C++编程(1)

唐晓晟
北京邮电大学电信工程学院

课程简介

- 参考资料：
  C++程序设计语言(特别版) 裘宗燕(北京大学)译
  Bjarne Stroupstrup 机械工业出版社
  ISBN 7-111-10202-9/TP.2424
  85RMB
  C++ Primer, Lippman
- 课时：34学时(17周)
- 教学网站：
  http://202.112.10.142/
  http://teach.academy.net.cn/

第一章 导论(致读者)

- 书本结构
- 学习C++
- C++的设计
- 历史注记
- C++的使用
- C和C++
- 忠告

1 书本结构

- 导论：1-3章，有关C++语言，所支持的关键性程序设计风格，有关C++标准库的综述
- 第一部分：4-9章，C++内部类型
- 第二部分：10-15章，使用C++做面向对象和通用型程序设计
- 第三部分：16-22章，C++标准库
- 第四部分：23-25章，设计和软件开发

2 学习C++

- 最重要的事情：关注概念，不要迷失在语言的技术细节中
- C++支持多种不同的程序设计风格
- C++支持一种逐步推进的学习方式
- 直接学习C++，而不是先学习C
- 大量查阅资料，每一种至少参考两个来源

3 C++的设计

- 重要设计原则：简单性，与C的高度兼容
- 无内部高级数据类型，没有高级的基本操作
- 尽力避免了那些即使不用也会带来运行时间或者空间开销的特征
- 能够使用传统的编译和运行时的环境
- C++的类型检查和数据隐藏特征依赖于编译时对程序的分析，以防止因为意外而破坏数据的情况

(1) 效率和结构

- C++以C为基础开发设计，大量维持了C作为一个子集，C++可以使用与C一样的函数调用及返回序列－或者其他效率更高的方式，而C的一个初始目标则是在大部分苛刻的系统程序设计中代替汇编
- C++中特别强调程序的结构，这反映了C以来程序规模增长的情况
- 本书强调的是为提供通用功能、普遍有用的类型、库等的各种技术

(2) 哲学注记

- 程序设计语言要服务于两个相互关联的目的
- 1 为程序员提供一种描述所需执行的动作的载体，这要求一种“尽可能接近机器的”语言
- 2 为程序员提供一组概念，使他们能利用这些概念去思考什么东西是能够做的，这要求该语言“尽可能接近需要解决的问题”
- C++的类概念已经被证明是一种极为强有力的概念工具

4 历史注记

- C++大大受惠于C[Kernighan,1978]
- 其前驱为BCPL[Richards,1980](//注释)
- 一些灵感来自Simula67[Dahl,1970,1972](如类的概念、派生和虚函数)
- Algol68[Woodward,1974]，如重载运算符和自由的将声明放置在可以出现语句的任何位置
- 模板机制功能部分受到Ada中generic的启发，部分受到Clu语言参数模块的影响

- 异常处理机制部分受到Ada[Ichbiah,1979]、Clu[Liskov,1979]和ML[Wikstrom,1987]语言的影响
- C++语言从1980年开始被研究组织使用，研究组织之外的最初使用起于1983年
- C++读做C plus plus
- 名称意义：C+,C++,++C,D
- C++设计的主要用途是为了个人程序员

- 87年，爆炸性使用导致了标准化工作的开始
- AT&T的贝尔实验室允许作者将C++参考手册的草稿和各种修订版本进行分发和共享
- ANSI的X3J16委员会1989年12月在HP的建议下建立起来
- 1991年该ANSI C++标准化变成ISO的C++标准化工作的一部分
- 标准化草案1995年4月给公众阅览，ISO C++标准1998年最终被批准(ISO/IEC 14882)

5 C++的使用

- C++被应用到几乎每个领域
- 高效率使得C++被用来写操作系统或者驱动程序
- 可靠性、可管理、易扩充、易测试使得C++被用于银行、贸易、保险、通信以及军事领域
- 由于编写用户界面，C++也被用于很多数值的、科学的以及工程计算中去

C++被广泛应用于教学与研究

- 对于教授基本概念而言足够清晰
- 对于深刻的项目而言足够现实、高效和灵活
- 对依赖各种不同开发和执行环境的组织或者研究机构而言，使用起来足够方便
- 对作为教高级概念和技术的媒介而言，足够的容易理解
- 对作为从学习到非学习使用的工具而言，足够商业化

6 C和C++

- C被选做C++的基础语言的原因
  - 通用的、简洁的、相对低级的
  - 适合用于大部分系统的程序设计工作
  - 可以在每个地方的任何系统运行
  - 适应于UNIX程序设计环境

C++继续与C兼容的原因

- 存在着成百万行的C代码可能从C++中获益，先决条件是不必将他们用C++重写
- 存在着成百万行的用C写出的库函数和功能软件代码可以从C++里使用，先决条件是C++能够与C连接兼容，语法相似
- 存在着数以十万计的程序员先了解C，这样他们能够很快速的学会C++
- C++和C将在很多年中被同一些人用于同样的系统，因此其差异必须或者很小、或者很大，以最大限度地减少错误和混乱的发生

(1) 给C程序员的建议

- C++里几乎不需要宏(const,enum定义明显的常量, inline避免函数调用开销, template刻画一组函数或类型, 用namespace避免名字冲突)
- 使用变量时可以随时声明
- 不要用malloc(new或者vector)
- 避免使用void*，指针算术，联合和强制
- 少用数组和C风格的字符串(string, vector)

(2) 给C++程序员的建议

- C++在不断发展中，应该注意引进新的特征，很多以前看来是全新的程序设计技术现在已经变成可行的东西了
- 不断的学习（作者本身在写这本书的过程中就学到了不少东西）

7 忠告

- 编程是在为某个问题的解决方案中的思想建立起一种具体表示
  - 如果能将“它”看成一个独立的概念，将其作成一个类
  - 如果能将“它”看成一个独立的实体，将它作成某个类的一个对象
  - 如果两个类有共同的界面，将此界面做成一个抽象类
  - 如果两个类的实现有某些显著的共同东西，将这些共性做成一个基类
  - 如果一个类是一种对象的容器，将它做成一个模板
  - 如果一个函数实现对某容器的一个算法，将它实现为对一族容器可用的模板函数
  - 如果一组类、模板等相互之间有逻辑关系，将它们放进一个名字空间去

- 在定义一个并不是实现某个像矩阵或复数这样的数学对象的类时，或者定义一个低层的类型如链表时：
  - 不要使用全局数据（使用成员）
  - 不要使用全局函数
  - 不要使用公用数据成员
  - 不要使用友元
  - 不要在一个类里面放“类型域”：采用虚函数
  - 不要使用内联函数，除非有效果显著的优化

C++编程(2)

唐晓晟
北京邮电大学电信工程学院

第四章 类型和声明

- 类型
- 布尔量
- 字符类型
- 整数类型
- 浮点类型
- 大小
- void
- 枚举
- 声明
- 忠告
- 练习

4.1 类型

- C++中，每个名字都有一个与之相关联的类型，类型决定了可以对其进行什么样的操作，并决定这些操作如何解释
- 基本类型有：
- 布尔、字符、整数、浮点、枚举、void、指针、数组、引用、数据结构和类
- 可以分为内部类型和用户自定义类型

4.2 布尔量(bool)

- 布尔量bool，可以具有两个值false或者true之一，用于表示逻辑运算的结果。
- 根据定义 true = 1, false = 0，实际：非0即true
- bool变量和整数（包括指针）可以相互转换

```
void f(int a, int b)
{
   bool bl = a == b;
   // ...
}

bool b = 7;

int i = true;

void g()
{
   bool a = true;
   bool b = true;
   bool x = a + b;
   bool y = a | b;
}
```

4.3 字符类型

- 类型为char的变量可以保存具体实现所用的字符集里面的一个字符
- 常用字符类型都至少包括8bit，可以保存256种不同的数值
- 字符集一般都采用ISO-646的某个变形，比如说ASCII(ANSI3.4-1968)

以下假设都是不安全的

- 8位字符集中共有不超过127个字符(有的字符集提供了255个字符)
- 不存在超出英语的字符(大部分欧洲语言提供了更多的字符)
- 字母字符是连续排列的(EBCDIC在"i"和"j"之间留有空隙)
- 写C++所需要的每个字符都是可用的(有些国家的字符集中没有提供{}[]\，C++提供的解决方法有关键字、二联符和三联符)

解决受限的字符集

关键字(keywords) C++		二联符(digraph) C++		三联符(trigraph) C	
and	**&&**	**<%**	**{**	??=	#
and_eq	**&=**	**%>**	**}**	??(	[
bitand	**&**	**<:**	**[**	??<	{
bitor	**|**	**:>**	**]**	??/	\
compl	**~**	**%:**	**#**	??)	]
not	**!**	**%:%:**	**##**	??>	}
or	**||**			??'	^
or_eq	**|=**			??!	|
xor	**^**			??-	~
xor_eq	**^=**				
not_eq	**!=**				

digraph and trigraph Examples

使用C++的digraph

```
#include <stdio.h>
main(int argc, char *argv<::>)
<%
  if (argc> 1 and argv<:0:> != NULL)
    printf("Hello, %s!??/n",argv<:1:>);
  else
    printf("Hello, world!??/n");
  return 0;
%>
```

使用C的trigraph

```
#include <stdio.h>
void main(int argc, char *argv??(??))
??<
    if (argc>1 && argv??(0??) != NULL)
        printf("Hello, %s!??/n",argv??(1??));
    else printf("Hello, world!??/n");
    return 0;
??>
```

转义字符

名字	ASCII名字	C++名字
换行符	NL(LF)	\n
水平制表符	HT	\t
垂直制表符	VT	\v
退格符	BS	\b
回车符	CR	\r
换页符	FF	\f
警铃符	BEL	\a
反斜线符	\	\\
问号	?	\?
单引号	‘	\’
双引号	“	\”
八进制	ooo	\ooo
十六进制	hhh	\xhhh

字符类型

- char, signed char, unsigned char
- char和int之间的相互转换(由实现决定)
- wchar_t类型，用于保存更大的字符集里的字符，如Unicode，具体大小由实现决定，该名字来源于C语言，是一个typedef
- 对于字符类型可以进行算术和逻辑运算
- 字符文字量(字符常量)，如’a’,’0’,’\n’等
- 宽字符文字量形式为L’ab’，引号中的字符个数由wchar_t决定，其类型是wchar_t

字符类型示例

八进制	十六进制	十进制	ASCII	大字符集
\6	\x6	6	ACK	\uXXXX 或者 \UXXXX
\60	\x30	48	‘0’	\u1e2b
\137	\x05f	95	‘_’	\uXXXX等价于\U0000XXXX

```
char v1[] = “a\xah\129”;
char v2[] = “a\xah\127”;
char v3[] = “a\xad\127”;
char v4[] = “a\xad\0127”;
```


字符类型程序示例

```
char c = 255; // 0xff
int i = c;
问题：i = ? // 不同的环境下数值不同
解决方法：使用unsigned char 或者 signed char标明
但是有些函数只接受普通char类型

void f(char c, signed char sc, unsigned char uc)
{
  char* pc = &uc;
  signed char* psc = pc;
  unsigned char* puc = pc;
  psc = puc;
  // 以上四条语句均产生错误
}
```

4.4 整数类型

- 三种形式：int, signed int (signed) , unsigned int (unsigned)
- 三种大小：short int (short), int, long int (long)
- 与char不同，默认的int 总是有符号的
- 整数文字量：7, 1234, 1234567890000
- 十六进制：0x3f 八进制：022
- L(大小写)结尾表示long，U(大小写)结尾表示unsigned，如：100UL

4.5 浮点类型

- 表示浮点数，即包括小数部分的数
- 三种大小：float, double, long double
- 浮点文字量：1.23 .23 0.23 1. 1.2e10
- 注意：浮点文字量的中间不能出现空格
- 以F(大小写)结尾表示float，不加后缀缺省表示为double类型，若需要long double类型的数字，可以加上L(大小写)后缀

4.6 大小

- C++基本类型的某些方面是由实现确定的，比如说int
- 为了保证程序的移植性，建议在所有的可能之处都使用标准库的功能
- 提供多种整数类型、无符号类型、浮点类型的原因是希望使程序员能够利用各种硬件特性。比方说：不同的硬件对不同的基础类型处理时，存储的需求、存储访问时间和计算速度方面存在明显的差异。

大小

C++对象大小由char的大小的倍数表示，定义char的大小为1，
则其他数据类型大小可以用sizeof运算符获得
1=sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)
1 <= sizeof(bool) <= sizeof(long)
sizeof(char) <= sizeof(wchar_t) <= sizeof(long)
sizeof(float) <= sizeof(double) <= sizeof(long double)
sizeof(N) = sizeof(signed N) = sizeof(unsigned N)

numeric_limits

```
#include <limits>
#include <iostream>
using namespace std;
int main()
{
   cout << "largest float == " <<
   numeric_limits <float> :: max()
   << ", char is signed == " <<
   numeric_limits<char> :: is_signed
   << endl;
}
```


4.7 void

- void类型是一个语法上的基本类型
- void可以用来标明一个函数并不返回数值，这可以理解为一个“伪返回类型”，可以加强语法的规范性
- 也可以用做指向不明类型的对象的指针的基础类型

例如：

```
void x;// error
void f();
void* pv;
```


4.8 枚举

- 枚举是一种类型，可以保存一组由用户刻画的值，一旦定义，使用起来很像整数类型
- 例如：
- enum{ ASM, AUTO, BREAK };
  - 定义三个枚举型的整数常量并赋值，默认方式下，数值从0开始
  - ASM = 0, AUTO = 1, BREAK = 2
  - 枚举也可命名，如enum keyword{ ASM, AUTO, BREAK }; 这样产生一个新的数据类型keyword

枚举类型的取值范围

- 如果某个枚举中的所有枚举符的值均非负，该枚举的表示范围就是”0”到”2的k次幂-1”，k是能够使得所有枚举符号都在此范围内的最小的2的幂
- 如果存在负的枚举符号值，该枚举的取值范围就是”负2的k次幂”到”2的k次幂-1”之间

```
enum e1 {dark, lignt};    // 0:1
enum e2 {a = 3, b = 9}; // 0:15
enum e3 {min = -10, max = 1000000}; //-1048576:1048576
```

枚举数值和整数之间的转换

- 一个整型数可以显式地转换到一个枚举值(前提是转换的结果位于该枚举范围之内，否则是无定义的)
- 默认情况下，枚举值可以转换到整数参加算术运算，此外，由于枚举是用户自定义类型，用户可以为枚举定义自身的操作，例如定义++或者<<

枚举数值转换示例

```
enum flag { x=1, y=2, z=4, e=8}; // 0:15

flag f1 = 5; // wrong

flag f2 = flag(5); // ok

flag f3 = flag(z | e); // ok, z|e = 12

flag f4 = flag(99); //not defined
```


4.9 声明

- 一个名字(标识符)在被使用之前必须声明，以刻画清楚它的类型，通知编译器这个名字所引用的是哪一类实体，如：double d;
- 通常，大部分声明同时也是定义
- C++中，每个命名实体必须有恰好一个定义，当然，可以有很多声明，但是所有声明必须类型完全一致

```
int error_number = 1;
int count;
int count; // Error

extern int error_number;
extern int error_number;
extern int error_number;
extern short error_number;
```


4.9.1 声明的结构

- 声明包括四部分：可选的“描述符”、基础类型、声明符、可选的初始式

描述符：virtual extern等，表明被声明事物的某些非类型的属性
char* kings[] = {“Seleucus”, “Ptolemy”};
基础类型是：char
声明符是：*kings[]，常用的声明符有*,*const,&等前缀，以及[],()等后缀，一般情况下，后缀的声明运算符比前缀的声明运算符约束力更强
初始式是：{… …}

注意：声明中不能没有类型

```
const c=7;  // Error
gt(int a, int b){ return (a>b)?a:b;} //Error
```

4.9.2 声明多个名字

- 单个声明中可以有多个名字，多个名字可以使用逗号分隔
- 例如：int x,y; // int x; int y;
- 需要注意：运算符只作用于一个单独的名字，不是同一声明中随后写的所有名字
- 例如：
- int* p, y; // int *p; int y; 不是int* y;

4.9.3 名字

- 名字(标识符)由一系列字母和数字组成，第一个字符必须是字母(包括下划线)
- C++对于名字中的字符个数没有任何限制，但是字符数目可能会受编译器或者连接器的限制
- 以下划线开头的名字是保留变量
- 编译器读程序时，总是设法寻找最长的能够组成一个名字的字符序列

- C++中，大小写字符是区分的，选择名字时，尽量避免选择不容易区分的字符，比如说1,l,L,l,O,0等
- 较少使用的名字可以选择相对比较长，频繁使用的名字可以选择比较短的
- 名字的选择应该反映一个实体的意义，而不是实现方式(phone_book比number_list更好)
- 设法保持统一风格

4.9.4 作用域

- 一个声明将一个名字引入一个作用域
- 作用域指这个名字的有效区间，包括局部作用域({}语句块)和全局作用域(整个文件内部可用，如果是在所有函数、类、名字空间之外定义)
- 注意同名局部变量和全局变量之间的覆盖问题

作用域程序示例

```
int x;
void f()
{
   int x;
   x = 1;
   {
      int x;
      x = 2;
   }
   x = 3;
}

int* p = &x;
```

```
int x;
void f2()
{
   int x = 1;
   ::x = 2;
   // ...
}
```

```
int x = 1;

void f()
{
   int y = x;
   int x = 22;
   y = x;
}
```

```
void f5(int x)
{
   int x;
}

//Error
```


4.9.5 初始化

- 如果为一个对象提供了初始式，这个初始式将确定对象的初始值，如果没有，全局的、名字空间的、局部静态的对象将被自动初始化为适当类型的0
  - int a; // a = 0
  - double d; // d = 0.0
- 局部对象(自动对象)和在自由存储区内建立的对象(动态对象或者堆对象)不会用默认值做初始化
  - void f()
    {
    int x; //x没有定义良好的值
    }

4.9.6 对象和左值

- 一个对象就是存储中一片连续的区域
- 左值就是引用某个对象的表达式，其原本意思是“某个可以放在赋值号左边的东西”，但是引用了某个常量的左值除外
- 没有被声明为常量的左值常常被称做是可修改的左值
- 左值在很多语法书中或者C++语言中称为lvalue

4.9.7 typedef

- typedef就是为类型声明了一个新名字，而不是声明一个指定类型的对象，更不是生成新的类型
  - typedef char* Pchar;
    Pchar p1,p2;
    char* p3 = p1;
- typedef的另一类使用是将对某个类型的直接引用限制到一个地方
  - typedef int int32;
    typedef short int16;

4.9.10 忠告

- [1]保持较小的作用域；
- [2]不要在一个作用域和它外围的作用域里采用同样的名字；
- [3]在一个声明中(只)声明一个名字；
- [4]让常用的和局部的名字比较短，让不常用的和全局的名字比较长；
- [5]避免看起来类似的名字；
- [6]维持某种统一的命名风格；
- [7]仔细选择名字，反映其意义而不是反映实现方式；

忠告

- [8]如果所用的内部类型表示某种可能变化的值，请用typedef为它定义一个有意义的名字；
- [9]用typdef为类型定义同义词，用枚举或类去定义新类型；
- [10]切记每个声明中都必须描述一个类型(没有“隐式的int”)；
- [11]避免有关字符数值的不必要假设；
- [12]避免有关整数大小的不必要假设；
- [13]避免有关浮点类型表示范围的不必要假设；
- [14]优先使用普通的int而不是short int或者long int；

忠告

- [15]优先使用double而不是float，或者long double；
- [16]优先使用普通的char而不是signed char或者unsigned char；
- [17]避免做出有关对象大小的不必要假设；
- [18]避免无符号算术；
- [19]应该带着疑问去看待signed到unsigned，或者从unsigned到signed的转换；
- [20]应该带着疑问去看待从浮点到整数的转换；
- [21]应该带着疑问去看待向较小类型的转换，如将int转换到char；

C++编程(3)

唐晓晟
北京邮电大学电信工程学院

第五章 指针、数组和结构

- 指针
- 数组
- 到数组的指针
- 常量
- 引用
- 指向void的指针
- 结构
- 忠告

5.1 指针

- 对于类型T，T*是“到T的指针”，也就是说，一个类型为T*的变量能保存一个类型T的对象的地址
- 对指针的基本操作是dereference，书中翻译为间接引用，间接运算符是(前缀的)*

```
char c = 'a';
char* p = &c;
//p 保存c的地址
```

```
char c = 'a';
char* p = &c;
char c2 = *p;
// c2 = 'a';
```


零

- 0是一个整数，可以用于任意整型、浮点类型、指针、指向成员的指针的常量等，其类型由具体环境确定
- 没有任何地址会被分配到地址0，所以0被当成一个指针常量，表示该指针没有指向任何对象
- C中，使用NULL代表0指针，C++中则使用数字0，可以const int NULL = 0;

5.2 数组

- 对于类型T，T[size]就是“具有size个T类型元素的数组”类型，元素下标从0到size-1
- 数组元素的个数必须是一个常量表达式
- 多维数组被表示为数组的数组

```
float v[3]; // v[0], v[1], v[2]
char* a[32]; // 32个到char的指针
// a[0] ... a[31]

int d2[10][20];
int bad[2,5]; // Error
```

```
void f(int i)
{
    int v1[i]; // Error
    vector<int> v2(i);
}
```


5.2.1 数组初始化

```
int v1[] = {1, 2, 3, 4}; // Size = 4
char v2[] = { ‘a’, ‘b’, ‘c’, 0}; // Size = 4
char v3[2] = { ‘a’, ‘b’, 0}; // Error
char v4[3] = { ‘a’, ‘b’, 0}; // ok
int v5[8] = {1, 2, 3, 4};
相当于int v5[8] = { 1, 2, 3, 4, 0, 0, 0, 0};

注意：不存在与数组初始化相对应的数组赋值
void f()
{
    v4 = { ‘c’, ‘d’, 0}; // Error
}
可以使用vector或者valarray进行赋值
```


5.2.2 字符串文字量

- 字符串文字量是用双引号括起来的字符序列
- 一个字符串文字量里包含的字符个数比它看起来的字符数多一个，它总是由一个空字符’\0’结束，空字符的值是0
- 例如：sizeof(“test”)=5，但是strlen(“test”)=4
- 字符串文字量的类型是“适当个数的const字符的数组”，所以“test”的类型就是const char[5]，此为常量，而且是静态分配的
- 可以用字符串文字量给一个char*赋值，若想修改一个字符串，可以先将其复制到一个数组中去

字符串示例

```
void f()
{
    char* p = “Plato”;
    p[3] = ‘e’;
    //Error: 给常量赋值，
      结果无定义
}

void f2()
{
    char p[] = “Zeno”;
    p[0] = ‘R’;
    // ok
}
```

```
const char* error_msg()
{
    // ...
    return “range error”;
} // ok

const char* p = “Heraclitus”;
const char* q = “Heraclitus”;

void g()
{
    if(p==q) cout << “One!”;
    //...
} // 结果由编译器实现决定
```

字符串示例

字符串中不能有真正的换行，比如

```
char str[] = " this is a two
     lines string";
//Error

char str[] = "this is a two \n lines string";
```

可以：`char alpha[] = "abcdefghijklmn" "opqrstuvwxyz";`

等价于

```
char alpha[] = "abcdefghijklmnopqrstuvwxyz";
```

另外：`char str[] = "Jers\000Munk"; // \0后面的将会被忽略`

5.3 到数组的指针

- 数组名字可以被用作它的开始元素的指针
- 注意：取得"开始元素之前的一个位置"没有任何意义，因为数组常常被分配在机器地址的边界上
- 数组名可以被隐式地转换成数组的开始元素的指针(这在C风格代码的函数调用中被广泛使用)，但是这将丢失数组的大小信息，vector和string没有此类问题

数组指针示例

```
int v[] = {1, 2, 3, 4};
int* p1 = v;
int* p2 = &v[0];
int* p3 = &v[4];
//最后元素之后的一个位置

extern "C" int strlen(const char*); // string.h
void f()
{
    char v[] = "Annemarie";
    char* p = v;  //隐式地从char[]转换到char*
    strlen(p);
    strlen(v);      //隐式地从char[]转换到char*
    v = p;          //Error: 不能给数组赋值
}
```


5.3.1 在数组中漫游

```
void fi(char v[])
{
    for(int i = 0; v[i] != 0; i++) use(v[i]);
}
```

等价于

```
void fi(char v[])
{
    for(char* p =v; *p!=0; p++) use(v[i]);
}
```

需要注意的是，普通数组(例如非char类型的数组)不具备自描述性，当遍历时，需要提供数组长度

```
#include <iostream>
int main()
{
   int vi[10];
   short vs[10];
   std::cout << &vi[0] << ' ' << &vi[1] << endl;
   std::cout << &vs[0] << ' ' << &vs[1] << endl;
}
```

可能输出

```
0x7fffaef0 0x7fffaef4
0x7fffaedc 0x7fffaede
```


指针的运算

- 假设T* p;
- p++相当于p变为指向类型为T的下一个元素的地址，即p和p++之间的实际距离为sizeof(T)
- 只有两个指针指向同一个数组的元素时，指针相减才有意义，结果为两个指针之间数组元素个数(一个整数)
- 指针相加没有意义，是不允许的

指针运算示例

```
void f()
{
   int v1[10];
   int v2[10];

   int i1 = &v1[5] - &v1[3]; // i1 = 2
   int i2 = &v1[5] - &v2[3]; //结果无定义

   int* p1 = v2 + 2;  // p1= &v2[2]
   int* p2 = v2  - 2;  // p2无定义
}
```


5.4 常量

- const是为了描述概念“不变化的值”
- 常量在编程中非常有用：
  - 许多对象在初始化之后就不再改变自己的数值
  - 采用符号常量写出的代码更容易维护
  - 指针常常是边读边移动，而不是边写边移动
  - 许多函数参数是只读不写的

常量示例

```
const int model = 90;
const int v[] = {1, 2, 3, 4};
const int x; // Error

void f()
{
    model = 200; // Error
    v[2] ++;       // Error
}
```

```
void g(const X* p)
{
    //这里不能修改*p
}

void h()
{
    X val; //val可以被修改
    g(&val); // ok
    //...
}
```

编程中，应系统化地使用符号常量，以避免出现“神秘的数值”(magic numbers)

5.4.1 指针和常量

- 使用一个指针时涉及到两个对象：指针本身和被它所指的对象
- 要将指针本身而不是被指对象声明为常量，必须使用声明运算符*const，而不能只是简单地用const

指针和常量示例(1)

```
char* const cp;
//到char的const指针
char const* pc;
//到const char的指针
const char* pc2;
//到const char的指针
```

对于const定义，从右向左读有助于我们理解
例如：
cp是一个const指针到char
pc2是一个指针指向const char

```
void f1(char* p)
{
    char s[] = “Gorm”;

    const char* pc = s; //指向常量
    pc[3] = ‘g’;  // Error
    pc = p; // ok

    char* const cp = s; //常量指针
    cp[3] = ‘a’;  // ok
    cp = p; // Error

    const char* const cpc = s;
    cpc[3] = ‘a’; // Error
    cpc = p; // Error
}
```


指针和常量示例(2)

```
char* strcpy(char* p, const char* q);
//const表示strcpy函数的执行体不能修改*q
```

可以将一个变量的地址赋给一个指向常量的指针，因为这么做不会造成任何伤害，但是不能将常量的地址赋给一个未加限制的指针，因为这样将会允许修改该对象的值了

```
void f4()
{
    int a = 1; const int c = 2;
    const int* p1 = &c; // ok
    const int* p2 = &a; // ok
    int* p3 = &c; // Error
    *p3 = 7; // Error
}
```

5.5 引用

- 一个引用就是某对象的另一个名字，引用的主要用途是为了描述函数的参数和返回值，特别是为了运算符的重载
- 记法 X& 表示到X的引用
- 为了确保引用总是某个东西的名字(总能约束到某个对象)，必须对引用进行初始化，而且一旦被初始化后就不可能改变了
- 引用的一种最明显的实现方式是作为一个(常量)指针，每次使用它时都自动地做间接访问

引用示例

```
void f()
{
   int i = 1;
   int& r = i;
   //r和i都是同一个int数值
   int x = r; // x = 1
   r = 2;      // i = 2
}

int i = 1;
int& r1 = i; // ok
int& r2; // Error 没有初始化
extern int& r3;
// ok r3在别处初始化
```

```
void g()
{
   int ii = 0;
   int& rr = ii;
   rr ++;   // ii被增加1
   int* pp = &rr; //pp指向ii
}
void increment(int& aa)
{ aa++;}
void f()
{
   int x = 1;
   increment(x); //x = 2
}
```


为提高程序的可读性，通常应该尽可能避免让函数修改传递过来的参数，相反应该让函数明确地返回一个值，或者明确要求一个指针参数，使用引用参数，会给他人一种该函数要修改这个参数的强烈暗示

```
int next(int p)
  { return p+1; }
void incr(int* p) { (*p)++; }
void g()
{
   int x = 1;
   increment(x);  // x = 2
   x = next(x);   // x = 3
   incr(&x);      // x = 4
}
```

另外一种错误的引用声明方式

```
double& dr = 1;
// Error, cannot convert
from 'const int' to 'double &'

const double& cdr = 1;
// ok
```


5.6 指向void的指针

- 一个指向任何对象类型的指针(不包括函数指针和成员指针)都可以赋值给类型为void*的变量
- void*可以赋值给另一个void*
- 两个void*可以比较相等与否
- 可以显式地将void*转换到另一个类型
- 其他任何操作都是不安全的
- void*的最重要的用途是需要向函数传递一个指针，而又不能对对象的类型做任何假设，或者从函数返回一个无类型的对象，注意：使用前必须经过显式的类型转换

void*示例

```
void f(int* pi)
{
   void* pv = pi; // ok
   *pv;    // Error
   pv++; // Error
   int* pi2 = static_cast<int*>(pv); // ok
   double* pd1 = pv; // Error
   double* pd2 = pi;  // Error
   double* pd3 = static_cast<double*>(pv); //不安全
}
```


5.7 结构

- 数组是相同类型的元素的一个集合，struct则是(几乎)任意类型数组的一个集合
- 结构类型对象的大小未必是其成员的大小之和，这是因为许多机器要求将对象分配在某种与机器系统结构有关的边界上，比如说机器的字边界，这种情况下，对象被称为是具有**对齐**的性质

结构示例(1)

```
struct address {
   char* name;
   long int number;
   char* street;
   char* town;
   char state[2];
   long zip;
}; //很多机器上，sizeof(address)是24
address jd;
jd.name = "Jim Dandy";
jd.number = 61;

address jd = { "Jim Dandy", 61, "South
St", "New Providence", {'N', 'J'}, 7974};
```

```
address *p;
...

cout << p->name ;
cout << p->number;

p->name
等价于
(*p).name
```


结构示例(2)

- 类型的名字在出现后立即就可以使用，不必等到看到完整声明，但是在完整声明被看到之前，不能去声明这个结构类型的新对象

```
struct Link{
   Link* previous;
   Link* successor;
}; // ok

struct No_Good{
   No_Good member;
}; // Error
```

```
struct List; //先声明一下
struct Link{
   Link* pre;
   Link* suc;
   List* member_of;
};
struct List{
   Link* head;
}
```

5.7.1 类型等价

- 两个结构总是不同的类型，即使它们有相同的成员，此外，结构类型也与各种基本类型不同，每个结构在程序里都是唯一的

```
struct S1 { int a; };
struct S2 { int a; };

S1 x;
S2 y = x; // Error

int i = x;  // Error
```


5.8 忠告

- [1] 避免非平凡的指针算术
- [2] 当心，不要超出数组的界限去写
- [3] 尽量使用0而不是NULL
- [4] 尽量使用vector和valarray而不是内部数组
- [5] 尽量使用string而不是以0结尾的char数组
- [6] 尽量少用普通的引用参数
- [7] 避免void*，除了在某些低级代码中
- [8] 避免在代码中使用非平凡的文字量(magic number)，相反，应该定义和使用符号常量

C++编程(4)

唐晓晟
北京邮电大学电信工程学院

第六章 表达式和语句

- 一个桌面计算器
- 运算符概览
- 语句概览
- 注释和缩进编排
- 忠告

6.1 一个桌面计算器

- 该计算器能够完成输入运算表达式的求解
- 由分析器、输入函数、符号表和一个驱动程序构成，可以看作是一个很小的编译器

6.2 运算符概览

- 在表格中，class_name表示类名，member表示一个成员的名字，object表示一个能产生出类对象的表达式，pointer是一个产生指针的表达式，expr是表达式，lvalue是一个表示非常量对象的表达式
- 表格中给出的只适合于内部类型的运算对象
- 每个间隔里的运算符具有相同优先级，上面间隔里面的运算符比下面间隔的运算符优先级更高

运算符概览(1)

作用域解析	class_name::member	全局	::name
作用域解析	namespace_name::member	全局	::qualified-name
成员选择	object.member	下标	pointer[expr]
成员选择	pointer->member	值构造	type(expr_list)
函数调用	expr(expr_list)	后增量	lvalue++
const转换	const_cast<type>(expr)	后减量	lvalue--
类型识别	typeid(type)	不检查的转换	reinterpret_cast<type>(expr)
运行时类型识别	type(expr)		
编译时检查的转换	static_cast<type>(expr)		
运行时检查的转换	dynamic_cast<type>(expr)		

运算符概览(2)

对象的大小	sizeof expr	前增量	++lvalue
类型的大小	sizeof(type)	前减量	--lvalue
地址	&lvalue	补	~expr
间接访问	*expr	非(否定)	!expr
建立(分配)	new type	一元负号	-expr
建立(分配并初始化)	new type(expr-list)	一元正号	+expr
建立(放置)	new (expr-list)type		
建立(放置并初始化)	new (expr-list)type(expr-list)	强制(类型转换)	(type)expr
销毁(释放)	delete pointer	销毁数组	delete []pointer

运算符概览(3)

成员选择	object.*printer-to-member		
成员选择	pointer->*pointer-to-member		
乘	expr*expr	除	expr/expr
取模(余数)	expr%expr	加(求和)	expr+expr
减(求差)	expr-expr		
左移	expr << expr	右移	expr>>expr
小于	expr < expr	小于等于	expr <= expr
大于	expr > expr	大于等于	expr >= expr

运算符概览(4)

等于	expr == expr	不等于	expr!=expr
按位与	expr&expr		
按位异或	expr^expr		
按位或	expr|expr		
逻辑与	expr&&expr		
逻辑或	expr||expr		

运算符概览(5)

条件表达式	expr?expr:expr				
简单赋值	lvalue=expr			左移并赋值	lvalue<<=expr
乘并赋值	lvalue*=expr	加并赋值	lvalue+=expr	右移并赋值	lvalue>>=expr
除并赋值	lvalue/=expr	减并赋值	lvalue-=expr	异或并赋值	lvalue^=expr
与并赋值	lvalue&=expr	或并赋值	lvalue|=expr	取模并赋值	lvalue%=expr
抛出异常	throw expr				
逗号(序列)	expr,expr				

运算符示例

```
a+b*c //a+(b*c)
*p++  //先取得p所指向的数据的数值，然后p指针+1
```

一元运算符合赋值运算符是右结合的，其他运算符是左结合的

```
a=b=c // a=(b=c)
a+b+c // (a+b)+c
```

有不多的几条语法规则无法通过优先级和结合性来说明

```
a=b<c ? d=e : f=g
//a=((b<c)? (d=e) : (f=g))
```


6.2.1 结果

- 算术运算符的**结果类型**由一组称为“普通算术转换”的规则确定
  - 例如：二元运算符中有一个是浮点数，那么计算就将通过浮点算术进行；如果有一个long运算对象，那么计算用长整型算术，结果就是long，比int小的运算对象在运算符作用之前将被转换到int
- 关系运算符，==、<=等产生布尔值
- sizeof的结果是一个无符号整型size_t，指针减的结果是一个有符号整型ptrdiff_t，这些类型在<cstddef>中定义

结果示例

```
void f(int x, int y){
    int j = x = y;    // x = y的值是赋值后x的值
    int* p = &++x; // p指向x
    int* q = &(x++); // (t=x,x=x+1,t)
    //错误：x++不是一个左值(它不是存储在x里的值)
    int* pp = &(x>y ? x : y) //较大的那个int的地址
}
void f(){
    int i = 1;
    while( 0<i) i++;
    cout << “i has become negative!” << i << endl;
}
```

6.2.2 求值顺序

- 在一个表达式里，子表达式的求值顺序是没有定义的
- 运算符、逗号、&&、和||保证了位于它们左边的运算对象一定在右边运算对象之前求值(对于&&和||，注意短路算法)
- 括号可以用作强制性的结组

求值顺序示例

```
int x = f(2) + g(3) // 没定义f()和g()哪个先被调用

int i=1;
v[i] = i++; // 结果无定义

b = (a=2, a+1) // b = 3

f1(v[i], i++)     // 两个参数
f2( (v[i], i++) )// 一个参数 i++

a*b/c // (a*b)/c
// != a*(b/c)
```


6.2.3 运算符优先级

- 优先级层次和结合性影响到最普通的使用情况
- 对于程序员来说，如果对某些规则不清楚，就应该使用括号
- 编译器对于很多包含不规范甚至错误使用运算符的表达式可以提出警告

运算符优先级示例

```
if(i<=0 || max<i) // if( (i<=0) || (max <i))
// NOT if( i<= (0||max) < i)

if( i&mask ==0)
// if( i& (mask==0) ) 确实如此,20051012

if(0<=x<=99) //  if( (0<=x) <= 99)
// if( 0<=x && x<=99)

if(a=7) //条件中的常量赋值 应该用==
```

6.2.4 按位逻辑运算符

- 按位逻辑运算符&(交)、|(或)、^(异或)、~(补)、>>和<<可以应用于整型和枚举(bool, char, short, int, long, enum)
- 方便的位操作有可能非常重要，但是，为了可靠性、可维护性、可移植性等，应该将它保持在系统的底层中
- 位域也可以用于作为在一个机器字里移位和掩盖，以便抽取一段二进制位的方便方式

按位逻辑运算符示例

```
enum ios_base :: iostate {
   goodbit=0, eofbit=1, failbit=2, badbit=4
};

state = goodbit;
//...
if(state & (badbit|failbit)) // stream有问题

unsigned short middle(long a)
{
   return (a>>8) & 0xffff;
} //获取a中的第8bit到第23bit
```


6.2.5 增量和减量

- ++运算符用于直接表示相加，使人不必通过加和赋值的组合去间接地表示这个操作，例如：++lvalue意思是lvalue += 1，即lvalue = lvalue + 1。减量也类似地采用--运算符
- 运算符++和--都可以用作前缀或者后缀运算符，++x的值是(增量之后的)新值，例：y=++x等价于y=(x+=1)，x++的值则是x原有的值，例如：y=x++等价于y=(t=x,x+=1,t)，其中t是一个与x同类型的变量

增量和减量

- 运算符++和--也可以用到在对数组元素进行操作的指针上：p++，使p指向下一个元素

```
void cpy(char* p, const char* q){
   while(*p++ = *q++);
}

int length=strlen(q);
for(int i=0;i<=length;i++)
   p[i] = q[i]
// q被遍历了两遍，效率低
```

增量和减量示例

```
int i;
for(i=0;q[i]!=0;i++)
    p[i] = q[i];
p[i] = 0;
//用作下标的i可以去掉

while(*q!=0){
    *p = *q; p++; q++;
}
*p = 0;
//后增量运算符使我们可以先用
//变量的值，然后再增加它
```

```
while(*q!=0){
    *p++ = *q++;
}
*p = 0;
//由于*p++的值也就是*q

while ((*p++ = *q++)!=0)
{ }
//不需要空block
//也不需要和0进行比较

while(*p++=*q++);
```


6.2.6 自由存储

- 一般命名对象的生存时间由它的作用域决定
- 能够使用new操作符建立起生存时间不依赖于建立它作用域的对象，这类对象被称为是“在自由存储里的”或者是“堆对象”或者“在动态存储中分配”
- new创建的对象一直存在，直到使用delete运算符显式地销毁
- delete操作符只能用到new返回的指针或者0，对0操作不会造成任何影响

6.2.6.1 数组

- 可以用new建立对象的数组，比如：char* s = new char[32];
- 删除数组需要使用delete []，比如delete []s;
- 为了释放new分配的空间，delete和delete[]需要能够确定对象分配的空间大小，这意味着通过new分配的对象将会占用比静态对象稍微大一点的空间
- delete[]运算符只能应用于new返回的数组指针，应用到0不会产生任何影响

6.2.6.2 存储耗尽

- 自由存储运算符通过一些在头文件<new>里描述的函数实现：
  - void* operator new(size_t);
  - void operator delete(void*);
  - void* operator new[](size_t);
  - void operator delete[](void*);
- new的标准实现并不对返回的存储做初始化
- 当new无法找到分配的空间时，函数将抛出一个bad_alloc异常

存储耗尽示例

```
void f() {
    try { for(;;) new char[10000]; }
    catch(bad_alloc) { cerr << "Memory exhausted!"; }
}
我们也可以规定存储耗尽时new应该做什么
void out_of_store(){
    cerr << "operator new failed: out of store";
    throw bad_alloc();
}
int main(){
    set_new_handler(out_of_store);
    for(;;) new char[10000];
    cout << "done!";
}
```


6.2.7 显式类型转换

- 有时我们需要处理"原始的数据"，也就是那种保存或者将要保存某种对象的存储，而编译器并不知道对象的类型，例如：一个存储分配程序可能返回一个新分配存储块的void*指针，或者我们希望把某一个整数当成一个I/O设备的地址等

```
void* malloc(size_t);

void f() {
    int* p = static_cast<int*>(malloc(100));
    IO_device* dl = reinterpret_cast<IO_device*>(0xff00);
}
//显式类型转换
```


显式类型转换

- 显式类型转换常常被称做**强制**
- static_cast完成相关类型之间的转换，例如整型到枚举、浮点到整型等的转换
- reinterpret_cast处理互不相干类型之间的转换，例如从整型到指针等的转换
- 其他的类型转换形式还有dynamic_cast(运行中检查)和const_cast(清除const和volatile限定符)

显式类型转换

- C++从C中继承来T(e)记法，这种形式可以执行能用static_cast, reinterpret_cast, const_cast的组合表述的任何转换，它从表达式e出发去做一个T类型的值
- 这种C风格的强制远比上述的命名转换更危险，因为在大的程序里这种记法极难看清楚
- 总之，若感到要使用显式类型转换，请花一些时间考虑是否确实有必要

6.2.8 构造函数

- 从值e构造出一个类型T的值可以用函数记法T(e)表述，比如：int i = int(1.3)，这种结构有时被称作函数风格的强制
- 对于内部类型而言，T(e)等价于(T)e，不安全
- 构造函数记法T()用于描述类型T的默认值，例如：int j = int();
  complex z = complex();
- 对于内部类型，得到的是将0转换到该类型

6.3 语句概览

语句的语法

```
statement:
   declaration
   { statement-list_opt }
   try { statement-list_opt } handler-list
   expression_opt ;

   if( condition ) statement
   if( condition ) statement else statement
   switch ( condition ) statement

   while( condition ) statement
   do statement while ( expression ) ;
   for (for-init-statement condition_opt ; expression_opt ) statement
```


语句的语法

```
   case constant-expression : statement
   default : statement
   break ;
   continue ;

   return expression_opt;

   goto identifier ;
   identifier : statement

statement-list:
   statement statement-list_opt
```


语句的语法

```
condition:
    expression
    type-specifier declarator = expression

handler-list:
    catch { exception-declaration } { statement-list_opt }
    handler-list handler-list_opt
```

6.3.1 声明作为语句

- 一个声明也是一个语句，除非一个变量被声明为static，否则它的初始式的执行就将在控制线程经过这个声明的时候进行
- 允许在所有写语句的地方写声明，是为了使程序员能够最大限度地减少由于未初始化的变量而导致的错误，并使得代码得到更好的局部化

6.3.2 选择语句

- if语句或者switch语句可以检测一个条件表达式的值，并以该值作为条件来决定需要执行的语句
- 比较运算符"== != < <= > >="返回true或者false
- 逻辑运算符"&& ||"除了在必要时，不会去对第二个运算对象求值
- 有时，使用三目表达式或者switch语句能够得到更简洁的代码

选择语句示例

```
if(x) // ...
相当于if(x!=0) // ...

对于指针p
if(p) // ...
相当于 if(p!=0) // ...

if(p && p->count > 1)
// 短路算法

if(a<=b) max = b;
else max = a;

max = (a <= b) ? b : a;
```

```
switch (val) {
   case 1:
      f();
      break;
   case 2:
      g();
      break;
   default:
      h();
      break;
}
```

```
switch (val) {
   case 1:
      cout << "case 1";
   case 2:
      cout << "case 2";
   default:
      cout << "default:";
}
```


6.3.2.1 在条件中的声明

- 为了避免意外地错误使用变量，在最小的作用域里引进变量是一个很好的想法，这样可以避免因为使用未初始化的变量而造成的麻烦
- 在条件中的声明只能声明和初始化单个的变量或者const

在条件中的声明示例

```
if( double d = prim(true)) {
    left /= d;
    break;
}

//上述原则的一个最优雅的应用
```

```
double d;
// ...
d2 = d;
// ...
if(d=prim(true)){
    left /= d;
    break;
}
// ...
d = 2.0;
//d的两个不相干的使用

//此为传统方式
```


6.3.3 迭代语句

- 循环语句可以用for,while或者do语句表述，这些语句都将反复执行一个称为受控语句或者循环体的语句，直到条件变为假，或者程序员要求以其他方式跳出该循环
- 根据作者经验，do语句是错误和混乱的一个根源

```
while ( condition ) statement
do statement while { expression };
for(for-init-statement; condition_opt; expression_opt)
    statement
```


6.3.3.1 for语句里的声明

- 可以在for语句的初始化部分中声明变量

```
void f(int v[], int max)
{
    for( int i = 0; i < max; i++)
        v[i] = i*i;
}
//i的值在for循环的整个过程中均有效
```


6.3.4 goto

- C++拥有臭名昭著的goto语句
- goto语句在高级程序设计中极少有用，但是在那些不是由人写的，而是由某个程序生成出来的C++代码就有可能很有用处
- 此外，在一些优化性能极端重要的程序中，goto也可能非常重要，例如：在某些实时应用的内存循环中
- 标号(label)的作用域是它所在的那个函数

6.4 注释和缩进编排

- 明智得使用注释、一致性地使用缩进编排形式，可以使阅读和理解一个程序的工作变得更轻松愉快
- 注释信息只对读者有用，编译器不会去理解注释的内容
- 糟糕的注释还不如没有

糟糕的注释示例

```
// 变量 “v” 必须初始化
// 变量 “v” 只能由函数 “f()” 使用
// 在调用这个文件中任何其他函数之前调用函数 “init()”
// 在你的程序最后调用函数 “cleanup()”
// 不要使用函数 “weird()”
// 函数 “f()” 有两个参数

a = b + c;  // a变为 b + c
count++;  // count 加1
```


作者的偏爱

- 为每个源文件写一个注释，一般性的陈述在它里面有哪些声明，对有关手册的引用，为维护而提供的一般性提示，如此等等
- 对每个类、模板和名字空间写一个注释
- 对每个非平凡的函数写一个注释，陈述其用途，所用的算法 (除非算法非常明显)，已经可能有的关于它对于环境所做的假设
- 对每个全局的和名字空间的变量和常量写一个注释
- 在非明显或不可移植的代码处的少量注释
- 极少其他的东西

6.5 忠告

- [1]应尽可能使用标准库，而不是其他的库和 “手工打造的代码”;
- [2]避免过于复杂的表达式;
- [3]如果对运算符的优先级有疑问，加括号;
- [4]避免显式类型转换 (强制);
- [5]若必须做显式类型转换，提倡使用特殊强制运算符，而不是C风格的强制;
- [6]只对定义良好的构造使用T(e)记法;
- [7]避免带有无定义求值顺序的表达式;
- [8]避免goto语句;

忠告

- [9]避免do语句；
- [10]声明变量时，最好同时进行初始化；
- [11]使注释简洁、清晰、有意义；
- [12]保持一致的缩进编排风格；
- [13]倾向于去定义一个成员函数operator new()去取代全局的operator new()；
- [14]在读输入的时候，总应考虑病态形式的输入；

C++编程(5)

唐晓晟
北京邮电大学电信工程学院

第七章 函数

- 函数声明
- 参数传递
- 返回值
- 重载函数名
- 默认参数
- 未确定数目的参数
- 指向函数的指针
- 宏
- 忠告

7.1 函数声明

- 函数是编程人员为了完成某一个任务而编写的可以被重复使用的代码
- 函数只有在预先声明之后才能被调用
- 函数声明中，需要给出函数的名称、返回值类型以及调用该函数时必须提供的参数个数和类型
- 参数传递的语义等同于初始化的语义，必要时，会进行隐式的类型转换
- 函数声明中可包含参数的名字，但是编译器将忽略这样的名字

函数声明示例

```
Elem* next_elem();
char* strcpy(char* to, char* from);
void exit(int);

double sqrt(double);

double sr2 = sqrt(2);
double sr3 = sqrt("three"); // Error

double sqrt(double temperature);
```

7.1.1 函数定义

- 程序中调用的函数必须在某个地方定义(仅仅一次)，函数定义相当于给出了函数体的函数声明
- 函数的定义和对它的所有声明都必须描述了同样的类型，因为参数名字不是类型的一部分，所以参数名不必保持一致
- 函数可以被定义为inline(内联)的

函数定义示例

```
extern void swap(int*, int*); // 声明

void swap(int* p, int* q)  // 定义
{
   int t = *p;
   *p = *q;
   *q = t;
}

inline int fac(int n)
{
   return (n<2) ? 1 : n*fac(n-1);
}
```


7.1.2 静态变量

- 局部变量将在运行线程达到其定义时进行初始化，这种情形发生在函数的每次被调用时，且函数的每次调用都有自己的一份局部变量副本
- 若局部变量被声明为static，那么将只有唯一的一个被静态分配的对象，在该函数的所有调用中该对象唯一，这个对象在该函数第一次被调用时被初始化

静态变量示例

```
void f(int a)
{
   while(a--) {
      static int n = 0; //初始化仅一次
      int x = 0;  //每次f()被调用都初始化
      cout << "n="<<n++<<",x="<<x++<<'\n';
   }
}

int main()
{
   f(3);
}
```

输出结果：
n=0,x=0
n=1,x=0
n=2,x=0

7.2 参数传递

- 当一个函数被调用时，将安排好其形式参数所需要的存储，并用实际参数对其进行初始化，必要时需要进行类型转换
- 传递一个引用参数一般意味着要修改这个参数，为了效率原因传递引用参数，可以将其声明为const类型，显式表明将不对其数值进行修改
- 文字量、常量和需要转换的参数都可以传递给const&参数，但不能传递给非const引用

参数传递示例(1)

```
void f(int val, int& ref)
{
   val ++;  ref ++;
}
```

```
void g()
{
   int i = 1;
   int j = 1;
   f(i,j);
   // i不变，j变为2
}
```

```
int strlen(const char*);
// 求C风格的字符串的长度
char* strcpy(char* to, const char* from);
// 复制C风格的字符串
int strcmp(const char*, const char*);
// 比较C风格的字符串
```


参数传递示例(2)

```
float fsqrt(const float&);  // 要求类型是常量引用类型
void g(double d)
{
   float r = fsqrt(2.0f);    // 传递的是保存2.0f临时量的引用
   r = fsqrt(r);    // 传递r的引用
   r = fsqrt(d);    // 传递的是保存float(d)的临时量的引用
}
void update(float& i);  // 要求参数为普通引用类型
void g(double d, float r)
{
   update(2.0f);   // Error: 传递的参数是const
   update(r);      // 传递r的引用
   update(d);      // Error: 要求类型转换，否则update将会
}                  // 更新临时变量float(d)的数值
```


7.2.1 数组参数

- 如果将数组作为函数的参数，传递的就是到数组的首元素的指针，即：类型T[]作为参数传递时将被转换为一个T*
- 对数组参数的某个元素的赋值，将改变实际参数数组中的那个元素的值，也就是说，数组不会(也不能)按传值的方式传递
- 传递数组时，注意数组参数的大小问题

数组参数示例

```
int strlen(const char*);
void f()
{
    char v[] = “an array”;
    int i = strlen(v);
    int j = strlen(“Nicholas”);
}

void compute1(int* vec_ptr, int vec_size); // 一种方法

struct Vec{
    int* ptr;
    int size;
}
        void compute2(const Vec& v);
        // 另一种方法
```


7.3 返回值

- 没有被声明为void的函数都必须返回一个值，void函数不能有返回值
- 返回值由返回语句(return)描述，一个函数中可以有多个返回语句，返回值的语义和初始化的语义相同
- 不要返回指向局部变量的指针

返回值示例

```
int f1(){ } // Error: 无返回值
void f2() { } // ok
int f3() { return 1; } // ok
void f4() { return 1; } // Error
int f5() { return; } // Error
void f6() { return; } // ok

int fac(int n)
{  return (n>1) ? n*fac(n-1):1;
}// 调用自己的函数称为递归函数

int fac2(int n){
   if(n>1) return n*fac2(n-1);
   return 1; } // 可以有多个返回语句
```

```
double f() { return 1; }
// 1被隐式转换为double(1)

int* fp() { int local = 1;
/* … */ return &local; }
// Error

int& fr() { int local = 1;
/* … */ return local; }
// Error

void g(int* p);
void h(int* p) { /* … */
return g(p); } // ok
```


7.4 重载函数名

- 函数在不同类型的对象上执行概念相同的工作时，可以给多个函数起相同的名字，叫做重载
- 当被重载的函数f被调用时，编译器必须弄清楚应该调用具有名字f的哪一个函数，依据是：实际参数的类型与哪个f函数的形式参数匹配的最好，就调用哪个，如果找不到匹配最好的函数，则给出编译错误
- 重载使得程序员不必为解决某类问题记忆过多的函数名称

编译器的匹配判断原则

- 准确匹配：无需任何转换或者只需做平凡转换(数组名到指针、函数名到函数指针、T到const T等)的匹配
- 利用提升的匹配：即包含整数提升(bool到int、char到int、short到int以及相应的无符号版本)以及从float到double的提升
- 利用标准转换(int到double、double到int、double到long double、Derived*到Base*、T*到void*、int到unsigned int)的匹配
- 利用用户定义转换的匹配
- 利用在函数声明中的省略号的匹配
- 若在匹配的某个层次上同时发现两个匹配，这个调用将被作为歧义而遭拒绝

重载函数名示例

```
void print(double);
void print(long);

void f()
{
   print(1L);
   // print(long)
   print(1.0);
   // print(double)
   print(1);
   // Error:歧义
}
```

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);

void h(char c, int i, short s, float f)
{  print(c); print(i); // 准确匹配
   print(s); // 整数提升：print(int)
   print(f); // float到double提升
   print('a'); print(49); // 准确匹配
   print(0);  // 准确匹配：print(int)
   print("a");
   // 准确匹配：print(const char*) }
```


7.4.1 重载和返回类型

- 重载解析中不考虑返回类型，其理由是：保持对重载的解析只是针对单独的运算符或者函数调用，与调用的环境无关

```
float sqrt(float);
double sqrt(double);
void f(double d, float f)
{
   float fl = sqrt(d);  // sqrt(double)
   double db = sqrt(d); // sqrt(double)
   fl = sqrt(f);  // sqrt(float)
   db = sqrt(f);  // sqrt(float)
}
```


7.4.2 重载与作用域

- 在不同的非名字空间作用域里声明的函数不算是重载

```
void f(int);

void g()
{
   void f(double);
   f(1);  // 调用f(double)
}
```

7.4.3 手工的歧义性解析

- 对一个函数，声明的重载版本过多(或者过少)都有可能导致歧义性
- 只要可能，应该做的就是将该函数的重载版本集合作为一个整体来考虑，有关问题通常可以通过增加一个消除歧义性的版本来解决
- 此外，也可以通过增加一个显式类型转换的方式去解决某个特定调用的问题，但是这样做通常只是权益之计

手工的歧义性解析示例

```
void f1(char);
void f1(long);

void f2(char*);
void f2(int*);

void k(int i)
{
   f1(i); // 歧义
   f2(0); // 歧义
}
```

解决方案

```
void f1(int n)
{
   f1(long(n));
}

f2(static_cast<int*>(0));
```


7.4.4 多参数的解析

- 基于上述重载解析规则，可以保证：当所涉及到的不同类型在计算效率或者精度方面存在明显差异时，被调用的将会是最简单的算法(函数)

多参数的解析示例

```
int pow(int,int);
double pow(double,double);
complex pow(double,complex);
complex pow(complex,int);
complex pow(complex,double);
complex pow(complex,complex);
void k(complex z)
{
   int i = pow(2,2);  // pow(int,int)
   double d = pow(2.0,2.0); // pow(double,double)
   complex z2 = pow(2,z);  // pow(double,complex)
   complex z3 = pow(z,2);  // pow(complex,int)
   complex z4 = pow(z,z);  // pow(complex,complex)
}
```

注意：
double d = pow(2.0,2);
将会引起歧义

7.5 默认参数

- 一个通用函数所需要的参数常常比处理简单情况时所需要的参数更多一些
- 默认参数的使用可以为编程增加灵活性
- 默认参数的类型将在函数声明时检查，在调用时求值
- 只能对排列在最后的那些参数提供默认参数

默认参数示例

```
void print(int value,
      int base = 10);
void f() {
   print(31);
   print(31,10);
   print(31,16);
   print(32,2);
}
```

输出：31 31 1f 11111

替代方案：

```
void print(int value, int base);
void print(int value){ print(value,10);}
```

但是，读者不容易看出原来的意图：一个函数加上一种简写形式

声明

```
int f(int,int =0, char* =0); // ok
int g(int =0, int =0, char*);//Error
int h(int =0,int, char* =0);//Error
int nasty(char*=0); //Syntax Error

void f(int x = 7);
void f(int = 7); // Error:参数重复
void f(int = 8); // Error:参数值改变
```

```
void g()
{
   void f(int x=9);
   // ok
}
```


7.6 未确定数目的参数

- 有些函数，无法确定在各个调用中所期望的所有参数的个数和类型，声明这种函数的方式就是在参数表的最后用省略号(...)结束，表示还可能有另外一些参数
- 对于省略号省略的参数，编译器在编译时无法对其进行参数检查，即：可能编译通过，但是运行出错，C++中可以通过重载函数和使用默认参数避免此类问题
- <cstdarg>里提供了一组标准宏，专门用于在这种函数里访问未加描述的参数

未确定数目的参数示例

```
int printf(const char* ...);

printf("Hello, world!\n");
printf("My name is %s %s\n", first_name, last_name);
printf("%d + %d = %d\n", 2,3,5);

#include <stdio.h>
int main()
{
   printf("My name is %s %s\n",2);
} // 最好情况下，会有一些很奇怪的输出

int fprintf(FILE*, const char* ...);
int execl(const char* ...);
```

请参看书中p139，error函数实现

7.7 指向函数的指针

- 对一个函数只能做两件事：调用它或者取得它的地址
- 通过取一个函数的地址而得到的指针即为函数指针，可以在后面用来调用这个函数
- 在指向函数的指针的声明中需要给出参数类型，就像函数声明一样，在指针赋值时，完整的函数类型必须完全匹配

函数指针示例

```
void error(string s) { /* ... */ }
void (*efct)(string); //函数指针
void f()
{
   efct = &error; // efct指向error
   efct("error");  // 调用error
}          // 不需要间接运算符*

void (*f1)(string) = &error;//ok
void (*f2)(string) = error; // ok

void g() {
   f1("Vasa");  // ok
   (*f2)("Mary Rose"); //ok }
```

```
void (*pf)(string);
void f1(string);
int f2(string);
void f3(int*);

void f() {
   pf = &f1; // ok
   pf = &f2;
   // Error:返回值不匹配
   pf = &f3; // Error
   pf("Hera"); // ok
   pf(1); // Error
   int i = pf("Zeus"); }
   // Error: void赋值给int
```


其他函数指针示例

```
// 摘自<signal.h>
typedef void (*SIG_TYP)(int); //对比 typedef char* PCH;
typedef void (*SIG_ARG_TYP)(int);
SIG_TYP signal(int,SIG_ARG_TYP);

typedef void (*PF)();
PF edit_ops[] = { &cut, &paste, &copy, &search };
PF file_ops[] = { &open, &append, &close, &write };

PF* button2 = edit_ops;
PF* button3 = file_ops;

button2[2](); //调用按钮2的第3个函数
```

请参看书中p141, ssort函数的实现，以及p142中对其的应用举例

函数指针示例

- 关于重载函数的指针，可以通过赋值或者初始化指向函数的指针方式，取得一个重载函数的地址
- 通过指向函数的指针调用的函数，其参数类型和返回值类型必须与指针的要求完全一致
- 当用函数对指针赋值或初始化时，没有隐含的参数或者返回值类型转换

```
void f(int);
int f(char);

void (*pf1)(int) = &f; // void f(int)
int (*pf2)(char) = &f; // int f(char)
void (*pf3)(char) = &f; // Error
```

7.8 宏

- 第一规则：绝不应该去使用它，除非你不得不这么做
- 几乎每个宏都表明了程序设计语言中、或者程序里、或者程序员的一个缺陷
- 宏使得编译器在真正处理正文之前需要进行预处理操作，编译器看到的是宏展开后的结果，可能导致非常难以理解的错误

关于宏使用中的注意事项

- 宏预处理器不能处理递归的宏
- 宏的名字不能重载
- 在宏中，引用全局名字时一定要使用作用域解析运算符::，并在所有可能的地方将出现的宏参数都用括号围起来
- 宏中的注释请使用/* */方式
- 通过##宏运算符可以拼接起两个串，构造出一个新串

宏的示例(1)

```
#define NAME rest of line
named = NAME
// named = rest of line

#define MAC(x,y) arg:x arg:y
expanded = MAC(foo bar, yuk yuk)
// expanded = arg: foo bar arg: yuk yuk

#define PRINT(a,b) cout << (a) << (b)
#define PRINT(a,b,c) cout << (a) << (b)<< (c) // Error
#define FAC(n) (n>1)?n*FAC(n-1):1 // Error

#define MIN(a,b) (((a) < (b)) ? (a) : (b) )
#define M2(a) something(a) /* 细心的注释 */
```


宏的示例(2)

```
可能有用的宏 #define FOREVER for(;;)
没有必要的宏 #define PI 3.14159
很危险的宏    #define SQUAR(a) a*a
    // int xx = 0; int y = square(xx+2);

#define NAME2(a,b) a##b
int NAME2(hack,cah)();
// 将产生 hackcah();

#undef X
//保证不再有称为X的有定义的宏
```

7.8.1 条件编译

- 根据条件决定某一段代码是否让编译器编译
- #ifdef identifier将条件性地导致随后的输入被忽略，直到遇到一个#endif指令

```
int f(int a
#ifdef arg_two
, int b
#endif
);
```

将产生 int f(int a
); //除非宏arg_two存在

7.9 忠告

- [1]质疑那些非const的引用参数；如果你想要一个函数去修改其参数，请使用指针或者返回值
- [2]当你需要尽可能减少参数复制时，应该使用const引用参数
- [3]广泛而一致地使用const
- [4]避免宏
- [5]避免不确定数目的参数
- [6]不要返回局部变量的指针或者引用

忠告

- [7]当一些函数对不同的类型执行概念上相同的工作时，请使用重载
- [8]在各种整数上重载时，通过提供函数去消除常见的歧义性
- [9]在考虑使用指向函数的指针时，请考虑虚函数或模板是不是更好的选择
- [10]如果你必须使用宏，请使用带有许多大写字母的丑陋的名字

C++编程(6)

唐晓晟
北京邮电大学电信工程学院

第八章 名字空间和异常

- 模块化和界面
- 名字空间
- 异常
- 忠告

8.1 模块化和异常

- 一个大型系统是由多个模块组成的，编程中，模块可以被理解为是完成某项功能的一段代码
- 一个模块使用另一个模块时，并不需要知道被调用模块的所有细节，因此，需要将一个模块和它的界面区分开来(形式上的)，这是开发大型程序的有用技术
- 划分模块并不难，难的是提供跨过模块边界的安全、方便而有效的通信

- 程序的整个结构中分布着错误处理，编写时，特别要注意将由错误处理造成的模块之间的相互依赖减到最小
- C++提供的异常机制，可以用于降低检查、报告错误和处理错误之间的联系程度

8.2 名字空间

- 名字空间是一种描述逻辑分组的机制，如果有一些声明按照某种准则在逻辑上属于同一个集团，就可以将它们放入同一个名字空间
- 名字空间描述了程序的界面，一般在实现具体细节前进行，所以一般和函数的具体实现是分开的
- 成员可以在名字空间的定义里声明，而后再采用namespace-name::member-name形式去定义
- 不能在名字空间定义之外用加限定的语法形式为名字空间引进新成员
- 一般来说，程序中的每个声明都必须位于某个名字空间内，但是main()函数例外

理想情况下，一个名字空间应该

- 描述一个具有逻辑统一性的特征集合
- 不为用户提供对无关特征的访问
- 不给用户强加任何明显的记述负担

示例

```
namespace Parser{
    double expr(bool);
    double prim(bool get){/* … */}
    double term(bool get){/* … */}
    double expr(bool get){/* … */}
} //声明和实现同时进行

namespace Parser{
    double expr(bool);
    double prim(bool);
    double term(bool);
}//实现与界面分开
double Parser::expr(bool get){/* …*/}
double Parser::prim(bool get){/* …*/}
double Parser::term(bool get){/* …*/}
```

名字空间的成员必须采用如下记法形式引入：

```
namespace name {
    // 声明和定义
}
```

```
void Parser::logical(bool);
// Error, Parser中无logical,
// 也无法引入新名字logical
void Parser::prim(int);
// Error, prim要求bool参数
```


8.2.1 带限定词的名字

- 名字空间是作用域，普通的作用域规则也对名字空间成立
- 如果一个名字先前已经在本名字空间里或者其外围作用域里声明过，就可以直接使用。
- 也可以使用来自另外一个名字空间的名字，但需要该名字所属的名字空间作为限定词

带限定词的名字示例

```
namespace Lexer{   //定义一个新的namespace
    enum Token_value { NAME, NUMBER, END, MUL='*'};
    Token_value curr_tok;
    double number_value;
    string string_value;
    Token_value get_token() { /* … */}
}
            double Parser::term(bool get) // Parser作用域
            {
                double left=prim(get);  // 不需要限定词
                for(;;) switch(Lexer::curr_tok) { // Lexer
                    case Lexer::NAME:                 // Lexer
                        left *= prim(true);  // 不需要限定词
                    // …
                 } /* … */ }
```


8.2.2 使用声明

- 当编程中频繁使用另外一个名字空间中的变量时，可以通过使用声明语句来避免反复书写名字空间限定词

```
double Parser::prim(bool get)
{
   if(get) Lexer::get_token();
   switch(Lexer::curr_tok){
      case Lexer::NUMBER:
         Lexer::get_token();
      // …
   }
}
```

```
double Parser::prim(bool get)
{
    using Lexer::get_token;
    using Lexer::curr_tok;

    if(get) get_token();
    switch(curr_tok){
       case Lexer::NUMBER:
            get_token(); // …
            }}
```


- 也可以把有关的使用说明放在Parser名字空间的定义里

```
namespace Parser{
   double prim(bool);
   double term(bool);
   double expr(bool);

   using Lexer::get_token;
   using Lexer::curr_tok;
   using Error::error;
}
```

```
double Parser::term(bool get)
{
   double left = prim(get);
   for(;;)
   switch(curr_tok) {
      case Lexer::MUL:
         left *= prim(true);
         break;
      case Lexer::DIV:
         if(double d = prim(true))
         {left /= prim(true); break;}
         return error("divided by 0");
   /* … …  */ }}
```


8.2.3 使用指令

- 一个使用指令能把来自一个名字空间的所有名字都变成可用的

```
namespace Parser{
   double prim(bool);
   double term(bool);
   double expr(bool);

   using namespace Lexer;
   using namespace Error;
}
```

8.2.4 多重界面

- 对于名字空间，编程者看到和用户看到的界面往往是没有必要相同的，用户看到的界面通常要比编程者所看到的简单的多
- 一般情况下，编程界面变化比较频繁
- 设计界面时要考虑两种界面，同时注意尽量避免编程界面的变化影响到用户界面

```
//给实现的界面
namespace Parser{
    double prim(bool);
    double term(bool);
    double expr(bool);

    using namespace Lexer;
    using namespace Error;
}

//给使用者的界面
namespace Parser{
    double expr(bool);
}
```


8.2.4.1 界面设计的各种选择

- 界面的作用就是尽可能减少程序不同部分之间的相互依赖，最小的界面将会使程序易于理解，有很好的数据隐蔽性质，容易修改，也编译的更快
- 对于上面描述的两种界面(用户和编程)，我们希望能做到：呈现一个易于理解的用户界面，该界面尽量少受编程界面的影响(至少表面看上去是这样)

界面设计的两种选择

```
//编程界面
namespace Parser{
    // …
    double expr(bool);
    // …
}
//用户界面
namespace Parser_interface{
    using Parser::expr;
}
```

看上去，用户界面还是很容易受到编程界面的伤害，原本希望能将其相互隔离

```
//编程界面不变
//用户界面
namespace Parser_interface{
    double expr(bool);
}

double Parser_interface::
    expr(bool get)
{
    return Parser::expr(get);
}
```

现在所有的依赖性都已最小化
当然，对于我们所面对的大部分问题而言，这种解决方案太过分了

8.2.5 避免名字冲突

- 名字空间就是为了表示逻辑结构
- 最简单的这类结构的应用就是为了分清楚一个人写的代码和另一个人写的代码
- 一般情况下，若只使用一个单独的名字空间，当从一些相互独立的部分组合起一个程序时，就可能遇到一些不必要的困难，比方说，同名函数或者变量的冲突问题

避免名字冲突示例

```
//my.h
char f(char);
int f(int);
class String{
 /* … */ };

//you.h
char f(char);
double f(double);
class String{
  /* … */ };
```

一般情况下，第三方很难同时使用my.h和you.h

解决方法：

```
namespace My{
    char f(char); int f(int);
    class String{/*…*/};
}
namespace You{
    char f(char); double f(double);
    class String{/*…*/};
}
```

使用时：

```
1 My::f(char), You::f(double)
2 using My::f; using You::String;
3 using namespace My; 或者
  using namespace You;
```


8.2.5.1 无名名字空间

- 有时，将一组声明包在一个名字空间内部就是为了避免可能的名字冲突
- 这样做的目的只是为了保持代码的局部性，而不是为了给用户提供界面
- 此时，我们可以使用无名名字空间

无名名字空间示例

```
#include “header.h”
namespace Mine{
    int a;
    voif f(){/*…*/}
    int g(){/*…*/}
}

//使用无名名字空间
#include “header.h”
namespace{
    int a;
    voif f(){/*…*/}
    int g(){/*…*/}
}
```

相当于：

```
#include “header.h”
namespace $$${
    int a;
    voif f(){/*…*/}
    int g(){/*…*/}
}
using namespace $$$;
//其中，$$$是在这个名字空间
//定义所在的作用域里具有唯一
//性的名字，在不同的编译单位
//里的无名名字空间也互不相同

//保证名称的私有性，避免冲突
```


8.2.6 名字查找

- 一个取T类型参数的函数常常与T类型本身定义在同一个名字空间里，因此，如果在使用一个函数的环境中无法找到它，我们就去查看它的参数所在的名字空间

```
namespace Chrono{
    class Date{/*…*/};
    bool operator==(const Date&,
        const std::string&);
    std::string
        format(const Date&);
}
```

```
void f(Chrono::Date d, int i){
    std::string s = format(d);
    //Chrono::format()
    std::string t = format(i);
    //Error:找不到使用int输入
    //参数的format()版本
}
```

名字空间示例

- 名字查找规则能够使程序员节省许多输入，同时也不必使用using指令污染名字空间，这个规则对于运算符的运算对象和模板参数特别有用
- 需要注意的是，名字空间本身必须在作用域里，函数也必须在它被寻找和使用之前声明

```
void f(Chrono::Date d, std::string s)
{
    if(d==s) { // …
    }
    else if(d == “August 4, 1914”) {  // …
    }
} //此函数最终会调用Chrono名字空间里的operator==
```


8.2.7 名字空间别名

- 用户给名字空间取很短的名字(例如namespace A)可能会出现冲突，但是如果给名字空间取很长的名字(例如namespace American_Telephone_and_Telegraph)又很不实用
- 可以通过为长名字提供较短的别名的方式解决此类问题

名字空间别名

```
namespace ATT = American_Telephone_and_Telegraph;
ATT::String s3 = “Grieg”;
ATT::String s4 = “Nielsen”;

namespace Lib = Foundation_library_v2r11;
//…

Lib::set s;
Lib::String s5 = “Sibelius”;
//将来Foundation库的版本更新了，只需要修改别名
//Lib的初始化语句并重新编译即可，极大的简化了升
//级工作
//注意：程序中过多的使用别名也会引起混乱
```


8.2.8 名字空间组合

- 有时候我们需要从现存的界面出发组合出新的界面

```
namespace His_string{
    class String{/*…*/};
    String operator+(const String&, const String&);
    String operator+(const String&, const char*);
    void fill(char);
}
namespace Her_vector{
    template<class T> class Vector{/*…*/};
}
namespace My_lib{
    using namespace His_string;
    using namespace Her_vector;
    void myfct(String&);}
```

名字空间组合示例

```
void f() {
   My_lib::String s = “Byron”;
  //系统能够查到是
  // My_lib::His_string::String
}

using namespace My_lib;
void g(Vector<String>& vs)
{
   //…
   my_fct(vs[5]);
   //…
}//Vector以及String都是My_lib
  //中可以查到的
```

仅当我们需要去定义什么东西时，才需要知道一个实体真正所在的名字空间

```
void My_lib::fill(char c){
    //…
} // Error
void His_string::fill(char c){
    //…
}// ok
void My_lib::my_fct(String& v)
{
    // …
}// ok String是My_lib::String
//即 His_string::String
```


8.2.8.1 选择

- 有时我们只是想从一个名字空间里选用几个名字，可以通过写一个仅仅包含我们想要的声明的名字空间来达到这个目的

```
namespace His_string{ // 只有His_string的一部分
    class String{/*…*/};
    String operator+(const String&, const String&);
    String operator+(const String&, const char*);
} // 注意：此处的His_string是一个声明
```

- 但是，这种做法会引起混乱，任何对原来His_string的实现部分做的修改在这里无法体现出来(比方说：添加一个+的重载函数)

改进方案

- 通过using语句，使得从名字空间里选择一些特征的事变得更加明确

```
namespace My_string{
    using His_string::String;
    //如果String被改动，这里自然会有所变化
    using His_string::operator+;
    //一条using语句就可以将operator+的所有重载都包含进来
}
```


8.2.8.2 组合和选择

- 将组合(通过using指令)和选择(通过using声明)结合起来能产生更多的灵活性，这些都是真实世界的例子所需要的

```
namespace His_lib{
   class String{/*…*/};
   template<class T> class
        Vector{/*…*/};
}
namespace Her_lib{
   template<class T> class
      Vector{/*…*/};
   class String{/*…*/};
}
```

```
namespace My_lib{
   using namespace His_lib;
   using namespace Her_lib;
   using His_lib::String;
   //以偏向His_lib的方式解析
   //潜在的冲突
   using Her_lib::Vector;
   //同上
   template<class T> class
      list{/*…*/};
}
```

组合与选择

- 查看一个名字空间时，其中显式声明的名字(包括通过using指令声明的名字)优先于在其他作用域里的那些通过using指令才能访问的名字
- 例如：对于String和Vector名字冲突的解析将分别偏向于His_lib::String和Her_lib::Vector，此外，My_lib::List总会被使用，和Her_lib或者His_lib中是否提供了List没有任何关系

8.2.9 名字空间和老代码

- 现存有数以百万行计的C和C++代码，我们若想使用这些代码，必须通过名字空间缓和这种代码之间的名字冲突问题
- 幸运的是，对于C代码，我们可以继续使用它们（就像它们是在名字空间里定义的一样），但是，对于C++代码则无法做到这一点，只能尽可能减少由于设计时名字空间的引入给已有的C++程序带来的破坏

8.2.9.1 名字空间和C

```
#include <stdio.h>
int main()
{
    printf("Hello, world!\n");
}

//标准C中stdio.h可以如下定义
namespace std{
    int printf(const char* ...);
    //...
}
using namespace std;
```

```
//cstdio，给那些不希望一大批
//名字都能隐式地随意使用的人
namespace std{
    int printf(const char* ...);
    //...
}

//真正C++标准库中的stdio.h
#include <cstdio>
using std::printf;
//...
```


8.2.9.2 名字空间和重载

- 重载可以跨名字空间工作

```
//老的A.h
void f(int);

//老的B.h
void f(char);

//老的user.c
#include “A.h”
#include “B.h”
void g()
{
    f(‘a’); //调用B.h中的f()
}
```

```
//新的A.h
namespace A{
    void f(int);
}
```

```
//新的B.h
namespace B{
    void f(char);
}
```

```
//新的user.c
#include “A.h”
#include “B.h”
using namespace A;
using namespace B;
void g() {
    f(‘a’); //调用B.h中的f()
}
```

8.2.9.3 名字空间是开放的

- 可以通过多个名字空间声明给它加入名字
- 注意，当定义一个名字空间中已经声明过的成员时，安全的方法是采用A::语法格式

```
namespace A{   namespace A{
    int f();          int g();
    //A中现有f()     //A中现有f()和g()
}                  }

void A::ff()
{    // ...
}
//Error:A中没有ff()

//不要使用如下方式
namespace A{
    void ff() {/*...*/};
} // 这样会引入一个新的函数ff()，而不是
// 实现已经声明过的ff()函数，编译器不报错
```


8.3 异常

- 当一个程序是由一些相互分离的模块组成时，特别是当这些模块来自某些独立开发的库时，错误处理的工作就需要分成两个相互独立的部分
  - 一方报告那些无法在局部解决的错误
  - 另一方处理那些在其他地方检查出来的错误
- 异常(exception)机制是C++中用于将错误报告与错误处理分开的手段

8.3.1 抛出和捕捉

- 异常概念就是为了帮助处理错误的报告
- 基本思想是：如果函数发现了一个自己无法处理的问题，它就抛出(throw)一个异常，希望它的(直接或间接)调用者能够处理这个问题，如果一个函数想处理某个问题，它就可以说明自己要捕捉(catch)用于报告该种问题的异常

异常处理示例

```
struct Range_error{
   int i;
   Range_error(int ii)
   {i=ii;}
};
char to_char(int i)
{
   if(i<numeric_limits<char>::min()
   || numeric_limits<char>::max()<i)
      throw Range_error(i);
   return i;
}

void g(int i)
{
   try {
      char c = to_char(i);
      //...
   }
   catch(Range_error x){
      cerr << “oops\n”;
      cerr << x.i << endl;
   }
}
```

其中catch(/*...*/){/*...*/}称为异常处理器，它只能紧接着由try关键字作为前缀的块之后，或者紧接在另一个异常处理器之后

异常的处理过程

- 如果在一个try块中的任何代码(或者由那里调用的东西)抛出了一个异常，代码马上会从产生异常的地方跳转到try后面的第一个异常处理器
- 如果所抛出的异常属于某个处理器所描述的类型，这个处理器就被执行
- 如果try未抛出异常，这些异常处理器都将被忽略
- 如果抛出了一个异常，而又没有catch块捕捉它，整个程序就将终止

8.3.2 异常的辨识

- 典型情况下，一个程序中可能存在多种不同的运行时错误，可以将这些错误映射到一些具有不同名字的异常，为每一种异常取一个系统中唯一的名字是比较好的解决方法
- 处理器中不需要break语句
- 一个函数不必捕捉所有可能的异常
- 异常处理器也可以嵌套

异常的辨识示例

```
struct Zero_divide{};
struct Syntax_error{
   const char* p;
   Syntax_error(const char* q)
   {   p = q; }
};

try {
   //...
   expr(false);
   //...
}
```

```
catch(Syntax_error se){
   // 处理语法错误
}
catch(Zero_divide zd){
   // 处理除以0错误
}
//若expr执行过程中没有异常
//或者出现异常但是被捕捉到
//程序代码将执行到这里
```


8.3.3 计算器中的异常

- 利用异常处理机制来完成以前计算器程序中的错误处理
- 这样做的好处是减少原来代码中程序各个部分的耦合程度，主要是其他部分和错误处理部分的耦合程度
- 原来的那种方式不能很好的用到由分别开发的库组合而成的程序中
- 例子见课本p170和p171

8.3.3.1 其他的错误处理策略

- 状态变量是混乱和错误的一个常见根源，特别是允许它们大量出现并影响程序中较大的片断时
- 一般的，使处理错误的代码与“正常”代码分离是一种很好的策略
- 在导致错误的代码的同一个抽象层次上处理错误是非常危险的(完成错误处理的代码有可能又产生了引起错误处理的那个错误)
- 修改“正常”的代码，加上错误处理代码，所无地工作量比增加单独的错误处理例程更多

8.4 忠告

- [1] 用名字空间表示逻辑结构
- [2] 将每个非局部的名字放到某个名字空间里，除了main()之外
- [3] 名字空间的设计应该让你能很方便地使用它，而又不会意外地访问了其他的无关名字空间
- [4] 避免对名字空间使用很短的名字
- [5] 如果需要，通过名字空间别名去缓和长名字空间名的影响
- [6] 避免给名字空间的用户添加太大的记法负担

忠告

- [7] 在定义名字空间的成员时使用namespace::member的形式
- [8] 只有在转换时，或者在局部作用域里，才用using namespace
- [9] 利用异常去松弛“错误”处理代码和正常代码之间的联系
- [10] 采用用户定义类型作为异常，不要使用内部类型
- [11] 当局部控制结构足以应付问题时，不要使用异常

C++编程(7)

唐晓晟
北京邮电大学电信工程学院

第九章 源文件和程序

- 分别编译
- 连接
- 使用头文件
- 程序
- 忠告

9.1 分别编译

- 文件是传统的存储单位和传统的编译单位
- 通常，将一个完整的程序放入一个文件是不可能的(标准库和操作系统)，而且不实际、不方便，将程序组织成一些文件的方式，可以强调它的逻辑结构，帮助读者理解程序
- 一些文件的方式，可帮助编译器去强迫实施这种逻辑结构，也能节省编译时间
- 源文件(source file)提交给编译器后进行预处理(处理宏、#include等)，形成编译单位

分别编译

- 为了使分别编译能够工作，程序员必须提供各种声明，为孤立地分析一个编译单位提供有关程序其他部分的类型信息
- 在一个由许多分别编译的部分组成的程序里，这些声明必须保持一致，连接器(linker有时也被混乱地称作装载器loader)能检查出许多类型的不一致
- 连接工作可以在程序开始运行之前完全做好，另外也允许程序开始运行后为其加入新代码(动态连接)

9.2 连接

- 在所有的编译单位中，对所有函数、类、模板、变量、名字空间、枚举和枚举符号的名字的使用必须保持一致
- 所有名字空间、类、函数等都应该在它们出现的各个编译单位中有适当的声明，而且所有声明都应该一致地引用同一个实体
- 如果一个名字可以在与其定义所在的编译单位不同的地方使用，就说它是具有"外部连接"的，否则就称其为具有"内部连接"的

连接示例(1)

```
//file1.c
   int x = 1;
   int f(){/*...*/}

//file2.c
   extern int x;
   int f();
   void g(){x=f();}

//ok
//file2.c使用了file1.c
//中的x和f()
```

```
//file1.c
   int x = 1;
   int b = 1;
   extern int c;

//file2.c
   int x;
   extern double b;
   extern int c;

//三个错误
//x定义了两次
//b类型不一致
//c只有声明，没有实现
```

```
//file1.c
   int x;
   int f(){ return x; }

//file2.c
   int x;
   int g() {return f();}

//两个错误
//x定义了两次
//file2中无法调用f()
//因为没有声明
```


连接示例(2)

```
//默认情况下
//const和typedef具有"内部连接"
//file1.c
   typedef int T;
   const int x=7;

//file2.c
   typedef void T;
   const int x=8;

//ok
//全局的const和inline一般
//应该放到头文件中去
//extern const int a = 77
//显式声明使const具有"外部连接"
```

```
//无名名字空间的效果很像"内部连接"
//file1.c
namespace {
   class X{/*...*/};
   void f();
   int i;
}
//file2.c
   class X{/*...*/};
   void f();
   int i;
//可以，但是自找麻烦

//关键字static也具有"内部连接"
```


9.2.1 头文件

- 为达到在不同的编译单位中声明的一致性而使用的不完美但是比较简单的方法，使用头文件
- #include机制是一种正文操作的概念，用于将源程序片断收集到一起
- #include "to_be_included"将用文件to_be_included的内容取代这行，该文件内容应该为源代码

头文件示例

```
#include <iostream>
// 来自标准库的包含目录
#include “myheader.h”
// 来自当前目录

#include < iosteam >
// <>或者””中的空格也是
// 有意义的
```

习惯上
头文件采用后缀.h，而包含函数或者数据定义的文件用.c后缀，
其他约定，比
如.C、.cxx、.cpp和.cc也常常可以看到

```
// 头文件被包含，每次都
// 需要重新编译，不过头
// 文件中只有声明，没有
// 需要被编译器深入分析
// 的代码，此外，很多
// C++实现都提供了头文
// 件的预编译形式
```


头文件中可以包括

命名名字空间	namespace N{/*...*/}
类型定义	struct Point {int x,y;};
模板声明	template <class T> class Z;
模板定义	template <class T> class V{..}
函数声明	extern int strlen(const char*);
数据声明	extern int a;
常量定义	const int a = 10;
名字声明	class Matrix;
包含指令	#include <algorithm>
条件编译	#ifdef _cplusplus
注释	/* check for end of file */
宏定义	#define VERSION 12

头文件中绝对不可以包括

常规的函数定义	char get(char* p){return *p++};
数据定义	int a;
聚集量(aggregate)定义	short tbl[] = {1, 2, 3};
无名名字空间	namespace {/*...*/}
导出的(exported)模板定义	export template <class T> f(T t){/* ... */ }

9.2.2 标准库中的头文件

- 标准库的功能是通过一组标准头文件给出的
- 标准库中的头文件不需要后缀，但并不意味这些头文件采用了某种特殊的存储方式
- 使用标准库中的头文件需要用#include <...>，而不是#include “...”
- 相对于每一个C标准库文件<X.h>，存在着一个与之对应的标准C++头文件<cX>，例如“#include <stdio.h>”和“#include <cstdio>”

标准库中的头文件示例

```
#ifdef __cplusplus
// 只为了C++编译器
namespace std {
    extern "C" {
    // 见9.2.4
#endif

    /* ..... */
    int printf(const char* ...);

#ifdef __cplusplus
    }
}
using namespace std;
#endif
```

左边代码为一个典型的<stdio.h>看起来的样子

实际声明都是共享的，但连接和名字空间问题需要另行处理，以便C和C++能共享这个头文件

9.2.3 单一定义规则

- 在一个程序里，任一个类、枚举和模板等都必须只定义唯一的一次，这种规则需要以一种复杂和精细的方式来描述，被称为"单一定义规则"(One-Definition Rule, ODR)
- 也就是说，一个类、模板或者内联函数的两个定义能够被接受为同一个唯一定义的实例，当且仅当
  - 它们出现在不同的编译单位里
  - 它们按一个个单词对应相同
  - 这些单词的意义在两个编译单位中也完全一样

单一定义规则(ODR)示例

```
//file1.c
    struct S{int a; char b;};
    void f(S*);

//file2.c
    struct S{int a; char b;};
    void f(S* p){/*...*/}

// ok
// S在两个源文件中被声明了两次
// 缺点是：
// 维护file2.c的人们不见得知道S
// 在不同文件中被声明了两次
// 或许认为可以自由去修改它
```

```
//s.h
    struct S{int a; char b;};
    void f(S*);

//file1.c
    #include "s.h"
    // 使用 f()

//file2.c
    #include "s.h"
    void f(S* p) {/*...*/}

// 基于ODR的设计
```


单一定义规则的错误示例

```
//file1.c
    struct S1{ int a; char b; };
    struct S1{ int a; char b; };
// Error 重复定义
//file1.c
    struct S2{int a; char b;};
//file2.c
    struct S2{int a; char bb;};
// Error 成员不同
//file1.c
    typedef int X;
    struct S3{X a;char b;};
//file2.c
    typedef char X;
    struct S3{X a;char b;};
//Error X的类型被改变了
```

```
//s.h
    struct S{Point a; char b;};
//file1.c
    #define Point int
    #include "s.h"
    //...
//file2.c
    class Point{/*...*/};
    #include "s.h"
    //...
// 如上写法是可以的
// 局部typedef和宏都可能改变通过
// #include包含进来的声明的意义
// 防范此类问题的方法是尽可能将头
// 文件做得自给自足
// 比如，在s.h中定义Point
```

9.2.4 与非C++代码的连接

- 一个C++程序可能由多个部分组成，这些部分不见得都使用C++语言编写
- 不同语言写出的程序片断、或者是同一种语言，但是由不同的编译器编译的片断之间的协作比较困难，为了能让C++使用，可以在一个extern声明中给出有关的连接约定
- 这种声明能够提醒编译器应该如何去连接这个片断(比如说：寄存器使用方式、参数入栈的顺序、整数等内部类型的布局)

与非C++代码的连接示例(1)

```
extern “C” char* strcpy(char*, const char*);
// 与 extern char* strcpy(char*, const char*);
// 的区别仅在于连接约定不同

extern “C”指令描述的只是一种连接约定，并不影响函数的语义，特别的，
声明为extern “C”的函数仍要遵守C++的类型检查和参数转换规则，而不是
C的较弱的规则

extern “C” int f();
int g() {  return f(1); }  // Error f()函数不应该有参数

extern “C” {
   char* strcpy(char*, const char*);
   int strcmp(const char*, const char*);
   int strlen(const char*);
} // 这种结构经常被称为连接块
```


与非C++代码的连接示例(2)

```
extern “C” {
   #include <string.h>
}

// 用extern “C”将整个C头文件包裹起来
// 以便整个文件能适合C++使用

#ifdef __cplusplus
extern “C” {
#endif
   char* strcpy(char*, const char*);
   int strcmp(const char*, const char*);
#ifdef __cpluscplus
}
#endif //__cplusplus是一个预定义的宏名
```


9.2.5 连接与指向函数的指针

- 当一个程序中混合使用C和C++代码片断时，有时会希望把在一个语言里定义的函数指针传递到另一个语言中定义的函数里
- 在这种情况下，注意不同语言之间的连接约定是否一致

连接与指向函数的指针示例

```
typedef int(*FT)(const void*, const void*);  // FT: C++连接
extern "C" {
    typedef int (*CFT)(const void*, const void*); // CFT: C连接
    void qsort(void* p, size_t n, size_t sz, CFT cmp); // cmp: C连接
}
void isort(void* p, size_t n, size_t sz, FT cmp);  // cmp: C++连接
void xsort(void* p, size_t n, size_t sz, CFT cmp); // cmp: C连接
int compare(const void*, const void*);  // compare: C++连接
extern "C" int ccmp(const void*, const void*); // ccmp: C连接
void f(char* v, int sz)
{
    qsort(v, sz, 1, &compare);    // Error
    qsort(v, sz, 1, &ccmp);       // ok
    isort(v, sz, 1, &compare);    // ok
    isort(v, sz, 1, &ccmp); }     // Error
```


9.3 使用头文件

- 单一头文件方式
- 对将一个程序划分成几个文件的最简单解决方案就是将所有定义放入合适数目的.c文件里，在一个头文件里声明这些.c文件之间通信所需要的类型，并让它们中的每个都#include该头文件
- 对于小的程序，这种风格的物理划分就很合适，但是如果程序大一些，这种方式就无法工作了，对公共头文件的修改就将导致对整个程序的重新编译，此外，几个程序员去修改一个头文件也极容易导致错误

多个头文件方式

- 让每一个逻辑模块有自己的头文件，其中定义它所提供的功能
- 这样，每个.c文件用一个对应的.h文件描述它所提供的东西(界面)，其中：为用户提供的界面放在它的.h文件中，为实现部分所用的界面放在以_impl.h(这个后缀可以随便起名)为后缀的文件里，而模块中的函数、变量等的定义则放在.c文件里

多个头文件方式

- 多个头文件的组织方式能够适应于比我们玩具式的分析器大几个数量级的模块，以及比我们的计算器大几个数量级的程序，基本原因就是它提供了我们所关心的更好的局部性
- 实际应用中，具体选择哪一种头文件的组织方式要根据实际情况来决定，过多过少都不好

9.3.3 包含保护符

- 为了避免某个头文件被多次包含，可以使用包含保护符的方式解决此类问题
- 为了避免潜在的宏的名称的冲突，包含保护符通常取比较长的非常难看的名字

```
//error.h
#ifndef CALC_ERROR_H
#define CALC_ERROR_H

namespace  Error{ /* ... */ }

#endif //CALC_ERROR_H
```


9.4 程序

- 一个程序就是由连接器组合到一起的一组分别编译单位
- 在这一集合中所使用的每个函数、对象、类型等都必须有唯一的定义
- 程序中必须包含恰好一个名字为main的函数，作为程序执行的入口，程序通过此函数的返回而结束，main函数返回的int数值被传递给调用main的系统，作为程序的结果

9.4.1 非局部变量的初始化

- 原则上说，在所有函数之外定义的变量(即那些全局的、名字空间的和类的static变量)应该在main()的调用之前完成初始化，在一个编译单位内的这种非局部变量将按照它们的定义顺序进行初始化，若某个变量没有显式的初始式，它将被初始化为有关类型的默认值

```
double x = 2;
double y; // 初始化为0
double sqx = sqrt(x+y); // sqx = sqrt(2)
```


非局部变量的初始化

- 不同编译单位里的全局变量，其初始化的顺序没有任何保证，此外，也没有办法捕捉到由全局变量的初始化式抛出的异常，一般来说，尽量减少全局变量的使用

```
int& use_count()
{
    static int uc = 0;
    return uc;
}

void f(){
    cout << ++use_count();
}
```

//对use_count()的调用在行为上就像是一个全局变量，除了它是在第一次被使用时才被初始化之外

9.4.1.1 程序终止

- 程序可能以多种方式终止
  - 通过从main()返回
  - 通过调用exit()，函数类型void exit(int)
  - 通过调用abort()
  - 通过抛出一个未被捕捉的异常
- 若程序调用exit()终止，所有已经构造起来的静态对象的析构函数都将被调用(调用它的函数及其调用者里面的局部变量的析构函数都不会被执行)，然而，如果程序使用abort()终止，那么析构函数就不会被调用
- 注意：析构函数里面调用exit()有可能导致无穷递归

程序终止

- 调用exit(int)时，其参数将被作为程序的值返回给系统(一般用0表示程序成功结束)
- 一般情况下，最好通过抛出一个异常的方式脱离一个环境，让异常处理器来决定应该做什么
- 可以通过使用atexit函数来让程序在终止之前执行一些代码
- exit(),abort(),atexit()在<cstdlib>里

9.5 忠告

- [1] 利用头文件去表示界面和强调逻辑结构
- [2] 用#include将头文件包含到实现有关功能的源文件里
- [3] 不要在不同编译单位里定义具有同样名字、意义类似但又不同的全局实体
- [4] 避免在头文件里定义非inline函数
- [5] 只在全局作用域和名字空间里使用#include
- [6] 只用#include包含完整的定义
- [7] 使用包含保护符

忠告

- [8] 用#include将C头文件包含到名字空间里，以避免全局名字
- [9] 将头文件做成自给自足的
- [10] 区分用户界面和实现界面
- [11] 区分一般用户界面和专家用户界面
- [12] 在有意向用于非C++程序组成部分的代码中，应避免需要运行时初始化的非局部对象

C++编程(8)

唐晓晟
北京邮电大学电信工程学院

第十章 类

- 引言
- 类
- 高效的用户定义类型
- 对象
- 忠告

10.1 引言

- C++里类的概念的目标是为程序员提供一种建立新类型的工具，使这些新类型的使用能够象内部类型一样方便
- 类型是某个概念的一个体现，类其实就是一种用户定义类型
- 如果一个程序里提供的类型与应用中的概念有直接的对应，与不具有这种情况的程序相比，这个程序很可能就更容易理解，也更容易修改

引言

- 定义新类型的基本思想就是将实现中那些并非必然的细节(例如，用户存储该类型的对象所采用的数据的布局)，与那些对这个类型的正确使用至关重要的性质(例如，能够访问其中数据的完整的函数列表)区分开来
- 表达这种划分的最好方式就是提供一个特定的界面，另对于数据结构以及内部维护例程的所有使用都通过这个界面进行

10.2 类

- 成员函数
- 访问控制
- 构造函数
- 静态成员
- 类对象的复制
- 常量成员函数
- 自引用
- 结构和类
- 在类内部的函数定义

10.2.1 成员函数

- 考虑用struct实现日期
- 改进版本，将函数加入到struct中
- 在一个类中声明的函数叫做成员函数(struct也是一种类)

```
struct Date {
    int d,m,y;
}
void init_date(Date& d, int, int, int);
void add_year(Date& d,int n);
void add_month(Date& d, int n);
void add_day(Date& d, int n);
// 函数和日期类型之间没有任何显示的联系
```

```
struct Date {
    int d,m,y;
    void init_date(Date& d, int, int, int);
    void add_year(Date& d,int n);
    void add_month(Date& d, int n);
    void add_day(Date& d, int n); }
// 改进版本
```


成员函数

- 成员函数只能通过适当类型的特定变量，采用标准的结构成员访问语法形式调用

```
Date my_birthday;
void f() {
    Date today;
    today.init(16,10,1996);
    my_birthday.init(30,12,1950);
    Date tomorrow = today;
    tomorrow.add_day();
    //...
};
```

```
//定义成员函数时使用的方式
void Date::init(int dd,
        int mm, int yy)
{
    d = dd; // d是结构内部的成员
    m = mm;
    y = yy;
}
// d m y 是函数被调用时所针对的
那个对象的成员
```


类定义

- class X { … };
  被称为一个类定义，因为它定义了一个新类型
- 由于历史的原因，一个类定义也常常被说成是一个类声明
- 就像其他不是定义的声明一样，类定义可以由于#include的使用而在不同的源文件里重复出现，这样并不违反唯一定义规则

10.2.2 访问控制

- 通过struct Date声明，提供了一组操作Date的函数，然而，它却没有清楚地说明这些函数就是直接依赖于Date表示的全部函数，而且也是仅有的能够直接访问Date类的对象的函数
- 这些限制可以通过用class替代struct而描述清楚

访问控制

标号public将这个类内部分成两个部分，其中第一部分(私用部分)只能由成员函数使用；而第二部分(共用部分)则构成了该类的对象的公用界面，struct也是class，不过其成员的默认方式是共用的

```
class Date {
   int d, m, y;
public:
   void init (Date& d, int, int, int);

   void add_year(Date& d,int n);
   void add_month(Date& d, int n);
   void add_day(Date& d, int n);
};
```

```
// 函数定义同前
void Date::add_year(int n)
{
   y += n;
}
```


访问控制

```
//非成员函数将禁止访问私有成员
void timewrap(Date& d)
{
   d.y -= 200; // Error
}
```

这种限定的好处是：
1 导致日期异常的错误必然是由某个成员函数造成的
2 一个潜在的用户要学习使用一个类也只需要考察其成员函数的定义

该类中init函数的原因
1 有一个函数来设置对象的值总是有用的
2 数据的私用性也要求我们这么做

10.2.3 构造函数

- 采用init一类的函数提供对类对象的初始化，这么做既不优美又容易出错(比如说，忘记调用)
- 更好的途径是让程序员有能力去声明一个函数，其明确目的就是去完成对象的初始化，此类函数称为构造函数
- 构造函数具有与类相同的名字

构造函数

```
class Date{
   //…
   Date(int, int, int);
   //构造函数 Constructor
   // ctor
};
```

若一个类有构造函数，这个类的所有对象的初始化都将通过对某个构造函数的调用完成初始化，注意参数匹配问题，可以创建多个构造函数来以多种方式完成对象的初始化工作

```
Date today = Date(23,6,1983);
Date xmas(25,12,1990);
// 简写形式
Date my_birthday;
// Error 缺少初始式
Date release1_0(10,12);
// Error 三个参数
```

```
class Date {
   int d, m, y;
public:
   Date(int, int, int);
   Date(int, int); // 构造函数的重载
   Date(int);
   Date();
}
```


构造函数

```
class Date {
   int d, m, y;
public:
   Date(int dd=0,
        int mm=0, int yy=0);
   //…
};
Date::Date(int dd, int mm, int yy)
{
   d = dd? dd : today.d;
   m = mm? mm : today.m;
   y = yy? yy : today.y;
}
```

也可以用带默认参数的函数来取代函数重载

10.2.4 静态成员

- 为Date提供默认值的方式隐藏着一个重要问题：Date类现在依靠一个全局变量today
- 如果设定一个这样的变量，而只在Date类中使用，用户会遇到很多不愉快的事情，而且维护工作也变得麻烦
- 对此的解决方法是：创建一个static静态成员，可以获得全局变量的方便，而又无须受到访问全局变量之累
- 对于所有对象，静态成员只有唯一的一个副本
- 使用一个静态成员，不必提及任何对象

静态成员

```
class Date{
   int d, m, y;
   static Date default_date;
public:
   Date(int dd=0,
        int mm=0, int yy=0);
   //…
   static void set_default(int, int, int);
};

Date::Date(int dd, int mm, int yy)
{
   d = dd? dd : default_date.d;
   m = mm? mm : default_date.m;
   y = yy? yy : default_date.y;
}
```

```
void f()
{
   Date::set_default(4,5,1945);
}

// 静态成员和函数需要在某个地方另行定义
Date
Date::default_date(16,12,1770);
void Date::set_default(int d,
        int m, int y)
{
   Date::default_date =
                Date(d,m,y);
}
//Date()相当于Date::default_date
```

10.2.5 类对象的复制

- 按照默认约定，类对象可以复制，例如：Date d = today; // 复制初始化
- 按照默认方式，类对象的复制就是其中各个成员的复制，如果某个类所需要的不是这种默认方式，那么就可以定义一个复制构造函数X::X(const X&)，由它提供所需要的行为
- 类似的，类成员也可以通过赋值进行按照默认方式的复制，例如：Date d; d = today;
- 对于赋值复制，默认语义是按照成员复制，用户可以定义合适的赋值运算符来完成自己需要的复制

10.2.6 常量成员函数

- 可以为Date类提供一些函数用来检查一个Date的值
- 注意，函数参数表后面的const，表示该函数不会修改Date的状态

```
class Date{
   int d, m, y;
public:
   int day() const {return d;}
   int month() const;
   int year() const;
}
inline int Date::year() const
{
   return y;
   // return y++; Error
}
```

```
void f(Date& d, const Date& cd)
{
   int i = d.year(); //ok
   d.add_year(1); // ok
   int j = cd.year(); // ok
   cd.add_year(1); //Error
}
// const或者非const对象都可以调用
const成员函数，而非const成员函数
则只能由非const对象调用
```


10.2.7 自引用

- 状态更新函数：add_year(), add_month(), add_day()被定义为不返回值的函数对于这样一组相关的更新函数，可以让他们返回一个到被更新的引用，以使对于对象的操作可以串接起来

```
class Date{
   // ...
   Date& add_year(int n);
   Date& add_month(int n);
   Date& add_day(int n);
};
//d.add_year(1).add_month(1).
add_day(1)
// 可以如上述方式使用
```

```
Date& Date::add_year(int n)
{
   if(d==29 && m==2 &&
            !leapyear(y+n)){
      d = 1;
      m = 3;
   }
   y+=n;
   return *this;
}
```


关于this

- 在一个非静态的成员函数里，关键字this是一个指针，指向该函数的当时这次调用所针对的那个对象
- 在类X的非const成员函数里，this的类型就是X*，但是，不能取得this的地址或者给它赋值
- 在类X的const成员函数里，this的类型是const X*，以防止对这个对象本身的修改
- 大部分对于this的应用都是隐含的，特别的，对于一个类中的非静态成员的引用都要依靠隐式地使用this，以获取相应对象的成员

this示例

```
Date& Date::add_year(int n)
{
   if(this->d==29 && this->m==2
       && !leapyear(this->y+n)){
      this->d = 1;
      this->m = 3;
   }
   this->y+=this->n;
   return *this;
}
// 使用this的例子
```

```
Date::Date(int y, int m, int d)
{
   this->y = y;
   this->m = m;
   this->d = d;
}
// 另外一种不得不使用this的例子
```


10.2.7.1 物理的和逻辑的常量性

- 偶尔有这种情形，一个成员函数在逻辑上是const，但它却仍然需要改变某个成员的值，对于用户而言，这个函数看似没有改变其对象的状态，然而，它却可能更新了某些用户不能直接访问的细节，这通常被称为逻辑的常量性
- 例如：Date类可能有一个函数，返回日期的字符串表示，构造字符串是相对费时的，实现中可以为其保留一个副本，重复需要时返回该副本即可

逻辑的常量性

```
class Date{
   bool cache_valid;
   string cache;
   void compute_cache_value();
   // …
public:
   string string_rep() const;
   //但是第一次调用string_rep时
   //需要构造cache字符串
}
```

```
string Date::string_rep() const
{
   if(cache_valid == false){
      Date* th =
         const_cast<Date*>(this);
      th->compute_cache_value();
      th->cache_valid = true;
   }
   return cache;
}
```

以上解决方式一点不优美，也无法保证总能工作，例如

```
Date d1;
const Date d2;
string s1 = d1.string_rep(); // ok
string s2 = d2.string_rep(); //无定义行为
```


10.2.7.2 可变的－mutable

- 为了解决上述显式“强制去掉const”的问题，可以将所涉及的数据声明为mutable

```
class Date{
   mutable bool cache_valid;
   mutable string cache;
   void compute_cache_value();
   // …
public:
   string string_rep() const;
}
```

```
string Date::string_rep() const
{
   if(cache_valid == false){
      compute_cache_value();
      cache_valid = true;
   }
   return cache;
}
```

```
Date d1;
const Date d2;
string s1 = d1.string_rep(); // ok
string s2 = d2.string_rep(); // ok
```

以上技术使得前面string_rep()的所有使用都合理化了

延迟求值(lazy evaluation)

- 若在某个表示中只有一部分允许改变，将这些成员声明为mutable式最合适的，但是如果有大批量都需要修改，最好将这些数据放入另外一个独立的对象里，并间接的访问它

```
struct cache{
    bool valid;
    string rep;
};
class Date{
    cache* c;
    void cache_value() const;
public:
    string string_rep() const;
};
```

```
string Date::string_rep() const
{
    if(!c->valid) {
        cache_value();
        c->valid = true;
    }
    return c->rep;
}
```

这种技术可以推广到各种方式的延迟求值

10.2.8 结构和类

- 按照定义，一个struct就是一个类，但其成员默认为公用的，也即：struct s{ ... 是class s { public: ... 的简写形式
- 一般情况下，struct被用于所有成员都是公用的那些类
- 这样的类并不是完整的类型，不过是个数据结构
- 结构中也可以有构造函数

```
class Date1 {
    int d, m, y;
public:
    Date1(int dd, int mm, int yy);
    void add_year(int n);
};
```

```
struct Date2{
private:
    int d, m, y;
public:
    Date2(int dd, int mm, int yy);
    void add_year(int n);
};
```

上述两个声明除了名字不同，完全等价

10.2.9 类内部定义的函数定义

- 如果一个函数是在类定义内部定义的(不是简单的声明)，那么这个函数就被做为内联函数处理
- 也就是说，在类内部定义的函数应该是小的，频繁使用的函数

```
class Date{
    int d, m, y;
public:
    int day() const
    {
        return d;
    }
}; // 函数和数据谁在前无所谓
```

<===>

```
class Date{
public:
    int day() const;
private:
    int d, m, y;
};

inline int Date::day() const
{  return d;  }
```


10.3 高效的用户定义类型

- 前面所提及的是设计一个Date类的各种细节，本部分主要内容讨论设计一个简单而高效的Date类，并阐明语言的各种特性来如何支持这种设计
- 应用系统中会频繁的使用很多种抽象，例如：字母、整数、复数、向量、数组等，C++直接支持其中一些最常见的抽象，但是，C++提供了使用户定义这些类型的机制

一个更好的Date类

```
class Date{
public:
    enum Month{ jan=1, feb, mar, apr,
may, jun, jul, aug, sep, oct, nov, dec};
// Month主要对付记忆的问题
// 欧美表示顺序不同，顺序反时能被检测出来
    class Bad_date{}; // 异常时抛出
    Date(int dd=0, Month mm=Month(0),
int yy=0); // 构造函数，注意日、月、年顺序
// 获取Date中数值的函数
    int day() const;
    Month month() const;
    int year() const;
    string string_rep() const;
    void char_rep(char s[]) const; // C风格
    static void set_default(int, Month, int);
// 修改Date的函数
    Date& add_year(int n);
    Date& add_month(int n);
    Date& add_day(int n);

private:
    int d, m, y;
    static Date default_date;
};

Date Date::default_date(22,
jan, 1901);
```


使用Date类示例

```
void f(Date& d)
{
    Date lvb_day = Date(16, Date::dec, d.year());
    if(d.day() == 29 && d.month() == Date::feb)
    {
        // ...
    }
    if(midnight()) d.add_day(1);
    cout << "day after:" <<d+1 << '\n';
}
```

实现Date类而不是简单的使用Date结构体可以使得Date的概念局部化，否则每个用户不得不直接操作Date的成分，会使得Date的概念散布到整个系统中，最终使得程序难于理解

10.3.1 成员函数

```
Date::Date(int dd, Month mm, int yy)
{
    if(yy==0) yy=default_date.year();
    if(mm==0)
        mm=default_date.month();
    if(dd==0) dd=default_date.day();

    int max;
    switch(mm) {
    case feb:
        max = 28 + leapyear(yy);
        break;
    case apr: case jun: case sep: case
nov:
        max = 30;
        break;
    case jan: case mar: case may:
case jul: case aug: case oct:
case dec:
        max = 31;
        break;
    default:
        throw Bad_date();
    }
    if(dd<1 || max<dd)
        throw Bad_date();
    y = yy; m = mm; d = dd;
}
// 构造函数完成构造一个合法的日
期，一旦完成，其他函数就只需要用
来对其赋值和更改，而不需要合法性
检查了，这样能极大地简化代码
```


10.3.2 协助函数

- 一个类可能有一批与它相关的函数，但是又没有必要定义在类里，传统方式下，它们的声明将直接与类Date的声明放在同一个文件里(一般为.h文件)，此外，我们也可以将这个类和它的协助函数包在同一个名字空间里
- 一般情况下，一个名字空间里的内容比下述Chrono要多的多

```
namespace Chrono{
    class Date { /*...*/ };
    int diff(Date a, Date b);
    bool leapyear(int y);
    Date next_weekday(Date d);
    Date next_saturday(Date d);
}
```

10.3.3 重载的运算符

- 增加一些具有习惯形式的函数很有用，比如，使用==来判断两个Date对象是否相等

```
bool operator==(Date a, Date b)
{
   return a.day() == b.day() && a.month() == b.month()
          && a.year() = b.year();
}
```

类似的：

```
bool operator!=(Date, Date);
bool operator<(Date, Date);
bool operator>(Date, Date);
Date& operator++(Date& d);
Date operator+(Date d, int n);
```


10.3.4 具体类型的意义

- 像Date这样的简单用户定义类型称作具体类型，是为了让它们与抽象类型与类层次结构区别，同时也是为了强调这种类型与内部int或者char等内部类型的相似性
- 这种类型有时也被称为值类型
- 定义良好的一组这种类型(小而有效)能够为应用提供一个基础，提高程序员开发效率

10.4 对象

- 对象的建立有多种可能，有的对象是局部变量，有的是全局变量，有的是类的成员等，本部分讨论这方面的不同情况、统辖着它们的规则、初始化对象所用的构造函数，以及在它们变得不可用之前清理他们的构造函数

10.4.1 析构函数

- 构造函数完成对象的初始化，因此，在类中需要一个函数完成对象的清理工作，并且该函数能够象构造函数一样被自动调用，这个函数称为析构函数(deconstructor)
- 析构函数通常完成一些清理和释放资源的工作，一般情况下，用户不需要显式地调用它
- 对于没有析构函数的类型，可以简单认为这种类型有一个什么都不做的析构函数

析构函数示例

```
class Name{ const char* s; };
class Table{
    Name* p;
    size_t sz;
public:
    Table(size_t s = 15) { p = new Name[sz=s]; }
    ~Table() { delete []p; }
    Name* lookup(const char*);
    bool insert(Name*);
};
```


10.4.2 默认构造函数

- 可以认为大部分类型都有一个默认构造函数
- 默认构造函数就是调用时不必提供参数的构造函数，上例中：Table::Table(size_t)就是默认构造函数
- 若用户没有声明构造函数，并且有必要，编译器会设法去生成一个
- 编译器生成的默认构造函数将隐式地为类类型(class type)的成员和基类调用有关的默认构造函数

默认构造函数

```
struct Tables{
    int i;
    int vi[10];
    Table t1;
    Table vt[10];
};
Tables tt;
```

tt将会被用一个生成出来的默认构造函数初始化，该构造函数为tt.t1以及tt.vt的每个成员调用Tables(15)，但是它不会去初始化tt.i和tt.vi，因为它们不是class type

```
struct X{
    const int a;
    const int& r;
};
X x; // Error
```

由于const和引用必须进行初始化，包含const或者引用成员的类就不能进行默认构造
此外，默认构造函数也可以被显示调用(10.4.10)，内部类型也有默认构造函数

10.4.3 构造和析构

- 对象可以有各种不同方式
- 10.4.4 一个命名的自动对象
- 10.4.5 一个自由存储对象
- 10.4.6 一个非静态成员对象
- 10.4.7 一个数组元素
- 10.4.8 一个局部静态对象
- 10.4.9 一个全局对象
- 10.4.10 一个临时对象
- 10.4.11 一个在用户函数获得的存储里放置的对象
- 10.4.12 一个union成员

10.4.4 局部变量

- 对一个局部变量的构造函数将在控制线程每次通过该变量的声明时执行
- 每次当控制离开该局部变量所在的块时，就会去执行它的析构函数
- 局部变量的析构函数将按照构造它们的相反顺序执行

```
void f(int i)
{
    Table aa;
    Table bb;
    if(i>0) { Table cc; }
    Table dd;
}
```


10.4.4.1 对象的复制

- 对象复制的默认含义就是对象按照成员逐个复制到另外一个中去
- 右边例子中，构造函数被调用了2次，而析构函数被调用了3次
- 为避免这类情形，可以将Table复制的意义定义清楚(复制构造函数，复制赋值)

```
void h()
{
    Table t1;
    Table t2 = t1;
    // 复制初始化
    Table t3;

    t3 = t2;
    // 复制赋值
}
```


对象的复制

```
class Table
{
    // ...
    Table(const Table&);
    Table& operator= (const
        Table&);
};

Table::Table(const Table& t)
{
    p = new Name[sz=t.sz];
    for(int i=0;i<sz;i++)
        p[i]=t.p[i];
}
```

```
Table& Table::operator=(const
Table& t)
{
    if(this != &t) { // 防止自赋值
        delete []p;
        p = new Name[sz=t.sz];
        for(int i=0;i<sz;i++)
            p[i]=t.p[i];
    }
    return *this;
}

// Table t1;
// Table t2 = t1;    // 复制初始化
// Table t3;
// t3 = t2;  // 复制赋值
```


10.4.5 自由存储

- 使用new的方式建立对象时，这些对象在自由存储里面，该对象将一直存在直到将delete函数用于指向它的指针
- 有些编程语言可以自动进行内存管理

```
int main() {
    Table* p = new Table;
    Table* q = new Table;
    delete p;
    delete p; // 行为没有定义
}
```

10.4.6 类对象做为成员

```
class Club{
    string name;
    Table members;
    Table officers;
    Date founded;
    Club(const string &,
        Date fd);
};
Club::Club(const string &,
    Date fd): name(n),
    member(),officers(),
    founded(fd)
{    // 上面的member()
}    // 和officers()也可无
```

- 包含类成员的类的构造函数
- 成员初始式列表由一个冒号开头，用逗号分割
- 成员的构造函数将在容器类的构造函数前被执行(严格按照在类中声明的顺序执行)
- 成员类的析构函数后于容器类的析构函数而执行，并按照与构造时相反的顺序逐个被调用

10.4.7 数组

- 如果在构造某个类的对象时可以不必提供显式的初始式，那么就可以定义这个类的数组
- 例如：Table tbl[10];
- 将建立一个10个Table的数组，并用默认参数15调用Table::Table()，对每个Table进行初始化
- 也可以使用指针，例如：Table* p1 = new Table[10];
  注意此时需要用delete[] p1来回收空间
- 而Table *p1=new Table;只需要delete p1;

10.4.8 局部静态存储

- 对于局部静态对象，构造函数是在控制线程第一次通过该对象的定义时调用
- 局部静态对象的析构函数将按照它们被构造的相反顺序逐一调用，没有规定确切的时间

```
void f(int i)
{
    static Table tbl;
    if(i) {
        static Table tbl2;
    }
}
```

```
int main()
{
    f(0);
    f(1);
    f(2);
}
```


10.4.9 非局部存储

- 在所有函数之外定义的变量(即全局变量、名字空间的变量，以及各个类的static变量)，在main()被激活之前完成初始化(构造)
- 对于已经构造起的所有这些变量，其析构函数将在退出main()之后调用
- 在一个编译单位里，对非局部变量的构造将按照它们的定义顺序进行，析构函数则按相反顺序逐个被调用
- 不同编译单位的非局部变量，构造和析构的顺序没有保证

非局部存储示例

```
class X{
    // ...
    static Table memtbl; // 注意这是个声明
};
Table tbl; // 第一个被构造
Table X::memtbl; // 第二个被构造
namespace Z{
    Table tbl2; // 第三个被构造
}
```


10.4.10 临时对象

- 临时对象最经常是作为算术表达式的结果出现的，例如：x*y+z过程中的某一点
- 除非一个临时对象被约束到某个引用，或者被用于做命名对象的初始化，否则它将总在建立它的那个完整表示式结束时销毁
- 完整表达式就是那种不是其他表达式的子表达式的表达式

临时对象示例

```
void f(string& s1, string& s2, string&
s3)
{
    const char* cs = (s1+s2).c_str();
    // s1+s2被保存为临时对象
    // 表达式运算完毕后会被删除
    cout << cs;
    if(strlen(cs=(s2+s3).c_str()) < 8
        && cs[0] == 'a') {
        // 在这里使用 cs
    }
}
```

```
void f(Shape& s, int x, int y)
{
    s.move(Point(x,y));
    // 构造一个Point并传递
    // 给Shape::move()
    // ...
}
```


10.4.11 对象的放置

- 按照默认方式，运算符new将在自由存储区创建对象，但是，也可以在其他地方分配对象

```
class X{
public:
    X(int);
    void* operator new(size_t,void* p){ return p; }
};

void* buf = reinterpret_cast<void*>(0xf00f);
X* p2 = new (buf)X; // 放置语法
// 在buf构造X时调用：operator new(sizeof(X), buf)
// 自由式放置可查阅15.6节
```

10.4.12 联合

- 命名联合的定义方式同struct，其中的各个成员将具有同样的地址
- 联合可以有成员函数，但是不能有静态成员
- 一般来说，编译器无法知道被使用的是联合的哪个成员，因此，联合就不能包含带构造函数或者析构函数的成员，因为无法保护其中的对象以防止破坏，也不可能保证在联合离开作用域时能调用正确的析构函数

10.5 忠告

- [1]用类表示概念
- [2]只将public数据(struct)用在它实际上仅仅是数据，而且对于这些数据成员并不存在不变式的地方
- [3]一个具体类型属于最简单的类。如果适用的话，就应该尽可能使用具体类型，而不要采用更复杂的类，也不要用简单的数据结构
- [4]只将那些需要直接访问类的表示的函数作为成员函数
- [5]采用名字空间，使类与其协助函数之间的关系更明确
- [6]将那些不修改对象值的成员函数做成const成员函数
- [7]将那些需要访问类的表示，但无须针对特定对象调用的成员函数做成static成员函数

忠告

- [8]通过构造函数建立起类的不变式
- [9]如果构造函数申请某种资源，析构函数就应该释放这一资源
- [10]如果在一个类里有指针成员，它就需要有复制操作(包括复制构造函数和复制赋值)
- [11]如果在一个类里有引用成员，它就可能需要有复制操作(复制构造函数和复制赋值)
- [12]如果一个类需要复制操作或析构函数，它多半还需要有构造函数、析构函数、复制赋值和复制构造函数
- [13]在复制赋值里需要检查自我赋值

忠告

- [14]在写复制构造函数时，请小心地复制每个需要复制的元素(当心默认的初始式)
- [15]在向某个类中添加新成员时，一定要仔细检查，看是否存在需要更新的用户定义构造函数，以使它能够初始化新成员
- [16]在类声明中需要定义整型常量时，请使用枚举
- [17]在构造全局的和名字空间的对象时，应避免顺序依赖性
- [18]用第一次开关去缓和顺序依赖性问题
- [19]请记住，临时对象将在建立它们的那个完整表达式结束时销毁

C++编程(9)

唐晓晟
北京邮电大学电信工程学院

第11章 运算符重载

- 引言
- 运算符函数
- 一个复数类型
- 转换运算符
- 友元
- 大型对象
- 基本运算符
- 下标
- 函数调用
- 间接
- 增量和减量
- 一个字符串类
- 忠告

11.1 引言

- 每个技术领域都发展了一套习惯性的简化记述形式，以便使涉及到常用概念的说明和讨论更加方便，例如：x+y*z
- 对于C++，已经为内部类型提供了一组运算符
- C++的用户希望也能在自己创建的新的类型(类)上也能使用这类运算符
- C++中解决类似问题的技术叫做运算符重载

示例

```
class complex {
    double re, im;
public:
    complex(double r, double i) : re(r), im(i) {}
    complex operator+(complex);
    complex operator*(complex);
};
void f() {
    complex a = complex(1,3.1);
    complex b = complex(1.2,2);
    complex c = b;
    a = b+c;
    b = b+c*a; // 运算符优先级仍然有效
    c = a*complex(1,2);
}
```

11.2 运算符函数

以下的运算符可以被重载(注意，不允许定义新的运算符)						
+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	()	new	new[]	delete	delete[]
以下操作符不能被重载						
::	.	.*	这些操作符以名字(不是值)来做为第二个参数			
? :	sizeof	typeid				

11.2.1 二元和一元运算符

- 二元运算符可以定义为取一个参数的非静态成员函数，也可以定义为取两个参数的非成员函数
- 对于任何二元运算符@，aa@bb可以解释为aa.operator@(bb)，或者解释为operator@(aa,bb)，如果两者都有定义，那么按照重载解析来确定究竟应该用哪个定义

二元和一元运算符

- 对于一元运算符，无论它是前缀的还是后缀的，都可以定义为无参数的非静态成员函数，或者定义为取一个参数的非成员函数
- 对任何前缀一元运算符@，@aa可以解释为aa.operator@()或者operator@(aa)
- 对任何后缀一元运算符@，aa@可以解释为aa.operator@(int)或者operator@(aa,int)

二元和一元运算符示例

```
class X{
public:
   void operator+(int);
   X(int);
};

void operator+(X,X);
void operator+(X,double);

void f(X a)
{
   a+1;  // a.operator+(1)
   1+a;  // ::operator+(X(1),a)
   a+1.0;// ::operator+(a,1.0)
}
```

```
class X{ // 隐含有this指针
   X* operator&(); // 前缀一元&取地址
   X operator&(X); // 二元& 与
   X operator++(int);// 后缀增量
   X operator&(X,X);// 错误：三元
   X operator/();  // 错误：一元
};

X operator-(X); // 前缀一元减
X operator-(X,X); // 二元减
X operator--(X&,int); // 后缀减量
X operator-(); // 错误：无操作数
X operator-(X,X,X); // 错误：三元
X operator%(X); // 错误：一元%
```

11.2.2 运算符的预定义意义

- 对于用户定义运算符只做了不多的几个假定，特别是operator=、operator[]和operator->只能做为非静态的成员函数，这就保证了它们的第一个运算对象一定是一个左值
- 有些内部运算符在意义上等价于针对同样参数的另外一些运算符的组合，但是对于用户定义运算符而言就没有这类关系，除非用户正好将它们定义成这样，例如：+=

11.2.3 运算符和用户定义类型

- 一个运算符函数必须或者是一个成员函数，或者至少有一个用户定义类型的参数(重新定义运算符new和delete的函数则没有此要求)，这一规则保证了用户不能改变原有表达式的意义，除非表达式中包含有用户定义类型的对象
- 如果某个运算符函数想接受某个内部类型做为第一参数，那么它自然就不可能是成员函数了，例如：2+aa，可以用非成员函数方便地处理这类问题
- 为枚举类型也可以定义运算符

运算符和用户定义类型示例

```
enum Day { sun, mon, tue, wed, thu, fri, sat };
Day operator++(Day& d)
{
   return d = (sat==d) ? sun: Day(d+1);
}
```


11.2.4 名字空间里的运算符

- 一个运算符或者是一个成员函数，或者是定义在某个名字空间里(也可以是全局名字空间)
- 注意：运算符查寻机制并不认为成员函数比非成员函数更应优先选取
- 此外，也不存在运算符屏蔽的问题，这保证了内部运算符绝对不会变得无法使用，也保证了用户可以为运算符提供新的意义，而不必取修改现存的类声明

名字空间里的运算符示例

```
namespace std{
   class ostream
   {
      ostream& operator<<(const
char*);
   };
   extern ostream cout;
   class string
   {
      // ...
   };
   ostream operator<<(ostream&,
const string&);
}
```

```
int main()
{
   char* p = "Hello";
   std::string s = "world";
   std::cout << p << ", "
<< s << "!\n";
}
// 本例没有使用
// using namespace std;
// 而是使用了std::string和
// std::cout
// 按照最好的方式行事，没
// 有污染全局名字空间，也
// 没有引进不必要的依赖性
```


二元运算符的解析方式

- 考虑二元运算符@，如果x的类型为X而y的类型为Y，x@y将按照如下方式解析
  - 若X是类，查询作为X的成员函数或者X的某个基类的成员函数的operator@
  - 在围绕x@y的环境中查询operator@的声明
  - 若X在名字空间N里定义，在N里查询operator@的声明
  - 若Y在名字空间M里定义，在M里查询operator@的声明
- 当查询到operator@重载时，按照重载规则处理
- 一元运算符的解析方式与之类似

11.3 一个复数类型

- 我们希望实现一个复数类，可以进行如下操作
- 此外，还希望提供一些运算符，例如以==做比较，<<做输出等

```
void f()
{
   complex a=complex(1,2);
   complex b=3;
   complex c=a+2.3;
   complex d=2+b;
   complex e=-b-c;
   b=c*2*c;
}
```


11.3.1 成员运算符和非成员符

- 作者喜欢让尽量少的函数能直接去操作对象的表示，因此，作者只在类本身中定义那些本质上需要修改第一个参数值的运算符，如+=，而像+这样基于参数的值简单产生新值的运算符，则在类之外利用基本运算符来定义

成员运算符和非成员符示例

```
class complex
{
   double re, im;
public:
   complex&
   operator+=(complex a);
};

complex operator+
   (complex a, complex b)
{
   complex r=a;
   return r+=b;
}
```

```
complex& complex::operator+=
(complex a) {
   re += a.re; im += a.im;
   return *this;
}
void f(complex x, complex y,
      complex z) {
   complex r1=x+y+z;
   // r1 = operator+
   //       (operator+(x,y), z)
   complex r2=x;
   r2+=y;
   // r2.operator+=(y)
   r2+=z;}
```


11.3.2 混合模式算术

- 为了处理complex d=2+b; 我们需要能够接受不同类型的运算对象的+运算符，为此，可以简单地增加运算符的版本来解决这个问题

```
class complex
{
   double re, im;
public:
   complex& operator+=
      (complex a)
   {
      re += a.re; im += a.im;
      return *this;
   }
```

```
   complex& operator+=
      (double a);
   {
      re += a;
      return *this;
   }
};
```


混合模式算术示例

```
complex operator+(complex a, complex b)
{
   complex r=a;
   return r+=b; // 调用complex::operator+=(complex)
}
complex operator+(complex a, double b)
{
   complex r=a;
   return r+=b; // 调用complex::operator+=(double)
}
complex operator+(double a, complex b)
{
   complex r=b;
   return r+=a; // 调用complex::operator+=(double)
}
```


11.3.3 初始化

- 我们可以使用标量对complex变量进行初始化和赋值，为此，可以定义一个只有一个参数的构造函数来解决这个问题

```
class complex{
   double re, im;
public:
   complex(double r) : re(r), im(0) {}
   complex(): re(0), im(0){}
};

complex b = 3; // b.re = 3, b.im = 0 即complex(3)
complex c;
```

11.3.4 复制

- 除了显式的构造函数之外，complex还按照默认规定，得到了一个定义好的复制构造函数，默认的构造函数就是简单地复制成员，这对于comple类来说已经足够好

```
class complex{
    double re, im;
public:
    complex(const complex& c) : re(c.re), im(c.im) {}
};
```


11.3.5 构造函数和转换

- 前面的例子中，我们为每个标准的算术运算定义了三个版本
  complex operator+(complex, complex);
  complex operator+(complex, double);
  complex operator+(double, complex);
- 若函数中需要处理的每个参数有三种类型，那么一个参数的函数需要3个版本，两个参数的需要9个版本，三个参数就需要27个版本，而且这些不同版本的函数的代码实现会非常相似
- 依靠类型转换可以大大简化此类问题

构造函数和转换示例

- 例如：complex类提供了一个构造函数，能够将double转换到complex，因此，我们就只需要为comple的==运算符定义一个版本

```
bool operator==(complex,complex);
void f(complex x, complex y)
{
    x == y; // operator==(x,y)
    x == 3; // operator==(x, complex(3))
    3 == y; // operator==(complex(3), x)
}
```


11.3.6 文字量

- 不能为类类型定义文字量(例如：double类型的1.2或者12e3等)
- 但是，我们常常可以利用内部类型的文字量，只需要通过类的成员函数对其提供一种解释
- 当构造函数很简单并且被内联时，把这种以文字量为参数的构造函数就看成是文字量也是相当合理的
- 例如：可以把complex(3)看成是complex类型的文字量，虽然从技术上说它并不是

11.3.7 另一些成员函数

- 为了使得complex更好的工作，还可以定义其他函数，比如说：检查实部和虚部的值，然后利用这些函数进行各种有用的运算

```
class complex{
    double re, im;
public:
    double real() const { return re; }
    double imag() const { return im; }
};
inline bool operator==(complex a, complex b)
{
    return a.real()==b.real() && a.imag()==b.imag();
}
```


11.3.8 协助函数

- 将所有的东西集中到一起，complex就变成了

```
class complex
{
    double re, im;
public:
    complex(double r=0, double i=0) : re(r), im(i) {}
    double real() const { return re; }
    double imag() const { return im; }
    complex& operator+=(complex);
    complex& operator+=(double);
    // -=, *=, /= ...
};
```


协助函数

- 除此之外，还必须提供一批协助函数

```
complex operator+(complex, complex);
complex operator+(complex, double);
complex operator+(double, complex);
// -, *, / ...

complex operator+(complex); //一元正号
complex operator-(complex); //一元负号
bool operator==(complex, complex);
bool operator!=(complex, complex);
istream& operator>>(istream&, complex&);
ostream& operator<<(ostream&, complex);
// 还可能提供一些以极坐标方式处理复数的函数、数学函数
// 可以参阅标准库的<complex>
```


11.4 转换运算符

- 虽然通过构造函数来进行类型转换确实很方便，但是构造函数无法刻画
  - 从用户类型到一个内部类型的转换(因为内部类型不是类)
  - 从新类型到某个已有类型的转换(而不去修改那个已有类的声明)
- 这些问题可以通过为源类型定义转换运算符的方式解决
- 成员函数X::operator T()，其中T是一个类型名，定义了一个从X到T的转换

转换运算符示例

```
class Tiny
{
    char v;
    void assign(int i)
        { if(i&~077) throw Bad_range(); v = i; }
public:
    class Bad_range{};
    Tiny(int i) { assign(i); }
    Tiny& operator=(int i)
        { assign(i); return *this; }
    operator int() const { return v; }
    // 转换到int的函数，不需要声明返回值
    // 注意：int operator int() const { return v; } 是错误的
};
```


转换运算符示例

```
int main()
{
   Tiny c1 = 2;
   Tiny c2 = 62;
   Tiny c3 = c1-c2; // c3 = 60
   Tiny c4 = c3; // 不检查值域(没必要)
   int i = c1+c2; // i = 64

   c1 = c1+c2; // 值域错误：c1不能是64
   i = c3-64; // i = -4
   c2 = c3-64; // 值域错误：c2不能是-4
   c3 = c4; // 不需要检查值域
}
```

istream和ostream也依靠类型转换函数，这使得我们可以写下面这类语句

```
while(cin>>x)
    cout << x;

// cin>>x返回istream&
// 这个值被隐式地转换到
// 一个表明cin状态的值
// 然后被while检测
// 但是这类转换丢失了
// istream&信息
```


11.4.1 歧义性

- 如果同时存在用户定义转换和用户定义运算符，那么也可能产生用户定义运算符和内部运算符之间的歧义性问题

```
int operator+(Tiny, Tiny);
voif f(Tiny t, int i)
{
    t+i; // 错误：歧义，operator+(t,Tiny(i)) 或 int(t)+i ?
}
// 因此，对于一个类型而言，最好是或者依靠用户定义转换
// 或者依靠用户定义运算符，但不要两者都用
```


歧义性示例

```
class X{/*...*/X(int);X(char*);};
class Y{/*...*/Y(int);};
class Z{/*...*/Z(X);};

X f(X);  Y f(Y);  Z g(Z);

void k1()  {
   f(1); // 错误：歧义的f(X(1))或f(Y(1))?
   f(X(1));  // ok
   f(Y(1));  // ok
   g("Mack");
   // 错误：需要两次用户定义转换，不会试验g(Z(X("Mack")))
   g(X("Doc")); // ok: g(Z(X("Doc")))
   g(Z("Suzy"));// ok: g(Z(X("Suzy")))
}
```

歧义性示例

- 只有在解析一个调用时有需要，才会考虑用户定义的转换

```
class XX {/*...*/XX(int); };

void h(double);
void h(XX);

void k2()
{
   h (1); // 使用h(double(1))还是h(XX(1))?
   // 结果是调用h(double(1))，因为这一选择只使用了标准转换
};
```


歧义性示例

```
class Quad
{
public:
   Quad(double);
};
Quad operator+(Quad, Quad);
void f(double a1, double a2)
{
   Quad r1 = a1+a2; // 双精度加法
   Quad r2 = Quad(a1) + a2; // 强制要求Quad算术
}
```


11.5 友元

- 一个常规的成员函数声明描述了三件在逻辑上相互不同的事情
  - 该函数能访问类声明的私用部分
  - 该函数位于类的作用域之中
  - 该函数必须经由一个对象去激活(有一个this指针)
- static函数具有前两种性质
- 友元(friend)则只具有第一种性质

友元示例

- 比如说，我们希望定义一个运算符去做Matrix和Vector的乘法，然而，我们所要的这个乘法例程不可能同时成为两者的成员函数，此外，我们也不希望提供一些低级访问函数，使每个用户都能对Matrix和Vector的完整表示进行读写操作

```
class Matrix;
class Vector{
   float v[4];
   friend Vector operator*(const Matrix&, const Vector&);
};
class Matrix{
   Vector v[4];
   friend Vector operator*(const Matrix&, const Vector&);
};
```

友元示例

- friend声明可以放在类声明中的任何部分(私用或者公用)，放在哪里都没有关系
- 一个类的成员函数也可以是另一个类的友元
- 某个类的所有函数可以全都是另一个类的友元

```
class List_iterator{
   int* next();
};
class List{
   firend int* List_iterator::next();
};
```

```
class List{
    friend class List_iterator;
};
```


11.5.1 友元的寻找

- 像成员的声明一样，一个友元声明不会给外围的作用域引进一个名字

```
class Matrix{
   friend class Xform;
   friend Matrix invert(const Matrix&);
};

Xform x; // 错误：作用域里无Xform
Matrix (*p)(const Matrix&) = &invert;
// 错误：作用域里无invert()
```


友元的寻找示例

- 一个友元类必须或者是在外围作用域里先行声明的
- 或者是在它作为友元的那个类的直接外围的非类作用域里定义的
- 在外围的最内层名字空间作用域之外的作用域不在考虑之列

```
class X{/*...*/}; // Y的友元
namespace N{
   class Y{
      friend class X;
      friend class Z;
      friend class AE;
   };
   class Z{/*...*/}; //Y的友元
}
class AE{/*...*/};//不是Y的友元
```


友元的寻找示例

- 友元函数可以像友元类一样明确地声明，此外，还可以通过它的参数找到，即使如果它并没有在外围最近的作用域里声明

```
void f(Matrix& m)
{
   invert(m);
   //Matrix的友元invert()
}
```

11.5.2 友元和成员

- 什么时候选用友元函数，什么时候用成员函数刻画一个运算才更好？
  - 应该尽可能地减少访问类表示的函数的数目
  - 尽可能地将能访问的函数集合做好

11.6 大型对象

- 前面complex的运算符都定义为以类型complex为参数，意味着对于每个运算符，各个运算对象都需要复制，为了避免过度的复制，我们需要定义以引用为参数的函数

```
class Matrix{
   double m[4][4];
public:
   Matrix();
   friend Matrix operator+(const Matrix&, const Matrix&);
   friend Matrix operator*(const Matrix&, const Matrix&);
};
```


大型对象示例(一种避免复制的技术)

- 使用静态对象的缓冲区

```
const int max_matrix_temp=7;
Matrix& get_matrix_temp()
{
   static int nbuf = 0;
   static Matrix
      buf[max_matrix_temp];

   if(nbuf==max_matrix_temp)
         nbuf=0;
   return buf[nbuf++];
}
```

```
Matrix& operator+(const Matrix&
         arg1, const Matrix& arg2)
{
   Matrix& res=get_matrix_temp();
   //...
   return res;
}
```


11.7 基本运算符

- 一般说来，对于类型X，复制构造函数X(const X&)负责完成从同类型X的一个对象出发的初始化
- 如果一个类X有一个完成非平凡工作的析构函数，这个类很可能需要一组完整的互为补充的函数来控制构造、析构和复制
- 对于一个类，如果程序员没有显式地声明复制赋值或者复制构造函数，编译器就会生成所缺少的操作

```
class X{
   X(Sometype);
   X(const X&);
   X& operator=(const X&);
   ~X();
};
```

11.7.1 显式构造函数

- 按照默认规定，只有一个参数的构造函数也定义了一个隐式转换，例如：complex z=2; // 用complex(2)初始化
- 但是有时候我们也不希望进行隐式转换，例如，我们可以用一个int作为大小去初始化一个string，但是，某人或许会写出：string s = ‘a’; // 将s做成一个包含int(‘a’)个元素的string
- 通过将构造函数声明为explicit(显式)的方式可以抑制隐式转换

显式构造函数示例

```
class String
{
   explicit String(int n);
   String(const char* p);
};
String s1='a';  //错误
String s2(10);  //可以
String s3=String(10);//可以
String s4="Brian";//可以
String s5("Fawlty");

void f(String);
```

```
String g()
{
   f(10); //错误
   f(String(10));//可以
   f(“Arthur”);//可以
   f(s1);
   String* p1=new String("Eric");
   String* p2=new String(10);
   return 10;
   // 错误
}
```


显式构造函数示例

```
class Year
{
   int y;
public:
   explicit Year(int i):y(i) {}
   operator int()const {return y;}
};
class Date
{
public:
   Date(int d, Month m, Year y);
};
Date d3(1978,feb,21);  //错误
Date d4(21,feb,Year(1978));//可以
```

Year只不过是包裹着int的一层布，由于有operator int()，很容易将Year转换成一个int数值

将构造函数声明为explicit，可以保证int到Year的转换只能在明确要求的地方进行

类似的技术也可以用于定义表示范围的类型

11.8 下标

- 函数operator[]可以用于为类的对象定义下标运算的意义
- operator[]的第二个参数(下标)可以具有任何类型，这就使我们可以去定义vector、关联数组等
- operator[]()必须是成员函数

下标示例

```
class Assoc {
   struct Pair {
      string name;
      double val;
      Pair(string n="",double v=0):name(n), val(v) {}
   };
   vector<Pair> vec;
   Assoc(const Assoc&); //私用，防止复制
   Assoc& operator=(const Assoc&);//私用，防止复制
public:
   Assoc(){}
   const double& operator[](const string&);
   double& operator[](string&);
   void print_all() const;
};
```


下标示例

```
double& Assoc::operator[](string& s){
// 检索s：如果找到就返回其值；
// 否则，做一个新的Pair并返回默认值0
   for(vector<Pair>::const_iterator p = vec.begin();
      p!=vec.end(); ++p)
         if(s==p->name) return p->val;
   vec.push_back(Pair(s,0));
   return vec.back().val;
}
void Assoc::print_all() const{
   for(vector<Pair>::const_iterator p=vec.begin();
      p!=vec.end();++p)
      cout << p->name<<":"<<p->val<<'\n';
}
```


下标示例

```
int main()
// 计算每一个单词在输入时出现的次数
{
    string buf;
    Assoc vec;
    while(cin>>buf) vec[buf]++;
    vec.print_all();
}
```


11.9 函数调用

- 函数调用，记法expression(expression-list)，也可以解释为一种二元符号，其中将expression作为左运算对象，而expression-list作为右运算对象，该调用运算符()也可以被重载
- 运算符()最明显、或许也是最重要的应用就是为对象提供常规的函数调用语法形式，使它们具有像函数似的行为方式，一个活动起来像函数的对象常常被称作一个拟函数对象，或简称为函数对象(仿函数对象)
- operator()()的另外一些很流行的应用包括作为子串运算符，以及作为多维数组的下标运算符
- operator()()必须作为成员函数

函数调用示例

```
template<class Iter, class Fct>Fct for_each
    (Iter b, Iter e, Fct f)
{
    while(b!=e) f(*b++);
    return f;
};
// for_each是一个模板，它反复地应用第三个参数的()
void negate(complex& c){ c = -c; }

void f(vector<complex>& aa, list<complex>& ll)
{
    for_each(aa.begin(), aa.end(), negate);
    for_each(ll.begin(), ll.end(), negate);
}
```


函数调用示例

```
void add23(complex c)
{
    c+=complex(2,3);
}
void g(vector<complex>& aa, list<complex>& ll)
{
    for_each(aa.begin(), aa.end(), add23);
    for_each(ll.begin(), ll.end(), add23);
}

// 解决向一个向量或者列表中所有元素加一个固定的值的问题
```


函数调用示例

```
class Add{
    complex val;
public:
    Add(complex c) { val=c; }
    Add(double r, double i){ val = complex(r,i); }
    void operator()(complex& c) const{ c+= val; }
};

void h(vector<complex>& aa, list<complex>& ll, complex z)
{
    for_each(aa.begin(), aa.end(), Add(2,3));
    for_each(ll.begin(), ll.end(), Add(z));
}
// 解决向一个向量或者列表中所有元素加一个任意的值的问题
```


11.10 间接(Dereferencing)

- 间接(提领)运算符->可以被定义为一个一元的后缀运算符，即：给出了类

```
class Ptr{
    /*...*/
    X* operator->();
};
```

就可以用类Ptr的对象访问类X的成员，其方式就像所用的Ptr的对象是一个指针，例如：

```
void f(Ptr p)
{
    p->m=7;//(p.operator->())->m=7
} // 其中m是类型X的成员
```

间接

- 从对象p到指针p.operator->()的转换并不依赖于被指向对象的成员m，这也是->被看作是一元后缀运算符的理由，但是，无论如何这里并没有引进一种新语法形式，所以，在->之后仍然要求写一个成员名字

```
void g(Ptr p)
{
   X* q1 = p->; // 语法错误
   X* q2 = p.operator->(); // ok
}
```


间接

- 间接是一个很重要的概念
- 迭代器是这个说法的一个重要佐证
- 认识运算符->的另一种方式是把它看成C++里提供的一种受限的，但也非常有用的委托(delegation)机制
- 运算符->必须是成员函数，如果使用，它的返回类型必须是一个指针，或者是一个你可以使用->运算符的类对象

间接示例

- 重载->的最有用之处就是创建所谓的“灵巧指针”，也就是一种行为像是指针的对象

```
class Rec_ptr {
   const char* identifier;
   Rec* in_core_address;
public:
   Rec_ptr(const char* p):identifier(p),in_core_address(0){}
   ~Rec_ptr(){ write_to_disk(in_core_adress, identifier);}
   Rec* operator->();
};
Rec* Rec_ptr::operator->() {
   if(in_core_address==0)
         in_core_address=read_from_disk(identifier);
   return in_core_address;}
```


间接示例

```
struct Rec
{
   string name;
};

void update(const char* s)
{
   Rec_ptr p(s); // 对s得到一个Rec_ptr
   p->name = “Roscoe”;
   // 更新s：如果需要，第一次将从磁盘读取
}
// 真正使用时，Rec_ptr可能是个模板，而Rec类型则是它的参数
// 此外，实际程序可能需要包含错误处理代码等
```

间接示例

- 对于常规指针，->的使用和一元*以及[]的某些使用方式意义相同，有了Y* p;则下面的关系成立：p->m==(*p).m==p[0].m(其中m是Y的成员)
- 对于用户自己定义的->()运算符，不保证这些关系，但若必要，用户也可以提供这些等价性

```
class Ptr_to_Y{
    Y* p;
public:
    Y* operator->(){ return p;}
    Y& operator*(){ return *p; }
    Y& operator[](int i) { return p[i]; }
};
```


11.11 增量和减量

- 一旦人们定义了“灵巧指针”，他们也常常会决定需要提供增量运算符++和减量运算符--，去模拟这些运算符对于内部类型的使用

```
class Ptr_to_T
{
    T* p;
    T* array;
    int size;
public:
    Ptr_to_T(T* p, T* v, int s);
    Ptr_to_T(T* p);
    Ptr_to_T& operator++(); // 前缀
    Ptr_to_T operator++(int);// 后缀
    Ptr_to_T& operator--();// 前缀
    Ptr_to_T operator--(int);// 后缀
    T& operator*();// 一元*
}; // 注意后缀操作符声明中虚设的参数
```


增量和减量示例

```
void f1(T a)
{
    T v[200];
    T* p=&v[0];
    p--;
    *p=a;
    // p越界，没有捕捉到
    ++p;
    *p=a; // ok
}
```

```
void f2(T a)
{
    T v[200];
    Ptr_to_T p(&v[0],v,200);
    p--; // p.operator--(0);
    *p=a;
    // p.operator*() = a;
    // 运行错误，p越界
    ++p;
    *p=a; // ok
}
```


11.12 一个字符串类

- 一个更实际的String类版本
- 提供了一种值语义、字符读写操作、检查和不检查的访问、流I/O、用字符串文字量作为文字量、相等和拼接运算符，将字符串表示为C风格的、用0结束的字符数组，并利用引用计数尽可能地减少复制

11.13 忠告

- [1] 定义运算符主要是为了模仿习惯使用方式
- [2] 对于大型运算对象，请使用const 引用参数类型
- [3] 对于大型的结果，请考虑优化返回方式
- [4] 如果默认复制操作对一个类很合适，最好是直接用它
- [5] 如果默认复制操作对一个类不合适，重新定义它，或者禁止它
- [6] 对于需要访问表示的操作，优先考虑作为成员函数而不是作为非成员函数
- [7] 对于不访问表示的操作，优先考虑作为非成员函数而不是作为成员函数

忠告

- [8] 用名字空间将协助函数与“它们的”类关联起来
- [9] 对于对称的运算符采用非成员函数
- [10] 用()作为多维数组的下标
- [11] 将只有一个“大小参数”的构造函数做成explicit
- [12] 对于非特殊的使用，最好是用标准string而不是你自己的练习
- [13] 要注意引进隐式转换的问题
- [14] 用成员函数表达那些需要左值作为其左运算对象的运算符

C++编程(10)

唐晓晟
北京邮电大学电信工程学院

第12章 派生类

- 引言
- 派生类
- 抽象类
- 类层次结构的设计
- 类层次结构和抽象类
- 忠告

12.1 引言

- C++从Simula那里借用了类以及类层次结构的概念，此外，还有有关系统设计的思想
- 概念不会孤立的存在，它总与一些相关的概念共存，用类来描述概念，不可避免的需要用类描述概念之间的关系
- 派生类的概念及其相关的语言机制使得我们能够表述一种层次性的关系，即：表述一些类之间的共性

- 例如：圆和三角形概念之间有关系，因为它们都是形状，即它们之间共有形状这个概念
- 若在程序中表示一个圆和一个三角形，但是却没有涉及到形状的概念，就应该认为是丢掉了某些最基本的东西
- 这个简单的思想就是面向对象程序设计的基础

12.2 派生类

- 考虑做一个程序，处理某公司雇佣的人员

```
struct Employee                 struct Manager
{                               {
    string first_name;              Employee emp;
    string family_name;             // 经理的雇佣记录
    char middle_initial;            list<Employee*> group;
    Date hiring_date;               // 所管理的人员
    short department;               short level;
};                              };
```

- 程序员很容易知道Employee和Manager之间的关系，但是编译器却对此一无所知
- 好的描述方法应该能够把Manager也是Employee的事实明确地表述出来

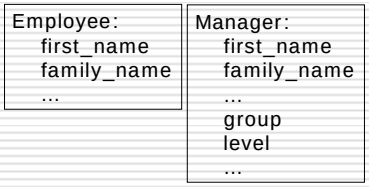
派生类示例

```
struct Manager : public Employee
{
    list<Employee*> group;
    short level;
}; // 注意，Employee必须在Manager之前定义，不能只声明
```

- 此Manager是从Employee派生(继承)出来的，也即：Employee是Manager的一个基类，而Manager则是Employee的一个子类
- 类Manager中包含了类Employee中的所有成员，再加上一些自己的成员

派生类的实现方式

一种常见的实现方式，就是将派生类的对象也表示为一个基类的对象，只是将那些特别属于派生类的信息附加在最后




```
void f(Manager ml, Employee el)
{
    list<Employee*> elist;
    elist.push_front(&ml); // 可以，因为Manager也是Employee
    elist.push_front(&el); // 可以
    // 注意，反之则不可，Employee不一定是Manager
}
```


派生类示例

- 可以用Derived*给Base*类型的变量赋值，不需要显式的转换，而相反的方向，则必须显式转换

```
void g(Manager mm, Employee ee)
{
    Employee* pe = &mm; // 可以
    Manager* pm = &ee; // 错误
    pm->level = 2;  // 错误：pm没有level变量
    pm=static_cast<Manager*>(pe); // 蛮力，可以
    pm->level = 2; // 没问题
}
```

12.2.1 成员函数

- 对于派生类，只能使用基类中的公用的和保护(protected)的成员，但是派生类不能使用基类的私用名字

```
class Employee{
   string first_name, family_name;
   char middle_initial;
   //...
public:
   void print() const;
   string full_name() const
   {
      return first_name + ' ' +
middle_initial + ' ' + family_name;
   }
};
```

```
class Manager : public Employee
{
   // ...
public:
   void print() const;
};
void Manager::print() const
{
   cout << "name is " <<
full_name() << "\n";
   //cout << family_name; // 错误
}
```


派生类中调用基类的同名函数

```
void Manager:print() const
{
   Employee::print();
   // 打印Employee的信息
   cout << level;
   // 打印Manager的特殊信息
}

void Manager:print() const
{
   print(); //错误，无穷递归调用
}
```


12.2.2 构造函数和析构函数

```
class Employee{
   string first_name, family_name;
   short department;
   //...
public:
   Employee(const string& n, int d);
};
class Manager : public Employee
{
   list<Employee*> group;
   short level;
public:
   Manager(const string& n, int d, int
level);
};
```

```
Employee::Employee(const
string& n, int d) :
family_name(n),
department(d) {}
// ok
Manager::Manager(const
string& n, int d, int lvl) :
Employee(n,d), level(lvl) {}
// ok
Manager::Manager(const
string& n, int d, int lvl) :
family_name(n),
department(d), level(lvl) {}
// 错误，无法访问
```

构造是自下而上进行：基类、成员、派生类，销毁时则顺序相反

12.2.3 复制

- 类对象的复制由复制构造函数和赋值操作定义
- 注意，如果没有为Manager定义复制运算符，编译器会为你生成一个，也即：该运算符是不继承的，构造函数也是如此

```
class Employee
{
   Employee& operator=(const Employee&);
   Employee(const Employee&);
};
void f(const Manager& m) {
   Employee e = m; // 从m的Employee部分创建e
   // 这种情况叫做切割(slicing)
   e = m; // 用m的Employee部分给e赋值
}
```

12.2.4 类层次结构(Class Hierarchies)

- 派生类也可以做为基类

```
class Employee{/*...*/};
class Manager:public Employee{/*...*/};
class Director:public Manager{/*...*/};
```

- 一组相关的类按照习惯被称为一个类层次结构

```
class Temporary{/*...*/};
class Secretary:public Employee{/*...*/};
class Tsec:public Temporary, public Secretary{/*...*/};
class Consultant:public Temporary, public Manager{/*...*/};
```

- C++可以表示类的有向无环图

12.2.5 类型域

- 为了使派生类不仅仅是一种完成声明的方便简写形式，必须解决如下问题：对于给定的类型为Base*的指针，被指的对象到底属于哪个派生类？
- 四种基本的解决方案
  - (1)保证被指的只能是唯一类型的对象
  - (2)在基类里安排一个类型域，供函数检查
  - (3)使用dynamic_cast
  - (4)使用虚函数

方案2的示例

```
struct Employee
{
   enum Empl_type{M,E};
   Empl_type type;
   Employee():type(E){}
   string ...
};

struct Manager:public
Employee
{
   Manager(){type=M;}
   list<Employee*> group;
   ...
};
```

```
void print_employee(const Employee* e)
{
   switch(e->type){
      case Employee::E:
         cout << e->family_name << e-
>department;
         break;
      case Employee::M:
         cout << e->family_name << e-
>department;
         const Manager* p =
static_cast<const Manager*>(e);
         cout << p->level;
         break;
   }}
```


方案2的示例

- 这种方式可以工作，特别是在由一个人维护的小程序里
- 致命弱点：程序员按照特定的方式来操纵这些类型，这种方式是编译器无法检查的
- 如果引入Employee的新的派生类，需要改动太多的原有代码，维护麻烦
- 对于模块化以及信息隐藏原则是一个很大的破坏，基类是某种“公用信息”的展台

12.2.6 虚函数

- 虚函数能克服类型域解决方案的缺陷，它使得程序员可以在基类里声明一些能够在各个派生类里重新定义的函数，编译器和加载程序能保证对象和应用于它们的函数之间正确的对应关系
- 派生类里，相关函数的参数类型必须和基类完全相同
- 虚函数第一次声明的那个类里，必须对其进行定义，除非为纯虚函数

```
class Employee{
    string first_name, family_name;
    short department;
public:
    Employee(const string& n, int d);
    virtual void print() const; // 指明print()的作用就像是一个界面
};
```


虚函数示例

```
void Employee::print() const{
    cout << family_name <<
department;
}
class Manager:public Employee {
    list<Employee*> group;
    short level;
public:
    Manager(const string& n, int d,
int lvl);
    void print() const;
};
void Manager::print() const {
    Employee::print();
    cout << level;
}
```

```
void print_list(const
list<Employee*>& s)
{
for(list<Employee*>::
const_iterator p=s.begin();
p!=s.end(); ++p)
    (*p)->print();
}
int main(){
    Employee e("Brown",1234);
    Manager m("Smith",1234,2);
    list<Employee*> empl;
    empl.push_front(&e);
    empl.push_front(&m);
    print_list(empl);
}
```


虚函数进一步说明

- 上述例子完全可行，即使print_list()是在派生类Manager之前写好并编译好的，这是类机制中最关键的一个方面
- 从Employee的函数中取得"正确的"行为，而又不依赖于实际使用的到底是哪个Employee，就是所谓的多态性
- 一个带有虚函数的类型被称为是一个多态类型，要在C++中取得多态性的行为，被调用的函数就必须是虚函数

12.3 抽象类

- 有些可以作为其他类的基类的类，但是不需要从为这种类生成一个实际的对象，或者说，该类仅仅提供了一种概念上的抽象
- 可以使用纯虚函数来表达这种抽象

```
class Shape
{
public:
    virtual void rotate(int){error("Shape::rotate");}
    virtual void draw(){error("Shape::draw");}
};
Shape s; // 愚蠢的声明，对s的任何操作都导致错误
```

抽象类

- 将类Shape声明为纯虚函数
- 如果一个类中存在一个或者几个纯虚函数，这个类就是一个抽象类
- 不能创建抽象类的对象，只能使用该类作为其他类的基类

```
class Shape
{
public:
    virtual void rotate(int) = 0;
    virtual void draw() = 0;
    virtual bool is_closed() = 0;
};
Shape s; // 错误
```


抽象类示例

```
class Point{/**/};
class Circle : public Shape
{
public:
    void rotate(int){}
    void draw();
    bool is_closed(){return true;}
    Circle(Point p, int r);
private:
    Point center;
    int radius;
};
```

一个未在派生类里定义的纯虚函数仍旧还是一个纯虚函数，也使得该类仍为一个抽象类，这就使得我们可以分步骤构筑起一个实现

抽象类的最重要用途就是提供一个界面，而又不保露任何实现的细节

12.4 类层次结构的设计

- 一个简单的设计问题：为某个程序提供一种方式，通过它可以从某种用户界面取得一个整数
- 想法：采用一个类Ival_box，它知道能够接受的输入值的取值范围，程序可以向Ival_box要求一个值，必要时要求它去提示用户输入，此外，程序可以询问Ival_box，看看程序最后一次查看之后用户是否修改过有关的值

12.4.1 一个传统的层次结构

```
class Ival_box
{
protected:
    int val;
    int low, high;
    bool changed;
public:
    Ival_box(int ll, int hh){
        changed = false;
        val = low = ll;
        high = hh;
    }
    virtual int get_value(){
        changed = false;
        return val;
    }
    virtual void set_value(int i){
        changed = true;
        val = i;
    } // 为用户
    virtual void reset_value(int i){
        changed = false;
        val = i;
    } // 为应用
    virtual void prompt() {}
    virtual bool was_changed() const{
        return changed;
    }
};
```

Ival_box的使用

```
void interact(Ival_box* pb)
{
    pb->prompt();
    int i = pb->get_value();
    if(pb->was_changed()){
        //...
    }
    else {
        //...
    }
}
```

```
class Ival_slider:public Ival_box{};
void some_fct()
{
    Ival_box* p1 = new Ival_slider(0,5);
    interact(p1);
    Ival_box* p2 = new Ival_dial(1,12);
    interact(p2);
}
```


- 大部分图形界面系统都提供了一个类，其中定义了屏幕实体的基本性质，如果采用“Big Bucks Inc”的系统，各个类定义如下

```
class Ival_box : public BBwindow{/*…*/};
class Ival_slider : public Ival_box{/*…*/};
class Ival_dial : public Ival_box{/*…*/};
class Flashing_ival_slider : public Ival_slider{/*…*/};
class Popup_ival_slider : public Ival_slider{/*…*/};
```


12.4.1.1 评论

- 缺点1：使用BBWindow作为Ival_box的基类(BBWindow只是一个实现细节)，如果想让Ival_box能够在其他的系统中使用，必须定义多个不同版本

```
class Ival_box : public BBwindow{/*…*/};
class Ival_box : public CWwindow{/*…*/};
class Ival_box : public IBwindow{/*…*/};
class Ival_box : public LSwindow{/*…*/};
```

- 缺点2：每个派生类都共享着在Ival_box里声明的基本数据，也即“在Ival_box中不应该包含具体数据
- 缺点3：修改BBWindow将使得重新编译甚至重写相关的代码
- 缺点3：无法在一个混合式的环境(多种图形界面系统)里使用

12.4.2 抽象类

- 解决上述问题
  - 用户界面系统应该是一个实现细节，对于不希望了解它的用户应是隐蔽的
  - 类Ival_box应该不包括数据
  - 在用户界面修改后，不应该要求重新编译那些使用了Ival_box这一族类的代码
  - 针对不同界面系统的Ival_box应能在我们的程序里共存
- 本小节介绍的是一种能够最清晰地映射到C++语言中的途径

将Ival_box设计为一个抽象界面

```
class Ival_box {
public:
    virtual int get_value() = 0;
    virtual void set_value(int i)
= 0;
    virtual void reset_value(int
i) = 0;
    virtual void prompt() = 0;
    virtual bool was_changed()
const = 0;
    virtual ~Ival_box() {}
};
```

```
class Ival_slider : public Ival_box,
protected BBwindow
{
public:
    Ival_slider(int, int);
    ~Ival_slider();
    int get_value();
    void set_value(int i);
    //..
protected:
    //覆盖BBwindow虚函数的函数
    //例如：draw(), mousehit()...
private:
    // slider所需要的数据
};// 多重继承的方式
```


其他

- 虚析构函数假定派生类需要某种清理工作，因此，用户需要在派生类里面适当地覆盖该虚析构函数，虚析构函数保证会调用有关的类析构函数，完成正确的清理工作

```
void f(Ival_box* p){
    /*…*/ delete p;
}
```

- 现在的类层次结构

```
class Ival_box {/*…*/};
class Ival_slider : public Ival_box, protected BBwindow{/*…*/};
class Ival_dial : public Ival_box, protected BBwindow{/*…*/};
class Flashing_ival_slider : public Ival_slider{/*…*/};
class Popup_ival_slider : public Ival_slider{/*…*/};
```


12.4.3 其他实现方式

- 与传统方式相比，这种设计更加清晰也更容易维护，而且并不低效
- 但是没有解决版本控制问题

```
class Ival_box {/*…*/};
class Ival_slider : public Ival_box, protected BBwindow{/*…*/};
class Ival_slider : public Ival_box, protected CWwindow{/*…*/};
```

- 一个明显的解决方案是采用不同的名字，定义几个不同的Ival_slider类

```
class Ival_box {/*…*/};
class BB_ival_slider : public Ival_box, protected BBwindow{/*…*/};
class CW_ival_slider : public Ival_box, protected CWwindow{/*…*/};
```


其他实现方式(2)

- 进一步设计，可以将Ival_box和实现细节隔离开，从Ival_box派生出来一个抽象的Ival_slider类，然后从它派生出来针对各个系统的Ival_slider类

```
class Ival_box {/*…*/};
class Ival_slider:public Ival_box{/*…*/};
class BB_ival_slider:public Ival_slider, protected BBwindow{/*…*/};
class CW_ival_slider:public Ival_slider, protected CWwindow{/*…*/};
```

- 通常，还可以利用实现层次方面的一些更特殊的类，把事情做的更好一些

```
class BB_ival_slider:public Ival_slider, protected BBslider{/*…*/};
class CW_ival_slider:public Ival_slider, protected CWslider{/*…*/};
```

其他实现方式(3)

- 最终，完整的类层次可以是将我们原来的面向应用的概念层次结构当作界面而组成的

```
class Ival_box {/*…*/};
class Ival_slider : public Ival_box{/*…*/};
class Ival_dial : public Ival_box {/*…*/};
class Flashing_ival_slider : public Ival_slider{/*…*/};
class Popup_ival_slider : public Ival_slider{/*…*/};

class BB_ival_slider:public Ival_slider,protected BBslider{/*…*/};
class BB_flashing_ival_slider:public Flashing_ival_slider, protected
BBwindow_with_bells_and_whistles{/*…*/};
class BB_popup_ival_slider:public Popup_ival_slider,protected
BBslider{/*…*/};
class CW_ival_slider:public Ival_slider,protected CWslider{/*…*/};
```


12.4.3.1 评论

- 这种抽象类的设计非常灵活，而且，与那种依赖于一个定义用户界面系统的公共基类的等价设计方式比较，处理起来几乎同样容易
- 从应用系统的角度看，这两种设计的意义是等价的，几乎所有的代码都能不加修改，而且能以同样的方式使用
- 使用抽象类设计，几乎所有的用户代码都得到了保护，能够抵御实现层次上的结构变化，而且在这种变化后不需要重新编译

12.4.4 对象创建的局部化

- 对象的创建必须通过特定于实现的名字去做，例如CW_ival_dial和BB_flashing_ival_slider等，我们希望尽可能减少出现这些特殊名字的地方
- 解决的方法是引入一个间接(可以通过多种方式做到)，一种简单的方式就是引入一个抽象类，用来表示一组创建操作

```
class ival_maker
{
public:
    virtual Ival_dial* dial(int,int)=0; // 做拨盘
    virtual Popup_ival_slider* popup_slider(int,int) = 0;
    //做弹出式滑块
}; // 这样的类被称为是一个工厂(factory)，它的函数有时被称为
    // 虚构造函数(有点易于令人误解)
```


对象创建的局部化

```
class BB_maker : public ival_maker // 做BB版本
{
public:
    Ival_dial* dial(int,int);
    Popup_ival_slider*
        popup_slider(int,int);
};
class LS_maker : public ival_maker // 做LS版本
{
public:
    Ival_dial* dial(int,int);
    Popup_ival_slider*
        popup_slider(int,int);
};
```

对象创建的局部化

```
Ival_dial* BB_maker::dial(int a, int b)
{
   return new BB_ival_dial(a,b);
}

Ival_dial* LS_maker::dial(int a, int b)
{
   return new LS_ival_dial(a,b);
}

void user(Ival_maker* pim)
{
   Ival_box* pb=pim->dial(0,99);
   //...
}
```

```
BB_maker BB_impl;
// 为BB用户
LS_maker LS_impl;
// 为LS用户
void driver()
{
   user(&BB_impl);
   // 使用BB
   user(&LS_impl);
   // 使用LS
}
```


12.5 类层次结构和抽象类

- 一个抽象类就是一个界面
- 类层次结构是一种逐步递增地建立类的方式，位于其中的类一方面为用户提供了有用的功能，同时也作为实现更高级或者更特殊的类的构造基础块，这种层次结构对于支持以逐步求精方式进行的程序设计是非常理想的
- 抽象类的层次结构是一种表述概念的清晰而强有力的方法，又不会用实现细节或者运行时的额外开销去妨碍这种表述

类层次结构和抽象类

- 虚函数的调用是廉价的，与它跨过的抽象边界的种类无关，调用一个抽象类的成员函数的开销与调用任何其他virtual函数完全一样
- 结论：一个系统展现给用户的应该是一个抽象类的层次结构，其实现所用的是一个传统的层次结构

12.6 忠告

- [1]避免类型域
- [2]用指针和引用避免切割问题
- [3]用抽象类设计的中心集中到提供清晰的界面方面
- [4]用抽象类使界面最小化
- [5]用抽象类从界面中排除实现细节
- [6]用虚函数使新的实现能够添加进来，又不会影响用户代码
- [7]用抽象类去尽可能减少用户代码的重新编译
- [8]用抽象类使不同的实现能够共存
- [9]一个有虚函数的类应该有一个虚析构函数
- [10]抽象类通常不需要构造函数
- [11]让不同概念的表示也不相同

C++编程(11)

唐晓晟
北京邮电大学电信工程学院

第13章 模板

- 引言
- 一个简单的string模板
- 函数模板
- 用模板参数描述策略
- 专门化
- 派生和模板
- 源代码组织
- 忠告

13.1 引言

- 模板提供了一种很简单的方式来表述范围广泛的一般性概念，以及一些组合它们的简单方法，利用模板产生出来的类和函数，在运行时和空间效率方面，可以与手工写出的更特殊的代码媲美
- 模板直接支持采用类型作为参数的程序设计
- 这里对于模板的介绍将集中关注与标准库的设计、实现和使用有关的各种技术

引言

- 标准库要求的是更高水平的通用性、灵活性和效率
- 每个主要的标准库抽象都被表示为一个模板(string, ostream, complex, list和map等)，所有的关键操作(string比较、输出运算符<<、complex的加法、取得list的下一个元素、sort()等)也是如此
- 下面通过一些小例子阐述模板的各种基本技术

引言

- 13.2 定义和使用类模板的基本机制
- 13.3 函数模板，函数重载和类型推断
- 13.4 用模板参数刻画通用型算法的策略
- 13.5 通过多个定义为一个模板提供多种实现
- 13.6 派生和模板(运行时的和编译时的多态性)
- 13.7 源代码组织

13.2 一个简单的String模板

- 串是一个能够保存一些字符的类，而且提供了各种串的基本操作
- 实际应用中，我们可能希望为多种不同种类的字符(有符号字符、无符号字符、中文字符、希腊字符)提供这样的功能，因此希望能够以最不依赖于特定字符种类的方式给出串的概念
- 因此可以将以前的char类型的串抽象为一个更具普遍性的串类型

一个简单的String模板

```
template<class C> class String
{
    struct Srep;
    Srep* rep;
public:
    String();
    String(Const C*);
    String(const String&);

    C read(int i) const;
    //...
};
```

template<class C>前缀说明当前正在声明的是一个模板，它有一个将在声明中使用的类型参数C

C的作用域将一直延伸到由这个template<class C>作为前缀的声明的结束处

C代表了一种类型，不必一定是某个类的名称

示例

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;
class Jchar{
    // ...
};
String<Jchar> js;
```

```
// 统计输入各个单词出现的次数
int main()
{
    String<char> buf;
    map<String<char>, int> m;
    while(cin>>buf) m[buf]++;
}
// 日文版本
int main()
{
    String<Jchar> buf;
    map<String<Jchar>,int> m;
    while(cin>>buf) m[buf]++;
}
```

示例

```
// 标准库提供了一个模板类basic_string，其中，string被定义为basic_string<char>的同义词
typedef basic_string<char> string;

int main()
{
    string buf;
    map<string, int> m;
    while(cin>>buf) m[buf]++;
}
```


13.2.1 定义一个模板

- 一个由类模板生成的类也是一个完全正常的类
- 方法：先做好特定的类，如String，并排除其中错误，然后再将它转化为String<C>一类的模板
- 模板类中成员的声明和定义与在非模板类里完全一样，模板类的成员也不一定非要在类内部定义，可以在外部定义，成员本身也是模板参数化的

定义一个模板示例

```
template<class C>
class String
{
    struct Srep;
    Srep* rep;
public:
    String();
    String(Const C*);
    String(const String&);

    C read(int i) const;
    //...
};
```

```
template<class C> struct
String<C>::Srep {
    C* s;
    int sz;
    int n;
};

template<class C> C
String<C>::read(int i) const {
    return rep->s[i];
}

template<class C> String<C>::String()
{
    rep = new Srep(0,C());
};
```


定义一个模板示例

类模板的名字不能重载

```
template<class T> class String{/*…*/};

class String{/*…*/}; // 错误，重复定义
```

13.2.2 模板实例化

- 从一个模板类和一个模板参数生成一个类声明的过程通常被称为模板实例化
- 该说法同样用于由模板函数加上模板参数产生一个函数的过程
- 针对一个特定模板参数的模板版本被称为是一个专门化

模板实例化

- 一般来说，具体实现负责保证对所用的每组模板参数能够生成模板函数的相应版本，程序员不必考虑该问题
- 下例中，实现应该生成String<char>和String<Jchar>的声明、与它们对应的Srep类型、构造和析构函数、以及赋值String<char>::operator=(char*)，其他的成员函数没有被使用，也就不应该生成

```
String<char> cs;
String<Jchar> js;
cs = " It's the implementation's job to figure out what
code needs to be generated.";
```


13.2.3 模板参数

- 模板的参数可以有常规类型参数如int，也可以有模板参数，一个模板也可以有多个参数，例如：一个模板参数可以用于定义跟随其后的模板参数

```
template<class T, T def_val> class Cont{/*…*/};
```

- 整数模板参数常常用于提供大小或者界限，必须为常量

模板参数示例

```
template<class T, int i> class Buffer
{
   T v[i];
   int sz;
public:
   Buffer():sz[i]{}
};
Buffer<char,127> cbuf;
Buffer<Record,9> rbuf;

// 当运行效率和紧凑性特别重要时，可以使用Buffer这类容器，将数组大小作为参数传递，可以使Buffer的实现避免使用自由存储(例如vector或者string)
```

模板参数可以使常量表达式，具有外部连接的对象或者函数的地址，或者非重载的指向成员的指针，用作模板参数的指针必须具有&of的形式，其中of是对象或者函数的名字；或者具有f的形式，f必须是一个函数名，到成员的指针必须具有&X::of的形式，这里的of是一个成员名

13.2.4 类型等价

- 给出一个模板，可以为它提供模板参数，以生成一些类型

```
String<char> s1;                    对于同一模板使用同一组模板
String<unsigned char> s2;           参数，将总是表示同一个生成
Sring<int> s3;                      出的类型

typedef unsigned char Uchar;        typedef并不引入新的类型，
String<Uchar> s4;                   所以String<Uchar>和
String<char> s5;                    String<unsigned char>是同
                                    样的类型(同一组模板参数)
Buffer<String<char>,10> b1;
Buffer<char,10> b2;                 编译器能对常量表达式求值，
Buffer<char,20-10> b3;              左边10和20-10是同样的类型
```


13.2.5 类型检查

- 模板定义时，要检查这个定义的语法错误，编译器可以帮助捕捉一些简单的语义错误
- 与模板参数的使用有关的错误在模板使用之前不可能检查出来
- 与模板参数相关的错误能被检查出来的最早位置，也就是在这个模板针对该特定模板参数的第一个使用点，被称为第一个实例化点，简称实例化点

类型检查示例

```
template<class T> class List {
   struct Link { Link* pre;  Link* suc; T val;
      Link(Link* p, Link* s, const T& v) : pre(p), suc(s), val(v) {}
   }  // 错误
   Link* head;
public:
   List() : head(7) {} // 错误
   List(const T& t):head(new Link(0,o,t)) {} // 错误
   void print_all() const {
      for(Link* p=head;p;p=p->suc)
         cout << p->val << '\n';
   }};
class Rec{/*...*/};
void f(const List<int>& li, const List<Rec>& lr){
   li.print_all();    // ok          lr.print_all();
} // 错误，Rec没有定义<<运算符号
```


13.3 函数模板

- 一般在应用了容器类(basic_string, vector, vector, list, map)后，就会提出对于模板函数的要求
- 在调用模板函数时，函数参数的类型决定到底应使用模板的哪个版本

函数模板示例

```
template<class T> void sort(vector<T>&);
void f(vector<int>& vi, vector<string>& vs) {
   sort(vi); // sort(vector<int>&);
   sort(vs); // sort(vector<string>&);
}

template<class T> void sort(vector<T>& v)
{ // Shell sort(Knuth, Vol. 3, pg.84)
  const size_t n = v.size();
  for(int gap=n/2; 0<gap; gap/=2)
    for(int i=gap; i<n; i++)
      for(int j=i-gap; 0<=j; j-=gap)
          if(v[j+gap]<v[j]) swap(v[j], v[j+gap]);
          // 注意，用<做比较运算，并非所有类型都支持该运算符
}
```


13.3.1 函数模板的参数

- 模板函数对于写出通用型算法是至关重要的，这种算法可以应用于广泛且多样化的容器类型
- 对于一个调用，能从函数的参数推断出模板参数的能力是其中最关键的

函数模板的参数示例

- 编译器能够从一个调用推断出类型参数和非类型参数，条件是由这个调用的函数参数表能够唯一地标识出模板参数的一个集合

```
template<class T, int i> T& lookup(Buffer<T,i>& b, const char* p);
class Record{
  const char v[12];
  // ...
};
Record& f(Buffer<Record,128>& buf, const char* p)
{
  return lookup(buf,p); // 使用lookup()，其中T是Record, i是128
}
```


函数模板的参数示例

- 如果不能从模板函数的参数推断出某个模板参数，那么就必须显式地去描述它

```
template<class T> class vector{/*...*/};
template<class T> T* create();
// 返回一个指向T的指针
void f()
{
  vector<int> v;
  int* p = create<int>(); // 注意create后面的<int>参数
}
```

函数模板的参数示例

- 显式描述的最常见用途是为模板函数提供返回值类型
- 可以将implicit_cast函数理解为隐式转换的一个显式版本

```
template<class T, class U> T implicit_cast(U u) {return u;}
void g(int i)
{
  implicit_cast(i); // 错误：无法推断T
  implicit_cast<double>(i); // T是double，U是int
  implicit_cast<char,double>(i); // T是char，U是double，可以
  implicit_cast<char*,int>(i); // T是char*，U是int
                               // 错误，无法将int转换为char*
}
```


13.3.2 函数模板的重载

- 可以声明多个具有同样名字的函数模板，甚至可以声明具有同一个名字的多个函数模板和常规函数的组合

```
template<class T> T sqrt(T);
template<class T> complex<T> sqrt(complex<T>);
double sqrt(double);

void f(complex<double> z)
{
  sqrt(2); // sqrt<int>(int)
  sqrt(2.0); // sqrt(double)
  sqrt(z); // sqrt<double>(complex<double>)
}
```


模板函数重载规则

- 找出能参与这个重载解析的一组函数模板的专门化。例如：sqrt(z)将产生候选函数sqrt<double>(complex<double>)和sqrt<complex<double>>(complex<double>)
- 如果两个模板函数都可用，而其中一个比另外一个更专门，在随后的步骤中就只考虑那个最专门的模板函数。例：sqrt(z)意味着sqrt<double>(complex<double>)而非sqrt<complex<double>>(complex<double>)

模板函数重载规则

- 在这组函数上做重载解析，包括那些常规函数，如果某个函数模板参数已经通过模板参数推断确定下来，这个参数就不能再同时应用提升、标准转换或者用户定义的转换。对于sqrt(2)，sqrt<int>(int)是确切匹配，优先于sqrt(double)
- 如果一个函数和一个专门化具有同样的匹配，那么就选用函数，因此，sqrt(2.0)将选用sqrt(double)而不是sqrt<double>(double)
- 如果找不到匹配或者找到多个匹配，都产生错误

模板函数重载示例

```
template<class T> T max(T,T);
const int s = 7;
void k(){
  max(1,2); // max<int>
  max('a','b'); // max<char>
  max(2.7,4.9); // max<double>
  max(s,7); // max<int>(int(s),7)
  max('a',1); // 错误，歧义
  max(2.7,4); // 错误，歧义
}
void f() { // 显式限定，避免歧义
  max<int>('a',1);
  max<double>(2.7,4);
}
```


模板函数重载示例

- 也可以增加适当的声明来避免歧义

```
int max(int i, int j) {return max<int>(i,j);}
double max(int i, double d){return max<double>(i,d);}
double max(double d, int i){return max<double>(d,i);}
double max(double d1, double d2)
            {return max<double>(d1,d2);}

void g()
{
  max('a',1); // max(int('a'),1)
  max(2.7,4); // max(2.7,double(4))
}
```


模板函数重载示例

- 重载解析规则保证模板函数能够正确地与继承机制相互作用

```
template<class T> class B{/*...*/};
template<class T> class D : public B<T>{/*...*/};
template<class T> void f(B<T>*);

void g(B<int>* pb, D<int>* pd)
{
  f(pb); // f<int>(pb)
  f(pd); // f<int>(static_cast<B<int>*>(pd))
}
```


模板函数重载示例

- 在对模板参数的推断中未涉及到的那些函数参数，就完全按照非模板函数的参数处理。特别是可以使用普通的转换规则

```
template<class T, class C> T get_nth(C& p, int n);
            // 取容器C中的第n个元素并将其返回
struct Index{
  operator int();
};
void f(vector<int>& v, short s, Index i)
{
  int i1 = get_nth<int>(v,2); // 准确匹配
  int i2 = get_nth<int>(v,s); // 标准转换，short到int
  int i3 = get_nth<int>(v,Index); // 用户定义转换：Index到int
}
```

13.4 用模板参数描述策略

- 考虑串排序，涉及到三个概念：串(容器)、元素类型和排序算法
- 一般来说，无法将排序算法硬编码到容器中，而且也无法将排序算法硬编码到元素类型
- 通用的解决方案是以通用的方式表述排序算法，并且该算法可以用于各种类型的数据排序

```
template<class T, class C>
int compare(const String<T>& str1, const String<T>& str2)
{   // 使用C中的静态方法进行比较
  for(int i=0;i<str1.length()&&i<str2.length();i++)
    if(!C::eq(str1[i],str2[i])) return C::lt(str1[i],str2[2]) ? -1:1;
  return str1.length()-str2.length();
}
```


用模板参数描述策略示例

```
template<class T> class Cmp{ // 常规，默认比较
public:
  static int eq(T a, T b) {return a==b;}
  static int lt(T a, T b) {return a<b;}
};
class Literate{
public:
  static int eq(char a, char b){return a==b;}
  static int lt(char,char);
}; // 根据文化习惯比较瑞典人的名字

void f(String<char> swede1,
       String<char> swede2)  {
  compare<char,Cmp<char>>(swede1,swede2);
  compare<char,Literate>(swede1,swede2);
}
```

将比较操作作为模板传递有两个重要的优点：可以通过一个参数传递几个操作；没有任何运行时额外开销

13.4.1 默认模板参数

- 每次调用都需要显式地给出比较准则也会使人厌烦，为此，可以使用函数重载，也可以使用默认模板参数

```
// 函数重载
template<class T, class C> // 用C比较
int compare(const String<T>& str1, const String<T>& str2);
template<class T> // 用Cmp<T>比较
int compare(const String<T>& str1, const String<T>& str2);

template<class T, class C=Cmp<T>> // 默认模板参数
int compare(const String<T>& str1, const String<T>& str2) {
  for(int i=0;i<str1.length()&&i<str2.length();i++)
    if(!C::eq(str1[i],str2[i])) return C::lt(str1[i],str2[2]) ? -1:1;
  return str1.length()-str2.length();
}
```


默认模板参数示例

```
// 然后可以按照如下方式调用compare
void f(String<char> swede1, String<char> swede2)
{
  compare(swede1,swede2); // 用Cmp<char>
  compare<char,Literate>(swede1,swede2); // 用Literate
}
```

- 通过模板参数支持执行策略，以及在此基础上采用默认值支持最常用策略的技术在标准库中广泛使用
- 用于表示策略的模板参数常常被称为“特征”(traits)，例如：标准库string依赖于char_traits，标准算法依赖于迭代器特征类，所有标准库容器依赖于allocator

13.5 专门化

- 用户可以提供多个模板设计定义，编译器在实际应用中基于模板参数来进行选择
- 一个模板的多个可以互相替代的定义被称为用户的专门化，或者简称为用户专门化

```
template<class T> class Vector{
  T* v;
  int sz;
public:
  Vector();
  explicit Vector(int);
  T& elem(int i){return v[i];}
  T& operator[](int i);
};
// 大部分Vector中存储某种指针
// 基本原因是为了保持多态的行为方式
Vector<int> vi;
Vector<Shape*> vps;
Vector<string> vs;
Vector<char*> vpc;
Vector<Node*> vpn;
```


专门化

- 大部分C++实现的默认方式是对模板函数建立代码段的副本，这可能导致严重的代码膨胀
- 可以通过设计，使得所有指针容器都能共享同一份实现代码，这可以通过专门化做到
- 首先，定义一个void的指针的(专门化)Vector版本
- 而后这个专门化就会被用作所有指针的Vector的共同实现，因为所有特定的指针类型都可以被转化为void*

专门化示例

- template<>表示这是一个专门化，不需要模板参数，<void*>表示这个定义应该用在所有的T是void*的Vector实现中
- Vector<void*>将用于如下定义的Vector： Vector<void*> vpv;

```
template<> class Vector<void*>{
  void** p;
  void*& operator[](int i);
};
```


专门化示例

- 为了使得该专门化能够用于所有指针的Vector，且仅仅用于指针的Vector，需要一个部分专门化(partial specification)
- 名字后面的<T*>说明这个专门化将被用于每一个指针类型

```
template<class T> class Vector<T*> : private Vector<void*>{
public:
  typedef Vector<void*> Base;
  Vector() : Base() {}
  explicit Vector(int i) : Base(i) {}
  T*& elem(int i){return reinterpret_cast<T*&>(Base::elem(i));}
  T*& operator[](int i){return
          reinterpret_cast<T*&>(Base::operator[](i));}
}; // 部分专门化示例
```

专门化示例

- 上面的定义将用于每一个其模板参数能够表述为T*的Vector
- 现在可以使得所有指针的Vector共享了同一个实现，Vector<T*>实际上只是Vector<void*>的界面，完全通过派生和内联展开实现
- 实际应用表明，这种技术在抑制代码膨胀方面非常有效

```
Vector<Shape*> vps; //<T*>是<Shape*>所以T是Shape
// 注意，当运用到某个部分专门化时，模板参数是从专门化模式匹
// 配出来的，所以此处模板参数T不是Shape*，而是Shape
Vector<int**> vppi; //<T*>是<int**>所以T是int*
```


13.5.1 专门化的顺序

- 一个专门化比另一个更专门的意思是：能够与它匹配的每个实际参数表也能与另外的那个专门化匹配，但是反之则不行
- 更专门可以被理解为更特殊
- 在声明对象、指针等的时候，在做重载解析的时候，最专门化的那个版本都将比别的版本优先被选中

```
template<class T> class Vector; // 一般情况
template<class T> class Vector<T*>; // 对任何指针的专门化
template<> class Vector<void*>; // 对void*的专门化，最专门
```


13.5.2 模板函数的专门化

- 专门化对于模板函数也非常有意义，考虑前面提到过的shell排序，其中用<比较元素
- 一个更好的定义是使用模板函数less()替代<比较符号
- 可以通过less()对const char*的一个专门化做好这类比较

```
template<class T> bool less(T a, T b) {return a<b;}
// 对于T为char*时，无法正确比较
template<> bool less<const char*>(const char* a,
                const char* b) {
  return strcmp(a,b)<0;
}
```


模板函数的专门化

```
// 模板参数可以从函数的实际参数表推断，不需要显式描述
template<> bool less<>(const char* a,
                const char* b){
  return strcmp(a,b)<0;
}
// 给出了template<>前缀，后面的<>也就多余了
template<> bool less (const char* a,
                const char* b){
  return strcmp(a,b)<0;
}
```

13.6 派生和模板

- 模板和派生都是从已有类型创建新类型的机制，通常都被用于写利用各种共性的代码
- 从一个非模板类派生出一个模板类，这是为一组模板提供一个公用实现的一个方法，例如：
  template<class T> class Vector<T*> : private Vector<void*>{};
- 从模板类派生模板类也很有用，如果某个基类的成员依赖于其派生类的模板参数，那么这个基类本身也必须参数化

```
template<class T> class vector{/*...*/};
template<class T> class Vec : public vector<T>
{/*...*/};
```


13.6.1 参数化和继承

- 模板是将一个类型或者一个函数用其他的类型进行参数化，该模板对于所有参数类型的代码实现都是相同的
- 抽象类定义了一个界面，抽象类的不同实现中的许多代码可以被该类层次结构所共享，同时，利用该抽象类的大部分代码不依赖于它的具体实现
- 两者在“声明一次，然后被应用于多种不同类型”方面非常相似，可用多态性来描述之

参数化和继承

- 为了区分，将虚函数提供的东西称作运行时多态性(run-time polymorphism)，而把模板提供的多态性称为编译时多态性(compile-time polymorphism)或者参数式多态性(parametric polymorphism)
- 选择虚函数或者模板的依据：若所操作的一组对象之间不需要层次性的关系，选择模板；若编译时无法得知这些对象的类型，选择抽象类；若对运行效率要求严格，选择模板可以将各种操作方便地内联化

13.6.2 成员模板

- 一个类或者模板的成员也可以是模板
- 但是成员模板不能是virtual

```
template<class Scalar> class complex{
  Scalar re, im;
public:
  template<class T> complex(const complex<T>& c) :
                    re(c.real()), im(c.imag()){}
};
complex<float> cf(0,0);
complex<double> cd = cf; // 可以，用float到double的转换
class Quad{
  // 没有到int的转换
};
complex<Quad> cq;
complex<int> ci = cq; // 错误，无从Quad到int的转换
```

13.6.3 继承关系

- 可以将类模板理解为一种有关如何生成特定类的规范
- 因此，类模板有时也被称为类型生成器(type generator)
- 如果只考虑C++的规则，由同一个类模板生成的两个类之间并不存在任何关系

继承关系示例

```
class Shape{/*...*/};
class Circle:public Shape{/*...*/};
基于上述声明，人们有时会想到把set<Circle*>当作set<Shape*>
但这是错误的，Circle是Shape，但是Circle的集合并非Shape的集合

class Triangle:public Shape{/*...*/};
void f(set<Shape*>& s) {
  s.insert(new Triangle()); // 可以
}
void g(set<Circle*>& s) {
  f(s); // 错误，类型不匹配，s是set<Circle*>不是set<Shape*>
}
// 不存在内部的从set<Circle*>到set<Shape*>的转换，也不应该
// 有这种转换，set<Circle*>应该能保证所有成员一定是Circle
```


13.6.3.1 模板转换

- 模板生成的各个类之间没有任何默认的关系，然而对于某些模板，我们可能希望能表述这种关系

```
template<class T> class Ptr{
  T* p;
public:
  Ptr(T*);
  Ptr(const Ptr&);
  template<class T2> operator Ptr<T2>();
  // 将Ptr<T>转换到Ptr<T2>
};
void f(Ptr<Circle> pc) {
  Ptr<Shape> ps = pc; // 应该能行
  Ptr<Circle> pc2 = ps; // 应该指出错误
}
```


模板转换

- 我们希望上例中，只有当Shape是Circle的直接或者间接基类时才能够进行指针的转化
- 如下方式，当且仅当p(其类型是T*)可以作为Ptr<T2>(T2*)构造函数的参数时，这个返回语句才能编译

```
template<class T> // 模板的模板参数表
  template<class T2> // 模板成员的模板参数表
    Ptr<T>::operator Ptr<T2>(){return Ptr<T2>(p);}

void f(Ptr<Circle> pc) {
  Ptr<Shape> ps = pc; // 可以：Circle*可以转换到Shape*
  Ptr<Circle> pc2 = ps; // 错误：Shape*不能转换到Circle*
}
```

13.7 源代码组织

- 使用了模板的代码有两种明显的组织方式
  - 在使用模板的编译单位最前面包含(include)有关模板的定义
  - 只在使用模板的编译单位最前面包含(include)有关模板的声明，并且分别编译它们的模板定义
- 此外，模板函数有时是首先声明，然后使用，最后在一个编译单位里定义

两种源代码组织方式示例(1)

```
//out.c
#include <iostream>
template<class T> void out(const T& t) { std::cerr<<t;}

//user1.c
#include “out.c”

//user2.c
#include “out.c”
```

out()的定义和它所依赖的所有声明都被#include在几个不同的编译单位里，(仅)在需要时才去生成代码，这会增大编译器必须处理的信息量同时也可能使得out.c中的代码和用户代码互相影响，此外，优化对冗余定义的读入过程等工作都是编译器的责任

两种源代码组织方式示例(2)

```
// out.h
template<class T> void out(const T& t);
// out.c
#include <iostream>
#include "out.h"
export template<class T> void out(const T& t){std::cerr<<t;}

//user1.c
#include “out.h”

//user2.c
#include “out.h”
```

这种策略对模板函数的处理与对非内联函数的处理一样，out()定义被单独进行编译，在需要该定义时，实现必须负责找到那个唯一的定义

源代码组织的理想方式

- 无论作为一个单位编译，还是分成一些独立的编译单元，这些代码都应能以同样的方式工作
- 接近这种理想的方式应该是限制模板定义对其环境的依赖性，而不是在进入实例化过程时为它的定义带上尽可能多的环境信息

13.8 忠告

- [1] 用模板描述需要使用到许多参数类型上去的算法
- [2] 用模板表述容器
- [3] 为指针的容器提供专门化，以减小代码规模
- [4] 总是在专门化之前声明模板的一般形式
- [5] 在专门化的使用之前先声明它
- [6] 尽量减少模板定义对于实例化环境的依赖性
- [7] 定义你所声明的每一个专门化
- [8] 考虑一个模板是否需要有针对C风格字符串和数组的专门化

忠告

- [9] 用表述策略的对象进行参数化
- [10] 用专门化和重载为同一概念的针对不同类型的实现提供统一界面
- [11] 为简单情况提供简单界面，用重载和默认参数去表述不常见的情况
- [12] 在修改为通用模板前，在具体实例上排除程序错误
- [13] 如果模板定义需要在其他编译单位里访问，请记住写***export***
- [14] 对大模板和带有非平凡环境依赖性的模板，应采用分开编译的方式
- [15] 用模板表示转换，但要非常小心地定义这些转换

忠告

- [16] 如果需要，用***constraint***( ) 成员函数给模板的实参增加限制
- [17] 通过显式实例化减少编译和连接时间
- [18] 如果运行时的效率非常重要，那么最好用模板而不是派生类
- [19] 如果增加各种变形而又不重新编译是很重要的，最好用派生类而不是模板
- [20] 如果无法定义公共的基类，最好用模板而不是派生类
- [21] 当有兼容性约束的内部类型和结构非常重要时，最好用模板而不是派生类

C++编程(12)

唐晓晟
北京邮电大学电信工程学院

第14章 异常处理

- 错误处理
- 异常的结组
- 捕捉异常
- 资源管理
- 不是错误的异常
- 异常的描述
- 未捕捉的异常
- 异常和效率
- 处理错误的其他方式
- 标准异常
- 忠告

14.1 错误处理

- 前面8.3节，已经介绍了异常处理的基本语法、语义和使用风格等方面的问题
- 异常的概念就是为了帮助程序员更好的进行错误处理，其基本想法是：让一个函数在发现了自己无法处理的错误时抛出(throw)一个异常，希望它的直接或者间接调用者能够处理这个问题，希望处理这类问题的函数可以表明它将要捕捉(catch)这个异常

错误处理

- 异常处理机制提供了一种更规范的错误处理风格，能明确地把错误代码从正常代码中分离出来，使程序更容易读，也更容易用工具进行处理，也能简化分别写出的代码片断之间的相互关系
- 异常处理机制可以看作是编译时的类型检查和歧义性控制机制在运行时的对应物，使得设计过程更为重要，虽然需要做的工作增加，但也使得代码的可用性大为提高
- 错误处理仍然是一件很困难的工作

14.1.1 关于异常的其他观点

- C++异常处理机制的设计是为了处理错误和其他非正常的情况，特别的，希望能够支持由分别开发的组件组合而成的程序中的错误处理
- 这个机制的设计只是为了处理同步异常，例如数组范围检查和I/O错误，异步问题的处理不能由这种机制处理(有些系统提供了信号机制)
- 标准C++中没有线程或者进程的概念，但是C++的异常处理设计能够有效用于并发程序，要求程序员遵守基本的并发规则
- 抛出大量异常会使得程序结构模糊

14.2 异常的结组(Grouping of Exceptions)

- 一个异常也就是某个用于表示异常发生的类的一个对象
- 一个throw的作用就是导致堆栈的一系列回退，直到找到某个适当的catch
- 异常经常可以自然地形成一些族，意味着可以借助于继承来描述异常的结构
- 异常的类层次结构表示对于代码的健壮性可能很重要，考虑：若没有结组机制，如何处理来自数学库的所有异常

异常的结组示例

```
class Matherr{};
class Overflow:public Matherr{};
class Underflow:public Matherr{};
class Zerodivide:public Matherr{};
void f(){
    try {
    // ...
    }
    catch(Overflow){
    // 处理Overflow或者任何由Overflow派生的异常
    }
    catch(Matherr){    // 处理所有不是Overflow的Matherr
    }
}
```


异常的结组示例

```
void g()
{
    try{
        //...
    }
    catch(Overflow){/*...*/}
    catch(Underflow){/*...*/}
    catch(Zerodivice){/*...*/}
    //...
}
```

若不使用结组机制，需要列举所有异常

缺点：

1 忘记列出某个异常

2 当向数学库添加新异常时，每段试图处理所有数学异常的代码都必须修改，其结果就是，某个库一旦发布，就无法再加入任何新异常了

注意：

无论是内部的数学操作，还是基本的数学库，都没有将算术错误报告为异常(比如除0，在许多流水线机器系统结构中都是非同步的操作)

14.2.1 派生的异常

- 由于异常的类层次结构，很多情况下，异常捕捉代码块只针对基类(而不是特定的子类)异常进行捕捉
- 捕捉和命名异常的语义等同于函数接受参数语义

派生的异常示例

```
class matherr{
  //...
  virtual void debug_print() const {
     ceer << "Math error";
  }
};

class Int_overflow:public Matherr{
  const char* op;
  int a1, a2;
public:
  Int_overflow(const char* p,int a,int b){
     op = p; a1 = a; a2 = b;    }
  virtual void debug_print() const {
     cerr << op << a1 << a2;    }
};
```

```
void f()
{
  try { g(); }
  catch(Matherr m){/*...*/ }
}
```

即使g()抛出的实际上是一个Int_overflow，这里m仍然一个Matherr对象，也即：Int_overflow所携带的附加信息将是不可访问的

若需要访问附加信息，catch后面括号中应该声明为Matherr的引用或者指针

14.2.2 多个异常的组合

- 并不是每组异常都具有树形结构，也常有一个异常同属于两个组的情况，如下，这样的一个Netfile_err异常能够被处理网络的异常捕获，也能被处理文件系统异常的函数捕获
- 对于错误处理的这种非层次性结构的组织形式很重要，否则用户或许根本意识不到自己捕获了某个异常

```
class Netfile_err:public Network_err, public File_system_err
{/*...*/};
```


14.3 捕捉异常

```
void f() {
  try{
     throw E();
  }
  catch(H){
     //何时运行到这里？
  }
}
```

[1] 如果H是和E相同的类型
[2] 如果H是E的无歧义的公用基类
[3] 如果H和E是指针类型，且[1]或[2]对它们所引用的类型成立
[4] 如果H是引用类型，且[1]或[2]对H所引用的类型成立

此外，还可以给用于捕捉异常的类型加上const，限制我们不能修改捕捉到的异常

从原则上说，异常在抛出时被复制，异常处理器得到的只是原始异常的一个副本(甚至已经被复制多次)，因此，我们不能抛出一个不允许复制的异常

14.3.1 重新抛出

- 在捕捉到了一个异常之后，很多情况下，处理器并不能完成对这个错误的全部处理，典型的方式是：该处理器做完局部能够做的事情，然后再一次抛出这个异常，这将使得错误恢复动作分布在几个处理器里
- 重新抛出采用一个不带运算对象的throw语句
- 重新抛出的异常就是当初捕捉到的那个异常，而不仅仅是异常处理器能访问到的那个部分

重新抛出示例

```
void h() {
   try { // 可能抛出Matherr的代码
   }
   catch(Matherr){
      if(can_handle_it_completed){
         //处理Matherr
         return;
      }
      else{
         //完成这里能做的事情
         throw;
      }
   }
}
```


14.3.2 捕捉所有异常

- 对于函数，省略号"..."表示"任何参数"，同样，catch(...)表示要"捕捉所有异常"

```
void m()
{
   try {
      // 一些代码
   }
   catch(...) {//处理所有异常
      // 清理工作
      throw;
   }
}
```


14.3.2.1 处理器的顺序

- 派生异常可能被多于一个异常类型的处理器捕捉，因此，在写try语句时处理器的排列顺序非常重要

```
void f() {
   try{
      //...
   }
   catch(std::ios_base::failure){
      //处理任何流io错误
   }
   catch(std::exception& e){
      //处理所有标准库异常
   }
   catch(...){
      //处理任何其他异常
   }
}
```

处理器的顺序

```
void f() {
   try{
      //...
   }
   catch(...){
      //处理任何其他异常
   }
   catch(std::exception& e){
      //处理所有标准库异常
   }
   catch(std::bad_cast){
      //处理dynamic_cast失败
   }
}
```

如果次序安排的不好，有些异常处理代码将永远不会运行

左例中，bad_cast部分将永远不被执行，即使将catch(...)部分去掉也不行

14.4 资源管理

- 当函数申请了某种资源(打开文件或者分配空间)时，一个很重要的问题就是需要正确释放这些资源
- 但是如果程序在运行过程中产生了异常，就会导致程序的流程不是所希望那样，此时需特别注意资源问题
- 异常产生时，实现会“向上穿过堆栈”去为某个异常查找对应处理器，叫做“堆栈回退”，此时，会对所有局部对象调用析构函数

资源管理示例

```
void use_file(const char* fn)
{
   FILE* f = fopen(fn, "r");
   // 使用f资源
   fclose(f);
}

// 若在使用f资源的过程中产生了异常，则fclose(f)将不会被调用，资源释放失败
```

```
void use_file(const char* fn) {
   FILE* f = open(fn, "r");
   try {
      // 使用f
   }
   catch(...){
      fclose(f);
      throw;
   }
   fclose(f);
}

// 一种解决方法，但是非常罗嗦
```


资源管理示例

考虑问题的一般形式：

```
void acquire() {
   // 申请资源1
   // ...
   // 申请资源n
   // 使用资源
   // 释放资源n
   // ...
   // 释放资源1
}
```

// 一般是按照申请的相反顺序来释放资源，这和构造和析构对象的顺序很相似

定义一个类File_ptr

```
class File_ptr{
   FILE* p;
public:
   File_ptr(const char* n,
        const char* a){p=fopen(n,a);}
   File_ptr(FILE* pp) { p=pp; }
   // 适当的复制机制
   ~File_ptr() { if(p) fclose(p); }
   operator FILE*() { return p; }
};
```

这样处理，主程序只需要一句话

```
File_ptr f(fn,"r");
```

析构函数总会被调用，资源总会被释放

14.4.1 构造函数和析构函数的使用

- 利用局部对象管理资源的技术通常被称为“资源申请即初始化(resource acquisition is initialization)”
- 只有在一个对象的构造函数执行完毕时，这个对象才被看作已经建立起来了，**只有在此之后**，异常产生时的堆栈回退才为该对象调用析构函数
- 一个由子对象组成的对象的构造将一直持续到它所有的子对象都完成了构造

示例

```
class X{
   File_ptr aa;
   Lock_ptr bb;
public:
   X(const char* x,
     const char* y)
      :aa(x,"rw"), // 申请'x'
       bb(y)// 申请y
   {}
};
```

考虑X，构造函数需要两种资源，文件x和锁y，这些请求都可能失败并抛出异常，若普通设计，程序员需要写大量代码

为此，可以使用两个类(子对象)来实现该类，这使得程序员根本不必去考虑任何异常情况

若在创建了aa但还没有bb的情况下出现了异常，aa的析构函数会被调用，bb的则不会

示例

```
class Y{
   int* p;
   void init();
public:
   Y(int s) {
      p=new int(s);
      init();
   }
   ~Y(){delete[] p;}
   //...
};

// 一段有问题的代码，可能
导致存储流失
```

```
class Z{
   vector<int> p;
   void init();
public:
   Z(int s) : p(s) { init(); }
   // ...
};


// 改进后的代码
```


14.4.2 auto_ptr

- 标准库提供的auto_ptr支持“资源申请即初始化”的技术
- 下例中，Rectangle和由pc所指向的Circle都将被删除，无论是否有异常被抛出

```
void f(Point p1, Point p2, auto_ptr<Circle> pc, Shape* pb)
{
   auto_ptr<Shape> p(new Rectangle(p1,p2)); // p指向一个矩形
   auto_ptr<Shape> pbox(pb);

   p->rotate(45); // auto_ptr<Shape>的使用与Shape*一样
   // ...
   if(in_a_mess) throw Mess();
} // 记住，退出时需要删除pb
```

auto_ptr

- 为了获得这种所有权语义(ownership semantics)，也常被称为破坏性复制语义(destructive copy semantic)，auto_ptr具有与常规指针很不一样的复制语义：在将一个auto_ptr复制给另一个之后，原来的auto_ptr将不再指向任何东西
- 因为复制auto_ptr将导致对它自身的修改，所以const auto_ptr就不能复制

auto_ptr的声明

```
// auto_ptr在<memory>中声明
template<class X> class std::auto_ptr{
   template<class Y> struct auto_ptr_ref{/*...*/}; // 协助类
   X* ptr;
public:
   typedef X element_type;
   explicit auto_ptr(X* p=0) throw(){ptr=p;}
      // throw()表示不抛出异常
   ~auto_ptr() throw() { delete ptr; }
   //注意，复制构造函数和赋值都用非const参数
   auto_ptr(auto_ptr& a) throw(); // 复制，而后a.ptr=0
   template<class Y> auto_ptr(auto_ptr<Y>& a) throw();
   // 复制，而后a.ptr=0
   auto_ptr& operator=(auto_ptr& a) throw(); // 复制，而后a.ptr=0
   template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw();
   // 复制，而后a.ptr=0
```


auto_ptr的声明

```
   // 运算符重载
   X& operator*() const throw() { return *ptr; }
   X* operator->() const throw() { return ptr; }
   X* get() const throw() { return ptr; } // 提取指针
   X* release() throw() { X* t=ptr; ptr=0; return t; }
   // 放弃所有权
   void reset(X* p=0) throw() { if(p!=ptr) {delete ptr; ptr=p; }}

   auto_ptr(auto_ptr_ref<X>) throw(); // 从auto_ptr_ref复制
   template<class Y> operator auto_ptr_ref<Y>() throw();
   // 复制到auto_ptr_ref
   template<class Y> operator auto_ptr<Y>() throw();
   // 从auto_ptr的破坏性复制
};
```


auto_ptr说明

- auto_ptr_ref的用途就是为普通auto_ptr实现破坏性复制语义，这使得const auto_ptr不可能复制
- 如果指针D*能转换到B*，那么这里的模板构造函数和模板赋值都能将auto_ptr<D>转换到auto_ptr<B>

auto_ptr示例

```
void g(Circle* pc)
{
    auto_ptr<Circle> p2(pc); // 现在p2负责删除
    auto_ptr<Circle> p3(p2); // 现在p3负责删除(且p2不再负责)
    p2->m = 7; // 程序错误：p2.get() == 0
    Shape* ps = p3.get(); // 从auto_ptr抽取指针
    auto_ptr<Shape> aps(p3); // 转移所有权，并转换类型
    auto_ptr<Circle> p4(pc); // 程序错误：现在p4也负责删除
} // p4(多个auto_ptr拥有同一个对象)的效果是无定义的
// 最可能的情况是被删除两次

vector<auto_ptr<Shape>>& v; // 危险，在容器里使用auto_ptr
sort(v.begin(), v.end()); // 该排序可能造成v混乱
// auto_ptr的破坏性复制语义意味着它不满足标准容器或标准算法
```


14.4.3 告诫

- 并不是所有的程序都需要有从所有破坏中恢复的能力
- 并不是所有资源都重要的需要使用“资源申请即初始化”技术、auto_ptr和catch(...)等去保护它们
- 当然，标准库的设计需要好好设计资源的处理问题

14.4.4 异常和new

- 若X的构造函数抛出异常，会如何？
- 标准内存申请p1，p2不会产生存储流失
- 若采用放置语法进行非标准分配，则需要进行非标准的释放

```
void f(Arena& a, X* buffer)
{
    X* p1 = new X;
    X* p2 = new X[10];

    X* p3 = new(&buffer[10])X;
    // 将X放入buffer(无需分配存储)
    X* p4 = new(&buffer[11])X[10];

    X* p5 = new(a)X;
    // 在Arena a中分配存储(从a中释放)
    X* p6 = new(a)X[10];
}
```


14.4.5 资源耗尽

- 讨论资源申请失败时应该做什么
- 两种风格
  - 唤醒：请求某个调用程序纠正问题，而后继续执行
  - 终止：结束当前计算并返回某个调用程序
- C++中，唤醒模型由函数调用机制支持，而终止模型由异常处理机制支持

示例

```
void* operator new(size_t size) // 某个new的实现
{
   for(;;){
      if(void* p = malloc(size)) return p; // 申请内存
      if(_new_handler == 0) throw bad_alloc(); // 无处理器
      _new_handler(); // 寻求帮助，如果也找不到内存
   }                          // 或许会抛出一个异常，否则无穷循环
}
```

// _new_handler()是当new无法申请到内存时，系统内定的处理函数，当然，也可以用自己的函数替代之

// set_new_handler(&my_new_handler);

每个C++实现都要求保留足够内存，在存储耗尽的情况下还能抛出bad_alloc，当然，全部耗尽的情形还是可能发生的

示例

```
void my_new_handler();
void f()
{
   void(*oldnh)() = set_new_handler(&my_new_handler);
   try{
      // ...
   }
   catch(bad_alloc){
      // ...
   }
   catch(...) {
      set_new_handler(oldnh); // 重置处理器
      throw; // 重新抛出
   }
   set_new_handler(oldnh); // 重置处理器
} // 也可以使用"资源申请即初始化"技术避免使用catch语句
```


14.4.6 构造函数里的异常

- 异常也为从构造函数里报告出错的问题提供了一个解决方案，构造函数无法返回一个独立的值供调用程序检查，传统的解决方法有
  - 返回一个处于错误状态的对象，相信用户有办法检查其状态
  - 设置一个非局部变量，指出创建失败，相信用户能去检查这个变量
  - 在构造函数中不做初始化，依靠用户在第一次使用对象之前调用某个初始化函数
  - 将对象标记为"未初始化的"，让对这个对象调用的第一个成员函数去完成实际的初始化工作，并让这个函数在初始化失败时报告错误

构造函数里的异常

- 异常处理机制使构造失败的信息能从构造函数内部传出来

```
class Vector{
public:
   class Size{};
   enum {max = 32000};
   Vector(int sz) {
      if(sz<0 || max<sz)
         throw Size();
      // ...
   }
};
```

```
Vector* f(int i) {
   try {
      Vector* p = new Vector(i);
      // ...
      return p;
   }
   catch(Vector::Size){
      // 处理错误
   }
}
```

14.6.1 异常和成员初始化

- 当成员的初始式抛出异常，该异常将传到调用这个成员的类的构造函数的位置
- 构造函数可以通过将完整的函数体包在一个try块里，自己设法捕捉这种异常

```
class X{
   Vector v;
public:
   X(int);
};
```

```
X::X(int s)
try
    :v(s) { // 用s初始化v
}
catch(Vector::Size){
   // ...
}
```


14.4.6.2 异常和复制

- 与其他构造函数一样，复制构造函数也可以通过抛出异常的方式发出一个失败信号
- 在这种情况下，实际上并没有创建对象
- 在抛出异常之前，复制构造函数就需要释放它已经申请到的所有资源
- 复制赋值函数与复制构造函数类似

14.4.7 析构函数里的异常

- 从异常处理的角度看，析构函数可能在两种情况下被调用
  - 正常调用
  - 在异常处理中被调用
- 对于后一种情况，绝不能让析构函数里抛出异常，如果抛出了，就将被认为是异常处理机制的一次失败，并调用std::terminate()，这么做的原因是没有一种普遍有效的方式来确定应该偏向处理那个异常(导致析构函数被调用的异常和析构函数抛出的异常)

析构函数里的异常

- 如果某个析构函数要调用一个可能抛出异常的函数，它可以保护自己
- 如果有某个异常已经被抛出，但是尚未被捕捉，标准库函数uncaught_exception()就会返回true，析构函数能够根据此信息来进行不同的动作

```
X::~X()
try {
   f();
}
catch(...){
   //
}
```

14.5 不是错误的异常

- 有些异常是预期内的，就算被捕捉到也不会对程序行为产生不良影响

```
void f(Queue<X>& q)
try {
    for(;;) {
        X m = q.get();
        // ...
    }
}
catch(Queue<X>::Empty) {
    return;
}
```

换一种看法，我们也可以认为异常处理机制只不过是另外一种控制结构

需要注意的是，异常处理是一种结构化更差的机制，通常在实际抛出时效率也更低

注意不要滥用

示例

- 有时候，采用异常作为返回方式是一种很优雅的技术
- 任何有助于维持一种关于一个错误是什么以及它被如何处理的清晰模型，都是非常宝贵的

```
void fnd(Tree* p, const string& s)
{
    if(s==p->str) throw p;
    if(p->left) fnd(p->left,s);
    if(p->right) fnd(p->right,s);
}
```

```
Tree* find(Tree* p,
           const string& s)
{
    try{ fnd(p,s); }
    catch(Tree* q){
        return q;
    }
    return 0;
}
```


14.6 异常的描述

- 将可能抛出的异常集合作为函数声明的一部分是有价值的，它可以有效的为调用者提供一种保证

```
void f(int a) throw(x2,x3);
// 保证f函数只会抛出异常x2、x3以及从这些类型派生的异常
```

- 若函数不遵循自己作出的保证，这个企图将会被转换为一个对std::unexpected()的调用，其默认意义是std::terminate()，通常将被转为对abort()的调用

异常的描述

```
void f() throw(x2,x3)
{ /*...*/ }
```

等价于

```
void f()
try
 {
    // ...
 }
catch(x2){throw;}
catch(x3){throw;}
catch(...){
    std::unexpected();
}
```

这里最重要的优点在于函数的声明属于界面，而界面是函数的调用者可以看到的

另外，函数的定义一般不是可用的东西，即使有源码，也不希望经常去查看

最后，带有异常描述的函数也比手工写出的等价版本更短更清晰

若函数不抛出任何异常，可以声明为：

```
int g() throw();
```

14.6.1 对异常描述的检查

- 编译时不可能捕捉到所有违反界面描述的情况，但是编译时可以完成大部分工作

```
int f() throw(std::bad_alloc);
int f() // 错误：缺少异常描述
{/*…*/}

void f() throw(X);
void (*pf1)() throw(X,Y) = &f; // ok
void (*pf2)() throw() = &f;  // 错误：f()不如pf2那么受限

void g(); // 可能抛出任何异常
void (*pf3)() throw(X) = &g; //错误：g()不如pf3那么受限
```


对异常描述的检查

```
class B{
public:
   virtual void f();
   virtual void g() throw(X,Y);
   virtual void h() throw(X);
};
```

异常描述不是函数类型的一部分，typedef不能带有异常描述

```
typedef void(*PF)()
throw(X);
// Error
```

```
class D : public B{
public:
   void f() throw(X); // ok
   void g() throw(X); // 可以，更受限
   void h() throw(X,Y); // 错误
};
```


14.6.2 未预期的异常

- 异常描述可能导致调用unexpected()，这是设计者所不希望看到的
- 一个设计良好的子系统Y常常将它的所有异常都从一个类Yerr派生出来

```
class Some_Yerr:public Yerr{/*…*/};
void f() throw(Xerr, Yerr, exception);
// 能把所有的Yerr传给它的调用者，这样f()中的Yerr都不会触发unexpected()
```

- 标注库抛出的所有异常都是从exception派生出来的

14.6.3 异常的映射

- 有时候，希望能将unexpected()的行为方式修改为其他的能够接受的方式，而不是简单的终止程序
- 最简单方式就是将标准库异常std::bad_exception加入某个异常描述，此时，unexpected()将直接抛出一个bad_exception，而不是去调用某个试图应付困难的函数

```
class X{}; class Y{};
void f() throw(X, std::bad_exception)
{
   throw Y();
}
```

虽然没有调用terminate()，bad_exception异常仍然是很粗鲁的，特别是丢失了引起问题的那个异常的所有信息

14.6.3.1 异常的用户映射

- 未预期异常的响应由_unexpected_handler决定，它又是通过<exception>中的std::set_unexpected()设置的
- 重新定义unexpected()函数时，我们先使用“资源申请即初始化”技术，为unexpected()函数定义一个类STC

```
typedef void(*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler);
class STC{
    unexpected_handler old;
public:
    STC(unexpected_handler f){old=set_unexpected(f);}
    ~STC(){set_unexpected(old);}
};
```


异常的用户映射示例

假设void g() throw(Yerr);是可能引发调用unexpected()的函数，假设引发的原因是现在应用到网络环境中…

定义一个函数，使它具有所希望的unexpected()的意义

```
class Yunexpected:public Yerr{};
void throwY() throw(Yunexpected) { throw Yunexpected();}
// 在用作unexpected()函数时，throwY将所有未预期的异常都映射为Yunexpected

void networked_g() throw(Yerr){
    //STC xx(&throwY); // 并没有违反异常描述，
                       // 因为Yunexpected是从Yerr派生出来的
    g();
} // 现在unexpected()抛出Yunexpected，但是丢失了异常的类型
```


14.6.3.2 找回异常的类型

```
class Yunexpected:public Yerr{
public:
    Network_exception* pe;
    Yunexpected(Network_expection* p):pe(p?p->clone():0){}
    ~Yunexpected(){delete pe;}
}; // clone用于在自由存储区为异常建立一个副本，异常处理完毕仍然存在
void throwY() throw(Yunexpected) {
    try{
        throw; //重新抛出将立即被捕捉
    }
    catch(Network_exception& p){
        throw Yunexpected(&p);
    }
    catch(...){
        throw Yunexpected(0);
    }}
```

throwY由unexpected()调用，而unexpected()是由一个catch(…)处理器调用的，所以现在一定存在着某个能够重新抛出的异常

实际的异常是Network_exception

14.7 未捕捉的异常

- 如果抛出的一个异常未被捕捉，那么就会调用函数std::terminate()
- 未捕捉的异常由_uncaught_handler确定，它由<exception>里的std::set_terminate()设置
- 提出terminate()的原因是，少数情况下，必须能用不那么精细的错误处理技术终止异常处理过程
- 如果希望在发生未捕捉异常时保证进行清理工作，可以为main()添加一个捕捉一切的处理器(但是仍然无法捕捉在全局变量的构造和析构中抛出的异常)

14.8 异常和效率

- 原则上说，当没有异常时，异常处理的实现在运行时没有任何开销，此外，也可以做到使抛出异常并不总比调用函数的开销更大
- 尽管异常处理会影响效率，但是，那些能够代替异常的东西也不是没有代价的
- 需要系统化地处理错误的地方，最好使用异常处理机制
- 异常描述可能非常有助于改进所生成的代码

14.9 处理错误的其他方式

- 使用异常处理方式，需要一种整体策略
- 异常处理策略最好使在设计的初始阶段予以考虑，而且必须是简单而明确的
- 成功的容错系统只能是多层次的，不可能有某种单一机制能够处理所有错误
- 将系统划分为一些独立的子系统，使它们或是成功结束，或是以某种定义良好的方式失败，是至关重要的

14.10 标准异常

- 标准异常、抛出它们的函数、运算符和一般性功能

标准异常(由语言抛出)			
名字	抛出	参考	头文件
bad_alloc	new	6.2.6.2 19.4.5	<new>
bad_cast	dynamic_cast	15.4.1.1	<typeinfo>
bad_typeid	typeid	15.4.4	<typeinfo>
bad_exception	异常描述	14.6.3	<exception

标准异常

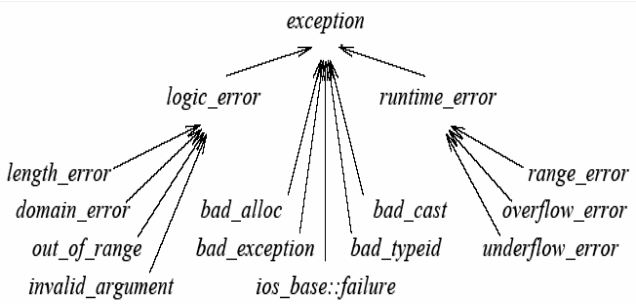
标准异常(由标准库抛出)			
名字	抛出	参考	头文件
out_of_range	at() bitset<>::operator[]()	3.7.2 16.3.3 20.3.3	<stdexcept> <stdexcept>
invalid_argument	按位设置构造函数	17.5.3.1	<stdexcept>
overflow_error	bitset<>::to_ulong()	17.5.3.3	<stdexcept>
ios_base:failure	ios_base::clear()	21.3.6	<ios>

标准异常

- 所有库异常都是同一个类层次结构中的部分，其根是标准库异常类exception

```
class exception{
public:
   exception() throw();
   exception(const exception&) throw();
   exception& operator=(const exception&) throw();
   virtual ~exception() throw();
   virtual const char* what() const throw();
private:
   // ...
};
```


标准异常类层次结构图



14.11 忠告

- [1] 用异常做错误处理
- [2] 当更局部的控制机构足以应付时，不要使用异常。
- [3] 采用“资源申请即初始化”技术去管理资源
- [4] 并不是每个程序都要求具有异常时的安全性
- [5] 采用“资源申请即初始化”技术和异常处理器去维持不变式
- [6] 尽量少用try块，用“资源申请即初始化”技术，而不是显式的处理器代码
- [7] 并不是每个函数都需要处理每个可能的错误
- [8] 在构造函数里通过抛出异常指明出现失败

忠告

- [9] 在从赋值中抛出异常之前，使操作对象处于合法状态
- [10] 避免从析构函数里抛出异常
- [11] 让***main***( ) 捕捉并报告所有的异常
- [12] 使正常处理代码和错误处理代码相互分离
- [13] 在构造函数里抛出异常之前，应保证释放在此构造函数里申请的所有资源
- [14] 使资源管理具有层次性
- [15] 对于主要界面使用异常描述

忠告

- [16] 当心通过***new***分配的内存在发生异常时没有释放，并由此而导致存储的流失
- [17] 如果一函数可能抛出某个异常，就应假定它一定会抛出这个异常
- [18] 不要假定所有异常都是由***exception***类派生出来的
- [19] 库不应该单方面终止程序。相反，应该抛出异常，让调用者去做决定
- [20] 库不应该生成面向最终用户的错误信息。相反，它应该抛出异常，让调用者去做决定
- [21] 在设计的前期开发出一种错误处理策略

C++编程(13)

唐晓晟
北京邮电大学电信工程学院

第15章 类层次结构

- 引言和概述
- 多重继承
- 访问控制
- 运行时类型信息
- 指向成员的指针
- 自由存储
- 忠告

15.1 引言和概述

- 本章讨论派生类和虚函数如何与其他语言功能相互作用，例如访问控制、名字查找、自由存储管理、构造函数、指针和类型转换等

15.2 多重继承

- 一个类可以有多个直接基类，采用多个直接基类的情况通常称为多重继承
- 假设Satellite从Task和Displayed多重继承，那么在实际应用中，就可以将一个Satellite传递给那些期望Task或者Displayed的函数

15.2.1 歧义性解析

```
class Task{
   //...
   virtual debug_info* get_debug();
};
class Displayed {
   //...
   virtual debug_info* get_debug();
};
void f(Satellite* sp){
   debug_info* dip = sp->get_debug();
   // 错误，歧义
   dip = sp->Task::get_debug();
   dip=sp->Displayed::get_debug();
}
```

通过左边这种明确写出的方式消除歧义性显得比较混乱，最好是通过在派生类里定义新函数的方法消除这类问题

例如：在Satellite中定义get_debug()函数即可消除此类问题

15.2.2 继承和使用说明

- 重载解析的使用不会跨越不同类的作用域(7.4)，特别地，来自不同基类的函数之间的歧义性不能基于参数类型完成解析

```
class Task{ void debug(double p); };
class Displayed{ void debug(int v); };
class Satellite: public Task, public Displayed{/*...*/}
// 注意：Satellite中没有定义debug()
void g(Satellite* p){
   p->debug(1); // 歧义，错误
   p->Task::debug(1); // ok
   p->Displayed::debug(1); // ok
}
```


继承和使用说明示例

- 假如在不同基类中使用同样名字是一种精心筹划的设计决策，可以如下解决

```
class A{ public: int f(int); char f(char); };
class B{ public: double f(double); };
class AB:public A, public B{
public:
   using A::f;
   using B::f;
   char f(char);
   // 遮蔽A::f(char)
   AB f(AB);
};

void g(AB& ab){
   ab.f(1); // A::f(int)
   ab.f('a');// AB::f(char)
   ab.f(2.0);// B::f(double)
   ab.f(ab); // AB::f(AB)
}
```


15.2.3 重复的基类

- 一个类两次作为基类是可能的，例如：Task和Displayed都是从Link派生而来，在一个Satellite中就会出现两个Link
- 当引用Link中的元素时，注意必须加以限定

```
struct Link { Link *next; };
void mess_with_links(Satellite* p){
   p->next = 0; // 错误，歧义性
   p->Link::next = 0; // 错误，歧义性
   p->Task::next = 0; // ok
   p->Displayed::next = 0; // ok
}
```

15.2.3.1 覆盖

- 重复基类中的虚函数可以由派生类里的函数覆盖

```
class Storable{
   public: virtual void write() = 0; virtual void read()=0;
   virtual const char* get_file() = 0; };
class Transimitter: public Storable{ public: void write();};
class Receiver:public Storable{ public: void write(); };
class Radio:public Transimitter, public Receiver{
   void write(); };
void Radio::write(){
   Transmitter::write();
   Receiver::write();
}// 一种典型应用方式，覆盖函数首先调用基类中的版本，然后做
 // 有关派生类的特殊工作
```


15.2.4 虚基类

```
class Storable{
public:
   Storable(const char* s);
   virtual void read() = 0;
   virtual void write() = 0;
   virtual ~Storable();
private:
   const char* store;
   Storable(const Storable&);
   Storable& operator=(const Storable&);
};
class Transmitter:public virtual Storable{
   public: void write();  };
class Receiver:public virtual Storable{
   public: void write();  };
```

```
class Radio:
   public Transmitter,
   public Receiver
{
   public:
   void write();
};
```

Storable稍加修改，继承方式需要改变，以便能处理一个对象的被存储了多个副本的共享问题
解决方法是使用虚基类

15.2.4.1 用虚基类的程序设计

- 在为存在虚基类的类定义函数时，一般来说，程序员并不知道这个基类是与其他派生类共享的
- 在实现某种服务，其中要求调用基类的某个函数恰好一次时，这种情况就可能引起问题

用虚基类的程序设计示例

```
class A{ /*...*/ }; // 无构造函数
class B{ public: B(); }; // 默认构造函数
class C{ public: C(int); }; // 无默认构造函数
class D:virtual public A, virtual public B, virtual public C{
   D(){/*...*/} // 错误，没有调用C的构造函数
   D(int i):C(i){/*...*/}; // ok
};
```

语言保证对于一个虚基类的构造函数将调用恰好一次
虚基类的构造函数将从最终派生类的构造函数里调用，而且要在派生类的构造函数之前调用

用虚基类的程序设计示例

```
class Window{     // 拥有基本功能
    virtual void draw(); };
class Window_with_border:
    public virtual Window{
    // 边框功能
    void own_draw(); // 显示边框
    void draw();
};
class Window_with_menu:
    public virtual Window{
    // 菜单功能
    void own_draw(); // 显示菜单
    void draw();
};// own_draw()不必是虚函数，
它只是被draw()调用，而draw()知
道调用哪一个
```

```
class Clock:
    public Window_with_border,
    public Window_with_menu
{
    // 时钟功能
    void own_draw();
    void draw();
};
```


用虚基类的程序设计示例

```
void Window_with_border::draw(){
    Window::draw();
    own_draw();
}

void Window_with_menu::draw(){
    Window::draw();
    own_draw();
}

void Clock::draw(){
    Window::draw();
    Window_with_border::own_draw();
    Window_with_menu::own_draw();
    own_draw();
}
```

借助own_draw()函数写出draw()函数，可以使任何对draw()的调用者都只能让Window::draw()被调用一次，做到这些并不依赖于到底对哪种Window调用draw()

15.2.5 使用多重继承

- 多重继承的最简单最明显的应用，就是利用它将两个原本不相干的类“粘合”起来，作为第三个类的实现的一部分
- 回顾以前讲到过的BB_ival_slider的(多重继承)实现方式，其一个基类作为公用的抽象基类，提供界面，另外一个则是受保护的具体类，提供细节
- 多重继承也使兄弟类之间能够共享信息，而又不会在程序里引进对同一基类的依赖性，通常被称为“钻石型继承diamond-shaped inheritance”，如刚才提到的Radio和Clock

使用多重继承

- 如果虚基类或者由虚基类直接派生的类是抽象类，钻石型继承将特别容易控制
- 将此思想进一步推广，得到的逻辑结论就是，组成应用的界面的抽象类的所有派生都**应该**是虚的，这是最符合逻辑、最具有一般性、也最灵活的解决方案
- 但是，由于历史的原因，作者没有这么设计

15.2.5.1 覆盖虚基类的函数

- 派生类可以覆盖其直接虚基类或者间接虚基类中的虚函数，特别地，两个不同的类也可能覆盖了来自虚基类的不同虚函数
- 不同的派生类可能覆盖其虚基类中的同一个函数，这当且仅当某个覆盖类是从覆盖此函数的类派生时，才可以这么做
- 如果两个类覆盖了同一个虚基类中的函数，但是它们又互不覆盖，这个类层次结构就是错误的
- 一个为虚基类提供部分实现(但是并非全部实现)的类通常被称为“混入类”(mixin.)

15.3 访问控制

- 类中的一个成员可以是private,protected或者public
- private:它的名字只能由其声明所在类的成员函数和友元使用
- protected:它的名字只能由其声明所在类的成员函数和友元，以及由该类的派生类的成员函数和友元使用
- public:它的名字可以由任何函数使用

15.3.1 保护成员

- 派生类只能访问它这种类型对象基类中的保护成员，这样能防止由于一个派生类破坏了属于另一个派生类的数据而产生的微妙错误

```
class Buffer{
    protected:    char a[128];
};
class Linked_buffer:public Buffer{/*...*/};
class Cyclic_buffer:public Buffer{
    void f(Linked_buffer* p){
        a[0] = 0; // 可以，自己的保护成员
        p->a[0] = 0; //  不可以，其他类型的保护成员
    }
};
```


15.3.1.1 保护成员的使用

- 简单的public/private模型就能很好的满足具体类型概念的需要，派生的引入使得这种模型无法迎合派生类的特殊需要
- 声明一些protected数据成员通常都是设计错误，将位于一个公共类的大量数据提供给派生类使用，将使这些数据容易遭到破坏
- private通常是更好的选择，也是默认情况
- 以上所有批评对于protected函数都不重要，protected是描述供派生类使用的操作的极好方式

15.3.2 对基类的访问

- 象成员一样，基类也可以是private, protected或者public

```
class X:public B{};
class Y:protected B{};
class Z:private B{};
```

- public派生使得派生类成为基类的一个子类型，protected在经常需要进一步派生的类层次结构中非常有用，private则提供更强的限定

对基类的访问

- 考虑从基类B派生出的类D
- 如果B是private基类，那么它的public和protected成员只能由D的成员函数和友元访问，只有D的成员和友元能将D*转换到B*
- 如果B是protected基类，那么它的public和protected成员只能由D的成员函数和友元、以及由D派生出的类的成员函数和友元访问。只有D的成员和友元以及由D派生出的类的成员和友元能将D*转换到B*
- 如果B是public基类，那么它的public成员可以由任何函数使用，此外，它的protected成员能由D的成员函数和友元，以及由D派生出的类的成员函数和友元访问，任何函数都能将D*转换到B*

15.3.2.1 多重继承和访问控制

- 在一个派生类的继承层次中，如果一个名字或者基类可以从多条路径到达，那么若有一条路径使它能够访问，它就是可访问的

```
struct B{
    int m;
    static int sm;
};

class D1:public virtual B{/*...*/};
class D2:public virtual B{/*...*/};
class DD:public D1, private D2{/*...*/};

DD* pd = new DD;
B* pb = pd;      // 可以：通过D1访问
int i1 = pd->m;// 可以：通过D1访问
```


多重继承和访问控制示例

- 如果一个实体可以通过多条途径到达，我们还是可能无歧义地引用它

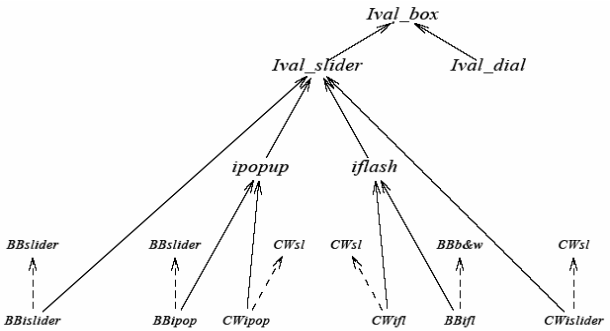
```
struct B{
    int m;
    static int sm;
};

class X1:public B{/*...*/};
class X2:public B{/*...*/};
class XX:public X1,public X2{/*...*/};

XX* pxx = new XX;
int i1 = pxx->m; // 错误：歧义
int i2 = pxx->sm; // 可以：在XX里只有B::sm
```

15.3.2.2 使用声明和访问控制

- 不能通过使用声明(using)取得对更多信息的访问权，using语句只是一种能使信息的使用更方便的机制

```
class B{
    private: int a;
    protected: int b;
    public: int c;
};
class D:public B{
public:
    using B::a; // 错误：B::a为private
    using B::b; // 可以：使B::b在整个D中可以直接使用
};
```


15.4 运行时类型信息



运行时类型信息

- 考虑以前设计过的Ival_box类层次结构
- 一般使用方式是将Ival_box的派生类生成的对象指针传递给一个需要Ival_box类型指针的控制显示屏系统进行绘制，然后，某些活动后，系统将对象送回应用程序，但是，用户系统对Ival_box一无所知，这当然是必要的，也是正确的，但是，我们确实会遗失掉送给系统而后又被送还我们的对象的类型

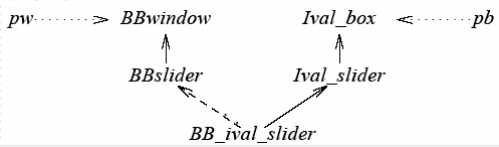
运行时类型信息示例

- 要寻回一个对象遗失的类型，需要能够以某种方式去向对象询问其类型，dynamic_cast做的就是这类事情
- 注意，在下例中，pw的实际类型应该是某种特殊的Ival_box，比如说Ival_slider或者BBslider等

```
void my_event_handler(BBWindow* pw){
    if(Ival_box* pb = dynamic_cast<Ival_box*>(pw)){
        pb->do_something();
    } // 若pw指向一个Ival_box，则do_something()
    else{
        //...
    }
}
```

运行时类型信息示例

- 动作pb = dynamic_cast<Ival_box*>(pw)可以表示为



- 出自pw和pb的箭头表示的是被传递的对象指针，其他箭头表示被传递对象中的不同部分之间的继承关系
- 在运行时对类型信息的使用一般称为运行时类型信息(RTTI RunTime Type Information)

运行时类型信息示例

- 从基类到派生类的强制转换通常被称为向下强制(downcast)，因为一般画继承树时，根在上面
- 从派生类向基类的强制转换称为向上强制(upcast)
- 从一个基类向其兄弟类的强制，例如上图中从BBwindow到Ival_box，称为交叉强制(crosscast)

15.4.1 dynamic_cast

- dynamic_cast与以前讲到过的static_cast类型，都是有两个参数

```
class BB_ival_slider:public Ival_slider, protected BBslider{};
void f(BB_ival_slider* p){
    Ival_slider* pil = p; // ok
    Ival_slider* pi1 = dynamic_cast<Ival_slider*>(p); // ok
    BBslider* pbb1 = p; // 错误：BBslider是保护的基类
    BBslider* pbb2 = dynamic_cast<BBslider*>(p);
    // 可以：pbb2变成0
}
// 上述例子说明：dynamic_cast也不能违背对private和protected基类的保护
```


dynamic_cast

- dynamic_cast的专长是处理那些编译器无法确定转换正确性的情况
- dynamic_cast<T*>(p)将查看p指向的对象，如果这个对象属于类T或者有唯一的类型为T的基类，将返回指向该对象的类型T*的指针；否则就返回0(要求该转换能唯一确定一个对象，否则转换失败，返回0)

dynamic_cast示例

```
class My_slider:public Ival_slider{// 多态基类(有虚函数)
};
class My_date:public Date{ // 基类不是多态类(无虚函数)
};
void g(Ival_box* pb, Date* pd){
   My_slider* pd1 = dynamic_cast<My_slider*>(pb); // ok
   My_date* pd2 = dynamic_cast<My_date*>(pd);
   // 错误：Date不是多态类
}
```

- dynamic_cast要求一个到多态类型的指针或者引用，以便做upcast或者crosscast
- dynamic_cast的目标类型不必是多态的
- 到void*的dynamic_cast可以用于确定多态类型的对象的起始地址

15.4.1.1 引用的dynamic_cast

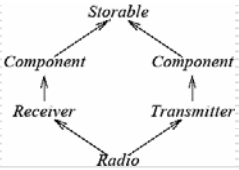
- dynamic_cast<T*>(p)时，对返回值必须检查是否为0，这表示一个询问："p所指的对象的类型是T吗？"
- dynamic_cast<T&>(r)，不是询问而是断言："由r引用的对象的类型是T"，其转换结果隐式地由dynamic_cast本身的实现去检查，如果引用的对象不具有所需要的类型，就会抛出一个bad_cast异常

引用的dynamic_cast示例

```
void f(Ival_box* p, Ival_box& r){
   if(Ival_slider* is = dynamic_cast<Ival_slider*>(p)){
      // p是指向一个Ival_slider吗?
   }
   else{ // *p不是一个slider
   }
   Ival_slider& is = dynamic_cast<Ival_slider&>(r);
   // r引用一个Ival_slider!
}
void g(){
   try { f(new BB_ival_slider, *new BB_ival_slider); // ok
         f(new BBdial, *new BBdial); // 抛出bad_cast异常
   }
   catch(bad_cast) {/*...*/}
}
```


15.4.2 在类层次结构中漫游

```
class Component:public virtual Storable{/*...*/};
class Record:public Component{/*...*/};
class Transmitter:public Component{/*...*/};
class Radio:public Receiver, public Transmitter{/*...*/};

void h1(Radio& r){
   Storable* ps = &r;
   Component* pc =
   dynamic_cast<Component*>(ps);
} // pc = 0
```



15.4.2.1 静态和动态强制(casts)

- dynamic_cast能从多态性的虚基类强制到某个派生类或者兄弟类，而static_cast不检查被强制的对象，所以它做不到这些

```
void g(Radio& r){
   Receiver* prec = &r; // Receiver是Radio的常规基类
   Radio* pr = static_cast<Radio*>(prec); // 可以，不检查
   pr = dynamic_cast<Radio*>(prec); // 可以，运行时检查

   Storable* ps = &r;  // Storable是Radio的虚基类
   pr = static_cast<Radio*>(ps); // 错误：不能从虚基类强制
   pr = dynamic_cast<Radio*>(ps); // 可以：运行时检查
}
```


静态和动态强制(casts)

- dynamic_cast要求多态性的操作对象，这是因为在非多态对象里没有存储有关的信息，而从一个对象触发，找到以它作为基类子对象的那些派生类对象，需要这种信息
- 之所以还存在使用static_cast进行的类层次结构的漫游，这是因为存在着数以百万计的代码是在dynamic_cast可以使用之前写出的，大都依靠C风格强制来保证合法性，常常遗留着隐蔽的错误
- 因此，尽管dynamic_cast有一些运行之外的开销，在所有可能的地方，还是应该去用更安全的dynamic_cast

静态和动态强制(casts)示例

- 编译器不能对由void*所指向的存储提供任何保证，这意味着dynamic_cast不能从void*出发进行强制，因为它必须去查看对象，以便确定其类型
- 这种情况需要使用static_cast
- dynamic_cast和static_cast都遵守const和访问控制规则

```
Radio* f(void* p){
   Storable* ps = static_cast<Storable*>(p);
   // 相信程序员！
   return dynamic_cast<Radio*>(ps);
}
```


15.4.3 类对象的构造和析构

- 一个类对象并不简单地就是一块存储
- 类对象是通过其构造函数从“原始存储”中构筑起来的，并通过其析构函数的执行使它重归于“原始存储”
- 构造使一个自下而上的过程，析构则自上而下
- 一个类对象也就是在它能得以构造或析构的意义上才称其为对象

15.4.4 typeid和扩展的类型信息

- 有时候，需要知道一个对象的确切类型，此时，可以使用typeid运算符，它取得一个对象，该对象代表着对应运算对象的类型
- typeid返回一个到标准库类型type_info的引用，该类型在头文件<typeinfo>里面定义
- typeid()经常被用于找出由一个引用或者指针所引用的对象的确切类型

typeid和扩展的类型信息

- 如果typeid是个函数，其声明大致如下：

```
class type_info;
const type_info& typeid(type_name) throw();
const type_info& typeid(expression) throw(bad_typeid);
```

- 如果一个多态类型的指针或者引用的操作对象的值是0，typeid()将抛出一个bad_typeid异常
- 如果typeid()的操作对象的类型不是多态的，或者它不是一个lvalue，其结果将在编译时确定

typeid和扩展的类型信息

```
class type_info{
public:
   virtual ~type_info(); // 多态的
   bool operator==(const type_info&)const; //可以比较
   bool operator!=(const type_info&)const;
   bool before(const type_info&)const; // 有序
   const char* name() const; // 类型名
private:
   type_info(const type_info&); // 禁止复制
   type_info& operator=(const type_info&); // 禁止赋值
};
#include <typeinfo>
void g(Component* p){
   cout << typeid(*p).name(); }
```


15.4.4.1 扩展的类型信息

- 典型情况下，找到一个对象的确切类型，只不过是作为获取和使用有关该类型的更详细信息的第一步
- 用户可以使用任何方式来对这类信息进行处理和建立关联

15.5 指向成员的指针

- C++提供了一种能间接引用类成员的功能，指向成员的指针就是一种标识类成员的值，可以将它们想象成位于该类的一个对象里的那个成员的位置，在这里，实现需要负责去处理数据成员、虚函数、非虚函数之间的差异
- 将地址运算符应用到完全限定的类成员名，就能得到指向成员的指针，要声明此类指针，需要使用形式为X::*声明符
- 静态成员不与任何对象相关联，指向静态成员的指针是一个常规指针

指向成员的指针示例

```
class Std_interface{
public:
   virtual void start() = 0;      virtual void suspend() = 0;
   virtual void resume() = 0;  virtual void quit() = 0;
   virtual void full_size() = 0; virtual void small() = 0;
   virtual ~Std_interface() {}
};
typedef void (Std_interface::*Pstd_mem)();
void f(Std_interface* p){
   Pstd_mem s = &Std_interface::suspend;
   p->suspend();
   (p->*s)();
}
```


15.6 自由存储

- 我们可以重新定义operator new()和operator delete()来取代全局的操作符或者为某个特定的类提供这些操作

```
class Employee{ // operator new和operator delete默认为static
public:  // size_t表示实际分配或者删除的对象的大小
   void* operator new(size_t);
   void operator delete(void*, size_t);
};
void* Employee::operator new(size_t s){
   // 分配s字节的内存，返回一个到它的指针
}
void Employee::operator delete(void* p, size_t s){
   // p指向new分配的内存，delete负责释放它们以便重用
}
```


自由存储示例

```
class Manager:public Employee {
   int level;
};
void f(){
   Employee* p = new Manager;
   // 确切类型丢失
   delete p; // 无法得到正确大小
}
// 解决方法，提供虚析构函数
class Employee {
public:
   void* operator new(size_t);
   void operator delete(void*, size_t);
   virtual ~Employee();
};
```

在Employee里给出了析构函数，能够保证每个它派生出的类都将提供一个析构函数(这样就能保证大小正确)，即使派生类里没有用户定义的析构函数

甚至是空的析构函数也可以
Employee::~Employee
{}

15.7 忠告

- [1] 利用常规的多重继承表述特征的合并
- [2] 利用多重继承完成实现细节与界面的分离
- [3] 用***virtual***基类表达在类层次结构里对某些类（不是全部类）共同的东西
- [4] 避免显式的类型转换（强制）
- [5] 在不可避免地需要漫游类层次结构的地方，使用***dynamic_cast***
- [6] 尽量用***dynamic_cast***而不是***typeid***

忠告

- [7] 尽量用***private***而不是***protected***
- [8] 不要声明***protected***数据成员
- [9] 如果某个类定义了***operator delete***( )，它也应该有虚析构函数
- [10] 在构造和析构期间不要调用虚函数
- [11] 尽量少用为解析成员名而写的显式限定词，最好是在覆盖函数里用它

C++编程(14)

唐晓晟
北京邮电大学电信工程学院

第16章 库组织和容器

- 标准库的设计
- 容器设计
- 向量
- 忠告

16.1 标准库的设计

- **C++标准库**
- 提供了对一些语言特征的支持，例如，存储管理
- 提供了有关实现所确定的语言方面的一些信息，例如最大的float值
- 提供了那些无法在每个系统上由语言本身做出最优实现的函数，例如sqrt()
- 提供了一些非基本的功能，使程序员可以为可移植性而依靠它们，例如表，映射
- 提供了一个为扩展它所提供功能的基本框架
- 为其它库提供一个公用的基础

16.1.1 设计约束

- **标准库所扮演的角色为它的设计提出了许多约束**
- 对于每个学生、每个专业程序员，包括其他库的构建者，都应该是及其宝贵而且又是能负担得起的
- 被每个程序员，在与库相关的领域中的每项工作中直接或者间接地使用
- 足够高效，能够在其他库的实现中成为手工编写的函数、类或模板的真正替代品

设计约束

- 在数学的意义上是最基础性的
- 对普通使用是方便而高效的、并具有合理的安全性在它们所做的工作方面是完全的
- 能很好地与内部类型和操作混合使用
- 按照默认方式是类型安全的
- 支持各种被广泛接受的程序设计风格
- 可以扩展，使之能以类似于内部类型和标准库类型的处理方式，去处理用户定义类型

16.1.2 标准库组织

- 标准库的功能都定义在std名字空间里，用一组头文件的方式呈现，这些头文件表明了这个库的各个主要部分，因此，列出这些头文件也就给出了标准库的一个概貌
- 所有名字以c开头的标准头文件等价于C标准库中的一个头文件，每个头文件<X.h>在全局名字空间和std名字空间里定义了C标准库的一个部分，与之对应的存在一个头文件<cX>，它只在名字空间std里定义同样的一组名字

标准库组织

Containers		
<vector>	*one-dimensional array of T*	§16.3
<list>	*doubly-linked list of T*	§17.2.2
<deque>	*double-ended queue of T*	§17.2.3
<queue>	*queue of T*	§17.3.2
<stack>	*stack of T*	§17.3.1
<map>	*associative array of T*	§17.4.1
<set>	*set of T*	§17.4.3
<bitset>	*array of booleans*	§17.5.3

- 关联容器multimap和multiset可以分别在<map>和<set>中找到，priority_queue在<queue>里声明

标准库组织

General Utilities		
<utility>	*operators and pairs*	§17.1.4, §17.4.1.2
<functional>	*function objects*	§18.4
<memory>	*allocators for containers*	§19.4.4
<ctime>	*C-style date and time*	§s.20.5

- 头文件<memory>还包括auto_ptr模板，主要用于指针与异常间的平滑交互

Iterators		
<iterator>	*iterators and iterator support*	Chapter 19

- 迭代器提供一些机制，使标准算法能通用于标准容器和类似类型

标准库组织

Algorithms		
<algorithm>	*general algorithms*	Chapter 18
<cstdlib>	***bsearch*** **( )** ***qsort*** **( )**	§18.11

- 一个典型的通用算法可以应用于任意的具有任何类型的元素的序列，C标准库函数bsearch()和qsort()可以应用于以任何类型为元素的内部数组，条件是该元素类型没有用户定义的复制构造函数和析构函数

Diagnostics		
<exception>	*exception class*	§14.10
<stdexcept>	*standard exceptions*	§14.10
<cassert>	*assert macro*	§24.3.7.2
<cerrno>	*C-style error handling*	§20.4.1

标准库组织

Strings		
<string>	*string of* ***T***	Chapter 20
<cctype>	*character classification*	§20.4.2
<cwtype>	*wide-character classification*	§20.4.2
<cstring>	*C-style string functions*	§20.4.1
<cwchar>	*C-style wide-character string functions*	§20.4
<cstdlib>	*C-style string functions*	§20.4.1

- 头文件<cstring>里声明了strlen()、strcpy()等一族函数。<cstdlib>里声明了atof()和atoi()，它们用于将C风格的字符串转换到数值

标准库组织

Input/Output		
<iosfwd>	*forward declarations of I/O facilities*	§21.1
<iostream>	*standard iostream objects and operations*	§21.2.1
<ios>	*iostream bases*	§21.2.1
<streambuf>	*stream buffers*	§21.6
<istream>	*input stream template*	§21.3.1
<ostream>	*output stream template*	§21.2.1
<iomanip>	*manipulators*	§21.4.6.2
<sstream>	*streams to/from strings*	§21.5.3
<cstdlib>	*character classification functions*	§20.4.2
<fstream>	*streams to/from files*	§21.5.1
<cstdio>	***printf*** *( ) family of I/O*	§21.8
<cwchar>	***printf*** *( ) -style I/O of wide characters*	§21.8

标准库组织

Localization		
<locale>	*represent cultural differences*	§21.7
<clocale>	*represent cultural differences C-style*	§21.7

- 一个locale使一些特征本地化，如日期的输出格式，用于表示现金的字符，字符串的排列准则等，这些特征体现了不同自然语言和文化之间的差异

标准库组织

Language Support		
<limits>	numeric limits	§22.2
<climits>	C-style numeric scalar-limit macros	§22.2.1
<cfloat>	C-style numeric floating-point limit macros	§22.2.1
<new>	dynamic memory management	§16.1.3
<typeinfo>	run-time type identification support	§15.4.1
<exception>	exception-handling support	§14.10
<cstddef>	C library language support	§6.2.1
<cstdarg>	variable-length function argument lists	§7.6
<csetjmp>	C-style stack unwinding	§s.18.7
<cstdlib>	program termination	§9.4.1.1
<ctime>	system clock	§s.18.7
<csignal>	C-style signal handling	§s.18.7

- <cstddef>头文件定义了由sizeof()的返回类型size_t，指针之差的结果类型ptrdiff_t和声名狼藉的NULL宏

标准库组织

Numerics		
<complex>	complex numbers and operations	§22.5
<valarray>	numeric vectors and operations	§22.4
<numeric>	generalized numeric operations	§22.6
<cmath>	standard mathematical functions	§22.3
<cstdlib>	C-style random numbers	§22.7

- 由于历史的原因，abs()、fabs()和div()位于<cstdlib>里，而不是与其他数学函数一起在<cmath>里

标准库组织

- 用户或者库的实现者都不允许在标准头文件里添加或者减少任何东西。试图通过在包含头文件之前定义一些宏的方式去修改头文件的内容，或者通过在其环境中写声明去改变头文件里的声明的意义，都是不可接受的
- 为了使用标准库的功能，必须包含有关的头文件

16.2 容器设计

- 容器就是能保存其他对象的对象，例如：表、向量和关联数组
- C++标准库容器的设计依据的是两条准则：在各个容器的设计中提供尽可能大的自由度，同时，又使各种容器能够向用户呈现出一个公共的界面，这将使容器实现能达到最佳的效率，也使用户能写出不依赖于所使用的特定容器类型的代码

16.2.1 专门化的容器和迭代器

- 要提供向量和表，最明显方式就是对它们中的每个都按照预想的方式给予定义：

```
Template<class T>class Vector{
public:
    explicit Vector(size_t n);
    T& operator[](size_t);
};
Template<class T>class List{
public:
    class Link{/*...*/};
    List();
    void put(T*);
    T* get();
};
```

每个类提供一组操作，对于使用而言，这些操作基本上是最理想的，而且，我们可以为各个类选择最合适的表示方式，而不去考虑其他容器的情况

有关操作的实现基本上是最优的

专门化的容器和迭代器

- 对于大部分容器，一种公共的使用方式就是迭代式地穿过整个容器，一个接一个地查看元素，这通常可以通过定义适合于容器类型的迭代器类做到
- 最理想的情况是：一个迭代类能够应用于各种容器类

```
template<class T>class Itor { // 公共界面
public:
    virtual T* first() = 0;  // 返回第一个元素的指针
    virtual T* next() = 0; // 返回下一个元素的指针
};
```


专门化的容器和迭代器

- 现在可以为Vector和List提供迭代器

```
template<class T>class Vector_itor:public Itor<T>{
    Vector<T>& v;
    size_t index;
public:
    Vector_itor(Vector<T>& vv):v(vv), index(0){}
    T* first(){ return (v.size())? &v[index=0]:0; }
    T* next(){ return (++index<v.size())? &v[index]:0;}
};
template<class T>class List_itor:public Itor<T>{
    List<T>& lst;
    List<T>::Link p;
public: List_itor(List<T>&);
    T* first(); T* next();
};
```


专门化的容器和迭代器

- 写容器上迭代的代码

```
int count(Itor<char>& ii, char term)
{
    int c = 0;
    for(char* p = ii.first(); p; p=ii.next()) if(*p==term) c++;
    return c;
}
```

- 注意，这种方式引起了一次虚函数的调用，很多情况下，这种额外开销是无关大局的，但是，这种方式对于标准库不合适
- 本例中：Itor甚至可以在Vector和List设计和实现很久以后才提供

16.2.2 有基类的容器

- Simula采用如下风格定义了它的标准容器

```
struct Link{
    Link* pre; Link* suc;
};
class List{
    Link* head;
    Link* curr;
public:
    Link* get();
    void put(Link*);
};
// 一个List就是一个Link的表，可以保存由Link派生的任何类型的对象
```

```
class Ship:public Link{/*...*/};

void f(List* lst){
    while(Link* po=lst->get()){
        if(Ship* ps =
            dynamic_cast<Ship*>(po)){
            //...
        }
        else {
            //...
        }
    }
}
```


有基类的容器

- 常见情况是定义一个公共容器类型

```
class Container:public Object{
 // 公共容器类型
public:    virtual Object* get();
    virtual void put(Object*);
    virtual Object*& operator[](size_t);
}; // 操作全部是虚的
class List:public Container{
public:
    Object* get();
    void put(Object*);
};
class Vector:public Container{
public:    Object*& operator[](size_t);
};
```

对于此类实现，Container中一般提供所希望支持的各种容器类基本操作集合的并集，针对一组概念的这样一个界面被称作一个肥大界面(fat interface)

这类设计将不必要的复杂性推给了用户，强加了显著的运行时开销，并限制了可以放入容器的对象的种类，对于标准库也不理想

16.2.3 STL容器

- 标准库容器和迭代器(STL框架)可以认为是一种能够同时取得前面描述了的两种传统模型的优点的途径，STL是一心一意追求真正高效的和通用的算法而结出的硕果
- 考虑到效率因素，STL中排除了采用难以内联化的虚函数去实现短小并使用频繁的访问函数
- 为了避免肥大界面，公共操作集合中不包括那些无法有效在所有集合中有效实现的操作
- 为每种容器提供自己的迭代器，并让这些迭代器都支持同一组标准的迭代器操作

STL容器

- 标准容器不是从一个公共基类派生出来的，而是每个容器实现了完整的标准容器界面
- 也不存在公共的迭代器基类，这样，在使用标准容器或者迭代器时就不涉及任何显式或隐式的运行时类型检查
- 最重要也是最困难的问题是如何为所有容器提供公共的服务，这个问题是通过模板参数传递的“分配器”(allocator)处理的，没有通过公共基类
- STL广泛而深入的依靠模板机制，为避免大量的代码重复，通常需要通过专门化的方式，为保存指针的容器提供共享的实现

16.3 向量(vector)

- 作为标准库的一个完整例子，除非另有说明，关于vector所说的情况都适用于每种标准库容器
- 17章将描述list, set, map等的专有特征
- 对vector的介绍分几个步骤进行：成员类型、迭代器、元素访问、构造函数、堆栈操作、表操作、大小和容量、协助函数、以及vector<bool>

16.3.1 类型

```
template <class T, class A = allocator<T> > class std::vector {
public:    // types:
   typedef T value_type; // 元素类型
   typedef A allocator_type; // 存储管理器类型
   typedef typename A::size_type size_type; // 一般做容器下标size_t
   typedef typename A::difference_type difference_type; // ptrdiff_t
   typedef implementation_dependent1 iterator; // T* (迭代器)
   typedef implementation_dependent2 const_ iterator; // const T*
   typedef std::reverse_iterator<iterator> reverse_iterator;
   typedef std::reverse_iterator<const_iterator> // 反向迭代器
                                    const_reverse_iterator;
   typedef typename A::pointer pointer; // 元素指针
   typedef typename A::const_pointer const_pointer;
   typedef typename A::reference reference; // 元素引用
   typedef typename A::const_reference const_reference;
};
```


类型

- 使用容器的代码
- 必须在作为模板参数的成员类型名字之前加typename，不加的话，编译器无法判断

```
template<class C> typename C::value_type sum(const C& c)
{
   typename C::value_type s = 0;
   typename C::const_iterator p = c.begin() ; // 从头开始
   while (p!=c.end()) { // 继续到结束处
      s += *p; // 取得元素的数值
      ++p; // 使p指向下一个元素
   }
   return s;
}
```


16.3.2 迭代器

```
template <class T, class A = allocator<T> > class vector {
public:
   iterator begin() ; // 指向首元素
   const_iterator begin() const;
   iterator end() ; // 指向末端元素的下一个位置
   const_iterator end() const;
   reverse_iterator rbegin() ; // 指向反向序列的首元素
   const_reverse_iterator rbegin() const;
   reverse_iterator rend() ; // 同上(end())
   const_reverse_iterator rend() const;
   // ...
};
```

迭代器

```
template<class C> typename C::iterator
    find_last(const C& c, typename C::value_type v)
{ // 对vector进行反向搜索，找到第一个出现v的位置
    typename C::reverse_iterator  ri = find(c.rbegin(), c.rend(), v);
    if(ri == c.rend()) return c.end(); // 表示未找到
    typename C::iterator i = ri.base();
    return --i;
}
```

- 对于reverse_iterator，ri.base()返回一个iterator，指向ri所指之处后面的一个位置，若没有reverse_iterator，必须使用iterator写一个显式的循环来完成同样的任务
- 注意：C::reverse_iterator和C::iterator不是同一个类型

16.3.3 元素访问

- 与其他容器相比，vector的一个重要特点就是可以方便高效地按照任何顺序访问其中的元素

```
template <class T, class A = allocator<T> > class vector {
public:
    reference operator[](size_type n) ; // 不加检查的访问
    const_reference operator[](size_type n) const;
    reference at(size_type n) ; // 受检查的访问
    const_reference at(size_type n) const;
    reference front() ; // 首元素
    const_reference front() const;
    reference back() ; // 尾元素
    const_reference back() const;
    // ...
};
```


元素访问示例

- 下标索引通过operator[]()和at()完成，at()进行下标检查，越界时抛出out_of_range异常
- 访问操作返回类型为const_reference或者reference，对于vector<X>，reference就是X&

```
void f(vector<int>& v, int i1, int i2)
try {
    for(int i = 0; i < v.size() ; i++) {
        // 已经做范围检查，这里用不检查的v[i]
    }
    v.at(i1) = v.at(i2) ; // 检查下标
}
catch(out_of_range) {
    // 错误: out_of_range
}
```


元素访问示例

- 在各种标准序列中，只有vector和deque支持下标
- 试图创建一个越界引用的效果是无定义的

```
void f(vector<double>& v)
{
    double d = v[v.size()] ; // 无定义：下标错误
    list<char> lst;
    char c = lst.front() ; // 无定义：表为空
}
```

16.3.4 构造函数

```
template <class T, class A = allocator<T> > class vector {
public:
    explicit vector(const A& =A()) ;
    explicit vector(size_type n, const T& val = T() , const A& =A());
        // n个val的副本
    template <class In> // In 必须是输入迭代器 (§19.2.1)
    vector(In first, In last, const A& =A()) ; // 从first到last进行拷贝
    vector(const vector& x) ;
    ~vector() ;
    vector& operator=(const vector& x) ;
    template <class In> // In 必须是输入迭代器(§19.2.1)
    void assign(In first, In last) ; //从first到last进行拷贝
    void assign(size_type n, const T& val) ; // n个val的副本
    // ...
};
// vector提供了完整的一组构造、析构和复制操作
```


构造函数示例

- vector提供了对任意元素的快速访问，但是修改它的值的代价是相当昂贵的，因此，我们在创建vector时通常给出一个初始大小

```
vector<Record> vr(10000) ;
void f(int s1, int s2) {
    vector<int> vi(s1) ;
    vector<double>* p = new vector<double>(s2) ;
}
// 这样分配的vector元素都将用元素类型的默认构造函数做初始化，
如果没有默认构造函数，就不能创建以这个类型为元素的向量
class Num{
public:   Num(long) ; // 无默认构造函数
};
vector<Num> v1(1000) ; // 错误: 无默认构造函数
vector<Num> v2(1000,Num(0)) ; // ok
```


构造函数示例

- vector的大小也可以通过初始元素的集合隐式给定，这样做需要给vector提供一个值的序列，此时，vector的构造函数会调整vector的大小

```
void f(const list<X>& lst)
{
    vector<X> v1(lst.begin() ,lst.end()) ; // 从表复制元素
    char p[] = "despair";
    vector<char> v2(p,&p[sizeof(p)-1]); // 从c风格字符串复制字符
}
```


构造函数示例

- 复制构造函数和复制赋值运算符拷贝vector的所有元素，其代价可能会很高，所以vector通常采用引用传递

```
void f1(vector<int>&) ; // 常见风格
void f2(const vector<int>&) ; // 常见风格
void f3(vector<int>) ; // 罕见风格
void h()
{
    vector<int> v(10000) ;
    // ...
    f1(v) ; // 传递引用
    f2(v) ; // 传递引用
    f3(v) ; // 会导致10000个元素的赋值操作
}
```

构造函数示例

- assign函数，可以被用来一次性地为vector中的多个元素赋值，这样做时，没有引进大量的构造函数或者转换函数
- 注意：赋值将完全改变一个向量里的全部元素，所有的老元素都被删除，新元素插入

```
class Book {   /*...*/  };
void f(vector<Num>& vn, vector<char>& vc,
            vector<Book>& vb, list<Book>& lb)
{
    vn.assign(10,Num(0)) ; // 用10个Num(0)给vn赋值
    char s[] = "literal";
    vc.assign(s,&s[sizeof(s)-1]); // 用“literal”给vc赋值
    vb.assign(lb.begin() ,lb.end()) ; // 用表的元素赋值
}
```


16.3.5 堆栈操作

- 最常见的情况是，我们把vector看作一种可以直接通过下标访问元素的紧凑数据结构
- vector的push_back将隐式地导致vector的size()增长，所以不会上溢，但是可能下溢

```
template <class T, class A = allocator<T> > class vector {
public:
    void push_back(const T& x) ; // 在最后加入
    void pop_back() ; // 删除最后元素
    // ...
};
```


堆栈操作

- 每次调用push_back时vector就增长一个元素，这个元素被加到最后，所以s[s.size()-1]就是s.back()
- vector可以做为realloc()的更通用、更优美而且丝毫不降低效率的替代机制

```
void f(vector<char>& s){
    s.push_back('a') ;
    s.push_back('b') ;
    s.push_back('c') ;
    s.pop_back() ; // 注意，并不返回数值，只是弹出
    if (s[s.size()-1]!='b') error("impossible!") ;
    s.pop_back() ;
    if (s.back() != 'a') error("should never happen!") ;
}
```


16.3.6 表操作

- push_back()、pop_back()和back()操作使vector可以有效地作为堆栈使用
- 然而，在vector的中间增加元素，从vector里删除元素有时也很有用

```
template <class T, class A = allocator<T> > class vector {
public:
    iterator insert(iterator pos, const T& x) ; // 在pos前添加x
    void insert(iterator pos, size_type n, const T& x) ;  // 添加n个x
    template <class In> void insert(iterator pos, In first, In last) ;
        // In必须是输入迭代器 // 插入一批来自序列的元素
    iterator erase(iterator pos) ; // 删除pos处的元素
    iterator erase(iterator first, iterator last) ; // 删除一段元素
    void clear() ; // 删除所有元素
}; // insert()返回的迭代器指向新插入的元素，erase()返回的迭代器指向被删除的最后元素之后的那个元素，这两种操作都可能导致迭代器指向另外的元素
```

表操作

```
vector<string> fruit;
fruit.push_back("peach");
fruit.push_back("apple");
fruit.push_back("kiwifruit");
fruit.push_back("pear");
fruit.push_back("starfruit");
fruit.push_back("grape");
sort(fruit.begin() ,fruit.end()); // 排序
vector<string>::iterator p1 =
    find_if(fruit.begin() ,fruit.end() ,initial('p');
    // 找到第一个以p开头的水果
vector<string>::iterator p2 =
    find_if(p1,fruit.end() ,initial_not('p');
    // 找到最后一个以p开头的水果
fruit.erase(p1,p2) ;
// 删除p1到p2之间的所有元素，但是不包括p2
```

注意，erase()要求两个类型一样的参数，而iterator和reverse_iterator是不同的类型，不可在这里混用。
删除元素将改变vector的大小，位于被删除元素之后的元素将被复制到空位中。
可以使用insert()插入元素，但是注意这将导致批量元素被移动，若插入元素太多，，请考虑使用list容器

16.3.7 元素定位

- 一般来说，insert或者erase函数的目标是某个明确的位置(begin()或end())，或者通过检索操作得到的结果(find())，或者在迭代中确定的某个下标
- 实际应用中，也可以直接使用数值下标(比如数字7)作为索引，方法是c.erase(c.begin()+7)

元素定位示例

```
template<class C> void f(C& c){
    c.erase(c.begin()+7); // ok
    // 但并非所有容器的迭代器都支持+运算，list不支持c.begin()+7
    c.erase(&c[7]); // 一般不行
    // 此处，c[7]引用相应的元素，其地址是合法的迭代器
    // 但是，对于其他容器，比如map，将返回一个mapped_type&
    // 而不是一个到元素的引用(其类型应该是value_type&)
    c.erase(c+7); // 错误：容器+7没有意义
    c.erase(c.back()); // 错误：c.back()是引用，不是迭代器
    c.erase(c.end()-2); // 可以(倒数第二个元素)
    c.erase(c.rbegin()+2);
        // 错误：vector::reverse_iterator和vector::iterator的类型不同
    c.erase((c.rbegin()+2).base()) ; // 难懂，但可行($19.2.5)
}
```


16.3.8 大小和容量

- 通常，vector在需要时将自动增长，然而，我们也可以直接提出vector应当如何使用存储的问题
- 任意时刻，vector中的元素个数可以通过size()获得，并可通过resize()改变(和C中的realloc()很类似)

```
template <class T, class A = allocator<T> > class vector {
public:
    size_type size() const; // 元素个数
    bool empty() const { return size()==0; } // 是否为空
    size_type max_size() const; // 最大可能大小
    void resize(size_type sz, T val = T());
        // 增加的元素用val初始化
    size_type capacity() const; // 当前已经分配的存储的大小
    void reserve(size_type n); // 做出n个元素空位，不初始化
        // 如果n>max_size()，则抛出length_error
};
```

大小和容量

- 对reserve(n)的调用保证，在v的大小增加到使v.size()超过n之前不需要再做任何存储分配

```
struct Link {
    Link* next;
    Link(Link* n =0) : next(n) {}
    // ...
};
vector<Link> v;
void chain(size_t n) // 填入n个Link，每个Link指向前一个
{
    v.reserve(n) ;
    v.push_back(Link(0)) ;
    for (int i = 1; i<n; i++) v.push_back(Link(&v[i1]));
    // ...
}
```


大小和容量

- 为后面工作预留空间有两个优点
  - 即使是没有多加思考的实现也可以先分配足够的空间，以避免随着工作进展慢慢地请求足够的内存，这种情况下可能会使得效率有所提高
  - 只有通过调用reserve()，保证向量构建过程中不再出现分配，才能保证v的元素间能建立正确的链接
- 此外，还可以确保潜在的存储耗尽问题和那些代价高昂的元素重新分配只可在预期的时刻出现，对于那些有着严格的运行时间约束的程序，这些极为重要

大小和容量

- reserve()并不改变vector的大小，就是说，并没有要求它对任何新元素做初始化
- capacity()给出当时所保留的所有存储位置的个数，c.capacity()-c.size()就是在不致于重新分配的情况下，可以插入的元素数
- 减小vector的大小不会减少其容量，这样做只能为vector随后的增长多留下了一些空间

16.3.9 其他成员函数

- 许多算法涉及到元素的交换，通常交换元素就是简单的复制所有元素
- 然而，vector常常是用一种类似于到元素组的句柄的结构实现的，这样，交换两个vector的工作可以通过交换它们的句柄的方式更有效的实现

```
template <class T, class A = allocator<T> > class vector {
  public:
  // ...
  void swap(vector&) ;
  allocator_type get_allocator() const;
};
```

16.3.10 协助函数

- 用==和<可以比较两个vector

```
template <class T, class A>
bool std: :operator==(const vector<T,A>& x, const vector<T,A>& y);
template <class T, class A>
bool std: :operator<(const vector<T,A>& x, const vector<T,A>& y);
```

- 标准库还提供了!=、<=、>和>=等协助函数

16.3.11 vector<bool>

- 标准库还提供了专门化vector(bool)，作为bool类型的紧凑vector，bool变量可以寻址，所以它至少要占一个字节
- 通常的vector操作都可以对vector(bool)使用，而且保持了它们原来的意义
- 标准库还提供了布尔值集合bitset，带有一组布尔集合运算

16.4 忠告

- [1] 利用标准库功能，以维持可移植性
- [2] 决不要另行定义标准库的功能
- [3] 决不要认为标准库比什么都好
- [4] 在定义一种新功能时，应考虑它是否能够纳入标准库所提供的框架中
- [5] 记住标准库功能都定义在名字空间***std***
- [6] 通过包含标准库头文件声明其功能，不要自己另行显示声明

忠告

- [7] 利用后续抽象的优点
- [8] 避免肥大的界面
- [9] 与自己写按照反向顺序的显式循环相比，最好是写利用反向迭代器的算法
- [10] 用***base***()从***reverse_iterator***抽取出***iterator***
- [11] 通过引用传递容器
- [12] 用迭代器类型，如***list<char>::iterator***，而不要采用索引容器元素的指针

忠告

- [13] 在不需要修改容器元素时，使用***const***迭代器
- [14] 如果希望检查访问范围，请(直接或间接)使用***at***()
- [15] 多用容器和***push_back***()或***resize***()，少用数组和***realloc***()
- [16] ***vector***改变大小之后，不要使用指向其中的迭代器
- [17] 利用***reserve***() 避免使迭代器非法
- [18] 在需要的时候，***reserve***() 可以使执行情况更容易预期

C++编程(15)

唐晓晟
北京邮电大学电信工程学院

第17章 标准容器

- 标准容器
- 序列
- 序列适配器
- 关联容器
- 拟容器
- 定义新容器
- 忠告

17.1 标准容器

- 标准库定义了两类容器：序列和关联容器
- 序列都很象vector，除了专门指明的情况之外，有关vector所论及的成员类型和函数都可以对任何其他容器使用，并产生同样效果
- 关联容器提供了基于关键码访问元素的功能

17.1.1 操作综述

- 本节列出标准容器的公共的或者几乎公共的成员

Member Types (§16.3.1)	
value_type	Type of element.
allocator_type	Type of memory manager.
size_type	Type of subscripts, element counts, etc.
difference_type	Type of difference between iterators.
iterator	Behaves like *value_type**.
const_iterator	Behaves like *const value_type**.
reverse_iterator	View container in reverse order; like *value_type**.
const_reverse_iterator	View container in reverse order; like *const value_type**.
reference	Behaves like *value_type*&.
const_reference	Behaves like *const value_type*&.
key_type	Type of key (for associative containers only).
mapped_type	Type of *mapped_value* (for associative containers only).
key_compare	Type of comparison criterion (for associative containers only).

操作综述

- 容器可以看成是按照该容器的iterator所定义的顺序形成的序列，或者按反向顺序形成的序列，对于关联容器，这个顺序则基于容器的比较准则(默认为<)

Iterators (§16.3.2)	
begin()	Points to first element.
end()	Points to one-past-last element.
rbegin()	Points to first element of reverse sequence.
rend()	Points to one-past-last element of reverse sequence.

Element Access (§16.3.3)	
front()	First element.
back()	Last element.
[]	Subscripting, unchecked access (not for list).
at()	Subscripting, checked access (not for list).

操作综述

- 向量和双端队列提供了对它们的元素序列中后端元素的有效操作，表和双端队列还对它们的开始元素提供了等价的操作

Stack and Queue Operations (§16.3.5, §17.2.2.2)	
push_back()	Add to end.
pop_back()	Remove last element.
push_front()	Add new first element (for list and deque only).
pop_front()	Remove first element (for list and deque only).

- 各种容器提供如下的表操作

List Operations (§16.3.6)	
insert(p,x)	Add *x* before *p*.
insert(p,n,x)	Add *n* copies of *x* before *p*.
insert(p,first,last)	Add elements from [*first*:*last*[before *p*.
erase(p)	Remove element at *p*.
erase(first,last)	Erase [*first*:*last*[.
clear()	Erase all elements.

操作综述

- 所有容器都提供了与元素个数有关的各种操作和若干其他操作

Other Operations (§16.3.8, §16.3.9, §16.3.10)	
size()	Number of elements.
empty()	Is the container empty?
max_size()	Size of the largest possible container.
capacity()	Space allocated for *vector* (for vector only).
reserve()	Reserve space for future expansion (for vector only).
resize()	Change size of container (for vector, list, and deque only).
swap()	Swap elements of two containers.
get_allocator()	Get a copy of the container's allocator.
==	Is the content of two containers the same?
!=	Is the content of two containers different?
<	Is one container lexicographically before another?

操作综述

- 容器提供了各种构造函数和赋值操作

Constructors, etc. (§16.3.4)	
container()	Empty container.
container(n)	*n* elements default value (not for associative containers).
container(n,x)	*n* copies of *x* (not for associative containers).
container(first,last)	Initial elements from [*first*:*last*[.
container(x)	Copy constructor; initial elements from container *x*.
~container()	Destroy the container and all of its elements.

Assignments (§16.3.4)	
operator=(x)	Copy assignment; elements from container *x*.
assign(n,x)	Assign *n* copies of *x* (not for associative containers).
assign(first,last)	Assign from [*first*:*last*[.

操作综述

- 关联容器提供如下基于关键码的检索操作

Associative Operations (§17.4.1)	
operator[](k)	Access the element with key *k* (for containers with unique keys).
find(k)	Find the element with key *k*.
lower_bound(k)	Find the first element with key *k*.
upper_bound(k)	Find the first element with key greater than *k*.
equal_range(k)	Find the *lower_bound* and *upper_bound* of elements with key *k*.
key_comp()	Copy of the key comparison object.
value_comp()	Copy of the *mapped_value* comparison object.

17.1.2 容器综述

Standard Container Operations					
	[]	List Operations	Front Operations	Back (Stack) Operations	Iterators
	§16.3.3 §17.4.1.3	§16.3.6 §20.3.9	§17.2.2.2 §20.3.9	§16.3.5 §20.3.12	§19.2.1
vector	const	O(n)+		const+	Ran
list		const	const	const	Bi
deque	const	O(n)	const	const	Ran
stack				const+	
queue			const	const+	
priority_queue			O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Bi
multimap		O(log(n))+			Bi
set		O(log(n))+			Bi
multiset		O(log(n))+			Bi
string	const	O(n)+	O(n)+	const+	Ran
array	const				Ran
valarray	const				Ran
bitset	const				

容器综述

- 上页表中，Iterators列的Ran表示随机访问、Bi表示双向迭代
- 表中其他项目表示的都是操作的效率，const表示该操作所用的时间不依赖于容器中元素的数目(相当于O(1))，后缀+表明偶尔会出现显著的额外时间开销，注意基本操作中没有非常糟的操作，比如说$O(n^2)$
- 有关复杂性和代价的度量值都是上界

17.1.3 表示

- 标准并没有为各种标准容器预先设定任何特定的实现方式。相反，标准只是描述了各种容器的界面，并提出了一些复杂性要求
- 实现者将选择适当的，一般也是经过最聪明地优化过的实现方式，以满足这些普遍要求
- 例如：vector(数组)、list(链表)、map(平衡树)、string(11章所描述的方式或者数组)

17.1.4 对元素的要求

- 容器里的元素是被插入对象的副本
- 一个对象要想成为容器的元素，它就必须属于某个允许容器的实现对他进行复制的类型
- 容器可以利用复制构造函数或者赋值做元素的赋值工作，在两种情况下，都要求复制结果是一个等价的对象

17.1.4.1 比较

- 关联容器要求其元素是有序的，许多可以应用于容器的操作也是这样(例如,sort())
- 按照默认方式，这个序由<运算符定义，如果<不合适，程序员必须另外提供合适的比较操作
- 排序准则必须定义一种严格的弱顺序(strict weak ordering)，即小于和等于操作必须具有传递性

比较示例

```
template<class Ran> void sort(Ran first, Ran last);
template<class Ran, class Cmp> void sort(Ran first,
                                         Ran last, Cmp cmp);
class Nocase{
   public: bool operator()(const string&, const string&) const;
};
bool Nocase::operator()(const string& x, const string& y) const{
   string::const_iterator p = x.begin();
   string::const_iterator q = y.begin();
   while(p!=x.end() && q != y.end() && toupper(*p)==toupper(*q)){
      ++p; ++q;
   }
   if(p == x.end() ) return q!=y.end();
   if(q == y.end() ) return false;
   return toupper(*p) < toupper(*q);
}
```


比较示例

- 使用上面的比较准则调用sort()，例如，给定fruit: apple pear Apple Pear lemon
- 用sort(fruit.begin(), fruit.end(), Nocase())将产生
  fruit: Apple apple lemon Pear pear
- 而简单的用sort(fruit.begin(), fruit.end())将给出
  fruit: Apple Pear apple lemon pear

17.1.4.2 其他关系运算符

- 按默认规定，容器和算法在需要小于比较时都采用<，如果默认方式不合时，程序员必须另行提供一个比较准则
- 标准库在名字空间std::rel_ops里定义了各种比较运算符，并通过<utility>给出

```
template<class T>bool rel_ops::operator!=(const T& x, const T& y)
    {return !(x==y);}
template<class T>bool rel_ops::operator>(const T& x, const T& y)
    {return y<x;}
template<class T>bool rel_ops::operator<=(const T& x, const T& y)
    {return !(y<x);}
template<class T>bool rel_ops::operator>=(const T& x, const T& y)
    {return !(x<y);} // 注意以上运算符都是定义在<运算符基础之上的
```


17.2 序列

- 序列遵循vector所描述的模式
- 标准库提供的基本序列包括：vector list deque
- 从它们出发，通过定义适当的界面，生成了stack queue priority_queue
- 这几个序列被称为容器适配器(container adapters)、序列适配器(sequence adapters)，或者简称适配器

17.2.1 向量－vector

- 前面已经描述了vector的细节
- 只有vector提供了预留空间的功能
- 按默认规定，用[]的下标操作不做范围检查，如果需要检查，请用at()
- vector提供随机访问迭代器

17.2.2 表－list

- list是一种最合适于做元素插入和删除的序列
- 由于下标访问操作太慢(与vector相比)，list没有提供下标操作，也因为这个原因，list提供的是双向迭代器
- list给出了vector所提供的几乎所有成员类型和操作，例外的就是下标、capacity()、和reserve()

17.2.2.1 粘接、排序和归并

- list提供了若干特别适合于对表进行处理的操作，这些操作都是稳定的

```
template <class T, class A = allocator<T> > class list {
public:
   // 表的特殊操作
   void splice(iterator pos, list& x);
   // 将x的所有元素移到本表的pos之前，且不作复制
   void splice(iterator pos, list& x, iterator p);
   // 将x中的*p移到本表的pos之前，且不作复制
   void splice(iterator pos, list& x, iterator first, iterator last);
   void merge(list&); // 归并排序的表
   template <class Cmp> void merge(list&,Cmp);
   void sort();
   template <class Cmp> void sort(Cmp);
};
```


粘接、排序和归并

例如：给出
fruit: apple pear
citrus: orange grapefruit lemon
我们可以

```
list<string>::iterator p = find_if(fruit.begin(), fruit.end(), initial('p'));
fruit.splice(p, citrus, citrus.begin());
```

将orange从citrus粘接入fruit，这样做的效果是从citrus删除了第一个元素，并将它放到fruit里的第一个名字以p开始的元素之前

注意，这里的迭代器参数必须是合法的迭代器，确实指向应该指向的那个list

```
list<string>::iterator p = find_if(fruit.begin(), fruit.end(), initial('p'));
fruit.splice(p, citrus, citrus.begin()); // ok
fruit.splice(p, citrus, fruit.begin());   //错误：fruit.begin()并不指向citrus
citrus.splice(p, frtui, fruit.begin());   //错误：p不是指向citrus
```


粘接、排序和归并

- merge组合起两个排好序的表，方式是将一个list的元素都取出来，将它们都放入另外一个表，且维持正确的顺序
- 如果一个表未排序，merge()仍然能够产生一个表，其中包含两个表的元素的并集，只是顺序不作保证

```
f1: apple quince pear
f2: lemon grapefruit orange lime
可以按：
f1.sort();
f2.sort();
f1.merge(f2); // merge也不复制元素
其结果是：
f1: apple grapefruit lemon lime orange pear quince
f2: <empty>
```


17.2.2.2 前端操作

- list也提供一些针对第一个元素的操作，与每个容器都提供的针对最后元素的操作相对应
- 对list而言，前端操作与后端操作一样方便而高效，如果可以选择的话，最好是用后端操作，这样写出来的代码也对vector适用

```
template <class T, class A = allocator<T> > class list {
public:
   // 元素访问
   reference front(); // 引用第一个元素
   const_reference front() const;
   void push_front(const T&); // 加入新的第一个元素
   void pop_front(); // 删除第一个元素
};
```

17.2.2.3 其他操作

- 对list的插入和删除操作效率特别高，当这类操作非常频繁时，会使得人们倾向于使用list，反过来，又使直接支持某些删除元素的通用方法变得很有价值

```
template <class T, class A = allocator<T> > class list {
public:
   void remove(const T& val); // 删除所有值为val的元素
   template <class Pred> void remove_if(Pred p);
   void unique(); // 根据==删除连续的重复元素
   template <class BinPred> void unique(BinPred b);
   // 根据b删除重复元素
   void reverse(); / / reverse order of elements
};
```


17.2.3 双端队列—deque

- deque(其发音起伏就像check)就是一种双端的队列，也就是说，deque是一个优化了的序列，对其两端的操作效率类似于list，而其下标操作具有接近vector的效率
- 注意：在deque的中间插入和删除元素具有与vector一样的低效率，而不是类似list的效率

17.3 序列适配器

- vector、list和deque序列不可能互为实现的基础而同时又不损失效率，另一方面，stack和queue则都可以在这三种基本序列的基础上优雅而高效地实现
- 因此，stack和queue就没有定义为独立的容器，而是作为基本容器的适配器(adapter)
- 容器适配器所提供的是原来容器的一个受限的界面(特别是适配器不提供迭代器，提供它们的意图就是为了只经由它们的专用界面使用)

17.3.1 堆栈—stack

- stack容器在<stack>中定义

```
template <class T, class C = deque<T> > class std: :stack{
protected: C c;
public:
   typedef typename C::value_type value_type;
   typedef typename C::size_type size_type;
   typedef C container_type;
   explicit stack(const C& a =C()) : c(a) { }
   bool empty() const { return c.empty(); }
   size_type size() const { return c.size(); }
   value_type& top() { return c.back(); }
   const value_type& top() const { return c.back(); }
   void push(const value_type& x) { c.push_back(x); }
   void pop() { c.pop_back(); }
};
```

堆栈－stack

- stack就是某容器的一个简单界面，容器的类型作为模板参数传递给stack。stack所做的全部事情就是从它的容器的界面中删除所有非stack操作，并将back()、push_back()和pop_back()改为人们所习惯的名字top()、push()和pop()
- top()函数只读取、pop()操作只删除栈顶元素
- 按默认规定，stack用一个deque来保存自己的元素，但也可以采用任何提供了back()、push_back()和pop_back()的序列

```
stack<char> s1;  // 用deque<char>保存char类型的元素
stack<int, vector<int>> s2; // 用vector<int>保存int类型的元素
```


17.3.2 队列－queue

- queue在<queue>里定义，它也是一个容器的界面，该容器应该允许在back()处插入新元素，且能从front()提取出来

```
template <class T, class C = deque<T> > class std::queue {
protected: C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;
    explicit queue(const C& a =C()) : c(a) { }
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    value_type& front() { return c.front(); }
    const value_type& front() const { return c.front(); }
    value_type& back() { return c.back(); }
    const value_type& back() const { return c.back(); }
    void push(const value_  type& x) { c.push_back(x); }
    void pop() { c.pop_front(); } };
```


队列－queue

- 按照默认规定，queue用deque保存自己的元素，但是任何提供了front()、back()、push_back()和pop_front()的序列都可以用(vector没有提供pop_front()，所以不能作为queue的基础容器)
- 对于queue，也是使用push()添加元素，使用pop()删除元素

17.3.3 优先队列－priority_queue

```
template <class T, class C = vector<T>,
   class Cmp = less<typename C::value_type>> class std::priority_queue{
protected: C c; Cmp cmp;
public:
   typedef typename C::value_type value_type;
   typedef typename C::size_type size_type;
   typedef C container_type;
   explicit priority_queue(const Cmp& a1 =Cmp() , const C& a2 =C())
         : c(a2) , cmp(a1) { }
   template <class In>
      priority_queue(In first, In last, const Cmp& =Cmp() , const C& =C());
   bool empty() const { return c.empty(); }
   size_type size() const { return c.size(); }
   const value_type& top() const { return c.front(); }
   void push(const value_type&);
   void pop();
};
```

优先队列－priority_queue

- 按默认规定，priority_queue简单地用<运算符做比较，top()返回最大的元素
- 实现priority_queue的一种有用方式是采用一个树结构保存元素的相对位置，这能给出O(log(n))代价的push()和pop()操作
- 按照默认规定，priority_queue用一个vector保存它的元素，对任何能提供front()、push_back()和pop_back()，并能使用随机访问迭代器的序列都可以用
- priority_queue最可能是用一个heap实现

17.4 关联容器

- 关联容器是用户定义类型中最常见的也最有用的一种
- 关联数组也常被称为映射(map)，有时也被称为字典(dictionary)，其中保存的是值的对
- 给定一个称为关键码的值，我们就能访问一个称为映射值的值
- 可以将关联数组想象为一个下标不必是整数的数组
- map是传统的关联数组，multiset运新许元素中出现重复关键码，set和multiset可以看成是退化的关联数组，其中没有与关键码相关联的值

```
template<class K, class V> class Assoc {
public:
    V& operator[](const K&); // 返回对应于K的V的引用
};
```


17.4.1 映射－map

- 一个map就是一个(关键码－值)对偶的序列
- map提供基于关键码的快速提取操作，关键码具有唯一性，map提供双向迭代器
- map要求其关键码类型提供一个小于操作，以保持自己元素的有序性
- 对于没有明显顺序的元素，或者不必保持容器有序的情况，可以考虑用hash_map

17.4.1.1 类型

```
template <class Key, class T, class Cmp = less<Key>,
   class A = allocator< pair<const Key,T> > > class std::map
{
public:
    typedef Key key_type;    // 关键码类
    typedef T mapped_type; // 映射值类
    typedef pair<const Key, T> value_type; // 注意：是pair
    typedef Cmp key_compare;
    typedef A allocator_type;
    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef implementation_defined1 iterator;
    typedef implementation_defined2 const_iterator;
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
};
```

17.4.1.2 迭代器的对偶

- map提供了一组在以pair<const Key, mapped_type>为元素类型的迭代器

```
template <class Key, class T, class Cmp = less<Key>,
    class A = allocator< pair<const Key,T> > > class map
{
public:
    // 迭代器
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    // ...
};
```


迭代器的对偶

我们可以用如下方式打印出一个电话薄的各项内容

```
void f(map<string,number>& phone_book)
{
    typedef map<string,number>::const_iterator CI;
    for (CI p = phone_book.begin(); p!=phone_book.end(); ++p)
    cout << p-> first << ´\t´ << p-> second << ´\n´;
}

template <class T1, class T2> struct std::pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second; // 下面为构造函数
    pair() :first(T1()), second(T2()) { }
    pair(const T1& x, const T2& y) :first(x), second(y) { }
    template<class U, class V> pair(const pair<U,V>& p) :first(p.first),
    second(p.second) { }
}; // 对任何pair，总用first和second索引第一个和第二个元素(关键码和映射值)
```


17.4.1.3 下标

- map的特征性操作就是采用下标运算符提供的关联查找，当没有找到关键码时，map的下标操作将加进一个默认元素，如果仅仅希望进行查找操作，可以使用find()

```
template <class Key, class T, class Cmp = less<Key>,
    class A = allocator< pair<const Key,T> > > class map
{
    public:
    mapped_type& operator[](const key_type& k); // 用关键码k访问元素
};
void f() {
    map<string,int>m; // map开始为空
    int x =m[“Henry”]; // 创建新项Henry，初始化为0，返回0
    m[“Harry”] = 7; // 创建新项Harry，其值为7
    int y =m[“Henry”] ; // 返回Henry的值
    m[“Harry”] = 9; // Harry关键码对应的值改为9
}
```


17.4.1.4 构造函数

- 复制一个容器隐含着为它的所有元素分配空间，并完成每个元素的复制工作，代价可能非常高

```
template <class Key, class T, class Cmp =less<Key>,
    class A =allocator<pair<const Key,T> > > class map
{
public:
    // 构造、复制、析构等函数
    explicit map(const Cmp& =Cmp(), const A& =A());
    template <class In>map(In first, In last, const Cmp& =Cmp(),
        const A& =A());
    map(const map&);
    ~map();
    map& operator=(const map&);
};
```

17.4.1.5 比较

- 为了能在map中找到对应于给定关键码的元素，map必须能够做关键码比较，此外，迭代器也会按照递增的方式遍历map，因此，插入元素时通常也需要元素比较
- 按默认规定，关键码比较采用<，map的value_type是(关键码、值)对偶，因此提供了value_comp()来进行比较

比较

```
template <class Key, class T, class Cmp = less<Key>,
   class A = allocator< pair<const Key,T> > >class map
{
public:
     typedef Cmp key_compare;
     class value_compare : public
        binary_function<value_type,value_type,bool>{
     friend class map;
     protected:
          Cmp cmp;
          value_compare(Cmp c) : cmp(c) {}
     public:
          bool operator()(const T& x, const T& y) const
               { return cmp(x.first, y.first) ; }
     };
     key_compare key_comp() const;
     value_compare value_comp() const;
};
```


比较

```
map<string,int> m1;
map<string,int,Nocase> m2; // 描述比较类型（§17.1.4.1)
map<string,int,String_cmp> m3; // 明确比较类型（§17.1.4.1)
map<string,int> m4(String_cmp(literary)); // 传递比较对象
```

有了key_comp()和value_comp()成员函数，就可以要求对一个map的关键码和值做各种比较

常见做法是将另外某个容器或者算法的比较准则传递给map

```
void f(map<string,int>&m)
{
     map<string,int>mm; // 按默认方式使用<做比较
     map<string,int> mmm(m.key_comp()); // 使用m的方式比较
     // …
}
```


17.4.1.6 映射操作

```
template <class Key, class T, class Cmp = less<Key>,
class A = allocator< pair<const Key,T> > > class map {
public:
     // 映射操作
     iterator find(const key_type& k); // 查找关键码为k的元素
     const_iterator find(const key_type& k) const;
     size_type count(const key_type& k) const; // 关键码为k的元素个数
     iterator lower_bound(const key_type& k); // 查找第一个关键码为k的元素
     const_iterator lower_bound(const key_type& k) const;
     iterator upper_bound(const key_type& k); // 找第一个关键码大于k的元素
     const_iterator upper_bound(const key_type& k) const;
     pair<iterator,iterator> equal_range(const key_type& k);
     pair<const_iterator,const_iterator> equal_range(const key_type& k)
     const;
     // …
};
```

映射操作

```
void f(multimap<string,int>&m)
{
    multimap<string,int>::iterator lb =m.lower_bound("Gold");
    multimap<string,int>::iterator ub =m.upper_bound("Gold");
    for (multimap<string,int>::iterator p = lb; p!=ub; ++p) {
    // ...
}
// 使用两个函数找到Gold出现的上界和下届
void f(multimap<string,int>&m)
{
    typedef multimap<string,int>::iterator MI;
    pair<MI,MI> g =m.equal_range("Gold");
    for (MI p = g.first; p!=g.second; ++p) {
        // ...
    }
} // 使用range_equal()一下子算出两个结果
```


17.4.1.7 表操作

```
template <class Key, class T, class Cmp = less<Key>,
  class A = allocator< pair<const Key,T> > > class map
{
    public:// 表操作
    pair<iterator, bool> insert(const value_type& val);
    // 插入(关键码、值)对
    iterator insert(iterator pos, const value_type& val);
    // pos只是一个提示，暗示insert从pos开始查找val的关键码
    template <class In> void insert(In first, In last); // 从序列中插入
    void erase(iterator pos); // 删除被指向的元素
    size_type erase(const key_type& k); // 删除关键码为k的所有元素，返回个数
    void erase(iterator first, iterator last); // 删除区间
    void clear(); // 删除所有元素
};
// 将一个值放入关联容器的最方便方式就是使用下标操作直接赋值
// phone_book["Order department"] = 8226339;
// m[k]的结果等价于(*(m.insert(makepair(k,V())).first)).second
// []操作总需要V()，因此，若map的值类型没有默认值，无法使用下标操作
```


表操作

```
void f(map<string,int>&m)
{
    pair<string,int> p99("Paul",99);
    pair<map<string,int>::iterator,bool> p =m.insert(p99);
    if (p.second) {
        // “Paul”被插入
    }
    else {
        // “Paul”已经存在
    }
    map<string,int>::iterator i = p.first; // 指向 m["Paul"]
    // ...
}
// m.insert(val)的返回值是pair<iterator,bool>，如果val被实际插
入，bool值为true，iterator引用的是m中一个元素，保存着val的关键码
val.first
```


17.4.1.8 其他函数

- map提供了一些处理元素个数的常用函数
- 此外，map还提供了==，!=，<，>，<=，>=和swap()，它们都作为非成员函数

```
template <class Key, class T, class Cmp = less<Key>,
class A = allocator< pair<const Key,T> > > class map
{
public:
    // capacity:
    size_type size() const; // number of elements
    size_type max_size() const; // size of largest possible map
    bool empty() const { return size()==0; }
    void swap(map&);
};
```

17.4.2 多重映射－multimap

- multimap很象map，但是允许重复的关键码
- 注意到insert函数总是真正的插入，所以返回值为iterator，而不是pair

```
template <class Key, class T, class Cmp = less<Key>,
  class A = allocator< pair<const Key,T> > > class std::multimap
{
public:
    // 类似map，除了：
    iterator insert(const value_type&); // 返回iterator，不是pair
    // 无[]运算符
};
```


17.4.3 集合－set

- 一个set也可以被看作是一个map，其中的值是无关紧要的，所以只保存了关键码，这样做只引起了用户界面的少许改动
- 将value_type定义为Key类型，使得map和set的代码在大部分情况下完全一样

```
template <class Key, class Cmp = less<Key>,
   class A = allocator<Key> > class std::set
{
public:
    // 类似map，除了：
    typedef Key value_type; // 关键码本身就是值
    typedef Cmp value_compare;
    // 无[]运算符
};
```


17.4.4 多重集合－multiset

- multiset是一种允许重复关键码的set

```
template <class Key, class T, class Cmp = less<Key>,
   class A = allocator<Key> >  class std::multiset
{
public:
    // 类似set，除了：
    iterator insert(const value_type&); // 返回iterator，不是pair
};
```


17.5 拟容器－Almost Containers

- 内部数组、string、valarray和bitset里也保存元素，因此，很多情况下也可以将它们看作容器
- 它们中的每个都缺乏标准容器的这些或那些特性，所以，这些“拟容器”不像开发完整的容器那样具有完全互换性

17.5.1 串－string

- basic_string提供下标操作、随机访问迭代器以及容器所能提供的几乎所有记法上的方便之处
- 不像容器那样支持广泛的元素类型选择
- 特别为作为字符串的使用做了优化

17.5.2 值向量－valarray

- 为数值计算而优化了的向量
- 提供了许多有用的数值操作
- 只提供了标准容器中的size()和下标操作
- 到valarray中的元素指针是一种随机访问迭代器

17.5.3 位集合－bitset

- C++可以通过整数上的按位运算，有效地支持小的标志集合的概念
- 类bitset<N>推广了这个概念，并通过提供了在N个二进制位的集合(下标从0到N-1)上的各种操作提供进一步的方便，这里的N需要在编译时已知
- 对于无法放进long int的二进制位的集合，用一个bitset比直接用一些整数方便的多，对于更小的集合，这里就有一个效率上的权衡问题
- 如果希望给二进制位命名，而不是给它们编号，请考虑使用set、枚举、或者位域(C.8.1)

位集合－bitset

- bitset<N>是N个二进制位的数组，与vector<bool>的不同之处在于它的大小是固定的；与set的不同点在于它通过下标索引其中的二进制位，而不是通过关键码关联；与vector<bool>和set都不同的地方在于它提供了同时对许多二进制位进行处理的操作
- 通过内部指针不能直接对单个的位寻址，因此，bitset提供了一种位引用类型，这是一种一般性的技术，用于对由于某些原因而使内部指针不能使用的对象寻址
- bitset模板在std名字空间定义，通过<bitset>给出

17.7 忠告

- [1] 如果需要用容器，首先考虑用***vector***
- [2] 了解你经常使用的每个操作的代价(复杂性，大O度量)
- [3] 容器的界面、实现和表示是不同的概念，不要混淆
- [4] 你可以依据多种不同准则去排序和搜索
- [5] 不要用C风格的字符串作为关键码，除非你提供了一种适当的比较准则
- [6] 你可以定义这样的比较准则，使等价的但是不相同的关键码值映射到同一个关键码

忠告

- [7] 在插入和删除元素时，最好是使用序列末端的操作 (***back***操作)
- [8] 当你需要在容器的前端或中间做许多插入和删除时，请用***list***
- [9] 当你主要通过关键码访问元素时，请用***map***或***multimap***
- [10] 尽量用最小的操作集合，以取得最大的灵活性
- [11] 如果要保持元素的顺序性，选用***map***而不是***hash_map***
- [12] 如果查找速度极其重要，选***hash_map***而不是***map***

忠告

- [13] 如果无法对元素定义小于操作时，选***hash_map***而不是***map***
- [14] 当你需要检查某个关键码是否在关联容器里的时候，用***find***()
- [15] 用***equal_range***() 在关联容器里找出所有具有给定关键码的所有元素
- [16] 当具有同样关键码的多个值需要保持顺序时，用***multimap***
- [17] 当关键码本身就是你需要保存的值时，用***set***或***multiset***