release()	u放弃对指针的控制权，返回指针，并将u置空
u.reset() u.reset(q) u.reset(nullptr)	释放u指向的对象，如果提供了内置指针q，将u指向q，否则将u置空

```
1 unique_str<string> p2(p1.release()); //指针的控制权
发生了转移
```

## 传递unique\_str参数和返回unique\_str

不能拷贝unique\_str的规则有一个例外：我们可以拷贝或赋值一个即  
将销毁的unique\_str。

```
1 unique_str<int> clone(int p){
2     return unique_str<int>(new int(p));
3 }
```

这种特殊的拷贝将在13.6.2节介绍

## 向unique\_str传递删除器

unique\_str管理删除器的方式与shared\_ptr不同（在16.1.6节介绍）

```
1 unique_str<objT,delT> p (new objT,fcn);
```

这种方式与重载关联容器的比较操作类似

## 12.1.6 weak\_ptr

weak\_ptr是一种不控制所指对象生存期的智能指针，它指向一个由shared\_ptr管理的对象。将一个weak\_ptr绑定到shared\_ptr不会改变shared\_ptr的引用计数。最后一个指向对象的shared\_ptr被销毁，对象就会被释放（即使weak\_ptr指向对象）

### weak\_ptr操作

## weak\_ptr操作

```
weak_ptr<T>
  w
weak_ptr<T>
  w(sp)
```

空weak\_ptr可以指向类型为T的对象  
与shared\_ptr sp指向相同对象的weak\_ptr。T必须  
能转换为sp指向的类型（是不是说反了？）

w = p

p可以是一个shared\_ptr或一个weak\_ptr。赋值后  
w与p共享对象

w.reset()

将w置为空

w.use\_count()

与w共享对象的shared\_ptr的数量

w.expired()

若w.use\_count()为0，返回true，否则返回false

w.lock()

如果expired为true，返回一个空shared\_ptr；否则  
返回一个指向w对象的shared\_ptr

由于对象可能不存在，我们不能使用weak\_ptr直接访问对象，而必  
须调用lock

## 12.2 动态数组

标准库提供了两种一次分配一个对象数组的方法。C++语言定义了另  
一种new表达式语法，可以分配并初始化一个对象数组。标准库中包含一  
个名为allocator的类，允许我们将分配和初始化分离。使用allocator通  
常会提供更好的性能和更灵活的内存管理能力，原因见12.2.2节

使用容器的类可以使用默认版本的拷贝、赋值和析构操作。分配动态  
数组的类必须定义自己版本的操作，在拷贝、复制以及销毁对象时管理关  
联的内存

## 12.2.1 new和数组

在类型名之后跟一对方括号，在其中指明要分配的对象的数目

```
1 int *pia = new int[get_size()];
```

由于返回的是一个元素类型的指针，不是一个数组类型（要注意二者的区别，数组类型与数组元素类型的指针是有区别的 $\star$ ），因此不能对动态数组调用begin或end。这些函数使用数组维度（维度也是数组类型的一部分）来返回首元素和尾后元素指针。也不能使用范围for语句

动态数组并不是数组类型 $\star$

## 初始化动态分配对象的数组

默认情况下，new分配的对象，都是默认初始化的。可以对数组中的元素进行值初始化

```
1 int *pia = new int[10];
2 int *pia2 = new int[10](); //值初始化
3 string *psa = new string[10];
4 string *psa2 = new string[10]();
```

还可以提供一个元素初始化器的花括号列表。如果初始化器数目大于元素数目，new表达式失败，不会分配任何内存，抛出  
bad\_array\_new\_length 异常

## 动态分配一个空数组是合法的

```
1 char arr[0];//错误，不能定义长度为0的数组
2 char *cp = new char[0]; //正确，但是cp不能解引用
```

## 释放动态数组

使用`delete[]`释放动态数组

```
1 delete p; //p必须指向一个动态分配的对象或为空
2 delete [] pa; //pa必须指向一个动态分配的数组或为空
```

数组中的元素按逆序销毁

## 智能指针和动态数组

标准库提供了可以管理`new`分配的数组的`unique_ptr`版本

```
1 unique_ptr<int[]> up(new int[10]);
2 up.release(); //自动用delete[]销毁其指针
```

当`unique_ptr`指向一个数组时无法使用点和箭头运算符，但是可以使用下标运算符来访问数组中的元素。

如果想使用`shared_ptr`管理一个动态数组，必须提供自己定义的删除器

```
1 shared_ptr<int> sp(new int[10], [](int *p)
  {delete[] p;});
2 sp.reset(); //使用lambda释放数组
```

`shared_ptr`未定义下标运算符，而且智能指针不支持指针算术运算。因此只能使用`get`获取一个内置指针，然后用它访问元素

## 12.2.2 allocator类

`new`灵活上的局限之一是它将内存分配和对象构造组合在了一起。类似的`delete`将对象析构和内存释放组合到了一起

对于大块内存的分配，我们可能希望将内存分配和对象构造分离，这可以让我们在真正需要时才执行对象创建操作

其他的不重要，但是使用new对于没有默认构造函数的类就不能动态分配数组了。✿

## allocator类

定义在头文件memory中

提供一种类型感知的内存分配方法，它分配的内存是原始的、未构造的

```
1 allocator<string> alloc;
2 auto const p = alloc.allocate(n); //分配n个未初始化的string
```

### 标准库allocator类及其算法

#### 标准库allocator类及其算法

allocator<T> a

定义了一个名为a的allocator对象，它可以为类型为T的对象分配内存

a.allocate(n)

分配一段原始的、未构造的内存，保存n个类型为T的元素

a.deallocate(p,n)

释放从T\*指针p中地址开始的内存，这块内存保存了n个类型为T的对象；p必须是一个先前由allocate返回的指针，且n必须是p创建时要求的大小。在调用deallocate之前，用户必须对每个在这块内存中创建的对象调用destroy

a.construct(p,args)

p必须是一个类型为T\*的指针，指向一块原始内存；arg被传递给类型为T的构造函数，用来在p指向的内存中构造一个对象

a.destroy(p)

p为T\*类型的指针，此算法对p指向的对象执行析构函数

## allocator分配未构造的内存

还未构造对象的情况下使用原始内存是错误的，我们必须用construct构造对象。

当我们用完对象后，必须对每个构造的元素调用destroy来销毁它们。只能对真正构造了的元素使用destroy。

元素被销毁后，可以重新使用这部分内存保存其他对象，也可以使用deallocate归还给系统

## 拷贝和填充未初始化内存的算法

### 拷贝和填充未初始化内存的算法

uninitialized\_copy(b,e,b2)

从迭代器b和e指出的输入范围内拷贝元素到迭代器b2指定的未构造的原始内存中。b2指向的内存必须足够大能够容纳这么多元素，返回递增后的目的迭代器

uninitialized\_copy\_n(b,n,b2)

从迭代器b指向的元素开始，拷贝n个元素到b2开始的内存中

uninitialized\_fill(b,e,t)

在迭代器b和e指向的原始内存中创建对象，对象的值均为t的拷贝

uninitialized\_fill\_n(b,n,t)

从迭代器b指向的内存地址开始创建n个对象。b必须指向足够大的未构造的原始内存，能够容纳给定数量的对象

## 12.3 使用标准库：文本查询程序

```
1 #include <fstream>
2 #include <iostream>
3 #include <map>
4 #include <memory>
5 #include <set>
6 #include <sstream>
7 #include <vector>
8 class QueryResult;
9 class TextQuery {
10 public:
11     using line_no =
12         std::vector<std::string>::size_type;
13     TextQuery(std::ifstream&); // 读取输入文件
14     QueryResult query(const std::string&)
15         const; // 常量成员函数
16
17 private:
18     std::shared_ptr<std::vector<std::string>>
19         file; // 输入文件
20     std::map<std::string,
21         std::shared_ptr<std::set<line_no>>> wm;
22 };
23 // 读取输入文件并建立单词到行号的映射
24 TextQuery::TextQuery(std::ifstream& is)
25     : file(new std::vector<std::string>)
26 {
27     std::string text;
28     while (getline(is, text)) { // 读取文件中每一行
29         file->push_back(text);
30         int n = file->size() - 1; // 当前的行号
31         std::istringstream line(text); // 将当前行文本给到string流
32         std::string word;
```

```

29         while (line >> word) {
30             auto& lines = wm[word]; // 通过key获
取值
31             if (!lines) {
32                 lines.reset(new
33                 std::set<line_no>);
34             }
35             lines->insert(n); // 将行号插入set中
36         }
37     }
38 // 这个const限定的是this指针
39 QueryResult TextQuery::query(const std::string&
sought) const
40 {
41     // 未找到一个单词对应的行号的时候，返回一个指向空
set的智能指针
42     static std::shared_ptr<std::set<line_no>>
nodata(new std::set<line_no>);
43     auto loc = wm.find(sought);
44     if (loc == wm.end()) {
45         return QueryResult(sought, nodata,
file); // 未找到
46     } else {
47         return QueryResult(sought, loc->second,
file);
48     }
49 }
50
51 class QueryResult {
52     friend std::ostream& print(std::ostream&,
53     const QueryResult&);
54 public:
55     using line_no =
std::vector<std::string>::size_type;

```

```

56     QueryResult(std::string s,
57     std::shared_ptr<std::set<line_no>> p,
58     std::shared_ptr<std::vector<std::string>> f)
59     : sought(s)
60     , lines(p)
61     , file(f)
62     {
63     }
64
65     std::string sought;
66     std::shared_ptr<std::set<line_no>> lines;
67     std::shared_ptr<std::vector<std::string>>
68     file;
69 }
70 std::ostream& print(std::ostream& os, const
71 QueryResult& qr)
72 {
73     os << qr.sought << " occurs " << qr.lines-
74     >size() << " " << make_plural(qr.lines->size(),
75     "time", "s") << std::endl;
76     for (auto num : *qr.lines)
77         os << "\t(line " << num + 1 << ") " <<
78     *(qr.file->begin() + num) << std::endl;
79     return os;
80 }
81 std::string make_plural(int i, const
82 std::string& s1, const std::string& s2)
83 {
84     return i > 1 ? s1 : s1 + s2;
85 }

```

# 第 III 部分 类设计者的工具

## 第十三章 拷贝控制

当定义一个类时，我们能够显式或隐式的指定在此类型的对象拷贝、移动、赋值和销毁时做什么。通过定义**拷贝构造函数**、**拷贝赋值运算符**、**移动构造函数**、**移动赋值运算符**和**析构函数**控制这些操作

- 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。（初始化与赋值不一样）
- 析构函数定义了对象销毁时做什么

上述操作称为**拷贝控制操作**，如果未定义这些操作，编译器会自动定义缺失的操作

### 13.1 拷贝、赋值与销毁

#### 13.1.1 拷贝构造函数

一个构造函数的第一个参数是自身类类型的引用，且任何额外的参数都有默认值，则此构造函数是**拷贝构造函数**。拷贝构造函数的第一个参数必须是一个引用类型（一般是const的），在几种情况下会被隐式的使用，所以通常不是[explicit](#)的。

#### 合成拷贝构造函数

即使我们定义了其他的构造函数，编译器也会合成一个拷贝构造函数（这点与默认构造函数不同）

合成拷贝构造函数对于某些类用来阻止我们拷贝该类类型的对象（见13.1.6节）。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。编译器从给定对象中将非static成员拷贝到正在创建的对象中。

成员的类型决定了如何被拷贝：

- 对类类型成员，使用拷贝构造函数来拷贝
- 对内置类型直接拷贝
- 对数组类型，合成拷贝构造函数会逐一拷贝数组类型的成员（我们无法直接拷贝一个数组）；如果数组元素是类类型，使用元素的拷贝构造函数

## 拷贝初始化

直接初始化要求编译器使用普通的函数匹配来选择与我们提供的参数最匹配的构造函数

拷贝初始化要求编译器将右侧运算对象拷贝到正在创建的对象中，如果需要的话还要进行类型转换

拷贝初始化通常使用拷贝构造函数来完成（有的情况下使用移动构造函数[13.6.2节]）

拷贝初始化在下列情况下会发生：

- 用`=`定义变量时
- 从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](#)中的成员

## 拷贝初始化参数为什么必须是引用类型？

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。如果其参数不是引用类型，则调用永远不会成功

## 编译器可以绕过拷贝构造函数

在拷贝初始化过程中，编译器可以（但不是必须）跳过拷贝/移动构造函数，直接创建对象

```
1 string null_book = "9-999-99999-9"; //拷贝初始化
2 //改写为
3 string null_book("9-999-99999-9"); //编译器略过了拷
贝构造函数
```

但是，即使编译器略过了拷贝/移动构造函数，但在这个程序点上，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

类可以控制其对象如何赋值

```
1 Sales_data trans, accum;
2 trans = accum; //使用Sales_data的拷贝赋值运算符
```

与拷贝构造函数一样，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

详细内容见第十四章

重载运算符本质上是函数，其名字由 `operator` 关键字后接表示要定义的运算符的符号组成

重载运算符的参数表示运算符的运算对象。某些运算符，包括赋值运算符，必须定义为成员函数。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。对于二元运算符，右侧运算对象作为显示参数传递

拷贝赋值运算符接受一个与其所在类类型相同的参数，赋值运算符通常应该返回一个指向其左侧运算对象的引用

## 合成拷贝赋值运算符

对于某些类，合成拷贝赋值运算符用来禁止该类型对象的赋值

[13.1.6]

如果拷贝赋值运算符不是上述目的，他会将右侧运算对象的每个非 static 成员赋予左侧运算对象的对应成员。合成拷贝赋值运算符返回一个指向其左侧运算对象的引用。

### 13.1.3 析构函数

析构函数执行与构造函数相反的操作：构造函数初始化对象的非 static 数据成员，外加一些其他工作；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和参数，所以析构函数只有一个无法重载

构造函数中，成员的初始化是在函数体执行之前完成的，按照在类中出现的顺序进行初始化

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



1. 析构函数释放对象在生存期分配的所有资源
2. 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。
3. 销毁类类型的成员需要执行成员自己的析构函数。
4. 内置类型没有析构函数所以什么也不做
5. 销毁一个内置指针类型的成员不会 delete 它所指向的对象
6. 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自动销毁

# 什么时候会调用析构函数

- 变量在离开其作用域时
- 当一个对象被销毁时，其成员被销毁
- 容器（标准库和数组）被销毁时，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，创建它的完整表达式结束时被销毁

```

1 { //新作用域
2     Sales_data *p = new Sales_data;
3     auto p2 = make_shared<Sales_data>(); //p2指向一个Sales_data类型的对象，make_shared会返回一个指向动态分配对象的shared指针
4     Sales_data item(*p);
5     vector<Sales_data> vec;
6     vec.push_back(*p2);
7     delete p;
8 } //退出局部作用域；对item、vec、p2调用析构函数
9 //销毁p2会递减其引用计数；如果引用计数为0，对象被释放
10 //销毁vec会销毁它的元素，它的元素是*p2的拷贝

```

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，那就需要我们`delete`释放

## 合成析构函数

对于某些类，合成析构函数用来阻止对象被销毁[13.1.6]。如果不是这种情况，合成析构函数的函数体为空

析构函数体自身并不直接销毁成员，成员是在析构函数体之后隐含的析构阶段中被销毁的。

## 13.1.4 三/五法则

拷贝构造函数、拷贝赋值运算符和析构函数外加移动构造函数、移动赋值运算符

C++并不要求我们定义所有的这些操作，但是这些操作通常应该被看做一个整体。只需要其中一个操作，而不需要定义所有操作的情况是很少见的

### 需要析构函数的类也需要拷贝和赋值操作

当我们决定一个类是否需要定义自己版本的拷贝控制成员时，一个基本原则是首先确定这个类是否需要析构函数

### 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

我们可以将拷贝控制成员定义为=default来显式地要求编译器生成合成的版本

```

1 class Sales_data{
2     public:
3         Sales_data() =default; //合成构造函数
4         Sales_data(const Sales_data&) = default;
5         //合成拷贝构造函数
6         Sales_data& operator=(const Sales_data
7         &) = default; //合成拷贝赋值运算符
8         ~Sales_data() = default; //合成析构函数
9 }
```

当我们在类内声明=default时，合成的函数会隐式的声明成内联。如果不想合成的成员是内联的，应该只对成员的类外定义使用=default

## 13.1.6 阻止拷贝

有些类的拷贝和操作没有意义，比如ostream。所以需要采用某种机制阻止拷贝或赋值

### 定义删除的函数

我们可以将拷贝构造函数和拷贝赋值运算符定义为删除的函数来阻止拷贝

删除的函数：我们虽然定义了它们，但不能以任何方式使用它们，在函数的参数列表后加上=delete来指出我们希望定义为删除的

```

1 struct NoCopy{
2     NoCopy() = default;
3     NoCopy(const NoCopy &) = delete; //阻止拷贝
4     NoCopy operator=(const NoCopy&) = delete; //阻止赋值
5 };

```

与=default不同=delete必须出现在函数第一次声明的时候

### 析构函数不能是删除的

```

1 struct NoDtor{
2     NoDtor() = default;
3     ~NoDtor() = delete;
4 };
5 NoDtor nd; //错误，NoDtor的析构函数是删除的，所以不能定义该类的变量或临时对象
6 NoDtor *p = new NoDtor(); //正确，虽然不能定义这种类型的变量或成员，但是可以动态分配 但是不能delete
7 delete p; //错误

```

# 合成的拷贝控制成员可能是删除的

如果我们未定义拷贝控制成员，编译器会为我们定义合成的版本。如果一个类未定义构造函数，编译器会为其合成一个默认构造函数。对某些类来说，编译器将这些合成的成员定义为删除的函数：

- 如果类的某个成员的析构函数是删除的或不可访问的（private），则类的合成析构函数被定义为删除的（析构函数被定义为删除的）
- 如果类的某个成员的拷贝构造函数是删除的或不可访问的，则类的合成拷贝构造函数被定义为删除的。如果类的某个成员的析构函数是删除的或不可访问的，则类合成的拷贝构造函数也被定义为删除的（拷贝构造函数被定义为删除的）
- 如果类的某个成员的拷贝赋值运算符是删除的或不可访问的，或是类有一个const的或引用成员，则类的合成拷贝赋值运算符被定义为删除的（拷贝赋值运算符被定义为删除的）
- 如果类的某个成员的析构函数是删除的或不可访问的，或是类有一个引用成员，它没有类内初始化器（类内初始值），或是类有一个const成员，它没有类内初始化器且类型未显式定义默认构造函数，则该类的默认构造函数被定义为删除的（构造函数被定义为删除的）

这些规则的含义是：如果一个类有数据成员不能默认构造、拷贝、复制或销毁，则对应的成员函数将被定义为删除的；成员有删除的或不可访问的析构函数会导致合成的默认构造函数和拷贝构造函数被定义为删除的，防止我们定义出来无法销毁的对象；对于具有引用或const成员的类，编译器不会合成默认构造函数；如果一个类有const成员，则它不能使用合成的拷贝赋值运算符，因为不能给const赋值新值；虽然能够赋值给引用一个新值，但是改变了所引用对象的值，而无法使得两个类引用同一个值，所以合成的拷贝赋值运算符被定义为删除的

## private拷贝控制

在新标准之前，类是通过将拷贝构造函数和拷贝赋值运算符声明为private来阻止拷贝。为了阻止友元和成员函数的访问，我们只声明不定义它们（声明但不定义一个成员函数是合法的，有一个例外见[15.2.1]）

## 13.2 拷贝控制和资源管理

通常，管理类外资源的类必须定义拷贝控制成员

定义拷贝操作，使得类的行为像一个值或像一个指针

类的行为像一个值，则拷贝一个对象时，副本和原对象是独立的，改变副本不会改变原对象

类的行为像一个指针，则拷贝一个对象时，副本和原对象指向的是同样的底层结构，改变副本也就改变了原对象

### 13.2.1 行为像值的类

类值版本的HasPtr

```

1 class HasPtr{
2     public:
3         HasPtr(const std::string &s =
4             std::string()):ps(new std::string(s)),i(0){}
5         HasPtr(const HasPtr &p):ps(new
6             std::string(*p.ps)),i(p.i){} //拷贝的时候重新分配
7             //一个string对象
8         HasPtr& operator=(const HasPtr &);
9         ~HasPtr() {delete ps;}
10        private:
11            std::string *ps;
12            int i;
13 }
```

### 类值拷贝赋值运算符

赋值运算符通常组合了析构函数和构造函数的操作。类似析构函数，赋值操作会销毁左侧运算对象的资源。类似拷贝构造函数，赋值操作会从右侧运算对象拷贝数据

```

1 HasPtr& HasPtr::operator=(const HasPtr &rhs){
2     auto newp = new string(*rhs.ps);
3     delete ps;
4     ps = newp;
5     i = rhs.i;
6     return *this;
7 }
```

## 13.2.2 定义行为像指针的类

- 使用shared\_ptr管理类中的资源（析构的时候比较方便）
- 引用计数（专门使用一个值来记录有多少个对象指向相同的内存，这样在析构的时候才知道能不能释放内存，相当于shared\_ptr的简单实现思路）

## 13.3 交换操作

除了定义拷贝控制成员，管理资源的类通常还定义一个名为swap的函数。

有时我们需要定义自己的swap，因为有时交换指针（不直接交换对象）更加的效率

### swap函数应该调用swap而不是std::swap

假定我们有一个命名为Foo的类，它有一个类型为HasPtr的成员h，如果我们没有定义Foo版本的swap，那就会使用标准库版本的swap。标准库版本swap对HasPtr管理的string进行不必要的拷贝（因为对于指针我们可以直接交换指针），不需要如下操作

```

1 HasPtr temp = v1;
2 v1 = v2;
3 v2 = temp;
```

这会对HasPtr对象进行一次拷贝和两次赋值

如果我们为Foo编写的swap是如下的代码，编译器能够通过且正常运行，但是人就是与简单的默认版本的swap一样没有使用HasPtr定义的swap方法

```
1 void swap(Foo &lhs, Foo &rhs) {
2     std::swap(lhs.h, rhs.h);
3 }
```

正确的swap函数如下

```
1 void swap(Foo &lhs, Foo &rhs) {
2     using std::swap;
3     swap(lhs.h, rhs.h); // 使用了HasPtr的版本
4 }
```

每个swap的调用应该都是未加限定的。每个调用都应该是swap而不是std::swap。如果存在特定的swap版本则会调用特定版本否则调用标准库版本，因为其匹配程度高于std中定义的版本[16.3]

至于为什么using声明没有覆盖HasPtr版本的swap声明将在[18.2.3]解释

## 在赋值运算符中使用swap

定义swap的类通常用swap来定义它们的赋值运算符。这些运算符使用了**拷贝并交换**的技术。这种技术将左侧运算对象与右侧运算对象的一个副本进行交换。

这种技术的有趣之处是自动处理了自赋值情况且天然就是异常安全的。通过在改变左侧运算对象之前拷贝右侧运算对象保证了自赋值的正确；代码中会发生异常的是拷贝构造函数的new表达式，如果真发生了异常，它也会在我们改变左侧运算对象之前发生。

## 13.4 拷贝控制示例

## 13.5 动态内存管理类

某些类需要自己进行内存分配，这些类一般来说必须定义自己的拷贝控制成员来管理所分配的内存

### 移动构造函数和std::move

移动构造函数通常是将资源从给定对象“移动”而不是拷贝到正在创建的对象。而且标准库保证移后源string仍然保持一个有效地、可析构的状态。对于string，我们可以想象每个string都有一个指向char数组的指针。可以假定string的移动构造函数进行了指针的拷贝，而不是为字符分配内存空间然后拷贝字符

**move函数：定义在头文件 utility 中**

- 当reallocate在新内存中构造string时，必须调用move来表示希望使用string的移动构造函数。如果漏掉了move的调用，将会使用拷贝构造函数。
- 使用move时通常不提供using声明，当我们使用move时，直接调用std::move而不是move

## 13.6 对象移动

移动而非拷贝对象会大幅度提升性能。

两个原因：

1. 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
2. IO类或unique\_ptr这样的类，它们都包含不能被共享的资源，这些类型不能拷贝但可以移动

## 13.6.1 右值引用

为了支持移动操作，引入了新的引用类型——**右值引用**：所谓右值引用就是必须绑定到右值的引用。我们通过`&&`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，因此可以自由的将一个右值引用的资源“移动”到另一个对象中

左值引用不能将其绑定到要求转换的表达式、字面常量或是返回右值的表达式。

右值引用可以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
4 int &r2 = i * 42; //错误: i * 42 是一右值
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，都是返回左值表达式的例子。我们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。不能将左值引用绑定到这类表达式上，但是可以将`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

右值引用只能绑定到临时对象，我们得知

- 所引用的对象将要被销毁
- 该对象没有其他用户

这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

## 变量是左值

变量可以看作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。所以我们不能将一个右值引用绑定到一个右值引用类型的变量上

```
1 int &&rr1 = 42;
2 int &&rr2 = rr1; //错误，表达式rr1是左值
```

## 标准库move函数

我们可以显式地将一个左值转换为对应的右值引用类型。我们还可以通过调用 `move` 的新标准库函数来获得绑定到左值上的右值引用，此函数定义在 `utility` 中。 `move` 函数使用了我们将在 16.2.6 节中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` 告诉编译器我们有一个左值，但是希望像一个右值一样处理它。`move` 意味着承诺：除了对 `rr1` 赋值或销毁它外，我们将不再使用它。在调用 `move` 之后，我们不能对移后源对象的值做任何假设

## 13.6.2 移动构造函数和移动赋值运算符

移动构造函数的第一个参数是该类类型的一个引用。这个引用参数在移动构造函数中是一个右值引用。与拷贝构造函数一样，任何额外的参数都必须有默认实参。

除了完成资源移动，移动构造函数还必须确保移后源对象处于这样一个状态——销毁它是无害的。特别是，一旦资源完成移动，源对象必须不再指向被移动的资源——这些资源的所有权已经归属新创建的对象。

## 移动操作、标准库容器和异常

`noexcept`是我们承诺一个函数不抛出异常的一种方法。我们在一个函数的参数列表和初始化列表开始的冒号之间指定`noexcept`

**我们必须在类头文件的声明中和定义中都指定`noexcept`**

## 移动赋值运算符

移动赋值运算符执行与析构函数和移动构造函数相同的工作。如果我们的移动赋值运算符不抛出任何异常，我们就应该将它标记为`noexcept`

## 移后源对象必须可析构

当我们编写一个移动操作时，必须保证移后源对象进入一个可析构的状态

除此之外，移动操作还必须保证对象仍然是有效的（对象有效是指可以安全的为其赋予新值或者可以安全地使用而不依赖其当前值）。另一方面，移动操作对移后源对象中留下的值没有任何要求

## 合成的移动操作

编译器也会合成移动构造函数和移动赋值运算符。但是合成移动操作的条件与合成拷贝操作的条件大不相同。与拷贝操作不同，编译器根本不会为某些类合成移动操作。特别是如果一个类定义了自己的拷贝构造函数、拷贝赋值运算符或者析构函数，编译器就不会为它合成移动构造函数和移动赋值运算符了。**只有当一个类没有定义任何自己版本的拷贝控制成员，且类的每个非`static`数据成员都可以移动时，编译器才会合成移动构造函数或移动赋值运算符。**编译器能够移动内置类型，如果一个类类型有对应的移动操作，编译器也能够移动这个成员。

移动操作和合成的拷贝控制成员间还有最后一个相互作用关系：一个类是否定义了自己的移动操作对拷贝操作如何合成有影响。如果类定义了一个移动构造函数和/或一个移动赋值运算符，则该类的合成拷贝构造函数和拷贝赋值运算符也会被定义为删除的

## 移动右值，拷贝左值

如果一个类既有移动构造函数又有拷贝构造函数，编译器使用普通的函数匹配规则来确定使用哪个构造函数

## 如果没有移动构造函数，右值也会被拷贝

## 拷贝并交换赋值运算符和移动操作

HasPtr定义了一个拷贝并交换赋值运算符，如果我们加入一个移动构造函数，它实际上也会获得一个移动赋值运算符

```

1 class HasPtr{
2     public:
3         HasPtr(HasPtr &&p) noexcept:ps(p.ps),i(p.i)
4             {p.ps = 0;}
5         HasPtr& operator=(HasPtr rhs)
6             {swap(*this, rhs); return *this;}
7     };

```

对于这个赋值运算符，有一个非引用参数，意味着此参数要进行拷贝初始化，要么使用拷贝构造函数，要么使用移动构造函数——左值被拷贝，右值被移动（因为赋值运算右侧运算对象是左值，那么它是不能传递给右值引用的，只能使用合成拷贝构造函数拷贝并且执行赋值操作，如果右侧运算对象是一个右值引用，那么在传入的时候能够调用移动构造函数，然后再执行赋值运算符的函数体），所以单一的赋值运算符就实现了拷贝赋值运算符和移动赋值运算符两种功能。

移动操作永远不会隐式定义为删除的函数。如果我们显式的要求编译器生成=default的移动操作，且编译器不能移动所有成员，则编译器会将移动操作定义为删除的函数。除了一个重要例外，什么时候将合成的移动操作定义为删除的函数遵循与定义删除的合成拷贝操作类似的原则。

- 与拷贝构造函数不同，移动构造函数被定义为删除的函数的条件是：有类成员定义了自己的拷贝构造函数且未定义移动构造函数，或者是有类成员未定义自己的拷贝构造函数且编译器不能为其合成移动构造函数。移动赋值运算符的情况类似
- 如果有类成员的移动构造函数或移动赋值运算符被定义为删除的或是不可访问的，则类的移动构造函数或移动赋值运算符被定义为删除的
- 类似拷贝构造函数，如果类的析构函数被定义为删除的或不可访问的，则类的移动构造函数被定义为删除的
- 类似拷贝赋值运算符，如果有类成员是const的或是引用，则类的移动赋值运算符被定义为删除的。

## 移动迭代器

一个移动迭代器通过改变给定迭代器的解引用运算符的行为来适配此迭代器。一般来说一个迭代器的解引用运算符返回一个指向元素的左值。移动迭代器的解引用运算符生成一个右值引用。我们通过标准库的 `make_move_iterator` 函数将一个普通迭代器转换为一个移动迭代器。此函数接受一个迭代器参数，返回一个移动迭代器

### 13.6.3 右值引用和成员函数

如果一个成员函数同时提供拷贝和移动版本，它也能从中受益。这种允许移动的成员函数通常使用与拷贝/移动构造函数和赋值运算符相同的参数模式——一个版本接受一个指向const的左值引用，第二个版本接受一个指向非const的右值引用。例如

```
1 void push_back(const X&);  
2 void push_back(X&&);
```

我们能够将转换为X类型的任何对象传递给第一个版本的push\_back。对于第二个版本我们只可以传递给它非const的右值，此版本与非const的右值是精确匹配的

## 右值和左值引用成员函数(引用限定符)

```
1 string s1 = "a value", s2 = "another";
2 s1 + s2 = "wow"; //旧标准中是ok的，新标准兼容了这个操作，但是可以在自己的类中阻止这种用法
```

我们能够强制左侧运算对象是一个左值。我们指出this的左值/右值属性的方式与定义const成员函数相同，即，在参数列表后放置一个引用限定符

```
1 class Foo{
2     public:
3     Foo &operator=(const Foo&) &;
4 }
```

引用限定符可以是&或者&&，分别指出this可以指向一个左值或右值，必须同时出现在函数的声明和定义中

引用限定符必须跟随在const限定符之后（如果有的话）

## 重载和引用函数

引用限定符也可以区分重载版本，而且如果一个版本加了引用限定符，那么所有的版本都要加（这一点与const限定符不同）

# 第十四章 操作重载与类型转换

## 14.1 基本概念

重载的运算符名字由关键字operator和其后要定义的运算符号共同组成

一元运算符有一个参数，二元运算符有两个；

对于二元运算符来说，左侧运算对象传递给第一个参数，右侧运算对象传递给第二个参数；

如果一个运算符是成员函数，则它的第一个运算对象绑定到this指针上

除了重载的函数调用运算符之外，其他重载运算符不能含有默认实参

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

可以像调用普通函数一行调用运算符函数

```
1 data1 + data2;
2 operator+(data1, data2); //等价的
```

```
1 data1 += data2;
2 data1.operator+=(data2); //等价的成员函数版
```

## 某些运算符不应该被重载

某些运算符指定了运算对象求值的顺序。因为重载运算符本质上是一次函数调用，所以这些关于运算对象求值顺序的规则无法应用到重载的运算符上。特别的逻辑与运算符、逻辑或运算符和逗号运算符的运算对象求值顺序规则无法保留下来。`||`和`&&`的短路求值属性也无法保留，两个运算对象总是会被求值。上述运算符重载版本无法与习惯相同，因此不建议重载它们。

我们一般不重载逗号运算符和取地址运算符：c++语言定义了这两种运算符用于类类型对象时的特殊含义

## 赋值和复合赋值运算符

赋值运算符的行为与复合版本的类似：赋值之后，左侧运算对象和右侧运算对象的值相等，并且运算符应该返回它左侧运算对象的一个引用。

如果类含有算术运算符或者位运算符，则最好也提供对应的复合赋值运算符

## 选择作为成员或者非成员

下面的准则有助于我们将运算符定义为成员函数还是非成员函数：

- 赋值（`=`）、下标（`[]`）和成员访问运算符（`->`）必须是成员

- 复合赋值运算符一般来说应该是成员，但并非必须，这一点与赋值运算符略有不同
- 改变对象状态的运算符或者与给定类型密切相关的运算符，如递增、递减和解引用运算符，通常应该是成员
- 具有对称性的运算符可能转换任意一端的运算对象，因此它们通常应该是普通的非成员

```

1 string s = "world";
2 string t = s + "!";
3 string u = "hi" + s;

```

如果将string的+定义为成员函数，那么第三条语句将会报错（因为当运算符定义成成员函数时，它的左侧运算对象必须是运算符所属类的一个对象）

## 14.2 输入输出运算符

### 14.2.1 重载输出运算符<<

输出运算符的第一个形参是一个非常量ostream对象的引用。ostream是非常量是因为写入内容会改变其状态，第二个形参一般来说是常量引用。返回值是它的ostream形参

```

1 ostream& operator<<(ostream &os, const
  Sales_data &item){
2     //输出操作
3     return os;
4 }

```

### 输入输出运算符必须是非成员函数

如果是成员函数

```

1 Sales_data data;
2 data << cout; //如果operator<<是Sales_data的成员

```

假设输入输出运算符是某个类的成员，我们想按默认的方式使用，则它们必须是istream或ostream的成员。但是我们无法为其添加任何成员

IO运算符通常定义为友元，因为会读取非公有成员

## 14.2.2 重载输入运算符>>

第一个形参要读取的流的引用，第二个形参将要读入到的对象的引用（不是常量，因为数据要读入到这个对象）

### 输入时的错误

在执行输入运算符时可能发生下列错误：

- 当流含有错误类型的数据时读取操作可能失败
- 当读取操作到达文件末尾或者遇到输入流的其他错误时也会失败

## 14.3 算术和关系运算符

通常情况下，我们将算术运算符和关系运算符定义成非成员函数以允许对左侧或右侧对象的运算对象进行转换。因为这些运算符一般不需要改变运算对象的状态，所以形参都是常量的引用。

一般类定义了算术运算符，也会定义对应的复合赋值运算符。此时最有效的方式是使用复合赋值来定义算术运算符。

### 14.3.1 相等运算符

设计准则：

- 如果一个类含有判断两个对象是否相等的操作，则它显然应该把函数定义成operator==而非一个普通的命名函数：因为用户肯定希望能使用==比较对象，所以提供了==就意味着用户无须在费力学习并记忆一个全新的函数名字
- 如果类定义了operator==，则该运算符应该能判断一组给定对象中是否含有重复数据

- 通常情况下，相等运算符应该具有传递性， $a==b$ 和 $b==c$ 都为真，则 $a==c$ 也应该为真
- 如果类定义了operator==，也应该定义operator!=
- 相等运算符和不等运算符中的一个应该把工作委托给另外一个，这意味着其中一个运算符应该负责实际比较对象的工作

## 14.3.2 关系运算符

通常情况下关系运算符应该

1. 定义顺序关系，令其与关联容器中对关键字的要求一致
2. 如果类同时也含有==运算符的话，则定义一种关系令其与==保持一致，特别是如果两个对象是!=的，那么其中一个应该<另一个

## 14.4 赋值运算符

除了拷贝赋值和移动赋值（一般是同类型对象的赋值），类还可以定义其他赋值运算符以使用别的类型作为右侧运算对象（比如vector除了拷贝赋值和移动赋值外，还能够接受花括号内的元素列表作为参数）

### 复合赋值运算符

复合赋值运算符不非得是类的成员，不过我们还是倾向于把包括复合赋值在内的所有赋值运算都定义在类的内部。

## 14.5 下标运算符

表示容器的类通常可以通过元素在容器中的位置访问元素，这些类一般会定义下标运算符operator[]。

为了和下标的原始定义兼容，下标运算符通常以所访问元素的引用作为返回值，这样做的好处是下标可以出现在赋值运算符的任意一端。我们最好同时定义下标运算符的常量版本和非常量版本，当作用于一个常量对象时，下标运算符返回常量引用以确保我们不会给返回的对象赋值。

```

1 class StrVec{
2     public:
3         std::string& operator[](std::size_t n){
4             return elements[n];
5         }
6         const std::string& operator[](std::size_t n)
7             const{
8                 return elements[n];
9             }
9 };

```

## 14.6 递增和递减运算符

C++并不要求递增和递减运算符必须是类的成员，但是因为它们改变的正好是所操作对象的状态，所以建议将其设定为成员函数

### 定义前置递增/递减运算符

```

1 class StrBlobPtr{
2     public:
3         StrBlobPtr& operator++();
4         StrBlobPtr& operator--();
5 };

```

为了与内置版本保持一致，**前置运算符应该返回递增或递减后对象的引用**。注意递增和递减实现上的差别

```

1 StrBlobPtr& StrBlobPtr::operator++() {
2     check(curr, "msg"); // 检查curr的索引值是否有效
3     ++curr;
4     return *this;
5 }
6 StrBlobPtr& StrBlobPtr::operator--() {
7     --curr; // 如果curr是0, 递减会产生一个表示无效下
8     // 标的非常大的值
9     check(curr, "msg"); // 检查curr的索引值是否有效
10    return *this;
11 }
```

## 区分前置和后置运算符

为了解决区分前置和后置的问题，后置版本接受一个额外的（不被使用）int类型的形参。当我们使用后置运算符时，编译器为这个形参提供一个值为0的实参

```

1 class StrBolbPtr{
2     public:
3     StrBolbPtr operator++(int);
4     StrBolbPtr operator--(int);
5 };
```

为了与内置版本一致，**后置运算符应该返回对象的原值**（而非引用，因为原值是要被改变的，实际上这里返回的是一个右值，用完即止）。

对于后置版本，在递增对象之前需要先记录对象的状态

```

1 StrBlobPtr StrBlobPtr::operator++(int){
2     StrBlobPtr ret = *this;
3     ++*this;
4     return ret;
5 }
6 StrBlobPtr StrBlobPtr::operator--(int){
7     StrBlobPtr ret = *this;
8     --*this;
9     return ret;
10}

```

后置运算符调用各自的前置版本来完成实际的工作

## 显式地调用后置运算符

如上所述，我们如果想显式调用后置++，为它的整型参数传入一个值

```

1 p.operator++(0);
2 p.operator++();

```

## 14.7 成员访问运算符

解引用运算符和箭头运算符

解引用运算符首先检查curr是否仍在作用范围内，如果是，则返回curr所指元素的一个引用。箭头运算符不执行自己的操作，通过调用解引用运算符并返回解引用结果元素的地址。

**箭头运算符必须是类的成员，解引用运算符通常也是**

### 对箭头运算符返回值的限定

**箭头运算符不能丢掉成员访问这个最基本的含义。当我们重载箭头时，可以改变的是箭头从那个对象当中获取成员，而箭头获取成员这一事实永远不变。**

对于形如point->mem的表达式来说，point必须是指向类对象的指针或者是一个重载了operator->的类的对象。根据point类型的不同，point->mem分别等价于

```
1 (*point).mem;
2 point.operator()->mem;
```

point->mem的执行过程如下所示：

1. 如果point是指针，则我们应用内置的箭头运算符，表达式等价于(\*point).mem
2. 如果point是定义了operator->的类的一个对象，则我们使用point.operator->()的结果来获取mem。其中如果该结果是一个指针，则执行第一步；如果该结果本身含有重载的operator->()，则重复调用当前步骤。最终，当这一过程结束时程序或者返回了所需的内容，或者返回一些表示程序错误的信息。

## 14.8 函数调用运算符

类重载了函数调用运算符，则我们可以像使用函数一样使用该类的对象。

```
1 class absInt{
2     int operator()(int val) const{
3         return val < 0 ? -val : val;
4     }
5 };
6 int i = -42;
7 absInt absObj;
8 int ui = absObj(i);
```

函数调用运算符必须是成员函数，一个类可以定义多个不同版本的调用运算符

如果类定义了调用运算符，则该类的对象称作**函数对象**。

## 14.8.1 lambda是函数对象

当我们编写一个lambda后，编译器将该表达式翻译成一个未命名类的未命名对象。在lambda表达式产生的类中含有一个重载的函数调用运算符

```

1 stable_sort(words.begin(),words.end(),[](const
2     string &a, const string&b){
3     return a.size() < b.size();
4 });
5 //行为类似于下面这个类的未命名对象
6 class ShorterString{
7     public:
8     bool operator()(const string &a, const
9         string&b) const{
10         return a.size() < b.size();
11     }
12 };

```

默认情况下lambda不能改变它捕获的变量，因此在默认情况下，由lambda产生的类当中的函数调用运算符是一个**const**成员函数。如果lambda被声明为可变的，则调用运算符就不是**const**的了

```

1 stable_sort(words.begin(),words.end(),ShorterStr
  ing);

```

### 表示lambda及相应捕获行为的类

上述的lambda没有捕获变量，所以只生成了函数调用运算符

当一个lambda表达式通过引用捕获变量时，将由程序负责确保lambda执行时引用所引对象确实存在，因此编译器可以直接使用该引用而无须在lambda产生的类中将其存储为数据成员

当一个lambda表达式通过值捕获时，该值被拷贝到lambda中，因此lambda生成的类必须为捕获的值生成对应的数据成员，同时创建构造函数来初始化数据成员

lambda产生的类不含默认构造函数、赋值运算符及默认析构函数；是否含有拷贝/移动构造函数则通常要视捕获的数据成员类型而定

## 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命名操作的调用运算符。

头文件functional

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>	

## 在算法中使用标准库函数对象

表述运算符的函数对象常用来替换算法中默认的运算符

```
1 sort(svec.begin(), svec.end(), greater<string>());
```

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，然而我们可能会希望通过比较指针的内存地址来sort指针的vector。直接这么做将产生未定义的行为，因此可以使用一个标准库函数对象来实现该目的：

```

1 vector<string*> nameTable; //指针的vector
2 //错误: nameTable中的指针彼此之间没有关系, 所以<产生未
  定义的行为
3 sort(nameTable.begin(),nameTable.end(),[](string
  *a,string *b){return a < b;});
4 sort(nameTable.begin(),nameTable.end(),less<stri
  ng*>()); //正确

```

### 14.8.3 可调用对象与function

可调用对象有函数、函数指针、lambda表达式、bind创建的对象（用来适配参数列表）以及重载了函数调用运算符的类

可调用对象也有类型。lambda有自己唯一的类类型，函数及函数指针的类型由返回值类型和实参类型决定。两个不同类型的可调用对象可能共享同一种调用形式。调用形式指明了调用返回类型以及传递给调用的实参类型。一种调用形式对应一个函数类型

```

1 int(int,int) //是一个函数类型, 接受两个int, 返回一个
  int

```

### 不同类型可能具有相同的调用形式

对于几个可调用对象共享同一种调用形式的情况，有时我们会希望把它们看成具有相同的类型

```

1 int add(int i, int j){return i + j;}
2 auto mod = [] (int i, int j){return i % j;};
3 struct divide{
4     int operator()(int i, int j){
5         return i/j;
6     }
7 };
8 //这几个对象共享同一种调用形式: int(int,int)

```

**函数表**: 用于存储指向这些可调用对象的“指针”，c++中函数表容易通过map实现，表示运算符的string作为键，实现运算符的函数作为值

```

1 map<string, int(*)(int,int)> binops;
2 binops.insert({"+", add}); //正确
3 binops.insert({"%", mod}); //错误: mod不是一个函数
指针

```

## 标准库function类型

我们使用function的标准库解决上述的问题

### function的操作

function<T> f;

f是一个用来存储可调用对象的空function，这些可调用对象的调用形式应该与函数类型T相同（即T是retType(args)）

function<T> f(nullptr);

显式的构造一个空function

function<T> f(obj);

在f中存储可调用对象obj的副本

f

将f作为条件：当f含有一个可调用对象时为真

f(args)

调用f中的对象，参数是args

## 定义为function<T>的成员 function的操作员的类型

result\_type

该function类型的可调用对象的返回的类型

argument\_type

当T有一个或两个实参时定义的类型。如果T只有一个实参，则argument\_type是该类型的同义词；如果T有两个实参，则first\_argument\_type和second\_argument\_type分别代表两个实参的类型

first\_argument\_type

second\_argument\_type

```

1 function<int(int,int)>
2 function<int(int,int)> f1 = add;
3 function<int(int,int)> f2 = divide();
4 function<int(int,int)> f3 = [](int i,int j)
    {return i * j;};
5 cout << f1(4,2) << endl;
6 cout << f2(4,2) << endl;
7 cout << f3(4,2) << endl;

```

直接使用function能够调用，function重载了调用运算符。使用function我们可以重新定义map

```
1 map<string,function<int(int,int)>> binops;
```

## 重载的函数与function

我们不能（直接）将重载函数名字存入function类型的对象中，解决二义性的问题的一条途径是存储函数指针而非函数名字

```

1 int (*fp)(int, int) = add;
2 binops.insert({"+", fp}); // fp就限制了我们到底存储的那个版本的重载

```

也可以使用lambda来消除二义性

```

1 binops.insert({"+", [](int a, int b){return
add(a,b);}});

```

## 14.9 重载、类型转换与运算符

转换构造函数和类型转换运算符共同决定了类类型转换，也称为用户定义的类型转换

### 14.9.1 类型转换运算符

类型转换运算符是类的一种特殊成员函数，它负责将一个类类型的值转换成其他类型。

```

1 operator type() const;

```

其中type表示某种类型。类型转换运算符可以面向任意类型（除了void）进行定义，只要该类型能够作为函数的返回类型。（所以数组和函数类型是不可以的）允许转换成指针或引用类型

类型转换运算符没有显式的返回类型，也没有形参，而且必须定义成类的成员函数。一般不会改变待转换对象内容，所以一般定义为const

### 定义含有类型转换运算符的类

```

1 class SmallInt{
2     public:
3         SmallInt(int i = 0):val(i){
4             if(i<0 || i>255)
5                 throw std::out_of_range("xxx");
6         }
7         operator int() const{return val;}
8     private:
9         std::size_t val;
10    };

```

既定义了向类类型的转换（构造函数），也定义了从类类型向其他类型的转换（类型转换运算符）。

尽管编译器一次只能执行一个用户定义的类型转换，但是隐式的用户定义类型转换可以置于一个标准类型转换之前或之后

```

1 SmallInt si = 3.14; //正确的
2 si + 3.14; //先将si转换为int，然后再内置转换为double
               进行相加

```

因为类型转换运算符是隐式执行的，所以无法传递实参，故不能在类型转换运算符的定义中使用任何形参。同时尽管类型转换函数不负责指定返回类型，但实际上每个类型转换函数都会返回一个对应类型的值

## 类型转换运算符可能产生意外结果

一般类很少提供类型转换运算符，大多数情况下，类型转换自动发生，用户可能会感觉比较意外，而不是感觉受到了帮助。然后这条经验法则存在例外情况，对于类来说，**定义向bool的类型转换还是比较普遍的现象。**

意向不到的结果有可能会发生（在C++标准的早期版本中），例如，如果istream含有向bool的类型转换时，下面的代码将编译通过：

```
1 int i = 42;
2 cin << i;
```

输出运算符被用到输入流上理应报错，但是如果cin发生了隐式的类型转换转换为bool，进而提升为int，然后移位42位，这样就不会报错，但是与我们的期望大相径庭（因为bool是一种算术类型，能够被用在任何需要算术类型的上下文中）

## 显式的类型转换运算符

C++11引入了显式的类型转换运算符

```
1 class SmallInt{
2     public:
3         //编译器不会自动执行这一类型转换
4         explicit operator int() const{return val;}
5     };
6 SmallInt si = 3; //正确，构造函数没有被explicit修饰
7 si + 3; //错误
8 static_cast<int>(si) + 3; //正确：显式地请求类型转换
```

当然存在例外，如果表达式被用于条件，编译器会将显式类型转换自动的应用。即在下列位置时，显式的类型转换将被隐式执行

- if、while及do语句的条件部分
- for语句头的条件表达式
- 逻辑非运算符、逻辑或运算符、逻辑与运算符的运算对象
- 条件运算符的条件表达式

## 转换为bool

向bool的类型转换一般发生在条件部分，所以operator bool一般定义为explicit的

## 14.9.2 避免有二义性的类型转换

两种情况下可能产生多重转换路径

- 两个类提供了相同的类型转换：当A类定义了接受B类对象的转换构造函数，同时B类定义了一个转换目标是A类的类型转换运算符时，我们就说它们提供了相同的类型转换。（都是B转换为A）
- 类定义了多个转换规则：这些转换设计的类型本身可以通过其他类型转换联系在一起。

### 实参匹配和相同的类型转换

定义两种从B到A的转换

```

1 struct B;
2 struct A{
3     A() = default;
4     A(const B&);
5 };
6 struct B{
7     operator A() const;
8 };
9 A f(const A&);
10 B b;
11 A a = f(b); //二义性错误，编译器不知道用哪个
12 A a1 = f(b.operator A()); //正确
13 A a2 = f(A(b)); //正确

```

### 二义性与转换目标为内置类型的多重类型转换

如果类定义了一组类型转换，它们的转换源或者转换目标类型本身可以通过其他类型转换联系在一起，同样会产生二义性问题。经常出现在多个算数类型上

## 重载函数和转换构造函数

```

1 struct C{
2     C(int);
3 }
4 struct D{
5     D(int);
6 }
7 void manip(const C&);
8 void manip(const D&);
9 manip(10); //二义性
10 manip(C(10)); //正确

```

## 重载函数与用户定义的类型转换

当调用重载函数时，如果两个（多个）用户定义的类型转换都提供了可行（不一致）匹配，则我们认为这些转换类型一样好。不会考虑可能出现的标准类型转换的级别。

```

1 struct E{
2     E(double);
3 }
4 void manip(const C&);
5 void manip(const E&);
6 manip(10); //同样二义性，因为我们认为两个转换是一样好的，尽管10还需要强制转换为double(10)

```

### 14.9.3 函数匹配与重载运算符

重载的运算符也是重载的函数，所以通用的函数匹配规则同样适用于判断在给定的表达式中到底应该使用内置运算符还是重载的运算符。

当运算符函数出现在表达式中时，候选函数集的规模要比我们使用调用运算符调用函数时更大。例如a是一种类类型则a sym b可能是

```
1 a.operatorsym(b); //成员函数
2 operatorsym(a,b); //普通函数
```

我们不能通过调用形式区分当前调用是成员还是非成员

重载运算符作用于类类型对象时，候选函数中来源有三

- 非成员版本
- 内置版本
- 如果左侧运算对象是类类型，还有定义在类中的重载版本

## 第十五章 面向对象程序设计 (OOP)

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（封装、继承、多态）

继承和动态绑定对程序的编写有两方面的影响：

1. 我们可以更容易的定义与其他类相似但不完全相同的新类
2. 在使用这些相似的类编写程序时，一定程度上能够忽略它们的区别

### 15.1 OOP概述

通过使用数据抽象，我们可以将类的接口与实现分离；使用继承，可以定义相似但有区别的类型；使用动态绑定，可以一定程度上忽略相似类型的区别，以统一的方式使用它们的对象

#### 继承

定义Quote表示基类，Bulk\_quote表示派生类

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适合自身的版本，此时基类就将这些函数声明为虚函数

```
1 class Quote{
2     public:
3         std::string isbn() const;
4         virtual double net_price(std::size_t n)
5             const;
6 };
```

派生类通过派生类列表明确指出它是从哪个（哪些）基类继承而来的。

派生类必须在其内部对所有重新定义的虚函数进行声明。派生类可以在这样的函数之前加上 `virtual` 关键字，但是不是非得这么做。

C++11标准允许派生类显式地注明它将使用哪个成员函数改写基类的虚函数，在形参列表之后增加 `override` 关键字。

## 动态绑定

## 15.2 定义基类和派生类

### 15.2.1 定义基类

```

1 class Quote{
2     public:
3         Quote() = default;
4         Quote(const std::string &book, double
5             sales_price):
6             bookNo(book), price(sales_price){}
7             std::string isbn() const {return
8             bookNo;}
9             virtual double net_price(std::size_t n)
10            const {return n * price;}
11            virtual ~Quote() = default;
12            private:
13             std::string bookNo;
14            protected:
15             double price = 0.0;
16         };

```

作为继承关系中根节点的类通常会定义一个虚析构函数

## 成员函数与继承

遇到如 `net_price` 这样与类型相关的操作时，派生类必须对其重新定义

基类必须将它的两种成员函数区分开来：

- 基类希望其派生类进行覆盖的函数（通常定义为虚函数）
- 基类希望派生类直接继承而不要改变的函数

任何构造函数之外的非静态函数都可以是虚函数，关键字 `virtual` 只能出现在类的声明语句之前而不能用于类外部的函数定义。如果一个函数在基类中声明成了虚函数，则该函数在派生类中隐式地也是虚函数  
[15.3]

## 访问控制与继承

派生类可以继承定义在基类中的成员，但是派生类的成员函数不一定有权访问从基类继承而来的成员。[15.5]节介绍关于受保护成员的知识

### 15.2.2 定义派生类

派生类必须将其继承而来的成员函数中需要覆盖的那些重新声明。

我们能将公有派生类型的对象绑定到基类的引用或指针上

大多数类都只继承自一个类，这种形式的继承被称作单继承，派生列表中含有多于一个基类的情况将在[18.3]中介绍

## 派生类中的虚函数

派生类经常（但不总是）覆盖它集成的虚函数。如果派生类没有覆盖则该虚函数的行为类似于其他成员，派生类继承基类的版本

## 派生类对象及派生类向基类的类型转换

一个派生类对象包含多个组成部分：一个含有派生类自己定义的（非静态）成员的子对象，以及一个与该派生类继承的基类对应的子对象。



继承自基类的部分和派生类自定义部分不一定是连续存储的，我们能够把派生类的对象当成基类对象来使用，而且我们也能将基类的指针或引用绑定到派生类对象中的基类部分上

## 派生类构造函数

派生类无法直接初始化基类成员，只能使用基类的构造函数来初始化它的基类部分。派生类构造函数同样是通过构造函数初始化列表来将实参传递给基类构造函数的。

```
1 Bulk_quote(const std::string& book, double p,
  std::size_t qty, double
  disc):Quote(book,p),min_qty(qty),discount(disc)
{}
```

首先初始化基类部分，然后按照声明的顺序依次初始化派生类的成员

## 派生类使用基类成员

派生类可以访问基类的公有成员和受保护成员

## 继承与静态成员

如果基类定义了一个静态成员，则在整个继承体系中只存在该成员的唯一定义。

静态成员遵循通用的访问控制规则，如果基类中的成员是private的，则派生类无权访问它

## 派生类的声明

**声明中包含类名但是不包含它的派生列表**

## 被用作基类的类

如果我们想将某个类用作基类，则该类必须已经定义而非仅仅声明

一个类是基类也可以是派生类

```

1 class Base{};
2 class D1:public Base{};
3 class D2:public D1{};

```

Base是D1的直接基类，同时是D2的间接基类

## 防止继承的发生

使用 `final` 阻止其他类继承该类

```
1 class NoDerived final {};
```

## 15.2.3 类型转换与继承

### 静态类型与动态类型

当我们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，直到运行时才可知

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

### 不存在基类向派生类的隐式类型转换

存在派生向基类的类型转换是因为每个派生类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。但是反过来不存在

即使一个基类指针或引用绑定在一个派生类对象上，我们也不能执行从基类到派生类的转换

```

1 Bulk_quote bulk;
2 Quote *itemP = &bulk; //正确
3 Bulk_quote *bulkP = itemP; //错误

```

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，这是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。如果在基类中含有一个或多个虚函数，我们可以使用dynamic\_cast[19.2.1]请求一个类型转换，该转换的安全检查将在运行时执行。如果我们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]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。

## 15.3 虚函数

当我们使用基类引用或指针调用一个虚函数时会执行动态绑定。我们直到运行时才能知道到底调用了哪个版本的虚函数，所以所有虚函数都必须有定义

### 对虚函数的调用可能在运行时才被解析

#### 派生类中的虚函数

当我们在派生类中覆盖了某个虚函数时，可以再一次使用virtual关键字指出该函数的性质。这么做非必须，因为一旦某个函数被声明成虚函数，则在所有派生类中它都是虚函数

一个派生类的函数如果覆盖了某个继承而来的虚函数，则它的形参类型必须与被它覆盖的基类函数完全一致

派生类中虚函数的返回类型也必须与基类函数匹配（当类的虚函数返回类型是类本身的指针或引用时，这一规则无效，即如果D由B派生得到，则基类的虚函数可以返回B\*而派生类的对应函数可以返回D\*，只不过这样的返回类型要求从D到B的类型转换时可访问的）

## final和override说明符

如果派生类定义的函数名字与虚函数相同但是形参列表不同是合法的，但是有时可能是因为我们编写代码时发生了错误，所以提供了 `override` 显式表明我们的意图，让编译器替我们检查

`final` 修饰的函数不能被覆盖，尝试覆盖该函数的操作都将引发错误

`final` 和 `override` 说明符出现在形参列表以及尾置返回类型之后

## 虚函数与默认实参

虚函数也可以有默认实参，如果某次调用使用了默认实参，则该实参值由本次调用的静态类型决定，即如果通过基类引用或指针调用函数，则使用基类中定义的默认实参，即使实际运行的是派生类中的函数版本。

## 回避虚函数的机制

使用作用域运算符可以让虚函数的调用不要进行动态绑定，强迫其执行虚函数的某个特定版本

```
1 double undiscounted = baseP->Quote::net_price(42);
```

该代码强调使用 `Quote` 的 `net_price` 函数，而不管 `baseP` 实际指向的对象类型是什么

## 15.4 抽象基类

### 纯虚函数

一个纯虚函数无须定义，我们通过在函数体的位置（声明语句的分号之前）书写 `=0` 就可以将一个虚函数说明为纯虚函数，`=0` 只能出现在类内部的虚函数声明语句处

```
1 double net_price(std::size_t) const = 0;
```

可以为纯虚函数提供定义，不过函数体必须定义在类的外部

## 含有纯虚函数的类是抽象基类

含有（或者未经覆盖直接继承）纯虚函数的类是**抽象基类**。抽象基类负责定义接口，其他的类可以覆盖该接口。不能（直接）创建一个抽象基类的对象。

派生类必须给出纯虚函数的定义（覆盖它），否则它们仍将是抽象基类

## 派生类构造函数只初始化它的直接基类

## 15.5 访问控制与继承

### 受保护的成员

一个类使用 `protected` 关键字来声明哪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`说明符可以看做是`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
- 和公有成员类似，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。

```

1 class Base{
2     protected:
3         int prot_mem;
4 };
5 class Sneaky:public Base{
6     friend void clobber(Sneaky&);
7     friend void clobber(Base&);
8     int j;
9 };
10 void clobber(Sneaky &s){s.j = s.prot_mem
11 = 0;} //正确
12 void clobber(Base &b){b.prot_mem = 0;}
13 //错误

```

## 公有、私有和受保护继承

1. 基类中成员的访问说明符
2. 派生类的派生列表中的访问说明符

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有什么影响。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。

公有继承中，则成员将遵循其原有的访问说明符；

私有继承中，基类的成员是全部私有的，因此类的用户不能访问其成员

保护继承中，公有成员是受保护的（保护访问权限用户是无法访问的，但是成员和友元可以访问）

基类的私有成员派生也不能访问的

## 派生类向基类转换的可访问性

派生类向基类的转换是否可访问由使用该转换的代码界定，同时派生类的访问说明符也会有影响。假定D继承自B：

- 只有当D公有继承B时，用户代码才能使用派生类向基类的转换；如果D继承B的方式是保护或私有的，则用户代码不能使用该转换
- 不论D以什么方式继承B，D的成员函数和友元都能使用派生类向基类的转换；派生类向其直接基类的类型转换对于派生类的成员和友元来说永远是可访问的
- 如果D继承B的方式是公有的或受保护的，则D的派生类的成员和友元可以使用D向B的类型转换；反之，如果D继承B的方式是私有的，则不能使用

## 友元与继承

友元关系不能继承。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

可以通过使用using声明改变继承的某个名字的访问级别

```

1 class Base{
2     public:
3         std::size_t size() const { return n; }
4     protected:
5         std::size_t n;
6 };
7 class Derived : private Base{
8     public:
9         using Base::size;
10    protected:
11        using Base::n;
12 };

```

Derived使用了私有继承，所以默认情况下的size和n是私有成员。我们使用using声明语句改变了可访问性

只能够改变该类的直接或间接基类中的任何可访问成员（因为基类 private 权限的派生类也无法访问）。

using 声明语句中名字的访问权限由该 using 声明语句之前的访问说明符来决定。

private 部分的 using 声明，只能被类成员和友元访问

public 部分的 using 声明，类的所有用户都可以访问

protected 部分的 using 声明，类的成员、友元和派生类可以访问

## 默认的继承保护级别

class 关键字定义的派生类是私有继承的；使用 struct 关键字定义的派生类是公有继承

```
1 class Base{};  
2 struct D1 : Base {};  
3 class D2 : Base {};
```

## 15.6 继承中的类作用域

每个类定义自己的作用域，在这个作用域中定义类的成员。当存在继承关系时，派生类的作用域嵌套在基类的作用域之内，如果一个名字在派生类的作用域内无法解析，编译器将继续在外层的基类作用域中寻找该名字的定义

### 在编译时进行名字查找

一个对象、引用或指针的静态类型决定了该对象的哪些成员是可见的。即使静态类型与动态类型可能不一致，但是我们能使用哪些成员仍然是由静态类型决定的

例如，如果 `Quote -> Disc_quote -> Bulk_quote` 是继承关系，其中 `Disc_quote` 中有成员 `discount_policy()`

```

1 Bulk_quote bulk;
2 Bulk_quote *bulkP = &bulk;
3 Quote *itemP = &bulk;
4 bulkP->discount_policy(); //正确
5 itemP->discount_policy(); //错误

```

即使bulk中确实有discount\_policy()成员，但是对于静态类型Quote它是不可见得

## 名字冲突与继承

派生类能重用定义在其直接基类或间接基类中的名字，此时定义在内层作用域（派生类）的名字将隐藏定义在外层作用域（基类）的名字

## 通过作用域运算符来使用隐藏的成员

```

1 struct Derived:Base{
2     int get_base_mem{return Base::mem;}
3 };

```

派生类最好不要重用定义在基类中的名字



## 名字查找与继承

理解函数调用的解析过程对于理解C++的继承至关重要，假定我们调用p->mem()，则依次执行以下四个步骤：

1. 确定p的静态类型。因为我们调用的是一个成员，所以该类型必须是类类型
2. 在p的静态类型对应的类中查找mem。如果找不到在其直接基类中不断查找直到到达继承链的顶端。如果找遍了整个继承层次仍然找不到，则编译器将报错

3. 一旦找到了mem，就进行常规的类型检查以确认对于当前找到的mem，本次调用是否合法

4. 假设调用合法，则编译器根据调用的是否是虚函数而产生不同的代码

- 如果mem是虚函数且我们是通过引用或指针调用的，则编译器根据对象的动态类型确定运行该虚函数的那个版本
- 如果mem不是虚函数或者我们是通过对象进行的调用，编译器将产生一个常规函数调用

## 名字查找先于类型检查

声明在内层作用域的函数并不会重载声明在外层作用域的函数，所以定义派生类中的函数也不会重载其基类中的成员（即使形参列表不同）

## 虚函数与作用域

### 通过基类调用隐藏的虚函数

调用的如果是非虚函数，不会发生动态绑定。实际调用的函数版本由指针的静态类型决定

## 覆盖重载的函数

成员函数无论是否是虚函数都能够被重载。派生类可以覆盖重载函数的0个或多个实例。如果派生类希望所有的重载版本对于它来说都是可见的，那么它就需要覆盖所有的版本，或者一个也不覆盖

如果仅需覆盖重载集合中的一些而非全部函数，那么就不得不覆盖基类中的每个版本，这样做非常的繁琐

一个解决方案是为重载的成员提供一条using声明语句，这样就无须覆盖基类中的每一个版本了。using声明语句指定一个名字不指定形参列表，所以一条基类成员函数的using声明语句就把该函数的所有重载版本实例添加到派生类作用域中，此时派生类只需要定义特有的函数就可以

了。

## 15.7 构造函数与拷贝控制

### 15.7.1 虚析构函数

继承关系对基类拷贝控制最直接的影响是基类通常应该定义虚析构函数，这样我们就能动态分配继承体系中的对象了

原因是当delete一个动态分配的对象的指针时将执行析构函数。如果该指针指向的是继承体系中的某个类型，那就可能出现指针的静态类型和被删除对象的动态类型不符的情况，所以我们需要虚析构函数（确保析构函数能够执行正确的版本）

与其他虚函数一样，虚析构函数的虚属性也会被继承。所以派生类使用合成的析构函数还是定义自己的析构函数，都将是虚析构函数

### 虚析构函数将阻止合成移动操作

基类需要一个虚析构函数还会对基类和派生类的定义产生另外一个间接的影响：如果一个类定义了析构函数，即使它通过=default的形式使用了合成的版本，编译器也不会为这个类生成合成移动操作

### 15.7.2 合成拷贝控制与继承

基类或派生类的合成拷贝控制成员的行为与其他合成的构造函数、赋值运算符或析构函数类似：它们对类本身的成员依次进行初始化、赋值或销毁操作。此外，合成的成员还负责使用直接基类的对应操作对一个对象的基类部分初始化、赋值或销毁

无论基类是合成的版本还是自定义的版本都没有太大影响。唯一的要求是相应的成员应该可访问并且不是一个被删除的函数

## 派生类中删除的拷贝控制与基类的关系

基类或派生类在某些情况下将其合成的默认构造函数或者任何一个拷贝控制成员定义成被删除的函数。此外某些定义基类的方式也有可能导致派生类成员是被删除的

- 如果基类中的默认构造函数、拷贝构造函数、拷贝赋值运算符或析构函数是被删除或者不可访问的，则派生类中对应的成员将是被删除的，原因是编译器不能使用基类成员来执行派生类对象基类部分的构造、赋值或销毁操作
- 如果在基类中有一个不可访问或删除掉的析构函数，则派生类中合成的默认和拷贝构造函数将是被删除的，因为编译器无法销毁派生类对象的基类部分
- 编译器不会合成一个删除掉的移动操作。当我们使用=default请求一个移动操作时，如果基类中的对应操作是删除的或不可访问的，那么派生类该函数将是被删除的，原因是派生类对象的基类部分不可移动。如果基类的析构函数是删除的或不可访问的，则派生类的移动构造函数也将是删除的

如果基类中没有默认、拷贝或移动构造函数，一般情况下派生类也不会定义相应的操作

## 移动操作与继承

大多数基类都会定义一个虚析构函数。因此默认情况下基类不会有合成的移动操作，而且派生类中也没有合成的移动操作

基类缺少移动操作会阻止派生类拥有自己的合成移动操作，所以当我们确实需要执行移动操作时应该先在基类中定义，基类中可以使用合成的版本，但是需要显式地定义，一旦基类定义了自己的移动操作，那么必须同时显式地定义拷贝操作

### 15.7.3 派生类的拷贝控制成员

派生类的构造函数在初始化阶段不但要初始化派生类自己的成员，还负责初始化派生类对象的基类部分。拷贝和移动构造函数也类似

析构函数只负责销毁派生类自己分配的资源，对象的成员是被隐式销毁的，了该死的，派生类对象的基类部分也是自动销毁的

## 定义派生类的拷贝或移动构造函数

当为派生类定义拷贝或移动构造函数时，通常使用对应的基类构造函数初始化对象的基类部分

```

1 class Base{};
2 class D : public Base{
3     public:
4         D(const D& d):Base(d); //会自动将D中的基类
      部分拿出来为Base做初始化
5         D(D&& d) : Base(std::move(d));
6     };

```

## 派生类赋值运算符

与拷贝和移动构造函数一样，派生类的赋值运算符也必须显式地为其基类部分赋值

```

1 Base::operator=(const Base&)//不会被自动调用
2 D &D::operator=(const D&rhs){
3     Base::operator=(rhs); //会自动将rhs中的基类部分
      拿出来操作
4     return *this;
5 }

```

## 派生类析构函数

派生类析构函数只负责销毁由派生类自己分配的资源

```

1 class D : public Base{
2     public:
3         ~D(){}
4 };

```

## 15.7.4 继承的构造函数

一个类只继承其直接基类的构造函数（因为一个类只初始化它的直接基类）。类不能继承默认、拷贝和移动构造函数，如果派生类没有直接定义这些，编译器将为它生成它们

派生类继承基类构造函数的方式是提供一条注明了基类名的using声明

```

1 class B : public D{
2     public:
3         using D::D;
4         //xxx
5 };

```

对于基类的每个构造函数，编译器都在派生类中生成一个形参列表完全相同的构造函数。编译器生成的构造函数形如：

```

1 derived(parms) : base(args){}

```

如果派生类有自己的数据成员，则这些成员将被默认初始化（因为这种生成的构造函数只有基类所需的参数）

### 继承的构造函数的特点

与普通成员的using声明不一样，一个构造函数的using声明不会改变构造函数的访问级别

一个using声明语句不能指定explicit或constexpr，如果基类的构造函数有这些属性，继承的构造函数也拥有相同的属性

当一个基类构造函数含有默认实参时，这些实参并不会被继承。相反，派生类将得到多个继承的构造函数，其中每个构造函数分别省略掉一个含有默认实参的形参。例如，如果基类有一个接受两个形参的构造函数，其中第二个形参含有默认实参，则派生类将获得两个构造函数：一个构造函数接受两个形参（没有默认实参），一个构造函数只接受一个形参

基类含有几个构造函数时，除了下述情况，派生类会继承所有这些构造函数

- 派生类定义的构造函数与基类的构造函数具有相同的形参列表，则该构造函数不会被继承。定义在派生类中的构造函数将替换继承而来的原构造函数
- 默认、拷贝和移动构造函数不会被继承。按照正常的规则被合成

## 15.8 容器与继承

当我们使用容器存放继承体系中的对象时，通常必须采取间接存储的方式。因为不允许在容器中保存不同类型的元素，所以不能把具有继承关系的多种类型的对象直接存放在容器中

假定一个vector保存派生类对象，那么我们不能将基类对象放入容器中（因为基类对象无法转换为派生类对象）；也不能使用vector保存基类对象，虽然此时可以将派生类对象放置在容器中，但是拷贝给容器时只有基类部分被拷贝，派生类部分被忽略掉

### 在容器中放置（智能）指针而非对象

在容器中放置基类对象的指针（智能指针）更好，这样继承层次中的类都能够加入到容器中

无法直接使用对象进行面向对象编程，必须使用指针和引用

# 第十六章 模板与泛型编程

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能处理程序在运行之前都不知道类型的情况；泛型编程则在编译时就能获知类型

## 16.1 定义模板

### 16.1.1 函数模板

一个函数模板就是一个公式，可以用来生成针对特定类型的函数版本

```

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

模板定义以关键字 `template` 开始，后跟一个模板参数列表

### 实例化函数模板

当我们调用一个函数模板时，编译器（通常）用函数实参来为我们推断模板实参

```
1 cout << compare(1,0) << endl; //T为int
```

编译器用推断出的模板参数来为我们实例化一个特定版本的函数。

### 模板类型参数

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

类型参数可以用来指定返回类型或函数的参数类型，以及在函数体内用于变量声明或类型转换

类型参数前必须使用关键字 `class` 或 `typename`

## 非类型模板参数

除了定义类型参数，还可以在模板中定义**非类型参数**。非类型参数表示一个值而非一个类型。我们通过一个**特定的类型名**而非关键字`class`或`typename`来指定非类型参数

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**。

例如，由于不能拷贝数组（所以用指针或者数组的引用），我们可以定义两个非类型的参数来表示不同长度的字符串字面量

```
1 template<unsigned N, unsigned M>
2 int compare(const char (&p1)[N], const char
3             (&p2)[M]){
4     return strcmp(p1, p2);
5 }
```

当我们调用这个版本的`compare`时，编译器会用字面常量的大小代替`N`和`M`

```
1 compare("hi", "mom");
```

一个非类型参数可以是一整型，或者是一个**指向对象或函数类型的指针**或（左值）引用。

绑定到非类型整型参数的实参必须是一个**常量表达式**。

绑定到指针或引用非类型参数必须具有**静态的生存期**。我们不能用一个普通（非`static`）局部变量或动态对象作为指针或引用非类型模板参数的实参。

指针参数也可以用`nullptr`或一个值为0的常量表达式来实例化。

## inline和constexpr的函数模板

函数模板可以声明为`inline`或`constexpr`的，如同非模板函数一样。`inline`或`constexpr`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

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`<`比较运算符

通过将函数参数设定为`const`的引用，我们保证了函数可以用于不能拷贝的类型，并且如果用来处理大对象，这种方式还能够提高效率

## 模板编译

编译器遇到一个模板定义时，并不生成代码。只有当我们实例化出模板的特定版本时，编译器才会生成代码。

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

## 大多数编译错误在实例化期间报告

通常，编译器会在三个阶段报告错误

1. 编译模板本身时，编译器此时能够检查语法错误
2. 编译器遇到模板使用时，对于函数模板调用，编译器通常会检查实参数目是否正确。参数类型是否匹配；对于类模板，编译器可以检查用户是否提供了正确数目的模板实参，但也仅限于此
3. 模板实例化时，这时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

编译器不能像函数模板一样为类模板推断模板参数类型，使用类模板必须在尖括号中提供额外信息

### 实例化类模板

一个类模板的每个实例都形成一个独立的类。类型Blob<string>与其他Blob类型都没有关联，也不会对其他任何Blob类型的成员有特殊访问权限

### 类模板的成员函数

类模板的成员函数本身是一个普通函数。但是，类模板的每个实例都有其自己的版本成员函数。类模板的成员函数具有和模板相同的模板参数，因此定义类模板之外的成员函数就必须以关键字template开始，后接类模板参数列表

当类外定义一个成员时，必须说明成员属于那个类。而且，从一个模板生成的类的名字中必须包含其模板实参

### 类模板成员函数的实例化

默认情况下，一个类模板的成员函数只有当程序用到它时才进行实例化。

如果一个成员函数没有被使用，则它不会被实例化。成员函数只有在被用到时才进行实例化，这一个特性使得即使某种类型不能完全符合模板操作的要求，我们仍能用该类型实例化类

### 在类代码内简化模板类名的使用

使用一个类模板时必须提供模板实参，但是有一个例外，在类模板自己的作用域中，我们可以直接使用模板名而不提供实参

```

1 template <typename T> class BlobPtr{
2     public:
3         BlobPtr() : curr(0){}
4         //xxx
5         BlobPtr& operator++();
6         //xxx
7     };

```

BlobPtr的前置递增运算符返回BlobPtr&而不是BlobPtr<T>，这样是可以的编译器处理时就像我们提供了与模板参数匹配的实参一样

## 在类模板外使用类模板名

当我们在类模板外定义其成员时，我们并不在类的作用域中，直到遇到类名才表示进入类的作用域

```

1 template <typename T>
2 BlobPtr<T> BlobPtr<T>::operator++(int){
3     BlobPtr ret = *this; //这里在类的作用域
4 }

```

返回类型位于类的作用域之外，我们必须指出返回类型是一个实例化的BlobPtr，它所用类型与类实例化所用类型一致。在函数体内我们已经进入类作用域，因此定义ret时无需重复模板实参

## 类模板和友元

当一个类包含一个友元声明时，类与友元各自是否是模板是相互无关的。

- 如果一个类模板包含一个非模板友元，则友元被授权可以访问所有模板实例。
- 如果友元自身是模板，类可以授权给所有友元模板实例，也可以只授权给特定实例

## 一对一友好关系

类模板与另一个（类或函数）模板间友好关系的最常见的形式是建立对应实例及其友元间的友好关系

为了引用（类或函数）模板的一个特定实例，必须首先声明模板自身。一个模板声明包括模板参数列表

```

1 //前置声明，在Blob中声明友元所需要的
2 template <typename> class BlobPtr;
3 template <typename> class Blob;
4 template <typename T>
5     bool operator==(const Blob<T>&, const
6     Blob<T>&);
7 template <typename T> class Blob{
8     //下面的声明建立起了一一对应的关系
9     friend class BlobPtr<T>;
10    friend bool operator==<T>(const Blob<T>&,
11        const Blob<T>&);
12 }
```

## 通用和特定的模板友好关系

一个类可以将另一个模板的每个实例都声明为自己的友元，或者限定特定的实例为友元

```

1 template <typename T> class Pal;
2 class C{
3     friend class Pal<C>; //用类C实例化的Pal是C的一个友元
4     template <typename T> friend class Pal2; //所有Pal2的实力都是C的一个友元，此时无须前置声明
5 };
6 template <typename T> class C2{
7     friend class Pal<T>; //C2的每个实例都将相同实例化的Pal作为友元
8     template <typename X> friend class Pal2; //Pal2的所有实例都是C2每个实例的友元
9     friend class Pal3; //非模板类Pal3是C2所有实例的友元
10 };

```

## 令模板自己的类型参数成为友元

```

1 template <typename Type> class Bar{
2     friend Type;
3     //xxx
4 };

```

## 模板类型别名

类模板的一个实例对应了一个类类型，与其他类类型一样，可以定义一个typedef来引用实例化的类：

```
1 typedef Blob<string> StrBlob;
```

模板不是一个类型，所以不能定义typedef引用一个模板，即无法定义typedef引用Blob<T>

新标准允许为类模板定义一个类型别名

```

1 template <typename T> using twin = pair<T,T>;
2 twin<string> authors; //authors 是一个
pair<string,string>

```

当定义一个模板类型别名时，可以固定一个或多个模板参数

```

1 template <typename T> using partNo =
pair<T,unsigned>;
2 partNo<string> books; //books 是一个
pair<string,unsigned>
3 partNo<Vehicle> cars; //cars 是一个
pair<Vehicle,unsigned>
4 partNo<Student> kids; //kids 是一个
pair<Student,unsigned>

```

## 类模板的static成员

类模板的每个实例都有其自己的static成员实例

### 16.1.3 模板参数

类型参数通常命名为T，但实际上可以使用任何名字

## 模板参数与作用域

模板参数遵循普通的作用域规则。一个模板参数名的可用范围是在其声明之后，至模板声明或定义结束之前

模板参数会隐藏外层作用域中声明的相同名字，但是与其他大多数上下文不同，在模板内不能重用模板参数名

```

1 typedef double A;
2 template <typename A, typename B> void f(A a, B
b) {
3     A tmp = a; //tmp 不是double类型的
4     double B; //错误: 重声明模板参数B
5 }

```

一个模板参数名在一个特定模板参数列表中只能出现一次

## 模板声明

模板声明必须包含模板参数，与函数参数相同，声明中的模板参数的名字不必与定义中相同，只需要保证每个定义和声明都有相同数量和种类（类型或非类型）的参数

一个特定文件所需要的所有模板的声明通常一起放置在文件开始位置，出现于任何使用这些模板的代码之前，原因见[16.3]

## 使用类的类型成员

使用作用域运算符(::)来访问static成员和类型成员。在普通（非模板）代码中，编译器掌握类的定义，因此它知道通过作用域运算符访问的名字是类型还是static成员，例如，string::size\_type的定义因为编译器有string的定义，所以它知道size\_type是一个类型

但是假定T是一个模板类型参数，编译器遇到T::mem时不会知道mem是一个类型成员还是一个static数据成员，直至类实例化时才能知道。但是为了处理模板，编译器必须知道名字是否表示一个类型，例如当出现如下代码时

```
1 T::size_type * p;
```

它需要知道我们是在定义名为p的指针还是一个乘法操作

默认情况下，C++语言假定通过作用域运算符访问的名字不是类型，因此如果我们希望使用一个模板类型参数的类型成员，就必须显式告诉编译器该名字是一个类型，通过关键字 `typename` 来实现这一点：

```

1 template<typename T>
2 typename T::value_type top(const T& c){
3     if(!c.empty()){
4         return c.back();
5     }
6     else{
7         return typename T::value_type();
8     }
9 }
```

当希望通知编译器一个名字表示类型时，必须使用关键字 `typename`，而不能使用 `class`

## 默认模板实参

可以提供默认模板实参，可以为函数和类模板提供默认实参，例如，

```

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

`F` 表示可调用对象的类型，并定义了一个新的函数参数 `f` 绑定到一个可调用对象上

对于一个模板参数，只有右侧的所有参数都有默认实参时，它才可以有默认实参

## 模板默认实参与类模板

无论何时使用类模板，都必须在模板名后接上<>。如果一个类所有模板参数都有默认实参且希望使用这些默认实参时，就必须在模板名之后跟一个空尖括号对

### 16.1.4 成员模板

一个类可以包含本身是模板的成员函数，称为**成员模板**。成员模板不能是虚函数

#### 普通类的成员模板

成员模板也是以模板参数列表开始的，可以通过实参类型来推断模板参数的值

#### 类模板的成员模板

类和成员各自有自己的、独立的模板参数

与类模板的普通函数成员不同，成员模板是函数模板。当我们在类模板外定义一个成员模板时，必须同时为类模板和成员模板提供模板参数列表。类模板的参数列表在前，后跟成员自己的模板参数列表

```
1 template <typename T>
2 template <typename It>
3 Blob<T>::Blob(It b, It e){};
```

#### 实例化与成员模板

必须同时提供类和函数模板的实参（一般类模板显式给出，函数模板是通过实参推断）

## 16.1.5 控制实例化

当模板被使用时才会实例化，这一特性意味着，相同的实例可能出现在多个对象文件中。当两个或多个独立编译的源文件使用了相同的模板，并提供了相同的模板参数时，每个文件中就都会有该模板的一个实例

在大系统中，这种实例化相同模板的额外开销可能非常严重

可以通过显式实例化来避免这种开销。一个显式实例化有如下形式：

```
1 extern template declaration; //实例化声明
2 template declaration; //实例化定义
```

`declaration` 是一个类或函数声明，其中所有模板参数已被替换为模板实参，例如

```
1 extern template class Blob<string>;
2 template int compare(const int&, const int&);
```

当编译器遇到 `extern` 模板声明时，它不会在本文中生成实例代码。将一个实例化声明为 `extern` 就表示承诺在程序其他位置有该实例化的一个非 `extern` 声明（定义）

### 实例化定义会实例化所有成员

一个类模板的实例化定义会实例化该模板的所有成员，包括内联的成员函数

## 16.1.6 效率与灵活性

## 16.2 模板实参推断

函数模板，编译器利用调用中的实参来确定其函数模板参数，这一过程叫做**模板实参推断**

### 16.2.1 类型转换与模板类型参数

一个函数形参的类型使用了模板类型参数，那么它采用特殊的初始化规则。只有很有限的几种类型转换会自动地应用于这些实参。编译器通常不是对实参进行类型转换而是生成一个新的模板实例

与往常一样，顶层 `const` 无论是在形参中还是实参中都会被忽略。在其他类型转换中，能在调用中应用于函数模板的包括如下两项：

- `const` 转换：可以将一个非 `const` 对象的引用（或指针）传递给一个 `const` 的引用（或指针）形参
- 数组或函数指针转换：如果函数形参不是引用类型，则可以对数组或函数类型的实参应用正常的指针转换。一个数组实参可以转换为一个指向其首元素的指针。一个函数实参可以转换为一个该函数类型的指针

其他类型转换，如算术类型转换[4.11.1]、派生类向基类的转换[15.2.2]以及用户定义的转换[7.5.4] [14.9]，都不能应用于函数模板

### 使用相同模板参数类型的函数形参

一个模板类型参数可以用作多个函数形参的类型。由于只允许有限的几种类型转换，因此传递给这些形参的实参必须具有相同的类型。如果推断出的类型不匹配，则调用就是错误的

### 正常类型转换应用与普通函数实参

函数模板也可以有普通类型定义的参数，即不涉及模板类型参数的类型。这种函数的普通类型实参能够进行正常的转换

## 16.2.2 函数模板显式实参

某些情况下，编译器无法推断出模板实参的类型。其他一些情况下，我们希望允许用户控制模板实例化。当函数返回类型与参数列表中任何类型都不相同时，这两种情况最常出现

### 指定显式模板实参

```
1 template <typename T1, typename T2, typename T3>
2 T1 sum(T2, T3);
```

没有实参可以用来推断 T1 的类型，每次调用 sum 时都必须为 T1 提供一个显式模板实参。显式模板实参在尖括号中给出，位于函数名之后，实参列表之前：

```
1 auto val3 = sum<long long>(i, lng); // 显式指定 T1
```

### 正常类型转换应用于显式指定的实参

用普通类型定义的函数参数，允许正常的类型转换；对于模板类型参数已经显式指定了的函数实参，也进行正常的类型转换

```
1 long lng;
2 compare(lng, 1024); // 错误：模板参数不匹配
3 compare<long>(lng, 1024); // 正确：实例化
    compare(long, long)
4 compare<int>(lng, 1024); // 正确：实例化
    compare(int, int)
```

## 16.2.3 尾置返回类型与类型转换

当返回类型时模板参数时，用户的负担比较重，因为无法从实参中推断模板参数的值。为了更好的定义我们使用尾置返回类型[6.3.3]，因为尾置返回类型出现在参数列表之后，它可以使用函数的参数

```

1 template <typename It>
2 auto fcn(It beg, It end) -> decltype(*beg) {
3     return *beg;
4 }

```

这样返回类型就能够从 `beg` 的类型中推断出来

## 进行类型转换的标准库模板类

为了获得元素类型，我们可以使用标准库的**类型转换模板**，位于头文件 `type_traits` 中。该头文件中的类通常用于所谓的模板元程序设计

### 标准类型转换模板

对 <code>Mod&lt;T&gt;</code> , 其中 <code>Mod</code> 为	若 <code>T</code> 为	则 <code>Mod&lt;T&gt;::type</code> 为	描述
<code>remove_reference</code>	<code>X&amp;</code> 或 <code>X&amp;&amp;</code> 否则	<code>X</code> <code>T</code>	能够去掉引用

<code>add_const</code>	<code>X&amp;、 const</code> <code>X</code> 或 函数 否则	<code>T</code> <code>const T</code>	为类型添加 <code>const</code>
<code>add_lvalue_reference</code>	<code>X&amp;</code> <code>X&amp;&amp;</code> 否则	<code>T</code> <code>X&amp;</code> <code>T&amp;</code>	左值引用
		<code>T</code>	添加右值引 用 或 中性

add_rvalue_reference 对Mod<T>,其中Mod为	$\wedge\alpha\wedge\wedge\alpha$ 否则 若T为	T及 Mod<T>::type 为	用，或木寸 左值引用不 描述
remove_pointer	X* 否则	X T	去掉指针
add_pointer	X&或X&& 否则	X* T*	
make_signed	unsigned X 否则	X T	将无符号数 变为有符号 数
remove_unsigned	带符号类型 否则	unsigned X T	将带符号类 型变为无符 号类型
remove_extent	X[n] 否则	X T	
remove_all_extent	X[n <sub>1</sub> ][n <sub>2</sub> ]... 否则	X T	

1 `remove_reference<decltype(*beg)>::type; //返回  
beg的类型（此时就不是引用了）`

## 16.2.4 函数指针和实参判断

当用一个函数模板初始化一个函数指针或为一个函数指针赋值时，编译器使用指针的类型来推断模板实参

1 `template <typename T> int compare(const T&,  
const T&);`  
2 `int (*pf1)(const int&, const int&) = compare;`

pf1中参数的类型决定了T的模板实参的类型。上述代码中T的类型为int。

如果不能从函数指针中确定模板实参，则发生错误

```
1 void func(int*)(const string&, const string&));
2 void func(int*)(const int&, const int&));
3 func(compare); //错误，两个函数都可以
4 func(compare<int>); //正确，显示指明了使用func的那个版本
```

## 16.2.5 模板实参推断和引用

```
1 template <typename T> void f(T &p);
```

编译器会应用正常的绑定规则；`const` 是底层的，不是顶层的（引用自己是 `const` 的）

### 从左值引用函数参数推断类型

当一个函数参数是模板类型参数的一个普通（左值）引用时，绑定规则告诉我们只能传递给它一个左值。实参可以是 `const` 类型也可以不是，如果实参是 `const` 的，则T将被推断为 `const` 类型

```
1 template <typename T> void f(T &p);
2 f(i); //i是一个int;模板参数类型T是int
3 f(ci); //模板参数类型是const int
4 f(5); //错误，传递给一个&参数的实参必须是一个左值
```

如果一个函数参数的类型是 `const T&`，正常的绑定规则告诉我们可以传递给它任何类型的实参——一个对象（`const`或非`const`）、一个临时对象或是一个字面常量值。当函数参数本身是`const`时，T的类型推断的结果不会是一个`const`类型，`const`是函数参数的一部分；因此它不会是模板参数类型的一部分

```

1 template <typename T> void f(T &p);
2 f(i); //i是一个int;模板参数类型T是int
3 f(ci); //ci是一个const int, 但模板参数类型是int
4 f(5); //一个const &参数可以绑定到一个右值; T是int

```

## 从右值引用函数参数推断类型

当一个函数参数是一个右值引用时，正常绑定规则告诉我们可以传递给它一个右值。这时，类型推断过程类似普通左值引用函数参数的推断过程。**推断出T的类型是该右值实参的类型**

```

1 template <typename T> void f3(T &&p);
2 f3(42); //实参是一个int类型的右值; 模板参数T是int

```

## 引用折叠和右值引用参数

假设i是一个int对象，我们可能认为像f3(i)这样的调用是不合法的，因为i是一个左值，而且通常不能将一个右值引用绑定到一个左值上。

但是，C++语言提供了两个例外规则，允许上述的绑定。也是move这种标准库设施正确工作的基础

**第一个例外规则影响右值引用参数的推断如何进行。**当我们将一个左值传递给函数的右值引用参数，且此右值引用指向模板类型参数时，编译器推断模板类型参数为实参的左值引用类型。因此调用上述f3(i)时，编译器推断T的类型为int&，而非int

这样看起来可能f3的参数是左值引用的引用，但是不能（直接）定义一个引用的引用。但是通过类型别名[2.5.1]或通过模板类型参数间接定义是可以的

这时使用**第二个例外规则绑定规则**：如果我们间接创建一个引用的引用，则这些引用形成了“折叠”。在所有情况下（除了一个例外），引用会折叠成一个普通的左值引用类型。在新标准中，折叠规则扩展到右值引用。只在一种特殊情况下引用会折叠成右值引用：右值引用的右值引用。

即，对于一个给定类型X：

- $X\& \&$ ,  $X\& \&\&$ 和 $X\&\& \&$ 都折叠成类型 $X\&$
- 类型 $X\&\& \&\&$ 折叠成 $X\&\&$

引用类型只能应用于间接创建的引用的引用，如类型别名或模板参数

## 16.2.6 理解std::move

std::move是如何定义的

```

1 template <typename T>
2 typename remove_reference<T>::type&& move(T&& t)
3 {
4     return static_cast<typename
5         remove_reference<T>::type&&>(t);
6 }
```

通过引用折叠，move的函数参数可以与任何类型的实参匹配

```

1 string s1("hi"), s2;
2 s2 = std::move(string("bye!"));
3 s2 = std::move(s1);
```

std::move是如何工作的

对于第一个std::move的调用

- 推断出T的类型为string
- 因此，remove\_reference用string进行实例化
- remove\_reference<string>的type成员是string
- move的返回类型是string&&
- move的函数参数t的类型为string&&

对于第二个std::move的调用

- 推断出的T的类型为string&

- 因此, `remove_reference`用`string&`进行实例化
- `remove_reference<string&>`的`type`成员是`string`
- `move`的返回类型是`string&&`
- `move`的函数参数`t`的类型为`string&&`

从一个左值`static_cast`到一个右值引用是允许的, 这是一条针对右值引用的特许规则: 虽然不能隐式地将一个左值转换为右值引用, 但可以使用`static_cast`显式地将一个左值转换为右值引用

## 16.2.7 转发

某些函数需要将其一个或多个实参连同类型不变地转发给其他函数。在此情况下, 需要保持被转发实参的所有性质, 包括实参是否是`const`的以及实参是左值还是右值

### 定义能保持类型信息的函数参数

某些函数需要将其一个或多个实参连同类型不变地转发给其他函数。需要保持被转发实参的所有性质, 包括实参类型是否是`const`的以及实参是左值还是右值

例如如下代码

```

1 template <typename F, typename T1, typename T2>
2 void flip2(F f, T1 t1, T2 t2){
3     f(t2, t1);
4 }
5 void f(int v1, int &v2){
6     ++v2;
7 }
```

将两个参数逆序传给`f`, 这在一般情况下工作的很好, 但是我们希望它调用一个接受引用参数的函数时会出现问题, `f`改变绑定到`v2`的值, 但是由于`flip2`是值传递的, 所以并不会改变真正的实参

通过将一个函数参数定义为一个指向模板类型参数的右值引用，可以保持其对应实参的所有类型信息。而使用引用参数（无论是左值还是右值）使得我们可以保持const属性，因为在引用类型中的const是底层的。

```

1 template <typename F, typename T1, typename T2>
2 void flip2(F f, T1 &&t1, T2&& t2){
3     f(t2,t1);
4 }
5 void f(int v1, int &v2){
6     //
7 }
```

当调用flip2的时候，能够保持传入参数的类型信息及const属性，这是因为T1被推断为int &，所以t1的类型会折叠成int &。当flip2调用f时，f中的引用参数v2被绑定到t1，当f修改v2时，也就相当于再修改最初传入flip2的实参

但是上述版本对于接受一个左值引用的函数工作的很好，但是不能用于接受右值引用参数的函数

```

1 void g(int &&i, int& j){
2     //
3 }
```

如果我们在flip2中调用g，则参数t2将被传递给g中的右值引用参数，会出现错误不能从一个左值实例化int&&（函数参数是左值表达式，即即使将一个右值传递给t2，t2的类型是右值引用，但是t2这个名字是左值表达式）

## 在调用中使用std::forward保持类型信息

可以使用一个名为 `forward` 的新标准库设施来传递flip2的参数，它能保持原始实参的类型，定义在头文件utility中，forward必须通过显式模板实参来调用，返回该显式实参类型的右值引用。即`forward<T>`的返回类型是`T&&`

```

1 template <typename Type> intermediary(Type
&&arg){
2     finalFn(std::forward<Type>(arg));
3 }

```

如果实参是一个普通（非引用）类型，`forward<Type>`将返回`Type&&`；如果实参是一个左值，那么通过引用折叠，`Type`本身是一个左值引用类型，所以`forward<Type>`返回一个指向左值引用的右值引用。再次对`forward`的返回类型引用折叠，将返回一个左值引用，这样右值实参将以右值引用的方式传递，左值实参将以左值引用的方式传递

## 16.3 重载与模板

函数模板可以被另一个模板或普通函数重载。名字相同的函数必须具有不同数量或类型的参数

设计函数模板时，函数匹配规则会在以下几方面受到影响：

- 对于一个调用，其候选函数包括所有模板实参推断成功的函数模板实例
- 候选的函数模板总是可行的，因为模板实参推断会排除任何不可行的模板
- 与往常一样，可行函数按类型转换来排序。当然可以用于函数模板调用的类型转换非常有限
- 与往常一样，如果恰有一个函数提供比其他任何函数都要好的匹配，则选择此函数。但是，如果有多个函数提供同样好的匹配，则：
  - 如果同样好的函数中只有一个是非模板函数，则选择此函数
  - 如果同样好的函数中没有非模板函数，而有多个模板函数，且其中一个模板比其他模板更特例化，则选择此模板
  - 否则，此调用有歧义

## 编写重载模板

```

1 template <typename T> string debug_rep(const T
&t) {
2     ostringstream ret;
3     ret << t;
4     return ret.str();
5 }
```

此函数可以用来生成一个对象对应的string表示，该对象可以是任意具备输出运算符的类型

接下来定义打印指针的debug\_rep版本

```

1 template <typename T> string debug_rep(T *p) {
2     ostringstream ret;
3     ret << "pointer: " << p;
4     if(p){
5         //xxx
6     }else{
7         //xxx
8     }
9     return ret.str();
10 }
```

此版本生成一个string，包括指针本身的值和调用debug\_rep获得的指针指向的值。此函数不能用于打印字符指针，因为char\*定义了一个<<版本，假定指针表示一个空字符结尾的字符数组，并打印数组的内容而非地址

```

1 string s("hi");
2 cout << debug_rep(s) << endl;
```

此调用只有第一个版本可行

如果用一个指针调用debug\_rep

```
1 cout << debug_rep(&s) << endl;
```

两个函数都生成可行的实例：

- debug\_rep(const string \*&), T被绑定到string \*
- debug\_rep(string\*), T被绑定到const string

第二个版本提供debug\_rep的精确匹配，第一个版本的实例需要进行普通指针到const指针的转换。

## 多个可行模板

```
1 const string *sp = &s;
2 cout << debug_rep(sp) << endl;
```

- debug\_rep(const string \*&), T被绑定到string \*
- debug\_rep(const string\*), T被绑定到const string

正常的函数匹配规则无法区分这两个函数，但是根据重载模板函数的匹配规则，此调用被解析为debug\_rep(T\*)这个更特例化的版本

## 16.4 可变参数模板

可变参数模板就是一个接受可变数目的模板函数或模板类。可变数目的参数被称为参数包。存在两种参数包

- 模板参数包，表示零个或多个模板参数
- 函数参数包，表示零个或多个函数参数

使用省略号来指出一个模板参数或函数参数表示一个包

- 在一个模板参数列表中，class...或typename...指出接下来参数表示零个或多个类型的列表；一个类型名后面跟一个省略号表示零个或多个给定类型的非类型参数的列表

- 在函数参数列表中，如果一个参数的类型是一个模板参数包，则此参数也是一个函数参数包

```
1 template <typename T, typename... Args>
2 void foo(const T &t, const Args& ... rest);
```



Args是一个模板参数包，rest是一个函数参数包

Args表示零个或多个模板类型参数

rest表示零个或多个函数参数

Args和rest是一一对应的

## sizeof...运算符

想要知道包中有多少元素时，可以使用sizeof...运算符，sizeof...运算符也返回一个常量表达式，而且不会对其实参求值

```
1 sizeof...(Args)
```

### 16.4.1 编写可变参数函数模板

使用 `initializer_list` 可以定义一个可接受可变数目实参的函数，但是所有实参必须是同样的类型

当实参数目和类型都不确定时，使用可变参数函数很有用

可变参数函数通常是递归的，

1. 调用处理包中的第一个实参
2. 用剩余实参调用自身

## 16.4.2 包扩展

对于一个参数包，除了获取其大小外，能够对它做的唯一的事情就是扩展。当扩展一个包时，还要提供用于每个扩展元素的模式。拓展一个包就是将它分解为构成的元素，对每个元素应用模式，获得扩展后的列表。通过在模式右边放一个省略号来触发扩展操作

```

1 template <typename T, typename... Args>
2 ostream &
3 print(ostream &os, const T &t, const Args& ...
       rest){ //扩展Args
4     //
5     return print(os,rest...); //扩展rest
6 }
```

第一个扩展模板参数包，为 print 生成函数参数列表

第二个扩展函数参数包，为 print 生成实参列表

## 理解包扩展

print 中的函数参数包仅仅将包扩展为其构成元素，C++语言支持更复杂的扩展模式

可以对其每个参数调用某个方法

```

1 template <typename... Args>
2 ostream &errorMsg(ostream &os, const Args&...
       rest){
3     return print(os, debug_rep(rest)...);
4 }
```

这个 print 调用使用了模式 debug\_rep(rest)。此模式表示希望对函数参数包 rest 中的每个元素调用 debug\_rep。扩展结果将是一个逗号分隔的 debug\_rep 调用列表

## 16.4.3 转发参数包

可以组合使用可变参数模板与 forward 机制来编写函数，实现将其实参不变地传递给其他函数

保持信息是一个两阶段的过程：

- 首先为了保持实参中的类型信息，我们必须将函数参数定义为模板类型参数的右值引用

```

1 class StrVec{
2     public:
3         template <class... Args> void
4             emplace_back(Args&&...);
5 }
```

- 当 emplace\_back 将这些实参传递给 construct 时，必须使用 forward 来保持实参的原始类型

```

1 template <class... Args>
2 inline
3 void StrVec::emplace_back(Args&&.. args){
4     //pass
5     alloc.construct(first_free++,
6                     std::forward<Args>(args)...);
7 }
```

即扩展了模板参数包Args，也扩展了函数参数包args

## 16.5 模板特例化

编写单一模板使其对任何模板实参都是最适合的非常困难

## 定义函数模板特例化

当我们特例化一个函数模板时，必须为原模板中的每个模板参数都提供实参。为了指出正在实例化一个模板，应使用关键字 `template` 后跟一个空尖括号对。空尖括号对指出我们将为原模板的所有模板参数提供实参

```

1 template <>
2 int compare(const char* const &p1,const char*
3           const &p2){
4     return strcmp(p1,p2);
5 }
```

## 函数重载与模板特例化

定义函数模板的特例化版本的时候，本质上是接管了编译器的工作。重要的是清楚：**一个特例化版本本质上是一个实例，而非函数名的一个重载版本**

## 类模板特例化

同样的需要 `template` 和 `<>`，定义时将需要的模板参数替换为对应的类型即可

## 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仍是一个模板，使用时用户还必须为未指定的模板参数提供实参

```

1 template <class T> struct remove_reference{
2     typedef T type;
3 };
4 template <class T> struct remove_reference<T&>{
5     typedef T type;
6 };
7 template <class T> struct remove_reference<T&&>{
8     typedef T type;
9 };

```

第一个模板提供了最通用的版本



由于一个部分特例化版本本质上是一个模板，与往常一样，我们首先定义模板参数。类似任何其他特例化版本，部分特例化版本的名字与原模板的名字相同。对每个未完全确定类型的模板参数，在特例化版本的模板参数列表中都有一项与之对应。在类名之后，为要特例化的模板参数指定实参，这些实参列于模板名之后的尖括号中。这些实参与原始模板中的参数按位置对应。

## 特例化成员而不是类

可以只特例化特定成员函数，例如 Foo 是一个模板类，包含一个成员 Bar

```

1 template <>
2 void Foo<int>::Bar(){}

```

只特例化了 Foo<int> 的一个成员，其他成员由 Foo 模板提供

# 第 IV 部分 高级主题

## 第十七章 标准库特殊设施

### 17.1 tuple 类型

tuple 是类似 pair 的模板。pair 的成员类型都不相同，但每个 pair 都恰好有两个成员。不同 tuple 类型的成员类型也不相同，但一个 tuple 可以有任意数量的成员。每个确定的 tuple 类型的成员数目是固定的，但一个 tuple 类型的成员数目可以与另一个 tuple 类型不同

定义在 tuple 头文件中

tuple 支持的操作

操作	说明
<code>tuple&lt;T1,T2,...,Tn&gt; t;</code>	<code>t</code> 是一个 tuple，成员数为 $n$ ，第 $i$ 个成员的类型为 $T_i$ ，所有成员都进行值初始化
<code>tuple&lt;T1,T2,...,Tn&gt; t(v1,v2,...,vn);</code>	每个成员用对应的 $v_i$ 初始化，此构造函数是 explicit 的
<code>make_tuple(v1,v2,...,vn)</code>	返回一个用给定值初始化的 tuple
<code>t1 == t2</code>	当两个 tuple 具有相同数量的成员且成员对应相等时，两个 tuple 相等。
<code>t1 != t2</code>	

操作	说明
t1 <b>relop</b> t2	tuple 的关系运算符使用字典序。两个 tuple 必须具有相同数量的成员。使用 <b>&lt;</b> 运算符比较 t1 和 t2
get <i>&lt;i&gt;</i> (t)	返回 t 的第 i 个数据成员的引用；如果 t 是一个左值，结果是一个左值引用，否则右值引用。tuple 的所有成员都是 public 的
tuple_size<tupleType>::value	一个类模板，可以通过一个 tuple 类型来初始化。它有一个名为 value 的 public constexpr static 数据成员，类型为 size_t，表示给定 tuple 类型中成员的数量
tuple_element <i>&lt;i, tupleType&gt;::type</i>	一个类模板，可以通过一个整型常量和一个 tuple 类型来初始化。它有一个名为 type 的 public 成员，表示给定 tuple 类型中指定的成员

## 17.1.1 定义和初始化 tuple

当定义一个 tuple 时，需要指出每个成员的类型

默认构造函数会对每个成员进行值初始化，也可以为每个成员提供一个初始值，但是这个构造函数是 explicit 的，因此必须使用直接初始化语法

```

1 tuple<size_t, size_t, size_t> threeD = {1,2,3};
// 错误
2 tuple<size_t, size_t, size_t> threeD{1,2,3}; // 正确

```

## 访问 tuple 成员

使用名为 `get` 的标准库函数模板来访问 `tuple` 的成员

```
1 auto book = get<0>(item);
```

尖括号中的值必须是整型常量表达式

## 关系和相等运算符

只有两个 `tuple` 具有相同数量的成员时，才可以比较

使用 `tuple` 的相等或不等运算符，对每对成员使用 `==` 运算符必须都是合法的；为了使用关系运算符，对每对成员使用 `<` 必须是合法的

## 17.2 bitset 类型

[4.8]节中介绍了将整型运算对象当做二进制位集合处理的一些内置运算符。标准库还定义了 `bitset` 类，使得位运算的使用更加容易，并且能够处理超过最长整型类型大小的位集合。定义在 `bitset` 头文件中

### 17.2.1 定义和初始化 bitset

类似 `array` 类，`bitset` 具有固定的大小，当我们定义 `bitset` 时，需要声明它包含多少个二进制位

```
1 bitset<32> bitvec(1U);
```

大小必须是一个常量表达式，`bitset` 中二进制位是未命名的，通过位置来访问它们，0是低位

初始化 `bitset` 方法

构造器

描述

构造器	描述
<code>bitset&lt;n&gt; b;</code>	<code>b</code> 有 <code>n</code> 位，每一个位均为0。此构造函数是一个 <code>constexpr</code>
<code>bitset&lt;n&gt; b(u);</code>	<code>b</code> 是 <code>unsigned long long</code> 值 <code>u</code> 的低 <code>n</code> 位的拷贝。如果 <code>n</code> 超过 <code>unsigned long long</code> 的大小，则 <code>b</code> 中超出 <code>unsigned long long</code> 的高位被置为0。此构造函数是一个 <code>constexpr</code>
<code>bitset&lt;n&gt; b(s, pos, m, zero, one);</code>	<code>b</code> 是 <code>string s</code> 从位置 <code>pos</code> 开始 <code>m</code> 个字符的拷贝。 <code>s</code> 只能包含字符 <code>zero</code> 或 <code>one</code> ；如果 <code>s</code> 包含其他字符，构造函数会抛出 <code>invalid_argument</code> 异常。字符 <code>b</code> 中分别保存为 <code>zero</code> 和 <code>one</code> 。 <code>pos</code> 默认为0， <code>m</code> 默认为 <code>string::npos</code> ， <code>zero</code> 默认为'0'， <code>one</code> 默认为'1'
<code>bitset&lt;n&gt; b(cp, pos, m, zero, one);</code>	与上一个构造函数相同，但从 <code>cp</code> 指向的字符数组中拷贝字符，如果为提供 <code>m</code> ，则 <code>cp</code> 必须指向一个C风格字符串。如果提供了 <code>m</code> ，则从 <code>cp</code> 开始必须至少有 <code>m</code> 个 <code>zero</code> 或 <code>one</code> 字符

## 用 `unsigned` 值初始化 `bitset`

当我们使用整型值初始化 `bitset` 时，此值将被转换为 `unsigned long long` 类型并被当做位模式来处理。`bitset` 中的二进制位是此模式的一个副本。

## 从一个 `string` 初始化 `bitset`

我们可以从一个`string`或一个字符数组指针来初始化 `bitset`。字符直接表示位模式（参考构造器）

## 17.2.2 bitset 操作

### bitset 操作

操作	描述
<code>b.any()</code>	<code>b</code> 中是否存在置位的二进制位
<code>b.all()</code>	<code>b</code> 中所有位都置位了吗
<code>b.none()</code>	<code>b</code> 中不存在置位的二进制位吗
<code>b.count()</code>	<code>b</code> 中置位的位数
<code>b.size()</code>	一个 <code>constexpr</code> 函数，返回 <code>b</code> 中的位数
<code>b.test(pos)</code>	若 <code>pos</code> 位置的位是置位的，则返回 <code>true</code> ，否则返回 <code>false</code>
<code>b.set(pos,v)</code> <code>b.set()</code>	将位置 <code>pos</code> 处的位设置为 <code>bool</code> 值 <code>v</code> 。 <code>v</code> 默认为 <code>true</code> 若未传递实参，则将 <code>b</code> 中所有位置位
<code>b.reset(pos)</code> <code>b.reset()</code>	将位置 <code>pos</code> 处的位复位或将 <code>b</code> 中所有位复位
<code>b.flip(pos)</code> <code>b.flip()</code>	改变位置 <code>pos</code> 处的位的状态或改变 <code>b</code> 中每一位的状态
<code>b[pos]</code>	访问 <code>b</code> 中位置 <code>pos</code> 处的位；如果 <code>b</code> 是 <code>const</code> 的，则当该位置位时 <code>b[pos]</code> 返回一个 <code>true</code> ，否则返回 <code>false</code>
<code>b.to_ulong()</code> <code>b.to_ullong()</code>	返回一个 <code>unsigned long</code> 或一个 <code>unsigned long long</code> 值，其位模式与 <code>b</code> 相同。如果 <code>b</code> 中位模式不能放入指定的结果类型，则抛出一个 <code>overflow_error</code> 异常

操作	描述
b.to_string(zero,one)	返回一个string，表示b中的位模式。zero和one的默认值分别为0和1，用来表示b中的0和1
os<<b; is>>b;	将b中二进制位打印为字符1或0，打印到流os 从is读取字符存入b。当下一个字符不是1或0时，或是已经读入b.size()个位时，读取过程停止

## 17.3 正则表达式

正则表达式是一种描述字符序列的方法，是一种极其强大的计算工具

C++正则表达式库RE库，定义在头文件regex中，它包含多个组件

组件	描述
regex	表示有一个正则表达式的类
regex_match	将一个字符序列与一个正则表达式匹配
regex_search	寻找第一个与正则表达式匹配的子序列
regex_replace	使用给定格式替换一个正则表达式
sregex_iterator	迭代器适配器，调用regex_search来遍历一个string中所有匹配的子串
smatch	容器类，保存在string中搜索的结果
ssub_match	string中匹配的子表达式的结果

**regex**类表示一个正则表达式。除了初始化和赋值外，regex还支持其他的操作

## regex\_search 和 regex\_match 的参数

(seq,m,r,mft)  
(seq,r,mft)

在字符序列seq中查找regex对象r中的正则表达式。  
seq可以是一个string、表示范围的一对迭代器以及一个指向空字符结尾的字符数组的指针  
m是一个match对象，用来保存匹配结果的相关细节。m和seq必须具有兼容的类型  
mft是一个可选的regex\_consts::match\_flag\_type值。它们会影响匹配过程。见[表17.13](#)

### 17.3.1 使用正则表达式库

```

1 string pattern("[^c]ei");
2 pattern = "[[:alpha:]]*" + pattern + "
3 regex r(pattern); //构造一个用于查找模式的regex
4 smatch results;
5 string test_str = "receipt freind theif
6 receive";
7 if(regex_search(test_str, results, r))
8     cout << results.str() << endl;

```

首先定义一个string保存希望查找的正则表达式。默认情况下regex使用的正则表达式语言是ECMAScript。使用该string初始化regex对象，smatch对象将会保存匹配位置的细节信息。接下来调用了regex\_search，如果找到匹配子串就返回true。regex\_search在输入序列中只要找到一个匹配子串就会停止查找

## 指定regex对象的选项

当我们定义一个regex或是对一个regex调用assign为其赋予新值时，可以指定一些标志影响regex如何操作

操作	描述
regex r(re) regex r(re,f)	re表示一个正则表达式，它可以是一个string、一个表示字符范围的迭代器对、一个指向空字符结尾的字符数组的指针、一个字符指针和一个计数器或是是一个花括号包围的字符列表。f是指出对象如何处理的标志。f通过下面的值来设置。如果未指定f，其默认值为ECMAScript
r1 = re	将r1中的正则表达式替换为re，re表示一个正则表达式，它可以是另一个regex对象、一个string、一个指向空字符结尾的字符数组的指针或是一个花括号包围的字符列表
r1.assign(re,f)	与赋值运算符效果相同，可选的标志f也与regex的构造函数中对应的参数含义相同
r.mark_count()	r中子表达式的数目[17.3.3]
r.flags()	返回r的标志集

定义regex时指定的标志

定义在regex和regex\_constants::syntax\_option\_type中

标志	描述
icase	在匹配过程中忽略大小写
nosubs	不保存匹配的子表达式
optimize	执行速度优先于构造速度

标志	描述
ECMAScript	使用ECMA-262指定的语法
basic	使用POSIX基本的正则表达式语法
extended	使用POSIX扩展的正则表达式语法
awk	使用POSIX版本的awk语言的语法
grep	使用POSIX版本的grep的语法
egrep	使用POSIX版本的egrep的语法

## 指定或使用正则表达式时的错误

正则表达式不是由C++编译器解释的，是在运行时当一个regex对象被初始化或被赋值一个新模式时，才被“编译”的。如果我们编写的正则表达式存在错误，则在运行时标准库会抛出一个类型为regex\_error的异常

类似标准异常类型，regex\_error有一个what操作来描述发生了什么错误。regex\_error还有一个code成员，返回某个错误类型对应的数值编码

定义在regex和regex\_constants::error\_type中

错误类型	描述
error_collate	无效的元素校对请求
error_ctype	无效的字符类
error_escape	无效的转义字符或无效的尾置转义
error_backref	无效的向后引用
error_brack	不匹配的方括号([或])

错误类型	描述
error_paren	不匹配的小括号((或))
error_brace	不匹配的花括号({或})
error_badbrace	{ }中无效的范围
error_range	无效的字符范围(如[z-a])
error_space	内存不足, 无法处理此正则表达式
error_badrepeat	重复字符(*、?、+、{})之前没有有效的正则表达式
error_complexity	要求的匹配过于复杂
error_stack	栈空间不足, 无法处理匹配

## 正则表达式类和输入序列类型

`regex`对应`char`或者`string`

`wregex`对应`wchar_t`或`wstring`

二者操作完全相同, 唯一差别是`wregex`的初始值必须使用`wchar_t`而不是`char`

匹配和迭代器类型更为特殊, 这些类型的差异不仅在于字符类型, 还在于序列是在标准库`string`中还是在数组中:

- `smatch`表示`string`类型的输入序列
- `cmatch`表示字符数组序列
- `wsmatch`表示宽字符串 (`wstring`) 输入
- `wcmatch`表示宽字符数组

RE库类型必须与输入序列类型匹配

## 如果输入序列类型

## 则使用正则表达式类

string	regex、smatch、ssub_match、sregex_iterator
const char*	regex、cmatch、csub_match、cregex_iterator
wstring	wregex、wsmatch、wssub_match、wsregex_iterator
const wchar_t*	wregex、wcmatch、wbsub_match、wcregex_iterator

### 17.3.2 匹配与Regex迭代器类型

默认的匹配只返回第一个匹配的单词。可以使用sregex\_iterator来获得所有匹配。regex迭代器是一种迭代器适配器，被绑定到一个输入序列和一个regex对象上

#### sregex\_iterator操作

也适用于cregex\_iterator、wsregex\_iterator、wcregex\_iterator

操作	描述
sregex_iterator it(b,e,r) sregex_iterator end;	一个sregex_iterator，遍历迭代器b和e表示的string。它调用sregex_search(b,e,r)将it定位到输入中第一个匹配的位置 sregex_iterator的尾后迭代器
*it it->	根据最后一个调用regex_search的结果，返回一个smatch对象的引用或一个指向smatch对象的指针
++it it++	从输入序列当前匹配位置开始调用regex_search。 前置版本返回递增后迭代器；后置版本返回旧值

操作	描述
<code>it1 == it2</code>	如果两个sregex_iterator都是尾后迭代器，则它们两个相等；两个非尾后迭代器是从相同的输入序列和regex对象构造，则它们相等
<code>it1 != it2</code>	

将一个sregex\_iterator绑定到一个string和一个regex对象时，迭代器自动定位到给定string中第一个匹配的为止（对string和regex调用了regex\_search）

## 使用匹配数据

smatch和ssub\_match允许我们获得匹配的上下文，匹配类型有两个名为prefix和suffix的成员，分别返回表示输入序列中当前匹配之前和之后部分的ssub\_match对象。一个ssub\_match对象有两个名为str和length的成员，分别返回匹配的string和该string的大小

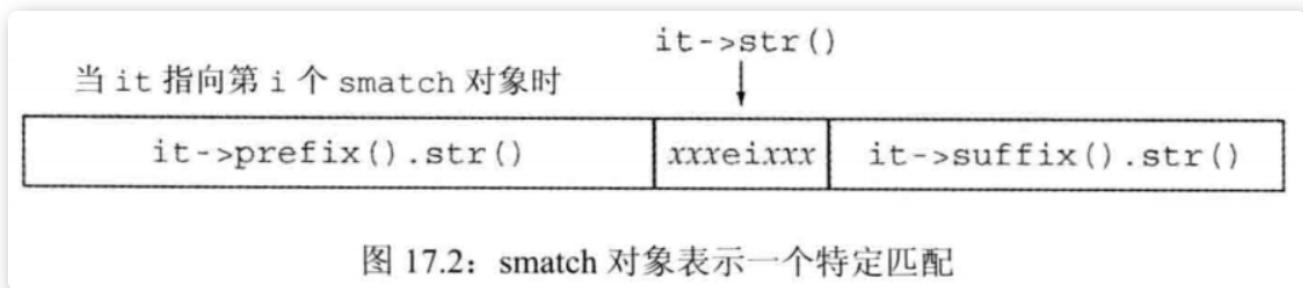


图 17.2: smatch 对象表示一个特定匹配

下述的操作也适用于cmatch、wsmatch、wcmatch和对应的csub\_match、wsub\_match和wssub\_match

操作	描述
<code>m.ready()</code>	如果已经通过调用regex_search或regex_match设置了m，则返回true；否则返回false。如果ready返回false，则对m进行操作时未定义的
<code>m.size()</code>	如果匹配失败，则返回0；否则返回最近一次匹配的正则表达式中子表达式的数目

操作	描述
m.empty()	m.size() == 0 返回true
m.prefix()	一个ssub_match对象，表示当前匹配之前的序列
m.suffix()	一个ssub_match对象，表示当前匹配之后的序列
m.format(...)	见[表17.12]
	在接受一个索引的操作中，n的默认值为0且必须小于m.size() 第一个子匹配（索引为0）表示整个匹配
m.length(n)	第n个匹配的子表达式的大小
m.position(n)	第n个子表达式距序列开始的距离
m.str(n)	第n个子表达式匹配的string
m[n]	对应第n个子表达式的ssub_match对象
m.begin(),m.end() m.cbegin(),m.cend()	表示m中sub_match元素范围的迭代器。

### 17.3.3 使用子表达式

正则表达式的模式中通常包含一个或多个子表达式，一个子表达式是模式的一部分，本身也具有意义。正则表达式语法通常用括号表示子表达式。正则表达式通常用括号表示子表达式

```
1 regex r("([:alnum:]+)\\.
(cpp|cxx|cc)$", regex::icase);
```

该模式中有两个括号括起来的子表达式：

- ([:alnum:]+), 匹配一个或多个字符的序列
- (cpp|cxx|cc), 匹配文件扩展名

```

1 if (regex_search(filename, results, r)){
2     cout << results.str(1) << endl; //打印第一个子
3     表达式
4 }
```

匹配对象除了提供匹配整体相关信息外，还提供访问模式中每个子表达式的能力。**子匹配是按位置来访问的。第一个子匹配位置为0，表示整个模式对应的匹配，随后是每个子表达式对应的匹配**

适用于ssub\_match、csub\_match、wssub\_match、wcsub\_match

操作	描述
matched	一个public bool 数据成员，指出此ssub_match是否匹配了
first second	public数据成员，指向匹配序列首元素和尾后位置的迭代器。如果未匹配，则first和second是相等的
length()	匹配的大小
str()	返回一个包含输入中匹配部分的string
s = ssub	将ssub_match对象ssub转换为string对象s。等价于s=ssub.str()

### 17.3.4 使用regex\_replace

正则表达式不仅用在查找一个给定序列的时候，还用在想将找到的序列替换为另一个序列的时候

希望在输入序列中查找并替换一个正则表达式时，可以调用  
regex\_replace

操作	描述
<pre>m.format(dest,fmt,mft) m.format(fmt,mft)</pre>	<p>使用格式字符串fmt生成格式化输出，匹配在m中，可选的match_flag_type标志在mft中。第一个版本写入迭代器dest指向的目的位置并接受fmt参数，可以是一个string也可以是表示字符数组中范围的一对指针。第二个版本返回一个string，保存输出，并接受fmt参数，可以是一个string，也可以是一个指向空字符结尾的字符数组的指针。mft的默认值为format_default</p>
<pre>regex_replace(dest,seq,r,fmt,mft) regex_replace(seq,r,fmt,mft)</pre>	<p>遍历seq，用regex_search查找与regex对象r匹配的子串。使用格式字符串fmt和可选的match_flag_type标志来生成输出。第一个版本将输出写到迭代器dest指向的位置，并接受一对迭代器seq表示范围。第二个版本返回一个string，保存输出，且seq即可以是一个string也可以是一个指向空字符结尾的字符数组的指针。在所有情况下，fmt既可以是一个string也可以是一个指向空字符结尾的字符数组的指针，且mft默认为match_default</p>

表 17.13

match_default	等价于 format_default
match_not_bol	不将首字符作为行首处理
match_not_eol	不将尾字符作为行尾处理
match_not_bow	不将首字符作为单词首处理
match_not_eow	不将尾字符作为单词尾处理
match_any	如果存在多于一个匹配，则可返回任意一个匹配
match_not_null	不匹配任何空序列
match_continuous	匹配必须从输入的首字符开始
match_prev_avail	输入序列包含第一个匹配之前的内容
format_default	用 ECMAScript 规则替换字符串
format_sed	用 POSIX sed 规则替换字符串
format_no_copy	不输出输入序列中未匹配的部分
format_first_only	只替换子表达式的第一次出现

## 17.4 随机数

新标准出现之前，C/C++都依赖于一个简单的C库函数rand生成随机数。此函数生成均匀分布的伪随机整数，每个随机数的范围在0和一个系统相关的最大值（至少32767）之间

rand函数的问题：一些程序需要不同范围的随机数、一些应用需要随机浮点数、一些程序需要非均匀分布的数

定义在头文件random中的随机数库通过一组协作的类来解决这些问题：**随机数引擎类**和**随机数分步类**

- 一个引擎类可以生成unsigned随机数序列
- 一个分部类使用一个引擎类生成指定类型的、在给定范围内的、服从特定概率分布的随机数

### 17.4.1 随机数引擎和分布

随机数引擎是函数对象类[14.8]，它们定义了一个调用运算符，该运算符不接受参数并返回一个随机unsigned整数。可以通过调用一个随机数引擎对象来生成原始随机数

```

1 default_random_engine e;
2 for(size_t i = 0; i < 10; ++i){
3     cout << e() << " ";
4 }

```

操作	描述
Engine e; Engine e(s);	默认构造函数：使用该引擎类型默认的种子 使用整型值s作为种子
e.seed(s)	使用种子s重置引擎的状态
e.min() e.max()	此引擎可生成的最小值和最大值
Engine::result_type	此引擎生成的unsigned整型类型
e.discard(u)	将引擎推进u步：u的类型为unsigned long long

大多数场合随机数引擎的输出是不能直接使用的，问题在于随机数的值范围通常与我们需要的不符

## 分布类型和引擎

为了得到一个在指定范围内的数，使用一个分布类型的对象

```

1 uniform_int_distribution<unsigned> u(0,9); //均匀分布 0-9 (包含9)
2 default_random_engine e;
3 for(size_t i = 0; i < 10; ++i){
4     cout << u(e) << " ";
5 }

```

分布类型也是函数对象类，接受一个随机数引擎作为参数，传递给分布对象的是引擎本身

# 比较随机数引擎和rand函数

`default_random_engine`对象的输出类似rand输出。随机数引擎生成的`unsigned`整数在一个系统定义的范围内，而rand生成的数的范围在0到`RAND_MAX`之间

## 引擎生成一个数值序列

随机数发生器有一个特性：即使生成的数看起来是随机的，但对一个给定的发生器，每次运行程序它都会返回相同的数值序列。序列不变在调试时非常有用，但是使用随机数发生器的程序也必须注意这一点

```

1  vector<unsigned> good_randVec(){
2      //由于我们希望引擎和分布对象保持状态，因此应该将它们
3      //定义为static的，从而每次调用都生成新的数
4      static uniform_int_distribution<unsigned>
5          u(0,9); //均匀分布 0-9 (包含9)
6      static default_random_engine e;
7      vector<unsigned> ret;
8      for(size_t i = 0; i < 100; ++i){
9          ret.push_back(u(e));
10     }
11 }
```



一个给定的随机数发生器一直会生成相同的随机数序列，一个函数如果定义了局部的随机数发生器，应该将其（包括引擎和分布对象）定义为`static`的。否则每次调用函数都会生成相同的序列

## 设置随机数发生器种子

种子就是一个数值，引擎可以利用它从序列中一个新位置重新开始生成随机数

- 在创建引擎时提供种子

- 调用引擎的seed成员

```

1 default_random_engine e1;
2 default_random_engine e2(2147483646);
3 default_random_engine e3;
4 e3.seed(32767);
5 default_random_engine e4(32767);

```

有相同种子的引擎应该生成相同的序列

## 17.4.2 其他随机数分布

### 生成随机实数

`uniform_real_distribution` 类型可以获取随机浮点数，并让标准库来处理从随机整数到随机浮点数的映射

`reset()` 成员能够重建分布的状态，使得随后对 `d` 的使用不依赖于 `d` 已经生成的值

### 使用分布的默认结果类型

分布类型都是模板，具有单一的模板类型参数，表示分布生成的随机数的类型

每个分布模板都有一个默认模板实参。生成浮点值的分布类型默认生成 `double` 值，生成整型值的分布默认生成 `int` 值

### 生成非均匀分布的随机数

`normal_distribution<> n(4, 1.5)` 正态分布均值为4，方差1.5

`bernoulli_distribution` 是一个普通类，此分布返回一个 `bool` 值，返回 `true` 的概率是一个常数，默认为0.5，  
`bernoulli_distribution b(.55)` 调整概率的方式

# 17.5 IO库再探

## 17.5.1 格式化输入输出

除了条件状态外，每个iostream对象还维护一个格式状态来控制IO如何格式化的细节。格式状态控制格式化的某些方面，如整型值是几进制、浮点值的精度、输出元素的宽度

标准库定义了一组操纵符来修改流的格式状态。一个操纵符是一个函数或一个对象，会影响流的状态，并能用作输入输出运算符的运算对象。类似输入输出运算符，操纵符也返回它所处理的流对象，因此可以在一条语句中组合操纵符和数据

`endl` 就是一个操纵符

### 很多操纵符改变格式状态

操纵符用于两大类输出控制：控制数值的输出形式以及控制补白的数量和位置。

大多数改变格式状态的操纵符都是设置/复原成对的，其中一个用来将格式状态设置为一个新值，另一个用来将其复原

### 控制布尔值的格式

默认情况下，bool值打印为1或0，我们可以使用 `boolalpha` 来覆盖这种格式

```

1 cout << "default bool values:" << true << " " <<
2     false
3     << "\nalpha bool values:" << boolalpha <<
4     true << " " << false << endl;
5 //1 0
6 //true false

```

一旦向流写入了 `boolalpha` 就改变了打印 `bool` 的方式，后续打印 `bool` 值的操作都会打印 `true` 或 `false`

取消这种格式使用 `noboolalpha`

## 指定整型值的进制

使用操纵符 `hex`、`oct`、`dec` 将其修改为十六进制、八进制或改回十进制

## 在输出中指出进制

默认情况下，即使使用了上述的修改进制的操纵符，输出中没有可见的线索指出是几进制

如果需要打印八进制或十六进制，应该使用 `showbase` 操纵符，输出结果中会显示进制；操纵符 `noshowbase` 恢复状态

默认情况下，十六进制值会以小写打印，前导字符也是小写 `x`，使用 `uppercase` 操纵符可以输出大写的 `X` 和 `a-f`

## 控制浮点数格式

可以控制浮点数输出三种格式：

- 以多高精度打印浮点值
- 数值是打印为十六进制、定点十进制还是科学计数法
- 对于没有小数部分的浮点值是否打印小数点

默认情况下，浮点值按六位数字精度打印；如果浮点值没有小数部分，则不打印小数点，非常大和非常小的值打印为科学计数法

## 指定打印精度

默认情况下，精度会控制打印的数字的总数（舍入而非截断）

调用 `IO` 对象的 `precision` 成员或使用 `setprecision` 操纵符来改变精度

- precision是重载的
  - 接受一个int值，将精度设为此值，并返回旧精度值
  - 不接受参数，返回当前精度
- setprecision接受一个参数用来设置精度

```
1 cout.precision(12);
2 cout << setprecision(3);
```

操纵符	作用
boolalpha noboolalpha	将true和false输出为字符串 将true和false输出为1,0
showbase noshowbase	对整型值输出表示进制的前缀 不生成表示进制的前缀
showpoint noshowpoint	对浮点值总是显示小数点 只有当浮点值包含小数部分时才显示小数点
showpos noshowpos	对非负数显示+ 对非负数不显示+
uppercase nouppercase	在十六进制中打印0X，科学计数法中打印E 在十六进制中打印0x，科学计数法中打印e
dec hex oct	整型值显示为十进制 整型值显示为十六进制 整型值显示为八进制
left right internal	在值的右侧添加填充字符 在值得左侧添加填充字符 在符号和值之间添加填充字符
fixed scientific hexfloat defaultfloat	浮点值显示为定点十进制 浮点值显示为科学计数法 浮点值显示为十六进制 重置浮点值显示为十进制

操纵符	作用
unitbuf nounitbuf	每次输出操作后都刷新缓冲区 恢复正常缓冲区刷新方式
skipws noskipws	输入运算符跳过空白 输入运算符不跳过空白符
flush ends endl	刷新ostream缓冲区 插入空字符，然后刷新ostream缓冲区 插入换行，然后刷新ostream缓冲区

## 指定浮点数计数法

`scoentific`、`fixed` 等等操作符

```
1 cout << scientific << 10 * sqrt(2.0) << fixed <<
hexfloat << defaultfloat
```

## 17.5.2 未格式化的输入/输出操作

标准库还提供了一组底层操作，支持未格式化IO。这些操作允许我们将一个流当做一个无解释的字节序列来处理

### 单字节操作

有几个未格式化操作每次一个字节的处理流。它们会读取而不是忽略空白符。例如未格式化IO操作get和put读取和写入一个字符：

```
1 char ch;
2 while(cin.get(ch))
3     cout.put(ch);
4 //此程序保留输入中的空白符，其输出与输入完全相同
```

## 单字节底层IO操作

is.get(ch)	从istream is 读取下一个字节存入字符ch中。返回is
os.put(ch)	将字符ch输出到ostream os， 返回os
is.get()	将is的下一个字节作为int返回
is.putback(ch)	将字符ch放回is， 返回is
is.unget()	将is向后移动一个字节， 返回is
is.peek()	将下一个字节作为int返回， 但不从流中删除它

一般情况下，在读取下一个值之前，标准库保证可以退回最多一个值。标准库不保证在中间不进行读取操作的情况下能连续调用putback或unget

函数peek和无参的get都以int类型返回一个字符，为什么不直接返回char？

其中一个原因是：可以返回文件尾标记。使用char范围中的每个值表示一个真实字符，因此，取值范围内没有额外的值可以用来表示文件尾

返回int的函数将它们要返回的字符先转换为unsigned char，然后再将结果提升到int。因此即使字符集中有字符映射到负值，这些操作返回的int也是正值。标准库使用负值表示文件尾，这样可以保证与任何合法字符的值都不同

头文件cstdio定义了一个名为EOF的const，可以用它来检测从get返回的值是否是文件尾

```

1 int ch;
2 while((ch = cin.get()) != EOF){
3     cout.put(ch);
4 }

```

## 多字节操作

一些未格式化IO操作一次处理大块数据。如果速度是考虑的重点的话，这些操作非常重要。

### 多字节底层IO操作

is.get(sink,size,delim)	从is中读取最多size个字节，并保存在字符数组中，字符数组的其实地址有sink给出。读取过程直至遇到字符delim或读取了size个字节或遇到文件尾时停止。如果遇到了delim，则将其留在输入流中，不读取出来存入sink
is.getline(sink,size,delim)	与接受三个参数的get版本类似，但会读取并丢弃delim
is.read(sink,size)	读取最多size个字节，存入字符数组sink中，返回is
is.gcount()	返回上一个未格式化读取操作从is读取的字节数
os.write(source,size)	将字符数组source中的size个字节写入os，返回os
is.ignore(size,delim)	读取并忽略最多size个字符，包括delim。与其他未格式化函数不同，ignore有默认参数：size默认为1，delim默认为文件尾

## 17.5.3 流随机访问

各种流类型通常都支持对流中数据的随机访问。我们可以重定位流，使跳过一些数据，首先读取最后一行，然后读取第一行。标准库定义了一对函数来定位（seek）到流中给定的位置，以及告诉（tell）我们当前位置

虽然标准库为所有流都定义了seek和tell函数，但是是否会做有意义的事情依赖于绑定到那个设备。大多数系统中，绑定到cin、cout、cerr、clog的流不支持随机访问。对这些流可以调用seek和tell，但是运行时会出错，将流置于一个无效状态



istream和ostream类型通常不支持随机访问，所以下述内容只适用于fstream和sstream类型

### seek和tell函数

为了支持随机访问，IO类型维护一个标记来确定下一个读写操作要在哪里进行。提供了两个函数：

- 通过将标记seek到一个给定位置来重定位它
- tell标记的当前位置

操作	描述
tellg() tellp()	返回一个输入流中（tellg）或输出流中（tellp）标记的当前位置
seekg(pos) seekp(pos)	在一个输入流或输出流中将标记重定位到给定的绝对地址。pos通常是前一个tellg或tellp返回的值

操作	描述
seekp(off,from) seekg(off,from)	在一个输入流或输出流中将标记定位到from之前或之后off个字符，from可以是下列值之一 - beg, 偏移量相对于流开始位置 - cur, 偏移量相对于流当前位置 - end, 偏移量相对于流结尾位置

## 第十八章 用于大型程序的工具

大规模应用程序的特殊要求包括：

- 在独立开发的子系统之间协同处理错误的能力
- 使用各种库进行协同开发的能力
- 对比较复杂的应用概念建模的能力

### 18.1 异常处理

异常处理机制允许程序中独立开发的部分能够在运行时就出现的问题进行通信并做出相应的处理。异常使得我们能够将问题的检测与解决过程分离开来。程序的一部分负责检测问题的出现，然后解决该问题的任务传递给程序的另一部分。检测环节无须知道问题处理模块的所有细节，反之亦然。

#### 18.1.1 抛出异常

通过抛出一条表达式来引发一个异常。

被抛出的表达式的类型以及当前的调用链共同决定了哪段处理代码将被用来处理该异常。被选中定的处理代码是在调用链中与抛出对象类型匹配的最近的处理代码，根据抛出对象的类型和内容，程序的异常抛出部分将会告知异常处理部分到底发生了何种错误

当执行一个throw时，跟在throw后面的语句将不再被执行。程序的控制权从throw转移到与之匹配的catch模块。该catch可能是同一个函数中的局部catch，也可能位于直接或间接调用了发生异常的函数的另一个函数中。控制权的转移有两个重要的含义：

- 沿着调用链的函数可能会提早退出
- 一旦程序开始执行异常处理代码，则沿着调用链创建的对象将被销毁

## 栈展开

当抛出一个异常后，程序暂停当前函数的执行过程并立即开始寻找与异常匹配的catch字句。当throw出现在一个try语句块内时，检查与该try块关联的catch字句。如果找到了匹配的catch，就是用该catch处理异常；如果没找到，且在该try语句嵌套在其他try块中，则继续检查与外层try匹配的catch。如果还是找不到，则退出当前函数，在调用当前函数的外层函数中继续寻找

上述过程被称为**栈展开**过程。栈展开过程沿着嵌套函数的调用链不断查找，直到找到了与异常匹配的catch子句为止；或者一直找不到匹配的catch，则退出主函数后查找过程终止。

假设找到了catch，在执行完catch块之后，找到与try块关联的最后一个catch子句后的点，从这里开始继续执行

如果没找到匹配的catch子句，程序将退出。当找不到匹配的catch时，程序将调用 `terminate` 负责终止程序的运行过程

## 栈展开过程中对象被自动销毁

如果栈展开过程中退出了某个块，编译器将负责确保在这个块中创建的对象能被正确地销毁。如果某个局部对象的类型是**类类型**，则该对象的析构函数将被自动调用。

如果异常出现在构造函数中，可能某个对象只构造了一部分，也要确保已构造的成员能被正确地销毁

如果异常发生在数组或标准库容器的元素初始化过程中，可能已经构造了一部分元素，也要保证这部分元素被正确地销毁

## 析构函数与异常

析构函数总是会被执行，但是函数中负责释放资源的代码却可能被跳过，这对如何阻止程序结构有重要影响。如果一个块分配了资源，那么负责释放这些资源的代码之前发生了异常，则释放资源的代码将不会被执行；另一方面，类对象分配的资源将由类的析构函数负责释放。因此，**如果使用类控制资源的分配，就能确保无论函数正常结束还是遭遇异常，资源都能被正确地释放**

析构函数在栈展开的过程中执行，这影响我们编写析构函数的方式。在栈展开过程中，已经引发了一场但是还未处理它。如果异常没有被正确捕获，则**terminate**函数被系统调用。因此，**出于栈展开可能使用析构函数的考虑，析构函数不应该抛出不能被它自身处理的异常**。所有标准库类型都能确保它们的析构函数不会引发异常

## 异常对象

**异常对象**是一种特殊的对象，编译器使用**异常抛出表达式**来对**异常对象**进行**拷贝初始化**。因此，**throw**语句中的表达式必须拥有完全类型[7.3.3]，并且如果表达式是类类型，则相应的类必须含有一个可访问的析构函数和一个可访问的拷贝或移动构造函数。如果该表达式是数组类型或函数类型，则表达式将被转换成与之对应的指针类型

异常对象位于编译器管理的空间中，编译器确保无论最终调用的是哪个**catch**子句都能访问该空间。当异常处理完毕后，异常对象被销毁

抛出一个指向局部对象的指针几乎肯定是一种错误的行为，因为退出块时局部对象使用的内存是要被释放的

当我们抛出一条表达式时，该表达式的**静态编译时类型**决定了异常对象的类型，如果一条**throw**表达式解引用一个基类指针，而该指针实际指向的是派生类对象，则抛出的对象将被切掉一部分，只有基类部分被抛出

## 18.1.2 捕获异常

catch子句中的异常声明看起来像是只包含一个形参的函数形参列表。如果catch无须访问抛出的表达式的话，我们可以忽略捕获形参的名字

**声明的类型决定了处理代码所能捕获的异常类型。必须是完全类型（有定义），可以是左值引用，但不能是右值引用**

当进入一个catch语句后，通过异常对象初始化异常声明中的参数。与函数参数类似，如果是非引用则参数是异常对象的一份副本，如果是引用类型，则是异常对象的一个别名

如果catch的参数是基类类型，则可以使用派生类类型的异常对象对其进行初始化。如果catch的参数是非引用类型，则异常对象将被切掉一部分；如果是基类的引用，则该参数将以常规方式绑定到异常对象上

异常声明的静态类型将决定catch语句所能执行的操作。如果catch的参数是基类类型，则catch无法使用派生类特有的任何成员



通常情况下，如果catch接受的异常与某个继承体系有关，则最好将该catch的参数定义成引用类型

### 查找匹配的处理代码

搜寻catch语句的过程中，最终找到的catch未必是最佳匹配，而是第一个匹配的catch语句，越是专门的catch越应该置于整个catch列表的前端

异常和catch异常声明的匹配规则受到更多的限制

- 允许从非常量向常量的类型转换
- 允许从派生类向基类的类型转换
- 数组被转换成指向数组元素类型的指针，函数被转换成指向该函数类型的指针

## 重新抛出

有时一个单独的catch不能完整地处理某个异常。一条catch语句通过重新抛出的操作将异常传递给另外一个catch语句，这里的重新抛出仍然是一条throw语句，只不过不包含任何表达式

```
1 throw ;
```

空的throw语句只能出现在catch语句或catch语句直接或间接调用的函数之内。如果在处理代码之外碰到了空throw语句，编译器将调用terminate。

一个重新抛出语句并不指定新的表达式，而是将当前的异常对象沿着调用链向上传递。

如果catch语句改变其参数的内容，然后重新抛出异常，只有在catch语句的异常声明是引用类型时改变才会保留并继续传播

## 捕获所有异常的处理代码

为了一次捕获所有异常，我们使用省略号作为异常声明，这样的处理代码称为**捕获所有异常的处理代码**

catch(...)通常与重新抛出一起使用，其中catch执行当前局部能完成的工作，随后重新抛出

### 18.1.3 函数 try 语句块与构造函数

程序执行的任何时刻都可能发生异常，特别是异常可能发生在处理构造函数初始值的过程中。构造函数在进入函数体之前先执行初始值列表。因为在初始值列表抛出异常时构造函数体内的try语句块还未生效，所以构造函数体内的catch语句无法处理构造函数初始值列表抛出的异常

要想处理构造函数初始值抛出的异常，必须将构造函数写成**函数try语句块**。函数try语句块使得一组catch语句既能处理构造函数体（或析构函数体），也能处理构造函数的初始化过程（或析构函数的析构过程）

```

1 template <typename T>
2 Blob<T>::Blob(std::initializer_list<T> il) try :
3     data(std::make_shared<std::vector<T>>(il)){
4
5     }catch(const std::bad_alloc &e){
6         handle_out_of_memory(e);
7     }

```

关键字try出现在表示构造函数初始值列表的冒号以及表示构造函数体的花括号之前。与这个try关联的catch既能处理构造函数体抛出的异常，也能处理成员初始化列表抛出的异常

初始化构造函数的参数时也可能发生异常，这异常不属于函数try语句块的一部分。函数try语句块只能处理构造函数开始执行后发生的异常，参数初始化过程的异常属于调用表达式的一部分，并将在调用者所在的上下文中处理

## 18.1.4 noexcept异常说明

- 知道函数不会抛出异常有助于简化调用该函数的代码
- 编译器确认函数不会抛出异常，它就能执行某些特殊的优化操作（这些优化操作并不适用于可能出错的代码）

C++11新标准中，通过 `noexcept` 说明指定某个函数不会抛出异常，紧跟在函数的参数列表后面

```

1 void recoup(int) noexcept; //不会抛出异常
2 void alloc(int); //可能抛出异常

```

`noexcept` 要不出现在所有的声明语句和定义中，要么不出现。

`noexcept`的出现位置：

- 该说明应该在函数的尾置返回类型之前
- 可以在函数指针的声明和定义中指定`noexcept`
- 在`typedef`或类型别名中则不能出现`noexcept`

- 成员函数中，`noexcept`说明符需要跟在`const`及引用限定符之后，而在`final`、`override`或虚函数的`=0`之前

## 违反异常说明

编译器并不会在编译时检查`noexcept`说明。所以一个函数在有`noexcept`的同时又含有`throw`语句或者调用了可能抛出异常的其他函数，编译器将顺利编译通过，并不会因为这种违反异常说明的情况而报错

一旦一个`noexcept`函数抛出了异常，程序就会调用`terminate`以确保不在运行时抛出异常的承诺

因此`noexcept`可以用在两种情况下：

- 确认函数不会抛出异常
- 根本不知道如何处理异常

## 异常说明的实参

`noexcept`接受一个可选的、能转换为`bool`类型的实参，如果实参是`true`，则函数不会抛出异常；如果是`false`，则可能抛出异常

```
1 void recoup(int) noexcept(true);
2 void alloc(int) noexcept(false);
```

## noexcept运算符

`noexcept`说明符的实参常与`noexcept`运算符混合使用

`noexcept`是一个一元运算符，返回值是一个`bool`类型的右值常量表达式，用于表示给定的表达式是否会抛出异常（和`sizeof`类似，`noexcept`也不会求其运算对象的值）

```
1 noexcept(recoup(i)); //返回true，因为recoup被定义为
                        //不抛出异常
```

更普通的形式是

1 `noexcept(e)`

当e调用的所有函数都做了不抛出说明且e本身不含有throw语句时，上述表达式为true，否则为false

1 `void f() noexcept(noexcept(g));` //保证f和g的异常说明一致

## 异常说明与指针、虚函数和拷贝控制

`noexcept`不属于函数类型的一部分，但是仍旧会影响函数的使用

**函数指针及该指针所指的函数必须具有一致的异常说明。**如果为某个指针做了不抛出异常的声明，则该指针只能指向不抛出异常的函数；如果我们显式或隐式的说明了指针可能抛出异常，则该指针可以指向任何函数（包括承诺了不抛出异常的函数）

1 `void (*pf1)(int) noexcept = recoup;` //二者都承诺不抛出异常  
 2 `void (*pf2)(int) = recoup;` //此时也是可以的  
 3 `pf1 = alloc;` //错误  
 4 `pf2 = alloc;` //正确

**如果一个虚函数承诺它不会抛出异常，则后续派生出来的虚函数也必须做出同样的承诺；如果基类的虚函数允许抛出异常，则派生类的对应函数既可以允许抛出也可以允许不抛出**

```

1 class Base{
2     public:
3         virtual double f1(double) noexcept;
4         virtual int f2() noexcept(false);
5         virtual void f3();
6     };
7 class Derived : public Base{
8     public:
9         double f1(double); //错误: 基类承诺不抛出异常
10        int f2() noexcept(false);
11        void f3() noexcept;
12    };

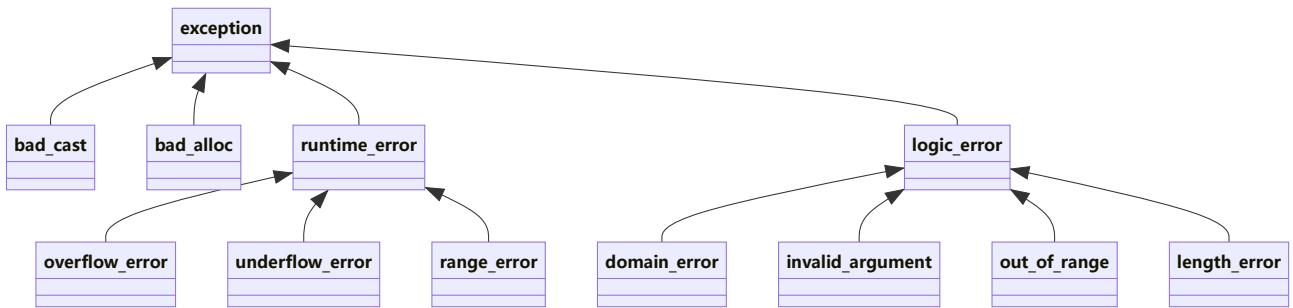
```

当编译器合成拷贝控制成员时，同时也生成一个异常说明。如果对所有成员和基类的所有操作都承诺了不会抛出异常，则合成的成员是noexcept的。如果合成成员调用的任意一个函数可能抛出异常，则合成的成员是noexcept(false)。

如果我们定义了一个析构函数但是没有为它提供异常说明，则编译器将合成一个。合成的异常说明将与假设编译器为类合成析构函数时所得的异常说明一致。

## 18.1.5 异常类层次

标准库异常类构成了下述的继承体系



类型 `exception` 仅仅定义了拷贝构造函数、拷贝赋值运算符、虚析构函数和 `what` 虚成员

类 `exception`、`bad_cast` 和 `bad_alloc` 定义了默认构造函数。类 `runtime_error` 和 `logic_error` 没有默认构造函数，但是有一个接受C风格字符串或标准string类型实参的构造函数，提供错误信息

在这些类中 `what` 负责返回用于初始化异常对象的信息。`what` 是虚函数，所以当捕获基类引用时，对 `what` 函数的调用将执行与异常对象动态类型对应的版本

异常类继承体系中位于最顶层的通常是 `exception`，`exception` 的含义是某处出错了，至于错误的具体细节未做描述

第二层将 `exception` 划分为两大类：运行时错误和逻辑错误

- 运行时错误：只有在程序运行时才能检测到的错误
- 逻辑错误：在程序代码中发现的错误

## 18.2 命名空间

命名空间为防止名字冲突提供了更加可控的机制，命名空间分隔了全局命名空间，其中每个命名空间是一个作用域

## 18.2.1 命名空间定义

一个命名空间定义包含两部分：关键字 `namespace` + 命名空间的名字

命名空间的名字后是一系列花括号括起来的声明和定义。只要能出现在全局作用域中的声明就能置于命名空间内，主要包括：

- 类
- 变量（及其初始化操作）
- 函数（及其定义）
- 模板
- 其他命名空间

**命名空间结束后无须分号**

**每个命名空间都是一个作用域**

位于命名空间内的其他成员及其内嵌作用域能够直接访问定义在命名空间的名字，位于命名空间之外的代码则必须明确指出所用的名字属于哪个命名空间

**命名空间可以是不连续的**

命名空间可以定义在几个不同的部分

**全局命名空间**

全局命名空间以隐式的方式声明，并且在所有程序中都存在。

```
1 ::member_name;
```

这种方式表示一个全局命名空间中的成员

## 嵌套的命名空间

命名空间可以嵌套

## 内联命名空间

与普通的嵌套命名空间不同，内联命名空间中的名字可以被外层命名空间直接使用

定义内联命名空间的方式是在关键字namespace前添加关键字

inline

```
1 inline namespace FifthEd{
2     //
3 }
4 namespace FirthEd{ //隐式内联
5     class Query_base{};
6 }
```

关键字inline必须出现在命名空间第一次定义的地方，后续再打开命名空间的时候可以写inline也可以不写

当应用程序的代码在一次发布和另一次发布之间发生了改变时，常常会用到内联命名空间。例如

```
1 namespace FourthEd{
2     class Item_base{};
3     class Query_base{};
4 }
```

命名空间cplusplus\_primer将同时使用这两个命名空间，假设命名空间定义在同名的头文件中

```

1 namespace cplusplus_primer{
2     #include "FifthEd.h";
3     #include "FourthEd.h";
4 }
```

因为FifthEd是内联的，所以形如cplusplus\_primer::的代码可以直接获取FifthEd的成员；如果想使用早期版本就需像其他嵌套的命名空间一样加上完整的外层命名空间名字

## 未命名的命名空间

未命名的命名空间是指**关键字namespace**后紧跟花括号括起来的一系列声明语句，为命名的命名空间中定义的变量拥有**静态生命周期**：它们在第一次使用前创建，并直到程序结束才销毁

一个未命名的命名空间可以在某个给定的文件内不连续，但是不能跨越多个文件。每个文件定义自己的未命名的命名空间，如果两个文件都有未命名的命名空间则二者相互无关。两个空间中可以定义相同的名字，但是是不同的实体。如果一个头文件定义了未命名的命名空间，则该命名空间中定义的名字将在每个包含了头文件的文件中对应不同的实体

未命名的命名空间中名字的作用域与该命名空间所在的作用域相同，如果未命名的命名空间定义在文件的最外层作用域中，则该命名空间中的名字一定要与全局作用域中的名字有所区别

```

1 int i;
2 namespace{
3     int i;
4 }
5 i = 10; //二义性
```

一个未命名的命名空间也能嵌套在其他命名空间中，此时可以通过外层命名空间的名字来访问

```

1 namespace local{
2     namespace{
3         int i;
4     }
5 }
6 local::i = 42; //正确

```

## 18.2.2 使用命名空间成员

### 命名空间的别名

命名空间的别名使得我们能够为命名空间的名字设定一个短的同义词

```

1 namespace cplusplus_primer{}
2 namespace primer = cplusplus_primer;

```

命名空间的别名也可以指向一个嵌套的命名空间

```

1 namespace Qlib = cplusplus_primer::QueryLib;
2 Qlib::Query q;

```

一个命名空间可以有多个别名

### using声明：扼要概述

一条using声明语句一次只引入命名空间的一个成员。使得我们清楚地知道程序使用的是哪个名字

using声明引入的名字遵守作用域规则：有效范围从using声明开始，一直到using声明所在的作用域结束。此时外层作用域的同名实体将被隐藏

一条using语句可以出现在全局作用域、局部作用域、命名空间作用域以及类作用域中。在类作用域中，这样的声明语句只能指向基类成员

## using指示

using指示和using声明类似的地方是，可以是用命名空间的简写形式；和using不同的地方是，无法控制哪些名字是可见的，因为所有名字都是可见的

using指示以关键字using开始，后面是关键字namespace以及命名空间的名字

using指示可以出现在全局作用域、局部作用域、命名空间作用域中，不能出现在类作用域中

using指示使得特定命名空间中的所有名字都可见

简写的名字从using指示开始，一直到using指示所在的作用域结束

## using指示与作用域

using声明引入的作用域与using声明语句本身的作用域一致。

using指示具有将命名空间成员提升到包含命名空间本身和using指示的最近作用域（using指示所属作用域？）的能力

## 头文件与using声明或指示

头文件如果在其顶层作用域中含有using指示或using声明，则会将名字注入到所有包含了该头文件的文件中。

### 18.2.3 类、命名空间与作用域

对命名空间内部名字的查找遵循常规的查找规则：由内向外一次查找每个外层作用域。外层作用域也可能是一个或多个嵌套的命名空间，直到最外层的全局命名空间查找过程终止

对于命名空间中的类来说，常规的查找规则仍然适用：当成员函数使用某个名字时，首先在该成员中进行查找，然后在类中查找（包括基类），接着在外层作用域中查找，这是一个或几个外层作用域可能就是命名空间

## 实参相关的查找与类类型形参

对下述程序

```
1 std::string s;
2 std::cin >> s;
3 //等价于
4 operator>>(std::cin, s);
```

operator>>定义在标准库string中，string又定义在命名空间std中，但是不用std::限定符和using声明就可以调用operator>>

对于命名空间中名字的隐藏规则来说有一个例外，使得可以直接访问输出运算符。这个例外是，当我们给函数传递一个类类型的对象时，除了在常规的作用域查找外还会查找实参类所属的命名空间，这一例外对于传递类的引用或指针的调用同样有效（所以这里除了查找运算符的时候还在std命名空间中查找了？√）

当编译器发现对operator>>的调用时，首先在当前作用域中寻找合适的函数，接着查找输出语句的外层作用域，随后，因为>>表达式的形参是类类型的，所以编译器还会查找cin和s的类所属的命名空间，即std。当在std中查找时，编译器找到了string的输出运算符函数

## 查找与std::move和std::forward

move函数和forward函数接受右值引用的形参，所以可以匹配任何类型。如果我们定义了一个接受单一形参的move函数，那么必定会产生冲突

因此move以及forward的名字冲突要比其他标准库函数的冲突频繁地多。而且，因为move和forward执行的是非常特殊的类型操作，所以应用程序专门修改函数的原有行为的概率非常小。所以建议使用它们的带限定语的完整版本 `std::move`

## 友元声明与实参相关的查找（与H5标题联合起来看）

当类声明了一个友元时，该友元声明并没有使得友元本身可见

但是，一个另外的未声明的类或函数如果第一次出现在友元声明中，则认为它是最近的外层命名空间的成员

```

1  namespace A{
2      class C{
3          //两个友元，在友元声明之外没有其他的声明
4          //这些函数被隐式地称为命名空间A的成员
5          friend void f2(); //除非另有声明，否则不会被
6          //根据实参的相关查
7          //找规则可以被找到
8      }

```

此时，f和f2都是命名空间A的成员，即使f不存在其他声明，也能通过实参相关的查找规则调用f

```

1  int main(){
2      A::C cobj;
3      f(cobj); //正确，通过A::C中的友元声明找到A::f
4      f2(); //错误，A::f2没有被声明
5  }

```

因为f接受一个类类型的实参，而且f在C说书的命名空间进行了隐式的声明，所以f能被找到；f2没有形参，所以无法被找到

## 18.2.4 重载与命名空间

命名空间对函数匹配过程有两方面影响

- using声明或using指示能将某些函数添加到候选函数集
- 另一个影响比较微妙

### 与实参相关的查找与重载

对于接受类类型实参的函数来说，其名字查找将在实参类所属的命名空间中进行。这对于如何确定候选函数集有影响。我们将在每个实参类（及其基类）所属的命名空间中搜寻候选函数。在这些命名空间中所有与被调用函数同名的函数都将被添加到候选集当中，即使其中某些函数在调用语句处不可见

```

1  namespace NS{
2      class Quote{};
3      void display(const Quote&){}
4  }
5  class Bulk_item : public NS::Quote{};
6  int main(){
7      Bulk_item book1;
8      display(book1);
9      return 0;
10 }
```

传递给display的实参属于类类型Bulk\_item，因此该调用语句的候选函数不仅应该在调用语句所在的作用域中查找，也应该在Bulk\_item及其基类Quote所属的命名空间中查找。命名空间中的函数display(const Quote&)也将被添加到候选函数集中

### 重载与using声明

using声明语句生命的是一个名字，而非一个特定的函数

```
1 using NS::print(int); //错误: 不能指定形参列表
2 using NS::print; //正确
```

该函数的所有版本都将被引入到当前作用域中

一个using声明囊括了重载函数的所有版本以确保不违反命名空间的接口

一个using声明引入的函数将重载该声明语句所属作用域中已有的其他同名函数。如果using声明出现在局部作用域中，则引入的名字将隐藏外层作用域的相关声明。如果using声明所在作用域已经有一个函数与新引入的函数同名且形参列表相同，则该using声明将引发错误。

## 重载与using指示

using指示将命名空间的成员提升到外层作用域中，如果命名空间的某个函数与该命名空间所属作用域的函数同名，则命名空间的函数将被添加到重载集合中

```
1 namespace libs_R_us{
2     extern void print(int);
3     extern void print(double);
4 }
5 void print(const std::string &);
6 using namespace libs_R_us;
7 void fooBar(int ival){
8     print("Value: ");
9     print(ival);
10 }
```

与using声明不同，对于using指示，引入一个与已有函数形参列表完全相同的函数并不会产生错误，此时只需要指明使用的是哪个版本

## 跨越多个using指示的重载

```

1  namespace Aw{
2      int print(int);
3  }
4  namespace Primer{
5      double print(double);
6  }
7  using namespace Aw;
8  using namespace Primer;
9  long double print(long double);
10 int main(){
11     print(1); //Aw::print
12     print(3.1); //Primer::print
13     return 0;
14 }
```

## 18.3 多重继承与虚继承

多重继承是指从多个直接基类中产生派生类的能力。多重继承的派生类继承了所有父类的属性

### 18.3.1 多重继承

```

1  class Bear : public ZooAnimal{};
2  class Panda : public Bear, public Endangered{};
```

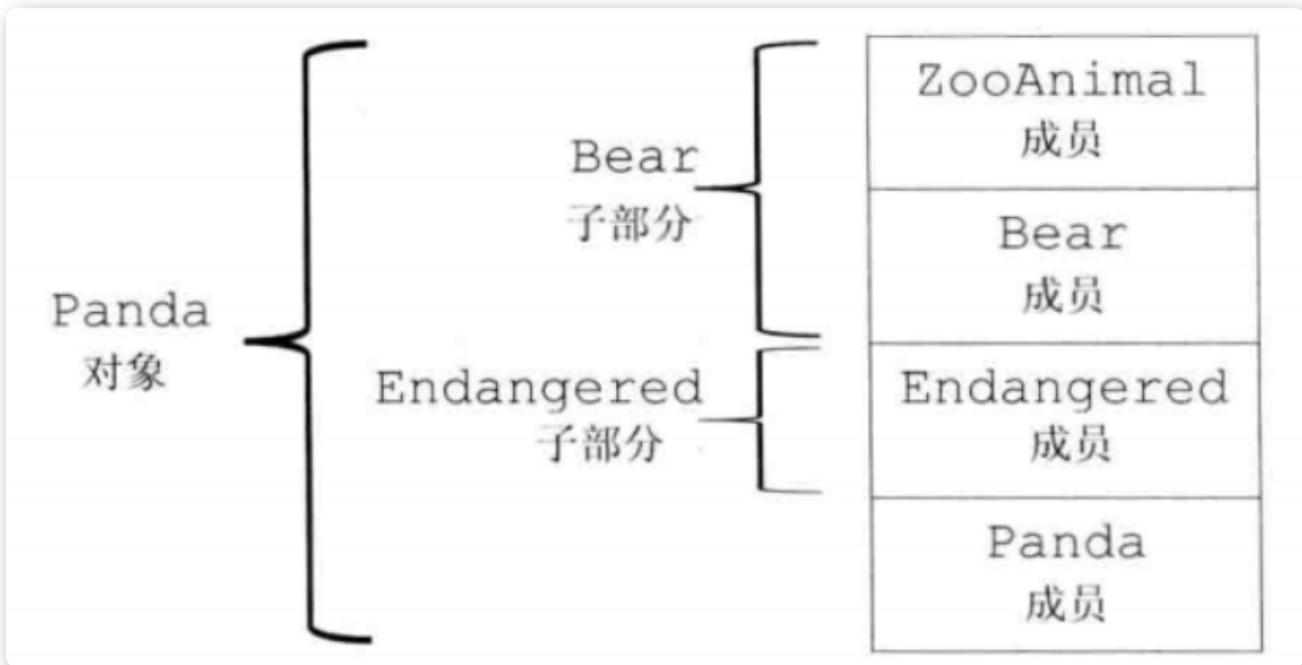
每个基类包含一个可选的访问说明符

与单继承一样，多继承的派生列表页只能包含已经定义过的类，而且这些类不能是final的

派生类能够继承的基类个数C++没有特殊规定，但是一个基类只能在派生列表中出现一次

## 多重继承的派生类从每个基类中继承状态

派生类的对象有每个基类的子对象



## 派生类构造函数初始化所有基类

构造一个派生类对象将同时构造并初始化它的所有基类子对象，多重继承的派生类构造函数也只能初始化它的直接基类

基类的构造顺序与派生类列表中基类的出现顺序保持一致，而与派生类构造函数函数初始值列表中基类的顺序无关

初始化顺序：

1. 首先初始化最终基类
2. 按照派生类列表中的顺序初始化直接基类
3. 初始化派生类

## 继承的构造函数与多重继承

允许派生类从它的一个或多个基类中继承构造函数[15.7.4]，但是如果继承的多个构造函数形参列表完全相同，则出现错误

```
1 struct Base1{
2     Base1() = default;
```

```

3     Base1(const std::string&);
4     Base1(std::shared_ptr<int>);
5 }
6 struct Base2{
7     Base2() = default;
8     Base2(const std::string&);
9     Base2(int);
10 }
11 struct D1 : public Base1, public Base2{
12     using Base1::Base1;
13     using Base2::Base2;
14 };//错误

```

如果一个类从它的多个基类中继承了相同的构造函数，则这个类必须为该构造函数定义它自己的版本：

```

1 struct D2 : public Base1, public Base2{
2     using Base1::Base1;
3     using Base2::Base2;
4     D2(const string& s) : Base1(s), Base2(s){}
5     D2() = default;
6 }

```

## 析构函数与多重继承

派生类的析构函数只负责清除派生类本身分配的资源，派生类的成员及基类都是自动销毁的。合成的析构函数体为空

析构函数的调用顺序正好与构造函数相反

## 多重继承的派生类的拷贝与移动操作

多重继承的派生类如果定义了自己的拷贝/赋值构造函数和赋值运算符，则必须在完整的对象上执行拷贝、移动或赋值操作

只有派生类使用的是合成版本的拷贝、移动或赋值成员时，才会自动对其基类部分执行这些操作，在合成的拷贝控制成员中，每个基类分别使用自己的对应成员隐式地完成构造、赋值或销毁工作

### 18.3.2 类型转换与多个基类

可以令某个可访问基类的指针或引用直接指向一个派生类对象

ZooAnimal、Bear、Endangered类型的指针或引用可以绑定到Panda对象上

编译器不会在派生类向基类的转换中进行比较和选择，在它看来转换到任意一种基类都一样好

```
1 void print(const Bear&);  
2 void print(const Endangered&);  
3 Panda ying_yang("xxx");  
4 print(ying_yang); //二义性错误
```

### 基于指针或引用类型的查找

对象、指针和引用的静态类型决定了我们能够使用哪些成员（决定成员的可见性）

### 18.3.3 多重继承下的类作用域

单继承时，派生类的作用域嵌套在直接基类和间接基类的作用域中。查找过程沿着继承体系自底向上进行，派生类的名字将隐藏基类的同名成员

多重继承中，相同的查找过程在所有直接基类中同时进行。如果名字在多个基类中都被找到，则对改名字的使用具有二义性。对于一个派生类来说，从它的几个基类中分别继承名字相同的成员是完全合法的，只不过在使用这个名字时必须明确指出它的版本

有时即使派生类继承的两个函数形参列表不同也可能发生错误；即使在一个类中是私有的，在另一个类中是公有的或受保护的同样可能发生错误。

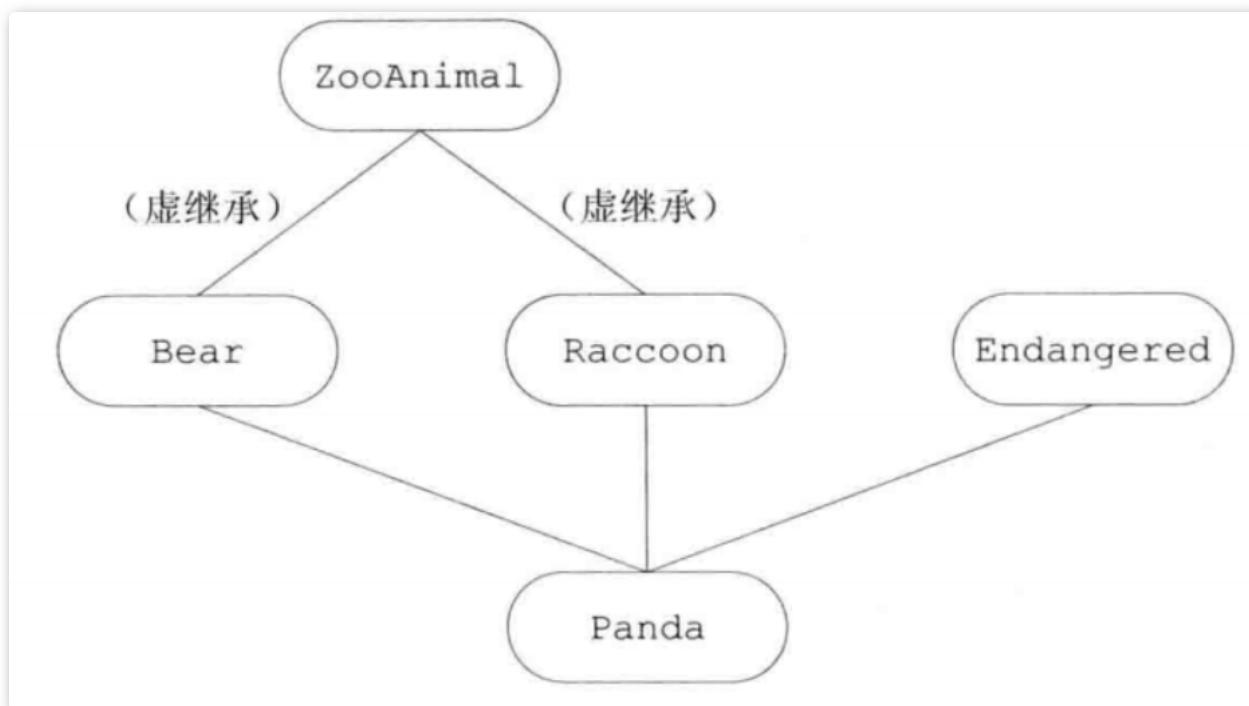
### 18.3.4 虚继承

尽管派生列表中同一个基类只能出现一次，但是派生类实际上也可能多次继承同一个类（如果派生类的两个直接基类继承自同一个间接基类或者派生类继承了直接基类的直接基类）

默认情况下，派生类中含有继承链上每个类对应的子部分，如果某个类在派生过程中继承了多次，则派生类中将包含该类的多个子对象

C++中使用虚继承的机制解决上述问题。虚继承的目的是令某个类做出声明，承诺愿意共享它的基类，其中共享的基类子对象称为虚基类，不论虚基类在继承体系中出现了多少次，在派生类中都只包含唯一一个共享的虚基类子对象

例如：



虚继承有一个不太直观的特征：必须在虚派生的真实需求出现前就已经完成虚派生的操作。当我们定义Panda时才出现了对虚派生的需求，但是如果Bear和Raccoon不是从ZooAnimal虚派生得到的，那么Panda的设计者就不太幸运了

实际的编程过程中，位于中间层次的基类将其继承声明为虚继承一般不会带来什么问题。

虚派生只影响从指定了虚基类的派生类中进一步派生出的类，它不会影响派生类本身

## 使用虚基类

使用关键字 `virtual`

```
1 class Raccoon : public virtual ZooAnimal{};  
2 class Bear : public virtual ZooAnimal{};
```

将ZooAnimal定义为Raccoon和Bear的虚基类

如果某个类指定了虚基类，则该类的派生类仍按常规方式进行

```
1 class Panda : public Bear, public Raccoon,  
public Endangered{};
```

## 支持向基类的常规类型转换

不论基类是不是虚基类，派生类对象都能被可访问基类的指针或引用操作

## 虚基类成员的可见性

该基类的成员可以直接被访问，不会有二义性（因为现在只有一份了）。如果虚基类的成员只被一条派生路径覆盖，则仍然可以直接访问这个被覆盖的成员；如果成员被多于一个基类覆盖，则一般情况下派生类必须为该成员自定义一个版本

例如，类B有一个名为x的成员，D1和D2从B虚继承得到的，D继承了D1和D2，则在D的作用域中，x通过D的两个基类都是可见的。如果我们通过D的对象使用x，有三种可能性：

- 如果D1和D2中都没有x的定义，则x被解析为B的成员，此时不存在二义性，一个D的对象只含有x的一个实例
- 如果x是B的成员，同时是D1和D2中某一个的成员，则同样没有二义性，派生类的x比共享虚基类B的x优先级更高
- 如果在D1和D2中都有x的定义，则直接访问x将产生二义性

与非虚的多继承体系一样，解决这种二义性问题最好的方法是在派生类中为成员自定义新的实例

### 18.3.5 构造函数与虚继承

在虚派生中，虚基类是由最底层的派生类初始化的。以上面的继承关系为例，当创建Panda时，由Panda的构造函数独自控制ZooAnimal的初始化过程。如果以普通初始化的方式，那么虚基类将会在多条继承路径上被重复初始化。

继承体系中的每个类都可能在某个时刻成为“最底层的派生类”。只要我们能创建虚基类的派生类对象，该派生类的构造函数就必须初始化它的虚基类。

例如，当创建一个Bear的对象时，它已经位于派生的最底层，因此Bear的构造函数将直接初始化其ZooAnimal基类部分；

```

1 Bear::Bear(std::string name, bool onExhibit) :
    ZooAnimal(name, onExhibit, "Bear");
2 Raccoon::Raccoon(std::string name, bool
    onExhibit) :
    ZooAnimal(name, onExhibit, "Raccoon");

```

当创建一个Panda对象时，Panda位于派生的最底层并由它负责初始化共享的的ZooAnimal基类部分。即使ZooAnimal不是Panda的直接基类，Panda的构造函数也可以初始化ZooAnimal

```

1 Panda::Panda(std::string name, bool onExhibit)
2 :ZooAnimal(name, onExhibit, "Panda"),
3 Bear(name, onExhibit),
4 Raccoon(name, onExhibit),
5 Endangered(Endangered::critical),
6 sleeping_flag(false){}

```

## 虚继承的对象的构造方式

含有虚基类的对象的构造顺序与一般的顺序稍有区别：首先使用提供给最底层派生类构造函数的初始值初始化该对象的虚基类子部分，接下来按照直接基类在派生列表中出现的次序依次对其进行初始化

如果Panda没有显式地初始化ZooAnimal基类，那么将调用ZooAnimal的默认构造函数

## 构造函数与析构函数的次序

一个类可以有多个虚基类，这些虚的子对象按照它们在派生列表中出现的顺序从左向右依次构造

编译器按照直接虚基类的声明顺序对其进行检查（不是构造），以确定其中是否含有虚基类。如果有，则先构造虚基类，然后按照声明顺序逐一构造其他非虚基类

合成的拷贝和移动构造函数按照完全相同的顺序执行，合成的赋值运算符中的成员也按照该顺序执行。

对象的销毁顺序与构造顺序正好相反

# 第十九章 特殊工具与技术

# 19.1 控制内存分配

## 19.1.1 重载new和delete

当我们使用一条 `new` 表达式时：

```
1 string *sp = new string("a value"); //分配并初始化
一个string对象
2 string *arr = new string[10]; //分配10个默认初始化
的string对象
```

实际执行了三步操作

1. `new` 表达式调用一个名为 `operator new` (或者 `operator new[]`) 的标准库函数。该函数分配一块足够大、原始的、未命名的内存空间以便存储特定类型的对象 (或对象的数组)
2. 编译器运行相应的构造函数以构造这些对象，并为其传入初始值
3. 对象被分配了空间并构造完成，返回一个指向该对象的指针

当我们使用一个 `delete` 表达式删除一个动态分配的对象时：

```
1 delete sp; //销毁*sp, 然后释放sp指向的内存空间
2 delete [] arr; //销毁数组中的元素, 然后释放对应的内存
空间
```

实际执行了两步操作

1. 对 `sp` 所指的对象或者 `arr` 所指的数组中的元素执行对应的析构函数
2. 编译器调用名为 `operator delete` (或者 `operator delete[]`) 的标准库函数释放内存空间

如果应用程序希望控制内存分配过程，则需要自定义operator new 和 operator delete，既可以在全局作用域中定义operator new 和 operator delete函数，也可以将它们定义为成员函数。

当编译器发现一条new表达式或delete表达式后，将在程序中查找可供调用的operator函数。如果被分配（释放）的对象时类类型，则编译器首先在类及其基类作用域中查找，然后再在全局作用域查找匹配的函数，否则使用标准库定义的版本

可以使用作用域运算符令new或delete表达式忽略定义在类中的函数，直接执行全局作用域中的版本。例如::new、 ::delete

## operator new接口和operator delete接口

标准库定义了operator new函数和operator delete函数的8个重载版本，其中前4个可能抛出bad\_alloc异常，后4个版本不会抛出异常

```

1 //这些版本可能抛出异常
2 void *operator new(size_t); //分配一个对象
3 void *operator new[](size_t); //分配一个数组
4 void *operator delete(void*) noexcept; //释放一个对象
5 void *operator delete[](void*) noexcept; //释放一个数组
6 //这些版本承诺不会抛出异常
7 void *operator new(size_t,nothrow_t&) noexcept;
8 void *operator new[](size_t,nothrow_t&)
    noexcept;
9 void *operator delete(void*,nothrow_t&)
    noexcept;
10 void *operator delete[](void*,nothrow_t&)
    noexcept;

```

类型**nothrow\_t**是定义在头文件中的一个**struct**，在这个类型中不包含任何成员。**new**头文件还定义了一个名为**nothrow**的**const**对象，用户可以通过这个对象请求**new**的非抛出版本。与析构函数类似，**operator delete**也不允许抛出异常。当我们重载这些运算符时，必须使用**noexcept**异常说明符指定其不抛出异常

应用程序可以自定义上面函数版本中的任意一个（位于全局作用域或者类作用域中），上述运算符函数定义为类的成员时，它们是隐式静态的（因为一个用在对象构造前，一个用在对象析构后，所以只能是静态的，而且不能操作类的任何成员）

对于**operator new**或者**operator new[]**，返回类型必须是**void\***，第一个形参必须是**size\_t**（存储指定类型对象所需的字节数或者存储数组中所有元素所需的空间）且不能含有默认实参，自定义**operator new**可以提供额外的形参，此时用到这些自定义函数的**new**表达式必须使用**new**的定位形式[12.1.2]将实参传给新增的形参。一般情况下我们可以定义具有任何形参的**operator new**，但是下面这个函数无论如何不能被用户重载：

```
1 void *operator new(size_t, void*); // 只供标准库使用，不能被用户重新定义
```

对于**operator delete**或者**operator delete[]**，它们的返回类型必须是**void**，第一个形参的类型必须是**void\***，指向待释放内存的指针将初始化**void\***形参

当使用**operator delete**或**operator delete[]**定义成类的成员时，该函数可以包含另外一个类型为**size\_t**的形参，该形参的初始值是第一个形参所指对象的字节数。**size\_t**形参可用于删除继承体系中的对象，如果基类有一个虚析构函数，则传递给**operator delete**的字节数将因待删除指针所指对象的动态类型不同而有所区别。而且，实际运行的**operator delete**版本也由对象动态类型决定



new 表达式与**operator new**函数

标准库函数operator new 和 operator delete的名字容易让人误解。和其他operator函数不同（如operator=），这两个函数并没有重载new表达式或delete表达式。实际上我们根本无法自定义new表达式或delete表达式的行为

一条new表达式的执行过程总是先调用operator new函数以获取内存空间，然后在得到的内存空间中构造对象。与之相反，一条delete的执行过程总是先销毁对象，然后调用operator delete函数释放对象所占的空间

改变operator new和operator delete的目的在于改变内存的分配方式

## malloc函数与free函数

cstdlib 头文件

malloc函数接受一个表示待分配字节数的size\_t，返回指向分配空间的指针或者返回0表示分配失败

free函数接受一个void\*，它是malloc返回的指针的副本，free将相关内存返回给系统。free(0)没意义

### 19.1.2 定位new表达式

普通的代码也可以调用operator new和operator delete

应该使用new的定位new形式构造对象。可以使用定位new传递一个地址

```
1 new (place_address) type;
2 new (place_address) type(initializers);
3 new (place_address) type [size];
4 new (place_address) type [size] {braced
    initializer list};
```

其中place\_address必须是个指针，同时在initializers中提供一个以逗号分隔的初始值列表用于构造新分配的对象

当仅通过一个地址值调用时，定位new使用operator new(size\_t,void\*)分配它的内存。该函数不分配任何内存，只是简单地返回指针实参；然后new表达式负责在指定的地址初始化对象以完成整个工作。定位new允许在一个特定的、预先分配的内存地址上构造对象

## 显式地析构函数调用

```
1 string *sp = new string("a value");
2 sp->~string();
```

调用析构函数可以清除给定的对象但是不会释放该对象所在的空间，如果需要的话，可以重新使用该空间

## 19.2 运行时类型识别

运行时类型识别 (run-time type identification,RTTI)的功能由两个运算符实现：

- typeid运算符，用于返回表达式的类型
- dynamic\_cast运算符，用于将基类的指针或引用安全的转换成派生类的指针或引用

当将这两个运算符用于某种类型的指针或引用，并且该类型含有虚函数时，运算符将使用指针或引用所绑定对象的动态类型

特别适用于以下情况：

使用基类对象的指针或引用执行某个派生类操作并且该操作不是虚函数

## 19.2.1 dynamic\_cast运算符

dynamic\_cast运算符的使用形式如下：

```
1 dynamic_cast<type*>(e);
2 dynamic_cast<type&>(e);
3 dynamic_cast<type&&>(e);
```

type必须是类类型（通常应该含有虚函数）。第一种形式e必须是有效指针，第二种形式e必须是左值，第三种形式e不能是左值

e的类型必须符合以下三个条件之一

- e的类型是目标type的公有派生类
- e的类型是目标type的公有基类
- e的类型就是目标type的类型

符合，则类型转换可以成功，否则失败。如果转换目标是指针类型并且失败了，结果为0；如果是引用并且失败了，抛出bad\_cast异常

### 指针类型的dynamic\_cast

Base至少有一个虚函数，有一个指向Base的指针bp，可以在运行时将它转换成指向派生类的指针

```
1 Derived *bp = dynamic_cast<Derived*>(bp);
```



可以对空指针执行dynamic\_cast，结果是所需类型的空指针

### 引用类型的dynamic\_cast

引用类型的dynamic\_cast与指针类型dynamic\_cast在表示错误发生时略有不同（因为不存在所谓空引用），当引用类型的转换失败时，程序抛出一个名为std::bad\_cast的异常（定义在typeinfo头文件）

## 19.2.2 typeid运算符

typeid运算符允许程序向表达式提问：你的对象类型是什么？

`typeid(e)`，e可以是任意表达式或类型的名字，操作结果是一个常量对象的引用，该对象的类型是标准库类型`type_info`（`typeinfo`头文件）或者`type_info`的公有派生类型

typeid可以作用于任意类型的表达式。顶层`const`被忽略，如果表达式是一个引用，则 typeid 返回该引用所引对象的类型。当 typeid 作用于数组或函数时，并不会执行向指针的标准类型转换，即对数组`a`执行`typeid(a)`所得结果是数组类型而非指针类型

运算对象不属于类类型或者是一个不包含任何虚函数的类时， typeid 运算符指示的是运算对象的静态类型。当运算对象是定义了至少一个虚函数的类的左值时， typeid 的结果直到运行时才会求得

### 使用 typeid 运算符

```

1 Derived *dp = new Derived;
2 Base *bp = dp;
3 if(typeid(*bp) == typeid(*dp)){
4     //
5 }
6 if(typeid(*bp) == typeid(Derived)){
7
8 }
```

typeid 应该作用于对象，因此我们使用`*bp`而非`bp`

```

1 if(typeid(bp) == typeid(Derived)){
2
3 }
```

比较的是类型Base和Derived，尽管指针所指类型是一个含有虚函数的类，但是指针本身不是一个类类型对象。类型Base在编译时求值，显然上面的条件永远不会满足

typeid是否需要运行时检查决定了表达式是否会被求值。只有当类型含有虚函数时，编译器才会对表达式求值，反之，返回表达式静态类型（无须对表达式求值也能知道表达式静态类型）

如果表达式的动态类型可能与静态类型不同，必须在运行时对表达式求值以确定返回的类型。这条规则适用于typeid(\*p)，如果p所指类型不含虚函数，则p不必非要有效；否则\*p将会在运行时求值，那么p必须是有效指针；如果p是一个空指针，则typeid(\*p)将抛出bad\_typeid异常

### 19.2.3 使用RTTI

示例，通过比较运算符展示，如果两个对象类型相等且对应数据成员取值相同，则两个对象可以认为是相等的。但是虚函数的基类版本和派生类版本必须具有相同的形参类型，如果定义一个虚函数equal，则函数形参必须是基类的引用，此时equal只能使用基类成员，这时就用到RTTI解决问题了

```

1 class Base{
2     friend bool operator==(const Base&, const
3     Base&);
4     public:
5         //
6     protected:
7         virtual bool equal(const Base&) const;
8 };
9 class Derived:public Base{
10    public:
11        //
12    protected:
13        bool equal(const Base&) const;
14 };

```

## 类型敏感的相等运算符

```

1 bool operator==(const Base& lhs, const Base& rhs)
{
2     return typeid(lhs) == typeid(rhs) &&
3     lhs.equal(rhs);
}

```

如果运算类型不同则返回false，否则运算类型相同，则运算符将工作委托给equal虚函数。这样能够实现多态，能够完成相对应的比较操作

### 虚equal函数

继承体系中的每个类必须定义自己的equal函数。第一件事就是将实参的类型转换为派生类类型

```
1 auto r = dynamic_cast<const Derived&>(rhs);
```

## 19.2.4 type\_info类

type\_info的精确定义随着编译器的不同略有差异。不过C++标准规定type\_info类必须定义在typeinfo头文件中并至少提供以下操作

操作	说明
<code>t1 == t2</code>	如果type_info对象t1和t2表示同一种类型，返回true
<code>t1 != t2</code>	与上述操作类似
<code>t.name()</code>	返回一个C风格字符串，表示类型名字的可打印形式。 类型名字的生成方式因系统而异
<code>t1.before(t2)</code>	返回一个bool值，表示t1是否位于t2之前。before所采用的顺序关系依赖于编译器

type\_info一般作为基类出现，所以它还应该提供一个公有的虚析构函数。当编译器希望提供而外的类型信息时，通常在type\_info的派生类中完成

type\_info没有默认构造，拷贝和移动构造以及赋值运算符都是删除的。所以无法定义type\_info类型对象，也不能type\_info类型赋值。创建type\_info对象的唯一途径是使用typeid运算符

对于name返回值的唯一要求是，类型不同则返回的字符串必须有所区别

## 19.3 枚举类型

枚举类型使一组整型常量组织在一起。枚举类型属于字面值常量类型

C++包含两种枚举：限定作用域或不限定作用域的（C++11引入）

- 限定作用域的枚举类型的一般形式是：首先关键字 enum class (enum struct)，随后是枚举类型名字以及花括号括起来的枚举成员列表

```
1 enum class open_modes{input, output,
append};
```

- 不限定作用域的枚举类型：省略掉关键字class (struct)，枚举类型的名字是可选的

```
1 enum color{red,yellow,green};
2 enum {floatPrec = 6, doublePrec = 10,
double_doublePrec = 10};
```

如果enum是未命名的，则只能在定义该enum时定义它的对象。在enum定义的右侧花括号和分号之间提供声明列表

## 枚举成员

限定作用域的枚举类型中，枚举成员的名字遵循常规的作用域准则，并且在枚举类型的作用域外不可访问；不限定作用域的枚举类型中，枚举成员的作用域与枚举类型本身的作用域相同

```

1 enum color{red,yellow,green};
2 enum stoplight{red,yellow,green}; //错误重复定义了
    枚举成员
3 enum class peppers {red,yellow,green}; //正确：枚
    举成员被隐藏了
4 color eyes = green; //正确：不限定作用域的枚举类型的
    枚举成员位于有效的作用域中
5 peppers p = green; //错误：peppers的枚举成员不在有效
    的作用域中 //color::green在作用域中，但是类型错误
6 color hair = color::red; //正确
7 peppers p2 = peppers::red; //正确：使用
    peppers::red

```

默认情况，枚举值从0开始，依次加1，也可以指定专门的值。枚举值不一定唯一，如果没有显式地提供初始值，则当前枚举成员的值为前一枚举成员的值加1

```

1 enum class intTypes{
2     charTyp = 8, shortTyp = 16, intTyp = 16,
    longTyp = 32, long_longTyp = 64
3 };

```

枚举成员是const，因此初始化枚举成员时必须是常量表达式

## 枚举定义新的类型

初始化enum对象或者为enum对象赋值，必须使用该类型的一个枚举成员或该类型的另一个对象

```

1 open_modes om = 2; //错误: 2不属于类型open_modes
2 om = open_modes::input; //正确

```

不限定作用域的枚举类型的对象或枚举成员自动地转换成整型，因此可以在需要整型值的地方使用它们

```

1 int i = color::red;
2 int j = pepers::red; //错误: 限定作用域的枚举类型不会
    隐式转换

```

## 指定enum的大小

实际上enum是由某种整数类型表示的。可以在enum的名字后加上冒号以及我们想使用的类型

```

1 enum intValues : unsigned long long{
2     charTyp = 255, shortTyp = 65535, intTyp =
    65536,
3     longTyp = 4294967295UL,
4     long_longTyp = 18446744073709661615ULL
5 };

```

默认情况下，是int类型的

对于不限定作用域的枚举类型来说，枚举成员不存在默认类型，但是类型足够大

## 枚举类型的前置说明

enum的前置声明必须指定成员的大小

```

1 enum intValues : unsigned long long; //不限定作用
    域的，必须指定成员类型，因为没有默认大小
2 enum class open_modes; //默认未int

```

enum的声明和定义必须匹配，即所有声明和定义中的成员大小必须一致。而且不能再同一作用域中声明同名的限定作用域的enum名字和不限定作用域的enum名字

## 形参匹配与枚举类型

初始化一个enum对象，必须使用enum类型的另一个对象或枚举成员，因此即使某个整型值与枚举成员值相等，也不能作为函数的enum使用

尽管不能直接将整型值传给enum形参，但是可以将一个不限定作用域的枚举类型的对象或枚举成员传给整型形参

## 19.4 类成员指针

成员指针是指可以指向类的非静态成员的指针。一般情况，指针指向一个对象，但是成员指针指示的是类的成员，而非类的对象

成员指针的类型囊括了类的类型以及成员的类型。当初始化这样一个指针时，我们令其指向类的某个成员，但是不指定该成员所属的对象；直到使用成员指针时，才提供成员所属的对象

```

1 class Screen{
2     public:
3         typedef std::string::size_type pos;
4         char get_cursor() const {return
5             contents[cursor];}
6         char get() const;
7         char get(pos ht, pos wd) const;
8     private:
9         std::string contents;
10        pos cursor;
11        pos height, width;
12    };

```

## 19.4.1 数据成员指针

声明成员指针时也使用\*表示当前声明的是一个指针。与普通指针不同，成员指针还必须包含成员所属的类（这与上面的初始化指针时不指定该成员所属的对象不矛盾？）

```
1 const string Screen::*pdata;
```

指向Screen类的const string成员的指针

当初始化成员指针时，需指定它所指的成员

```
1 pdata = &Screen::contents;
```

C++11中声明成员指针最简单的方法是使用auto或decltype

```
1 auto pdata = &Screen::contents;
```

## 使用数据成员指针

初始化一个成员指针或为成员指针赋值时，指针并未指向任何数据。成员指针指定了成员而非该成员所属的对象，只有解引用成员指针时才提供对象的信息

.\* 和 ->\* 可以解引用指针并获得该对象的成员

```
1 Screen myScreen, *pScreen = &myScreen;
2 auto s = myScreen.*pdata;
3 s = pScreen->*pdata;
```

## 返回数据成员指针的函数

常规的访问控制规则对成员指针同样有效。例如，Screen的contents成员是私有的，因此之前对于pdata的使用必须位于Screen类的成员或友元内部，否则程序将发生错误

返回值为指向该成员的指针的函数如下定义

```
1 class Screen{
2     public:
3         static const std::string Screen::*data()
4             {return &Screen::contents;}
5 };
```

返回指向contents成员的指针

```
1 const string Screen::*pdata = Screen::data();
```

## 19.4.2 成员函数指针

与指向数据成员的指针类似，想要创建一个指向成员函数的指针，使用auto来推断类型比较简单

```
1 auto pmf = &Screen::get_cursor;
```

使用classname::\*的方式声明一个指向成员函数的指针。指向成员函数的指针也需要指定目标函数的返回类型和形参列表。如果成员函数是const成员或者引用成员，则必须将const限定符或引用限定符包含进来

```
1 char (Screen::*pmf2)(Screen::pos, Screen::pos)
2     const;
3 pmf2 = &Screen::get;
```

与普通函数指针不同，成员函数和指向该成员的指针之间不存在自动转换规则

```
1 pmf = &Screen::get;
2 pmf = Screen::get; //错误
```

# 使用成员函数指针

使用 `.*` 或 `->*` 运算符作用于指向成员函数的指针，以调用类的成员函数

```
1 Screen myScreen, *pScreen = &myScreen;
2 (myScreen.*pmf2)(0, 0);
3 (pScreen->*pmf2)(0, 0);
```

## 使用成员指针的类型别名

使用类型别名或`typedef`可以让成员指针更容易理解

```
1 using Action = char (Screen::*)
(Screen::pos, Screen::pos) const;
```

`Action`是某类型的另外一个名字，该类型是指向`Screen`类的常量成员函数的指针，这个成员函数接受两个`pos`形参，返回一个`char`

```
1 Action get = &Screen::get; //get就类似于上述的pmf2
```

可以将指向成员函数的指针作为某个函数的返回类型或形参类型，指向成员的指针形参也可以拥有默认实参

```
1 Screen& action(Screen&, Action = &Screen::get);
```

```
1 Screen myScreen;
2 action(myScreen); //默认实参
3 action(myScreen, get); //使用的是定义的Action get
4 action(myScreen, &Screen::get); //显式传入地址
```

## 成员指针函数表

对于普通函数指着和指向成员函数的指针来说，一种常见用法是将其存入一个函数表中。如果一个类含有几个相同类型的成员，则这样一张函数表可以帮我们从这些成员中选择一个

```

1 class Screen{
2     public:
3         Screen& home();
4         Screen& forward();
5         Screen& back();
6         Screen& up();
7         Screen& down();
8     };

```

可以定义一个move函数，使其可以调用上面任意一个函数并执行对应的操作

```

1 class Screen{
2     public:
3         //其他定义保持不变
4         using Action = Screen& (Screen::*)();
5         enum Directions{Home, FORWARD, BACK, UP, DOWN};
6         Screen& move(Directions);
7     private:
8         static Action Menu[]; //函数表
9     }

```

数组Menu依次保存每个光标移动函数的指针

```

1 Screen& Screen::move(Directions cm){
2     return (this->*Menu[cm])();
3 }

```

## 19.4.3 将成员函数用作可调用对象

成员指针并不是一个可调用对象（因为需要先利用.\*和->\*运算符将指针绑定到特定对象上），所以不能直接将一个成员函数的指针传递给算法

```
1 auto fp = &string::empty; //fp指向empty函数
2 find_if(svec.begin(), svec.end(), fp); // 错误, fp不是一个可调用对象
```

find\_if需要一个可调用对象，但是我们提供的是一个指向成员函数的指针fp，因此再find\_if的内部将执行如下形式的代码，从而导致无法通过编译：

```
1 if(fp(*it)) //显然该语句试图直接调用fp
```

## 使用function生成一个可调用对象

```
1 function<bool (const string&)> fcn =
&string::empty;
2 find_if(svec.begin(), svec.end(), fcn);
```

我们告诉function一个事实，empty是一个接受string参数并返回bool值的函数。通常情况下，**执行成员函数的对象将被传给隐式的this形参**。当我们使用function为成员函数生成一个可调用对象时，必须首先“翻译”代码，**使得隐式的形参变成显式的**(我们自己传入一个形参，把这个认为是this)

当function包含有一个指向成员函数的指针时，function类直到它必须使用正确的指向成员的指针运算符来执行调用函数

```
1 if(fcn(*it)) //假设fcn是find_if内部的一个可调用对象的名字
2 if(((*it).*p)()) //p时fcn内部的一个指向成员函数的指针
```

当我们定义一个function对象时，必须指定该对象所能表示的函数类型，如果可调用对象是一个成员函数，第一个形参必须表示该成员是在哪个（一般是隐式的）对象上执行的。同时提供给function的形式中还必须指明对象是以指针还是引用的形式传参

```
1 vector<string*> pvec;
2 function<bool (const string*)> fp =
3   &string::empty;
4 find_if(pvec.begin(), pvec.end(), fp);
```

## 使用mem\_fn生成一个可调用对象

使用function必须提供成员的调用形式，也可以使用标准库功能mem\_fn来让编译器负责推断成员的类型

```
1 find_if(svec.begin(), svec.end(), mem_fn(&string::empty)); //该对象接受一个string实参，返回一个bool值
```

mem\_fn生成的可调用对象可以通过对象调用，也可以指针调用

```
1 auto f = mem_fn(&string::empty);
2 f(*svec.begin()); //使用.*
3 f(&svec[0]); //使用->*
4 //可以认为mem_fn生成的可调用对象含有一对重载的函数调用运算符，一个接受指针另一个接受对象
```

## 使用bind生成一个可调用对象

```
1 auto it =
  find_if(svec.begin(), svec.end(), bind(&string::empty, _1));
```

与function类似，使用bind时，必须将函数中用于表示执行对象的隐式形参转换成显式的。

与mem\_fn类似的地方是，bind生成的可调用对象的第一个实参既可以是string指针，也可以是string引用

```
1 auto f = bind(&string::empty,_1);
2 f(*svec.begin()); //使用.*  
3 f(&svec[0]); //使用->*
```

## 19.5 嵌套类

一个类可以定义在另一个类的内部

**嵌套类是一个独立的类，与外层类基本没什么关系。**外层类的对象和嵌套类的对象是相互独立的。嵌套类的对象中不包含任何外层类定义的成员；外层类的对象中也不包含任何嵌套类定义的成员

嵌套类的名字在外层类作用域中是可见的，在外层类作用域外不可见；嵌套类的名字不会和别的作用域中的同一个名字冲突

外层类对嵌套类的成员没有特殊的访问权限，嵌套类对外层类也是如此

嵌套类在其外层类中定义了一个类型成员。该类型的访问权限由外层类决定。

位于外层类public部分的嵌套类实际上定义了一种可以随处访问的类型；位于外层类protected部分的嵌套类定义的类型只能被外层类及其友元和派生类访问；位于外层类private部分的嵌套类定义的类型只能被外层类的成员和友元访问

### 声明一个嵌套类

```
1 class TextQuery{
2     public:
3         class QueryResult;
4     };
```

## 在外层类之外定义一个嵌套类

嵌套类必须声明在类的内部，但是可以定义在类的内部或外部。当在外层类之外定义嵌套类时，必须以外层类的名字限定嵌套类的名字

```

1 class TextQuery::QueryResult{
2     friend std::ostream& print(std::ostream&,
3         const QueryResult);
4     public:
5     QueryResult(//形参列表);
6 };

```

## 定义嵌套类的成员

使用外层类的名字限定嵌套类的名字

```
1 TextQuery::QueryResult::QueryResult(xxx):xxx{}
```

## 嵌套类的静态成员定义

如果嵌套类声明了一个静态成员，则该成员的定义将位于外层类的作用域之外

```
1 int TextQuery::QueryResult::static_mem = 1024;
```

## 嵌套类作用域中的名字查找

名字查找的一般规则在嵌套类中同样适用。因为嵌套类本身是一个嵌套作用域，所以还必须查找嵌套类的外层作用域

嵌套类是其外层类的一个类型成员，因此外层类的成员可以像使用任何其他类型成员一样使用嵌套的名字。所以TextQuery的query成员中可以直接使用QueryResult（返回类型并不在作用域中，所以返回类型需要指明作用域）

## 嵌套类和外层类是相互独立的

外层类的对象和嵌套类的对象没有任何关系。嵌套类的对象只包含嵌套类定义的成员；外层类的对象只包含外层类定义的成员

## 19.6 union一种节省空间的类

联合是一种特殊的类，一个union可以有多个数据成员，但是在任意时刻只有一个数据成员可以有值。当给union的某个成员赋值后，该union的其他成员就变成未定义状态了

分配一个union对象的存储空间至少要能够容纳它的最大的数据成员

union不能含有引用类型的成员

union可以为其成员指定public、protected、private等标记。默认是public的

union可以定义构造函数和析构函数和其他成员函数，但是不能继承自其他类，也不能作为基类使用，所以union不能含有虚函数

### 定义union

```

1 union Token{
2     char cval;
3     int ival;
4     double dval;
5 };

```

首先是关键字union，然后是union的名字（可选），以及成员声明

### 使用union

默认情况union是未初始化的，可以使用一对花括号内的初始值显示地初始化一个union

```

1 Token first_token = {'a'};
2 Token last_token;
3 Token *pt = new Token;

```

如果指定了初始值，该初始值被用于初始化第一个成员

## 匿名union

未命名的union，并且右花括号和分号之间没有任何声明的union是匿名union。定义匿名的union，编译器会自动地为该union创建一个未命名的对象

```

1 union{
2     char cval;
3     int ival;
4     double dval;
5 };
6 cval = 'c';//未匿名union对象赋一个新值
7 ival = 42;

```

匿名union的定义所在的作用域内该union的成员都是可以直接访问的

匿名union不能包含保护和私有成员，也不能定义成员函数

## 含有类类型成员的union

union包含的是内置类型的成员时，可以使用普通的赋值语句改变union的值；对于含有特殊类类型成员的union若想改变值，必须分别构造或析构该类类型成员：当将union的值改为类类型成员对应的值时，必须运行该类型的构造函数；反之，将类类型成员的值改为其他值时，必须运行该类型的析构函数

union包含的时内置类型成员时，编译器将按照成员的次序依次合成默认构造或拷贝控制成员。如果union含有类类型成员，并且该类型自定义了默认构造或拷贝控制成员，则编译器将为union合成对应的版本并将其声明为删除的



例如string定义了5个拷贝控制成员以及一个默认构造函数。如果union含有string类型的成员，并且没有自定义默认构造函数或某个拷贝控制成员，则编译器将合成缺少的成员并将其声明为删除的。如果某个类中含有union成员，而且该成员含有删除的拷贝控制成员，则该类与之对应的拷贝控制操作也将是删除的

## 使用类管理union成员

对于union，想要构造或销毁类类型的成员非常复杂。因此通常把含有类类型成员的union内嵌在另一个类中，使用该类管理并控制与union的类类型成员有关的状态转换



例如，为union添加了一个string成员，并将union定义成匿名union，最后将它作为Token类的一个成员。此时Token可以管理union成员

为了追踪union中到底存储了什么类型的值，通常会定义一个独立的对象，该对象称为**union的判别式**。可以使用判别式辨认union存储的值。



为了使判别式与union同步，将判别式也作为Token的成员，我们的类将定义一个枚举类型追踪union成员的状态

```
1 class Token{
```

```

2     public:
3         Token() : tok(INT), ival{0}{} // 初始化时用花括
4             号
5             Token(const Token &t) : tok(t.tok){
6                 copyUnion(t);
7             }
8             Token& operator=(const Token&);
9             ~Token(){
10                 if(tok == STR)
11                     sval.~string(); // 显式析构掉
12             }
13             Token& operator=(const std::string&);
14             Token& operator=(char);
15             Token& operator=(int);
16             Token& operator=(double);
17             private:
18                 enum{INT,CHAR,DBL,STR} tok;
19                 union{
20                     char cval;
21                     int ival;
22                     double dval;
23                     std::string sval;
24                 };
25             void copyUnion(const Token&);
26         };

```



主要说明一下析构函数

作为union组成部分的类成员无法自动销毁，因为析构函数不清楚union存储的值是什么类型，所以它无法确定应该销毁那个成员，所以我们自定义析构函数

## 管理判别式并销毁string

类的赋值运算符设置tok并为union赋值。与析构函数相同，这些运算符在为union赋值时必须首先销毁string（如果当前值确实是string的话，tok == STR）

赋值运算符形参是string时更加的复杂

```

1 Token& Token::operator=(const std::string&){
2     if(tok == STR)
3         sval = s;
4     else
5         new(&sval) string(s); //定位new表达式
6     tok = STR;
7     return *this;
8 }
```

## 管理需要拷贝控制的联合成员

拷贝时也需要考虑上述赋值运算符和析构函数中考虑的问题

```

1 void Token::copyUnion(const Token&){
2     switch(t.tok){
3         case Token::INT : ival = t.ival; break;
4         case Token::CHAR : cval = t.cval; break;
5         case Token::DBL : dval = t.dval; break;
6         case Token::STR : new(&sval)
7             string(t.sval); break;
8     }
```

如果本身union中就是string，copyUnion在该内存重新构造string；如果不是string，则构造string

```

1 Token& Token::operator=(const Token &t){
2     if(tok == STR && t.tok!=STR) sval.~string();
3     //本身是string, 赋值的参数不是string
4     if(tok == STR && t.tok == STR)
5         sval = t.sval; //本身和赋值参数都是string,
6         直接赋值
7     else
8         copyUnion(t); //本身不是string, 赋值是不是
9         string都直接copyUnion即可, 如果赋值的是内置类型, 则直接
10        赋值; 如果是string类型, 那么会调用定位new构造string
11        tok = t.tok;
12        return *this;
13    }

```

## 19.7 局部类

类可以定义在某个函数内部，称为**局部类**。局部类定义的类型只在定义它的作用域内可见，和嵌套类不同，局部类的成员受到严格的限制

局部类的所有成员都必须完整定义在类的内部

局部类中不允许声明静态数据成员（因为我们无法定义这样的成员）

### 局部类不能使用函数作用域中的变量

局部类只能访问外层作用域定义的类型名、静态变量以及枚举成员。如果局部类定义在某个函数内部，则该函数的普通局部变量不能被该局部类使用

```

1 int a, val;
2 void foo(int val){
3     static int si;
4     enum Loc {a = 1024, b};
5     struct Bar{
6         Loc locVal; //正确: 使用局部类型名
7         int barVal;

```

```

8     void fooBar(Loc l = a){ //正确: 默认实参
9         barVal = val; //错误, val是foo的局部
10        变量
11        barVal = ::val; //正确, 使用一个全局对
12        象
13        barVal = si; //正确, 静态局部对象
14        locVal = b; //正确, 枚举成员
15    }

```

## 常规的访问保护规则对局部类同样适用

外层函数对局部类的私有成员没有访问特权。局部类可以将外层函数声明为友元（就可以访问private了）。局部类封装在函数作用域中

## 局部类中的名字查找

如果某个名字不是局部类的成员，则继续在外层函数作用域中查找；如果找不到，则在外层函数所在的作用域查找

## 嵌套的局部类

可以在局部类的内部再嵌套一个类。嵌套类的定义可以出现在局部类之外，不过嵌套类必须定义在与局部类相同的作用域中

```

1 void foo(){
2     class Bar{
3         public:
4             class Nested;
5     };
6     class Bar::Nested{
7
8 };
9 }

```

局部类内的嵌套类也是一个局部类，必须满足局部类的各种规定。嵌套类的所有成员都必须定义在嵌套类内部

## 19.8 固有的不可移植的特性

不可移植的特性是指因机器而异的特性（从一台机器上转移到另一台机器上时，通常需要重新编写程序）

### 19.8.1 位域

类可以将其（非静态）数据成员定义为位域，在一个位域中含有一定数量的二进制位。当一个程序需要向其他程序或硬件设备传递二进制数据时，通常会用到位域



位于在内存中的布局是与机器相关的

位域的类型必须是整型或枚举类型。因为带符号位域的行为由具体实现确定，所以通常使用无符号类型保存一个位域。

位域的声明形式是在成员名字之后紧跟一个冒号以及一个常量表达式，该表达式用于指定成员所占的二进制位数

```

1 typedef unsigned int Bit;
2 class File{
3     Bit mode:2;
4     Bit modified:1;
5     Bit prot_owner:3;
6     Bit prot_group:3;
7     Bit prot_world:3;
8     public:
9     enum modes {READ = 01, WRITE = 02, EXECUTE
 = 03};
10    File &open(modes);
11    void close();

```

```

12     void write();
13     bool isRead() const;
14     void setWrite();
15 };

```

如果可能的话，在类的内部连续定义的位域压缩在同一整数的相邻位，从而提供压缩存储

取地址符不能作用于位域，因此任何指针都无法指向类的位域

## 使用位域

位域的使用方式与访问类的成员的方式非常相似：

```

1 void File::write(){
2     modified = 1;
3 }
4 void File::close(){
5     if(modified)
6 }

```

通常使用内置的位运算符操作超过1位的位域

```

1 File& File::open(File::modes m){
2     mode |= READ;
3     if(m & WRITE)
4         //
5     return *this;
6 }

```

如果一个类定义了位域成员，通常会定义一组内联成员函数以检验或设置位域的值

```

1 inline bool File::isRead() const {return mode &
  READ;}
2 inline bool File::setWrite() const {return mode
  | WRITE;}

```

## 19.8.2 volatile限定符

volatile的确切含义与机器有关

直接处理硬件的程序通常包含这样的数据元素，它们的值由程序直接控制之外的过程控制。当对象的值可能在程序的控制或检测之外被改变时（比如可能包含一个由系统时钟定时更新的变量），应该将该对象声明为 volatile。关键字 volatile 告诉编译器不应对这样的对象进行优化

```

1 volatile int display_register; //该int值可能发生改变
2 volatile Task *curr_task; //curr_task指向一个
  volatile对象
3 volatile int iax[max_size]; //iax的每个元素都是
  volatile
4 volatile Screen bitmapBuf; //bitmapBuf的每个成员都
  是volatile

```

const和volatile互相没什么影响，某种类型可能既是const的也是volatile的，此时具有二者的属性

类可以将成员函数定义为volatile的，只有volatile的成员函数才能被 volatile对象调用

volatile限定符和指针的关系与const限定符和指针的关系类似：可以声明volatile指针、指向volatile对象的指针以及指向volatile对象的 volatile指针

## 合成的拷贝对volatile对象无效

不能使用合成的拷贝/移动构造函数及赋值运算符初始化volatile对象或从volatile对象赋值。合成的成员接受的形参是（非volatile）常量引用，显然不能把非volatile引用绑定到volatile对象上。所以我们必须自定义拷贝或移动操作（普通对象可以赋值给volatile对象）

### 19.8.3 链接指示：extern "C"

C++程序有时需要调用其他语言编写的函数，最常见的是C语言编写的。

像所有其他名字一样，其他语言中的函数名字也必须在C++中进行声明，并且该声明必须指定返回类型和形参列表。对于其他语言编写的函数来说，编译器检查其调用的方式与处理普通C++函数的方式相同，但是生成的代码有所区别。

C++使用链接指示指出任意非C++函数所用的语言



要想将C++代码和其他语言（包括C）的代码放在一起使用，要求我们有权访问该语言的编译器，且该编译器与当前C++编译器兼容

### 声明一个非C++函数

链接指示有两种形式：单个的或复合的。链接指示不能出现在类定义或函数定义的内部。同样的链接指示必须在函数的每个声明中都出现

```

1 //<cstring>中的某些C函数是如何声明的
2 extern "C" size_t strlen(const char *);
3 extern "C" {
4     int strcmp(const char*, const char*);
5     char *strcat(char*, const char*);
6 }
```

链接指示的第一种形式包含一个关键字extern，后面是一个字符串字面值常量以及一个"普通的"函数声明，其中的字符串字面值常量指出了编写函数所用的语言。编译器应该支持对C语言的链接指示。编译器也可能会支持其他语言的链接指示，如extern "Ada"、extern "FORTRAN"

## 链接指示与头文件

链接指示后跟上花括号括起来的若干函数声明，从而一次性建立多个链接。花括号中声明的函数名字就是可见的，就好像在花括号之外声明的一样

```
1 extern "C" {
2     #include <string.h>
3 }
```

链接指示可以嵌套，因此如果头文件包含带自带链接指示的函数，则该函数的链接不受影响

## 指向extern "C" 函数的指针

编写函数所用的语言是函数类型的一部分，因此对于使用链接指示定义的函数来说，它的每个声明都必须使用相同的链接指示。所以指向其他语言编写的函数的指针必须与函数本身使用相同的链接指示

```
1 extern "C" void (*pf)(int);
```

使用pf调用函数时，编译器认定当前调用的是一个C函数。一个指向C函数的指针不能用在执行初始化或赋值操作后指向C++函数

## 链接指示对整个声明都有效

当我们使用链接指示时，它不仅对函数有效，而且对作为返回类型或形参类型的函数指针也有效

```
1 extern "C" void f1(void(*)(int));
```

链接指示不仅对f1有效，对参数也有效，必须给f1传递一个C函数的名字或者指向C函数的指针

## 导出C++函数到其他语言

通过使用链接指示对函数进行定义，可以在另一个C++函数在其他语言编写的程序中可用

```
1 extern "C" double calc(double dparm) {}
```

编译器将为该函数生成适合于指定语言的代码

可被多种语言共享的函数的返回类型或形参类型受到很多限制

## 重载函数与链接指示

链接指示与重载函数的相互作用依赖于目标语言。如果目标语言支持重载函数，则为该语言实现链接指示的编译器很可能也支持重载这些C++函数