

# 序——高质量 C&C++ 编程指南

## 第 1 章 文件结构

1. 版权和版本的声明位于头文件和定义文件的开头（参见示例 1-1），主要内容有：

- (1) 版权信息。
- (2) 文件名称，标识符，摘要。
- (3) 当前版本号，作者/修改者，完成日期。
- (4) 版本历史信息。

```
/*
* Copyright (c) 2001, 上海贝尔有限公司网络应用事业部
* All rights reserved.
*
* 文件名称: filename.h
* 文件标识: 见配置管理计划书
* 摘要: 简要描述本文件的内容
*
* 当前版本: 1.1
* 作者: 输入作者(或修改者)名字
* 完成日期: 2001 年 7 月 20 日
*
* 取代版本: 1.0
* 原作者: 输入原作者(或修改者)名字
* 完成日期: 2001 年 5 月 10 日
*/

```

- 2. 如果一个软件的头文件数目比较多，通常应该将头文件和定义文件分别保存在不同的目录，以便于维护。
- 3. 如果某些头文件是私有的，它不会被用户的程序直接引用，则没必要公开其“声明”。为了加强信息隐藏，这些私有头文件可以和定义文件存放于同一个目录。

## 第 2 章 程序的版式

- 4. 尽可能在定义变量的同时初始化该变量。

## 第3章 命名规则

5. 为了防止某一软件库中的一些标识符和其它软件库中的冲突，可以为各种标识符加上能反映软件性质的前缀。例如三维图形标准 OpenGL 的所有库函数均以 gl 开头，所有常量（或宏定义）均以 GL 开头。

## 第4章 表达式和基本语句

6. 如果循环体内存在逻辑判断，并且循环次数很大，宜将逻辑判断移到循环体外面。
7. 建议 for 语句的循环控制变量的取值采用“半开半闭区间”写法。

## 第5章 常量

8. 在 C++ 程序中尽量用 const 常量代替宏常量。
9. 需要对外公开的常量放在头文件中，不需要对外公开的常量放在定义文件的头部。为了方便管理，可以把不同模块的常量集中存放到一个公共的头文件中。
10. 怎样才能建立在整个类中都恒定的常量呢？别指望 const 数据成员了，应该用类中的枚举常量来实现。**为什么不能用const static常量？**

## 第6章 函数设计

11. 参数的书写要完整，不要省略参数类型而省略参数名字。如果没有参数，则用 void 填充。

```
void SetValue(int width, int height); // 良好的风格  
void SetValue(int, int); // 不良的风格  
float GetValue(void); // 良好的风格  
float GetValue(); // 不良的风格
```

12. 一般的，应将目的参数放在前面，源参数放在后面。**泛型算法刚好跟此规则相反。**
13. 不要将正常值和错误值混在一起返回。正常值用输出参数获得，而错误标志用 return 语句返回。

14. 如果输入参数以值传递方式传递对象，应该用“`const &`”方式来传递，省去临时对象的构造和析构，提高效率。

15. 有时候函数原本不需要返回值，但为了增加灵活性如支持链式表达，可以附加返回值。

```
char *strcpy(char *strDest, const char *strSrc);
```

16. 如果函数的返回值是一个对象，有些场合用“引用传递”替换“值传递”可以提高效率（返回对象的生存期稳定）。而有些场合（返回对象在栈上）只能用“值传递”而不能用“引用传递”，否则会出错，用“值传递”将其拷贝到外部。

17. 在函数体“入口处”对参数的有效性进行检查。

18. 在函数体“出口处”对 `return` 语句的正确性和效率进行检查。

```
return String(s1 + s2);
```

这是临时对象的语法，表示“[创建一个临时对象并返回它](#)”。不要以为它与“先创建一个局部对象 `temp` 并返回它的结果”是等价的，如

```
String temp(s1 + s2);
```

```
return temp;
```

实质不然，上述代码将发生三件事。首先，`temp` 对象被创建，同时完成初始化；然后拷贝构造函数把 `temp` 拷贝到保存返回值的外部存储单元中；最后，`temp` 在函数结束时被销毁（调用析构函数）。然而“[创建一个临时对象并返回它](#)”的过程是不同的，编译器直接把临时对象创建并初始化在外部存储单元中，省去了拷贝和析构的化费，提高了效率。

## 第 7 章 内存管理

19. 内存分配有 3 种

- 1) 静态存储区，存放全局变量和静态变量；
- 2) 栈，存放所有函数的局部变量和函数参数值，内存分配方向是从上到下，释放方式是先入后出，因此可以有效避免内存碎片的产生；
- 3) 堆，存放 `new` 出来的对象，分配方向是从下到上，释放方式是程序员控制，因此有可能产生内存碎片。

另外还有一种不能被程序员控制的内存区域，即常量存储区。

20. 用 `malloc` 或 `new` 申请内存之后，应该立即检查指针值是否为 `NULL`。防止使用指值为 `NULL` 的内存。

21. 不要忘记为数组和动态内存赋初值。防止将未被初始化的内存作为右值使用。

22. 用 `free` 或 `delete` 释放了内存之后，立即将指针设置为 `NULL`，防止产生“野指针”。

23. `char *p = "world"; // 注意 p 指向常量字符串  
p[0] = 'X'; // 编译器不能发现该错误`

24. 当数组作为函数的参数进行传递时，数组自动退化为同类型指针。

```
void Func(char a[100])  
{  
    cout<< sizeof(a) << endl; // 4 字节而不是 100 字节  
}
```

25. 字节对齐：结构，联合，或者类的数据成员，第一个放在偏移为 0 的地方，以后每个数据成员的对齐，按照#pragma pack 指定的数值和这个数据成员自身长度中，比较小的那个进行。而结构整体的对齐，则按照结构体中最大的数据成员和#pragma pack 指定值之间，较小的那个进行。
26. 对于非内部数据类型的对象而言，用 malloc/free 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free。

## 第 8 章 C++ 函数的高级特性

27. C++ 只能靠参数而不能靠返回值类型的不同来区分重载函数。

28. C++ 编译器会为函数生成一个内部名字如 int func(int i)，C++ 编译器会生成\_func\_int，但是 C 编译器只会生成\_func。这是 C++ 实现重载的基础。如果 C++ 程序员要调用已经被编译后的 C 函数，怎么办？

假设某个 C 函数的声明如下：

```
void foo(int x, int y);
```

该函数被 C 编译器编译后在库中的名字为 \_foo，而 C++ 编译器则会产生像 \_foo\_int\_int 之类的名字用来支持函数重载和类型安全连接。由于编译后的名字不同，C++ 程序不能直接调用 C 函数。C++ 提供了一个 C 连接交换指定符号 extern “C” 来解决这个问题。

例如：

```
extern "C"  
{  
    void foo(int x, int y);  
    ... // 其它函数  
}
```

或者写成

```
extern "C"  
{  
    #include "myheader.h"
```

```
... // 其它 C 头文件  
}
```

这就告诉 C++ 编译器，函数 foo 是个 C 连接，应该到库中找名字 \_foo，而不是找 \_foo\_int\_int。C++ 编译器开发商已经对 C 标准库的头文件作了 extern “C” 处理，所以我们可以用 #include 直接引用这些头文件。

## 29. 当心隐式类型转换导致重载函数产生二义性。

```
# include <iostream.h>  
void output( int x); // 函数声明  
void output( float x); // 函数声明  
  
void output( int x)  
{  
    cout << " output int " << x << endl ;  
}  
  
void output( float x)  
{  
    cout << " output float " << x << endl ;  
}  
  
void main(void)  
{  
    int    x = 1;  
    float y = 1.0;  
    output(x);        // output int 1  
    output(y);        // output float 1  
    output(1);        // output int 1  
//    output(0.5);      // error! ambiguous call, 因为自动类型转换  
    output(int(0.5)); // output int 0  
    output(float(0.5)); // output float 0.5  
}
```

## 30. 参数的缺省值只能出现在函数的声明中，不能出现在定义体中。

运算符	规则
所有的一元运算符	建议重载为成员函数
= () [] ->	只能重载为成员函数
+= -= /= *= &=  = ^= %= >>= <<=	建议重载为成员函数
所有其它运算符	建议重载为全局函数

31.

32. 内联关键字 `inline` 必须与定义体放在一起。如果函数体代码太长，或者函数体内出现循环，那么不宜使用内联。

## 第 9 章 类的构造函数、析构函数与赋值函数

33. 类的数据成员的初始化可以采用初始化表或函数体内赋值两种方式，这两种方式的效率不完全相同。非内部数据类型的成员对象应当采用第一种方式初始化，以获取更高的效率。

```
B::B(const A &a)
: m_a(a)
{
...
}
```

类 `B` 的构造函数在其初始化表里调用了类 `A` 的拷贝构造函数，从而将成员对象 `m_a` 初始化。

```
B::B(const A &a)
{
m_a = a;
...
}
```

类 `B` 的构造函数在函数体内用赋值的方式将成员对象 `m_a` 初始化。我们看到的只是一条赋值语句，但实际上 `B` 的构造函数干了两件事：先暗地里创建 `m_a` 对象（调用了 `A` 的无参数构造函数），再调用类 `A` 的赋值函数，将参数 `a` 赋给 `m_a`。

34. 成员对象初始化的次序完全不受它们在初始化表中次序的影响，只由成员对象在类中声明的次序决定。这是因为类的声明是唯一的，而类的构造函数可以有多个，因此会有多个不同次序的初始化表。
35. 如果不主动编写拷贝构造函数和赋值函数，编译器将以“位拷贝”的方式自动生成缺省的函数。
36. 如果我们实在不想编写拷贝构造函数和赋值函数，又不允许别人使用编译器生成的缺省函数，只需将拷贝构造函数和赋值函数声明为私有函数。
37. 析构函数永远声明为 `virtual`。
38. 编写派生类的赋值函数时，不要忘记对基类的数据成员重新赋值。

```
Derived & Derived::operator =(const Derived &other)
{
    // (1) 检查自赋值
    if(this == &other)
```

```

    return *this;

    // (2) 对基类的数据成员重新赋值
    Base::operate =(other); // 因为不能直接操作私有数据成员

    // (3) 对派生类的数据成员赋值
    m_x = other.m_x;
    m_y = other.m_y;
    m_z = other.m_z;

    // (4) 返回本对象的引用
    return *this;
}

```

39. 编写赋值函数时，注意判断自赋值。

```

String & String::operate =(const String &other)
{
    // (1) 检查自赋值
    if(this == &other)
        return *this;

    // (2) 释放原有的内存资源
    delete [] m_data;

    // (3) 分配新的内存资源，并复制内容
    int length = strlen(other.m_data);
    m_data = new char[length+1];
    strcpy(m_data, other.m_data);

    // (4) 返回本对象的引用
    return *this;
}

```

## 第一篇 C++概述

### 第 1 章 开始

40. 标准 C++ 中，如果 main() 没有返回语句，则它缺省返回 0。

41. C++文件在不同的实现版本中是不同的，所以标准C++头文件没有后缀名。而实现文件一般为 cpp/cxx。

42. 用以下代码来防止头文件的重复包含。只要不存在“[两个必须包含的头文件要检查一个同名的预处理常量](#)”这样的情形，这个策略就能很好的运作。

```
#ifndef XXX  
#define XXX  
/*头文件内容*/  
#endif
```

43. 编译 C++程序时，编译器自动定义了一个预处理常量 `_cplusplus`，可以根据它来判断该程序是否是 C++ 程序。在编译标准 C 时，编译器将自动定义名字 `_STDC_`，当然，`_cplusplus` 与 `_STDC_` 不会同时被定义。

```
#ifdef __cplusplus  
// 不错 我们要编译 C++  
// extern "C" 到第 7 章再解释  
extern "C"  
#endif
```

44. 几个有用的预定义名字是：`_LINE_`, `_FILE_`, `_TIME_`, `_DATE_`。

45. 对于一个 C 库的头文件一般有 2 个名字，C 名字和 C++名字。C 库头文件的 C++名字总是以 C 开头，后面是去掉后缀.h 的 C 库 C 名字。使用 C++名字时，因为所有 C++库名字是在名字空间 std 中被定义的，所以记得加上 `using namespace std`。

## 第 2 章 C++浏览

46. C++不允许成员函数与数据成员共享一个名字。

47. 类定义以及相关的常数值或 `typedef` 名通常都存储在头文件中，并且头文件以类名来命名。

48. 基类的构造函数(全部)、析构函数、拷贝赋值函数都没有被派生类继承。

49. 虚拟函数不能被内联，因为内联发生在编译时刻，而虚拟函数是在运行时刻被处理的。

50. 要使用泛型算法，必须`#include<algorithm>`。

# 第二篇 基本语言

## 第 3 章 C++ 数据类型

51. 典型情况下，float 是一个字，double 是两个字，long double 是三个或四个字。
52. 文字常量是不可寻址的，尽管它的值也存储在机器内存的某个地方，但是我们没有办法访问他们的地址。
53. 文字常量后缀：**U,UL,F,L**，不区分大小写，但是 L 最好用大写，以免与数字 1 混淆。注意，后缀只能用在十进制形式中。
54. 转义字符采用如下格式：\ooo，这里 ooo 代表三个八进制数字组成的序列。
55. 字符文字前面可以加“L”，例如：L'a'，称为宽字符文字，类型为 wchar\_t；宽字符常量用来支持某些字符集，如汉语和日语，这些语言中的某些字符不能用单个字符来表示。同样的，也有宽字符串，如：L"abc"。
56. 一个字符串文字可以扩展到多行。在一行的最后加上一个反斜杠，表明字符串文字在下一行继续。例如：

```
"a multi-line \
string literal signals its \
continuation with a backslash"
```

57. 如果两个字符串或宽字符串在源代码中相邻，C++ 就会把它们连接在一起，并在最后加上一个空字符。如

```
"two"    "some"
它代表了"twosome"
```

表 3.1 C++关键字

asm	auto	bool	break	case
catch	char	class	const	const_cast
Continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast
struct	switch	template	this	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
volatile	wchar_t	while		

58. 每种内置数据类型都支持“默认构造函数语法”，可将对象初始化为 0

```
// 设置 ival 为 0 dval 为 0.0
int ival = int();
double dval = double();
```

59. void\*类型指针，它可以被任何数据指针类型的地址值赋值，函数指针不能赋值给它。

60. string 类的 c\_str() 函数返回的类型是 const char\*。

61. A 引用类型的变量不能用 B 类型变量初始化，但是 const 引用可以用不同类型的对象初始化（只要能从一种类型转换到另一种类型即可），也可以是不可寻址的值，如文字常量。

```
double dval = 3.14159;

// 仅对于 const 引用才是合法的
const int &ir = 1024;
const int &ir2 = dval;
const double &dr = dval + 1.0;
```

原因有些微妙：引用在内部存放的是一个对象地址，它是该对象的别名。对于不可寻址的值，如文字常量，以及不同类型的对象，编译器为了实现引用，必须生成一个临时对象，引用实际上指向该对象，但用户不能访问它。例如，当我们写：

```
double dval = 1024;
const int &ri = dval;
```

编译器将其转换成

```
int temp = dval;  
const int &ri = temp;
```

如果我们给 ri 赋值，则这样做不会改变 dval，而是改变 temp。这对程序员来说不是好事。但是 const 引用不会暴露这个问题，它是只读的，它不允许赋值。

62. 下面的例子很难第一次就能正确声明。

```
const int ival = 1024;  
// 声明一个引用，指向 ival 的地址  
  
int *&pi_ref = &ival;  
// 错误  
  
const int ival = 1024;  
// 仍然错误，这是一个非 const 引用，引用了一个 const int*  
const int *&pi_ref = &ival;  
  
const int ival = 1024;  
// ok：这是可以被编译器接受的，一个 const 引用，引用了一个 const int*  
const int *const &pi_ref = &ival;
```

63. 枚举类型可以隐式转换为 int，但是 int 不能转换为枚举类型。

64. C++ 不允许声明一个引用数组。

65. 使用 vector 有两种不同的形式，即所谓的数组习惯和 STL 习惯。

66. typedef 不是宏，如下例

```
typedef char *cstring;  
在以下声明中 cstr 的类型是什么  
extern const cstring cstr;  
第一个回答差不多都是  
const char *cstr // 错  
正确答案是  
char *const cstr; // 正确
```

67. volatile 对象提示编译器“该对象可能在编译器的控制或检测之外被改变”，编译器执行的某些例行优化行为不能应用在 volatile 对象上。

68. 应用在引用类型上的 sizeof 返回的是被引用的对象所需的内存长度。

69. 由于位操作符在较低的层次上操纵位，所以它比较容易出错，最好将它包装成宏或内联函数。

70. bitset 用法

表格 4.4 bitset 操作

操作	功能	用法
test( pos )	pos 位是否为 1?	a.test( 4 )
any()	任意位是否为 1?	a.any()
none()	是否没有位为 1?	a.none()
count()	值是 1 的位的个数	a.count()
size()	位元素的个数	a.size()
[pos]	访问 pos 位	a[ 4 ]
flip()	翻转所有的位	a.flip()
flip( pos )	翻转 pos 位	a.flip( 4 )
set()	将所有位置 1	a.set()
set( pos )	将 pos 位置 1	a.set( 4 )
reset()	将所有位置 0	a.reset()
reset(pos)	将 pos 位置 0	a.reset( 4 )

```
bitset< 32 > bitvec;  
  
bitvec.flip( 0 ); // 翻转第一位  
bitvec[0].flip(); // 也是翻转第一位  
bitvec.flip(); // 翻转所有的位的值
```

```
bitset< 32 > bitvec2( 0xffff );  
将 bitvec2 的低 16 位设为 1  
00000000000000001111111111111111  
下面的 bitvec3 的定义  
bitset< 32 > bitvec3( 012 );  
将第 1 和 3 位的值设置为 1 假设位置从 0 开始计数  
000000000000000000000000000000001010
```

我们还可以标记用来初始化 bitset 的字符串的范围，例如，在下面的语句中

```
// 从位置 6 开始，长度为 4: 1010  
string bitval( "1111110101100011010101" ); //高位-----低位  
bitset< 32 > bitvec5( bitval, 6, 4 );
```

```
// 从位置 6 开始, 直到字符串结束: 1010101
string bitval( "1111110101100011010101" );
bitset<32> bitvec6( bitval, 6 ); //从位置 6 到位置 0
```

我们还可以传递一个代表 0 和 1 的集合的字符串参数来构造 bitset 对象，如下所示

```
string bitval( "1010" );
bitset<32> bitvec4( bitval );
bitset 类支持两个成员函数，它们能将 bitset 对象转换成其他类型，一种情况是用 to_string()
string bitval( bitvec3.to_string() );
另一种情况是用 to_ulong()操作
```

## 第 4 章 表达式

71. sizeof 操作符的返回值类型是 size\_t，这是一种与机器相关的 typedef 定义。
72. 所有含有小于整形的有序类型的算术表达式，在计算前，其类型都会被转换成整形。这叫做整值提升。
73. wchar\_t、bool 和枚举类型被提升为能够表示其底层类型所有值的最小整数类型。如 bool 类型的底层类型是 char，则 bool 类型转换为 int。
74. long 类型的一般转换有一个例外，如果一个操作数是 long 型，而另一个是 unsigned int 型，那么，只有机器上的 long 型足够长以便能够存放 unsigned int 的所有值时（一般来说，在 32 位操作系统中 long 型和 int 型都用一个字长来表示，所以不满足这里的假设条），unsigned int 才会被转换为 long 型，否则两个操作数都被提升为 unsigned long 型。
75. 任何非 const 数据类型的指针都可以被赋值给 void\*型的指针。

```
const int *pci = &ival;
void* pv = pci; // 错误: pv 不是一个 const void*.
const void *pcv = pci; // ok
```

76. const\_cast 用来转换掉表达式的常量性以及 volatile 对象的 volatile 性。试图用其他三种形式来转换掉常量性会引起编译错误，用 const\_cast 来执行一般的类型转换，也会引起编译错误。
77. 编译器隐式执行的任何类型转换都可以由 static\_cast 显示完成。
78. 行为不佳的静态转换

将 void\*型的指针强制转换成某种显示指针类型

**把一个算术值强制转换成枚举值**

**把一个基类强制转换成派生类或者这种类的指针或引用**

即特殊转一般是安全的，一般转特殊是危险的。

79. `reinterpret_cast` 通常对于操作数的位模式执行一个比较低层次的重新解释，它的正确性很大程度上依赖于程序员的主动管理。如

```
complex<double> *pcom;  
char *pc = reinterpret_cast< char*>( pcom );
```

80. `dynamic_cast` 支持在运行时刻识别由指针或引用指向的类对象。

## 第 5 章 语句

81. `isalpha()` 是标准 C 库的一个函数，为使用它，程序员必须包含 `ctype.h`。

82. 把一条声明语句放在与 `case` 或 `default` 相关联的语句中是非法的，除非它被放在一个语句块中。

```
case illegal_definition:  
    // 错误：声明语句必须被放在语句块中  
    string file_name = get_file_name();  
  
    // ...  
    break;
```

以上语句的意思是：`file_name` 在 `switch` 中可见，进入 `switch` 时，`file_name` 进栈初始化或调用构造函数，但是该 `case` 语句又要求只能在 `case` 满足时才进栈初始化或调用构造函数。所以错误。

83. 标号 `label` 语句只能用作 `goto` 的目标，必须由冒号结束，且标号语句不能紧接在结束右花括号的前面，在标号语句后面加一个空语句，是处理这种限制的典型方法，例如

```
end: ;    // 空语句  
}
```

84. `goto` 语句不能向前跳过没有被语句块包围的声明语句。然而，向后跳过一个已被初始化的对象定义不是非法的。

## 第 6 章 抽象容器类型

85. `map` 和 `set` 中，每个键只允许出现一次。`multimap` 和 `multiset` 支持同一个键的多次出现。

86. “连续存储的容器类型”有一个叫“容量”概念，list 不要求容量。容量是指在下一次需要增长自己之前能够被加入到容器中的元素总数。为了知道一个容器的容量，调用 capacity()。在 Rogue Wave 实现版本下，vector 对象定义后，其长度和容量都是 0，但是在插入第一个元素后，容量是 256，长度是 1。在 VC6.0 的实现版本中，插入第 1 个元素(int)后，容量是 1，2 容量 2，3 容量 4，4 容量 4，5 容量 8.....

表格 6.1 长度、容量、以及各种数据类型

数据类型	长度(字节)	初始插入后的容量
int	4	256
double	8	128
简单类 (simple class) #1	12	85
String	12	85
大型简单类 (large simple class)	8000	1
大型复杂类 (large complex class)	8000	1

表格 6.2 插入 1 千万个元素所需的时间

数据类型	list(s)	vector
int	1038	3.76
double	10.72	3.95
简单的类 (simpleclass)	12.31	5.89
string	14.42	11.80

表格 6.3 插入 1 万个元素的时间

数据类型	list(s)	vector
大型简单类 (large simple class)	0.36	2.23
大型复杂类 (large complex class)	2.37	6.70

在 list 中插入一个元素时，需要分配一次内存，然后调用元素的拷贝构造函数；而 vector 只在容量扩充时分配内存并调用拷贝构造函数，但是它在分配内存时要重新调用拷贝构造函数将之前的内容复制到新的内存中。当元素是内置类型对象或简单类时，vector 较好，它省去了许多分配内存的操作；当元素是复杂类型时，list 较好，因为这时 vector 多调用一次构造函数的毛病体现出来了。

87. `reserve()` 操作允许程序员将容器的容量设置成一个显示指定的值。增加大型复杂类的容量，会大大改善性能。

88. `svec.resize( 2 * svec.size() );` 将 svec 的长度加了一倍，每个新元素都被初始化为“与元素底层类型相关联的缺省值”。也可以 `svec.resize( 2 * svec.size(), "piglet" );` 将新元素都初始化为”piglet”。

89. 除了 iterator 类型，每个容器还定义了一个 `const_iterator`，它对遍历 const 容器是必需的。

90. iterator 算术运算不适用于 list。

91. istream\_iterator 用法

```
#include <vector>
#include <string>
#include <iterator>

int main()
{
    // 将字符串输入流 iterator 绑定到标准输入上，以字符串来解释流的每个元素
    istream_iterator<string> infile( cin );

    // 标记流结束位置的输入流 iterator
    istream_iterator<string> eos;

    // 利用通过 cin 输入的值初始化 svec
    vector<string> svec( infile, eos );

    // 处理 svec
}
```

92. 插入操作的一些用法

```
vector< string > svec;
list< string > slist;

string spouse( "Beth" );
slist.insert( slist.begin(), spouse );
svec.insert( svec.begin(), spouse );
//-----
string son( "Danny" );
list<string>::iterator iter;

iter = find( slist.begin(), slist.end(), son );
slist.insert( iter, spouse );
//-----
vector<string> svec;
...
```

```

string anna( "Anna" );
svec.insert( svec.begin(), 10, anna );
//-----
string sarray[4] = { "quasi", "simba", "frollo", "scar" };
//我们可以向字符串 vector 中插入数组中的全部或部分元素
svec.insert( svec.begin(), sarray, sarray+4 );

svec.insert( svec.begin() + svec.size()/2, sarray+2, sarray+4);

```

### 93. 删 除 操 作 的 一 些 用 法

```

string searchValue( "Quasimodo" );

list< string >::iterator iter = find( slist.begin(), slist.end(), searchValue );

if ( iter != slist.end() )
    slist.erase( iter );
//-----
// 删除容器中的所有元素
slist.erase( slist.begin(), slist.end() );

// 删除由 iterator 标记的一段范围内的元素
list< string >::iterator first, last;
first = find( slist.begin(), slist.end(), val1 );
last = find( slist.begin(), slist.end(), val2 );

// ... 检查 first 和 last 的有效性
slist.erase( first, last );

```

### 94. 标 准 库 函 数 支 持 `istream& getline( istream &is, string str, char delimiter )`

```

// 返回值是指向 string vector 的指针
vector<string,allocator>* retrieve_text()
{
    string file_name;
    cout << "please enter file name: ";
    cin >> file_name;

    // 打开文本文件以便输入 ...
    ifstream infile( file_name.c_str(), ios::in );
    if ( !infile ) {
        cerr << "oops! unable to open file "
            << file_name << "-- bailing out!\n";
        exit( -1 );
    }
}

```

```

else cout << '\n';
vector<string, allocator> *lines_of_text = new vector<string,allocator>;
string textline;
typedef pair<string::size_type, int> stats;
stats maxline;
int linenum = 0;

while ( getline( infile, textline, '\n' ) ) {
    cout << "line read: " << textline << '\n';
    if ( maxline.first << textline.size() ) {
        maxline.first = textline.size();
        maxline.second = linenum;
    }

    lines_of_text->push_back( textline );
    linenum++;
}
return lines_of_text;
}

```

95. 一些 string 用法,注意 erase 用法:

```

iterator erase( iterator pos );
iterator erase(iterator start, iterator end );
basic_string &erase( size_type index = 0, size_type num = npos );
// 程序片断
vector<string> words;
while (( pos = textline.find_first_of( ' ', pos ) ) != string::npos )
{
    words.push_back( textline.substr( prev_pos, pos-prev_pos));
    prev_pos = ++pos;
}
//-----
void filter_text( vector<string> *words, string filter )
{
    vector<string>::iterator iter = words ->begin();
    vector<string>::iterator iter_end = words ->end();

    // 如果用户没有提供 filter, 则缺省使用最小集
    if ( !filter.size() )
        filter.insert( 0, "\",," );

    while ( iter != iter_end ) {
        string::size_type pos = 0;

```

```

// 对于找到的每个元素，将其删除
while (( pos = (*iter).find_first_of( filter, pos ) ) != string::npos )
    (*iter).erase(pos,1);
iter++;
}
}

//-----
string::size_type pos3 = word.size()-3;
string ies( "ies" );
if ( ! word.compare( pos3, 3, ies ) ) {
    word.replace( pos3, 3, 1, 'y' );

    return;
}

//-----
string::size_type spos = 0;
string::size_type pos3 = word.size()-3;

// "ous", "ss", "is", "ius"
string suffixes( "oussisius" );

if ( ! word.compare( pos3, 3, suffixes, spos, 3 ) ||           // ous
     ! word.compare( pos3, 3, suffixes, spos+6, 3 ) ||      // ius
     ! word.compare( pos3+1, 2, suffixes, spos+2, 2 ) || // ss
     ! word.compare( pos3+1, 2, suffixes, spos+4, 2 )) // is
    return;

//-----
string_object.insert(
    string_object.size(), // string_object 中的位置
    new_string, pos,      // new_string 的开始位置
    posEnd-pos // 要拷贝字符的数目
)
//-----
s3.assign( s1, 0, 4 ).append( '' ).append( s2, 0, 4 );
beauty.assign( s2, s2.begin()+4, s2.end() );
//-----

用[]可能发生越界，另一个可替代的 at()操作提供了运行时刻对索引值的范围检查，如果索引是有效的，则 at()返回相关的字符元素：与下标操作符的方式相同，但是，如果索引无效则 at()抛出 out_of_range 异常。
void mumble( const string &st, int index )
{
    try {
        char ch = st.at(index);
        // ...
    }
}

```

```
    }
    catch( std::out_of_range ) { ... }
    // ...
}
//-----
```

96. 标准 C++ 库定义了一个 `ctype` 类，它封装了标准 C 库函数的功能以及一组非成员函数，如 `toupper()`、`tolower()` 等等。为了使用它，必须`#include<locale>`

97. 当我们写如下语句时 `word_count[ string("Anna") ] = 1;` 将发生以下事情：

- 1.一个未命名的临时 `string` 对象被构造并传递给与 `map` 类相关联的下标操作符,这个对象用“Anna”初始化;
- 2.在 `word_count` 中查找 `Anna` 项,没有找到;
- 3.一个新的键/值对被插入到 `word_count` 中,当然,键是一个 `string` 对象,持有”Anna”,但是,值不是 1,而是 0 ;
- 4.插入完成,接着值被赋为 1。

98. 一种比较好的插入单个元素的方法如下所示:

```
// the preferred single element insertion method
word_count.insert( map<string,int>::value_type( string("Anna"), 1 ) );
```

99. 从 `map` 中删除元素

```
string removal_word;
cout << "type in word to remove: ";
cin >> removal_word;
if ( text_map->erase( removal_word ) )
    cout << "ok: " << removal_word << " removed\n";
else cout << "oops: " << removal_word << " not found!\n";
```

100. 一段代码

```
typedef set< string >::difference_type diff_type;
set< string > exclusion_set;
ifstream infile( "exclusion_set" );

if ( !infile )
{
    static string default_excluded_words[25] = {
        "the", "and", "but", "that", "then", "are", "been",
        "can", "can't", "cannot", "could", "did", "for",
        "had", "have", "him", "his", "her", "its", "into",
        "were", "which", "when", "with", "would"
    };
}
```

```

        cerr << "warning! unable to open word exclusion file! -- "
        << "using default set\n";
        copy( default_excluded_words, default_excluded_words+25, inserter(exclusion_set,
exclusion_set.begin() ));
    }
else {
    istream_iterator<string,diff_type> input_set(infile),eos;
    copy( input_set, eos, inserter( exclusion_set, exclusion_set.begin() ) );
}

```

101. 在 multimap 和 multiset 中，相同的 key 之间是连续的。

```

typedef multimap< string,string >::iterator iterator;
pair< iterator, iterator > pos;

pos = authors.equal_range( search_item );
for ( ; pos.first != pos.second; pos.first++ )
    // 对每个元素进行操作

```

102. multimap 不支持下标操作符。

103. stack 是一种**容器适配器**，因为它是抽象在底层容器集上的，缺省情况下，它由 deque 实现。如果想用 list 实现，可以这样写：

```
stack< int, list<int> > intStack; //注意，最后两个尖括号之间要有空格
```

104. priority\_queue 的 push，是根据优先级排序的，默认情况下，根据元素类型相关的小于操作符执行。

## 第三篇 基于过程的程序设计

### 第 7 章 函数

105. 当函数的参数是一个数组的引用时，数组长度成为参数和实参类型的一部分，编译器检查数组实参的长度与在函数参数类型中指定的长度是否匹配。

```

void putValues( int (&arr)[10] );
int j[ 2 ];
putValues(j); // 错误：实参不是 10 个 int 的数组
void putValues( int[], int size );

```

106. 当参数是多维数组时，必须指明除第一维外的所有维长度。

```
void putValues( int matrix[][1a], int rowSize );
```

107. 函数的后继声明中，不能重新定义已设置的缺省实参，但可以重新定制其他缺省实参。

已知下列在头文件 ff.h 中声明的函数声明

```
int ff( int a, int b, int c = 0 ); // ff.h
```

怎样重新声明 ff(), 来把缺省实参提供给 b, 下列语句是错误的，因为它重新指定了 c 的缺省实参

```
#include "ff.h"
```

```
int ff( int a, int b = 0, int c = 0 ); // 错误
```

下列看起来错误的重新声明实际上是正确的

```
#include "ff.h"
```

```
int ff( int a, int b = 0, int c ); // ok
```

108. `int printf( const char* , ... );` 中,逗号是可选的，省略号挂起类型检查机制。

109. 链接指示符不能放在函数体中，最好放在头文件中。extern “C”是唯一被保证由所有 C++ 实现都支持的连接指示符。

```
// 单一语句形式的链接指示符
```

```
extern "C" void exit(int);
```

```
// 复合语句形式的链接指示符
```

```
extern "C" {
```

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

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

```
}
```

```
// 复合语句形式的链接指示符
```

```
extern "C" {
```

```
#include <cmath>
```

```
}
```

110. `int main( int argc, char *argv[] )` 中，`argv[0]`总是被设置为当前正被调用的命令。从索引 1 到 `argc-1` 表示被传递给命令的实际选项。

111. 定义一个命令行参数类

```
class CommandOpt {  
public:  
    CommandOpt() : _limit(-1), _debug_on(false) {}  
    int parse_options( int argc, char *argv[] );  
    string out_file() { return _out_file; }  
    bool debug_on() { return _debug_on; }  
    int files() { return _file_names.size(); }
```

```

// 访问 _file_names
string& operator[]( int ix );
private:
    inline int usage( int exit_value = 0 );

    bool    _debug_on;
    int     _limit;
    string  _out_file;
    vector<string, allocator> _file_names;

    static const char *const program_name;
    static const char *const program_version;
};

```

112. 省略号是函数类型的一部分

```

int printf( const char*, ... );
int strlen( const char* );

int (*pfce)( const char*, ... ); // 可以指向 printf()
int (*pfc)( const char* );      // 可以指向 strlen()

```

113. C.A.R.Hoare 的快速排序算法

```

void sort( string *s1, string *s2, PFI2S compare = lexicoCompare )
{
    // 递归的停止条件
    if( s1 < s2 ) {
        string elem = *s1;
        string *low = s1;
        string *high = s2 + 1;

        for (;;) {
            while ( compare( *++low, elem ) < 0 && low < s2 ); // 找到左边第一个>elem
            while ( compare( elem, *--high ) < 0 && high > s1 ); // 找到右边第一个<elem
            if ( low < high )
                low->swap(*high);
            else break;
        } // end, for(;)

        s1->swap(*high); // (*s1) 已经被放到正确的位置
        sort( s1, high - 1, compare );
        sort( high + 1, s2, compare );
    } // end, if( s1 < s2 )
}

```

114. 函数参数的类型如果是函数类型，函数类型的参数将被自动转换成该函数类型的指针。

然而函数返回值不能是函数类型，只能是函数指针，否则将导致编译出错。

```
typedef int functype( const string &, const string & );
void sort( string *, string *, functype );
    编译器把 sort() 当作已经声明为
void sort( string *, string *, int (*)( const string &, const string & ) );
// typedef 表示一个函数类型
typedef int func( int*, int );
func ff( int ); // 错误: ff() 的返回类型为函数类型
```

115. 指向 C 函数的指针与指向 C++ 函数的指针是不同类型的。

```
void (*pf1)(int);
extern "C" void (*pf2)(int);

int main() {
    pf1 = pf2; // 错误: pf1 和 pf2 类型不同
    // ...
}
```

在 C 函数指针与 C++ 函数指针有相同特性的编译器实现中，编译器可能会支持一种语言扩展，允许向 f1() 传递一个 C++ 函数指针作为实参

116. 当链接指示符应用在一个声明上，所有被它声明的函数都将受到链接指示符的影响。

```
// pfParm 是一个指向 C 函数的指针
extern "C" void f1( void(*pfParm)(int) );

如何声明一个含有 C 函数指针的 C++ 函数呢？利用 typedef
extern "C" typedef void FC( int );

// f2() 是一个带有一个参数的 C++ 函数，参数是一个 C 函数指针
void f2( FC *pfParm );
```

## 第 8 章 域和生命周期

117. if 语句的条件中定义的变量，在 if 语句和相关的 else 语句，以及这些语句内部的嵌套域中都可见。

118. 全局对象和函数在整个程序中只能有一个定义。Inline 函数和常量可以被定义多次，但每次定义必须一样。

119. 未经初始化的静态局部对象会被程序自动初始化为 0。相反，自动对象的值会是任意的。

120. delete pi 前不用判断 if( pi != 0 )。

121. 当一个 auto\_ptr 对象被另一个 auto\_ptr 对象初始化或赋值时，左边被赋值或初始化的对象就有了堆上底层对象的所有权，而右边 auto\_ptr 对象则撤销所有责任。

122. get()返回 auto\_ptr 对象内部的底层指针。我们不能够在 auto\_ptr 对象被定义之后，再用 new 表达式创建对象的地址来直接向其赋。

```
// 没有指向任何对象
auto_ptr< int > p_auto_int;
if ( p_auto_int.get() != 0 && *p_auto_int != 1024 )
    *p_auto_int = 1024;
else
    // ok, 让我们设置 p_auto_int 的底层指针
    p_auto_int.reset( new int( 1024 ) );
```

123. 注意，对于用 new 分配的数组，只有第一维可以用运行时刻计算的表达式来指定，其他维必须是在编译时刻已知的常量值。

```
int (*pia2)[ 1024 ] = new int[ 4 ][ 1024 ];
```

124. 在堆上创建的 const 对象必须被初始化。我们不能在堆上创建内置类型元素的 const 数组，因为我们不能初始化用 new 表达式创建的内置类型数组的元素。

```
const int *pci = new const int[100]; // 错误
```

125. 定位 new 表达式，不存在于定位 new 表达式相匹配的 delete 表达式

```
#include <new>
// 预分配内存
char *buf = new char[ sizeof(Foo) * chunk ];
// 在 buf 中创建一个 Foo 对象
Foo *pb = new (buf) Foo;
delete[] buf;
```

126. 名字空间定义，名字空间定义是可以累积的

```
// 名字空间的这部分定义了库接口
namespace cplusplus_primer {
    class matrix { /* ... */ };
    const double pi = 3.1416;
    matrix operator+ ( const matrix &m1, const matrix &m2 );
    void inverse ( matrix & );
}

// 名字空间的这部分定义了库实现
namespace cplusplus_primer {
```

```

void inverse ( matrix &m )
    { /* ... */ }
matrix operator+ ( const matrix &m1, const matrix &m2 )
    { /* ... */ }
}

//或者

cplusplus_primer::matrix cplusplus_primer::operator+( const matrix &m1, const matrix &m2 );
注意，上面的语句中参数部分没有名字限定修饰符，它被称为“名字空间成员的简短形式”，
它可以用在参数表和函数体中，不能用在返回值中。

```

## 127. 未命名名字空间将其成员的作用域限制为当前文件可见

```

// ----- SortLib.h -----
void quickSort( double *, double * );
void bubbleSort( double *, double * );
void mergeSort( double *, double * );
void heapSort( double *, double * );

// ----- SortLib.C -----
namespace {
    void swap( double *d1, double *d2 ) { /* ... */ }
}

// 只有下面四个函数使用 swap()
void quickSort( double *d1, double *d2 ) { /* ... */ }
void bubbleSort( double *d1, double *d2 ) { /* ... */ }
void mergeSort( double *d1, double *d2 ) { /* ... */ }
void heapSort( double *d1, double *d2 ) { /* ... */ }

```

## 128. using 指示符使名字空间成员名可见，就好像它们是在名字空间被定义的地方之外被声明的一样。using 声明就好象它们是在该声明所在的域被声明的一样。

using 指示符是域内的，在 foo() 中的 using 指示符只能应用在函数 foo() 块内。
另外，使用限定修饰符不会受 using 指示符的影响。
由 using 指示符引起的二义性错误只有在该名字的使用点检测到。而由 using 声明引起的二义性错误在声明点就能检测到
<pre> namespace blip {     int bi = 16, bj = 15, bk = 23; } int bj = 0;  void manip() {     using namespace blip; // using 指示符 -                         // ::bj 和 blip::bj 之间的冲突只在 bj 被使用时才被检测到     ++bi;           // 设置 blip::bi 为 17     ++bj;           // 错误：二义性，全局 bj 还是 blip::bj? } </pre>

```

++::bj;           // ok: 设置全局 bj 为 1
++blip::bj;       // ok: 使用限定修饰符不受 using 指示符的影响
int bk = 97;      // 局部 bk 隐藏 blip::bk
++bk;             // 设置局部 bk 为 98
}

namespace blip {
    int bi = 16, bj = 15, bk = 23;
    int func(void);
    // 其他声明
}
int bj = 0;
void manip() {
    using blip::bi;   // 函数 manip() 中的 bi 指向 blip::bi
    ++bi;             // 设置 blip::bi 为 17
    using blip::bj;   // 隐藏全局域中的 bj
    using blip::func(void); // 错误
    using blip::func;
    ++bj;             // 设置 blip::bj 为 16
    int bk;            // bk 在局部域中声明
    using blip::bk;   // 错误: 在 manip() 中重复定义 bk
}
int wrongInit = bk; // 错误: bk 在这里不可见 应该用 blip::bk

```

## 第 9 章 重载函数

129. 当一个函数名在一个特殊的域中被声明多次，编译器按以下步骤解释第二个声明：

- 1) 参数表中参数个数或类型不同——视为重载
- 2) 返回类型和参数表精确匹配——视为重复声明
- 3) 参数表相同，返回类型不同——视为错误重复声明
- 4) 只有缺省实参不同——视为重复声明

130. 识别函数声明是否相同时，并不考虑 `const` 和 `volatile`。但是，如果把 `const` 和 `volatile` 应用在指针或引用参数指向的类型上，则在判断函数声明是否相同时，就要考虑 `const` 和 `volatile`。

```

// 声明同一函数
void f( int );
void f( const int );

// 声明了不同的函数
void f( int* );
void f( const int* );

```

```
// 也声明了不同的函数
void f( int& );
void f( const int& );
```

131. 注意，`using` 指示符和 `using` 声明有可能和当前域的同名函数重载。

132. `extern "C"` 只能指定重载函数集中的一个函数。

133. 重载解析：

- 1) 确定候选函数——即确定重载函数集合。候选函数是“在调用点上可见的函数”以及“在实参类型所在的名字空间中声明的同名函数”的集合。

```
namespace basicLib {
    int print( int );
    double print( double );
}

namespace matrixLib {
    class matrix { /* ... */ };
    void print( const matrix & );
}

void display()
{
    using basicLib::print;
    matrixLib::matrix mObj;
    print( mObj );      // 调用 matrixLib::print( const matrix& )
    print( 87 );        // 调用 basicLib::print( int )
}
```

2) 确定可行函数——对于每个可行函数，调用中实参与函数的对应参数类型之间必须存在转换。

3) 确定最佳匹配函数——判断可行参数在“标准转换序列”（与用户自定义转换序列对应，见 223 条）的转换等级和转换次数。转换序列的转换次数的构成：[左值转换]—>[提升或标准转换]—>[限定修饰转换]，转换序列的等级是构成该序列最坏转换的等级，若等级相同，则转换次最少的最优。

134. 参数类型转换

1) 精确匹配：左值转换（左值转换包括：左值到右值的转换、数组到指针的转换、函数到指针的转换），限定修饰转换。

2) 类型转换：提升、标准转换、用户自定义转换  
提升——

Char、`unsigned char` 或 `short` 型的实参被提升为 `int` 型，如果机器上 `int` 型的字长比 `short` 整型的长，则 `unsigned short` 型的实参被提升到 `int` 型，否则，它被提升到 `unsigned int` 型；

float 型的实参被提升到 double 类型;  
枚举类型的实参被提升到下列第一个能够表示其所有枚举常量的类型 int、unsigned int、long 或 unsigned long;  
布尔型的实参被提升为 int 型

标准转换——

有五种转换属于标准转换

1. 整值类型转换，从任何整值类型或枚举类型向其他整值类型的转换，不包括前面提升部分中列出的转换；
2. 浮点转换，从任何浮点类型到其他浮点类型的转换，不包括前面提升部分中列出的转换；
3. 浮点—整值转换，从任何浮点类型到任何整值类型或从任何整值类型到任何浮点类型的转换；
4. 指针转换，整数值 0 到指针类型的转换和任何类型的指针到类型 void\* 的转换；
5. bool 转换，从任何整值类型，浮点类型，枚举类型或指针类型到 bool 型的转换。

3) 无匹配

## 第 10 章 函数模板

135. 模板参数分为模板类型参数和模板非类型参数，模板非类型参数代表了一个常量表达式（按值调用的普通参数）。

```
template <class Type, int size>
    Type min( Type (&arr) [size] );

// 错误: 模板参数名 Type 的非法重复定义
template <class Type, class Type>
    Type min( Type, Type );

// 三个 min() 的声明都指向同一个函数模板
// 模板的前向声明
template <class T> T min( T, T );
template <class U> U min( U, U );
// 模板的定义
template <class Type>
Type min( Type a, Type b ) { /* ... */ }

// 错误: inline 指示符放置的位置错误
inline template <typename Type>
Type min( Array<Type>, int );

// ok: 关键字跟在模板参数表之后
template <typename Type>
```

```
inline Type min( Type, Type );
```

136. 模板类型参数中 `typename` 的必要性，为了分析模板定义，编译器必须能区分哪些是类型，哪些不是类型。而编译器有时候只有在模板实例化之后才能知道这个信息。如下例，只有在实例化后才能找到 `Parm` 的定义，才知道 `Parm::name` 是不是类型。[告诉编译器一个表达式是类型表达式的机制是在表达式前加上关键字 `typename`.](#)

```
template <class Parm, class U>
Parm minus( Parm* array, U value )
{
    Parm::name * p; // 这是一个指针声明还是乘法
}

template <class Parm, class U>
Parm minus( Parm* array, U value )
{
    typename Parm::name * p; // ok: 指针声明
}
```

137. 函数模板在它被调用或取其地址时被实例化

```
template <typename Type, int size>
Type min( Type (&p_array)[size] ) { /* ... */ }

// pf 指向 int min( int (&)[10] )
int (*pf)(int (&)[10]) = &min;

template <typename Type, int size>
Type min( Type (&r_array)[size] ) { /* ... */ }
typedef int (&rai)[10];
typedef double (&rad)[20];
void func( int (*(rai) );
void func( double (*(rad) );
int main() {
    // 错误: 哪一个 min() 的实例?
    func( &min );
}

int main() {
    // ok: 强制转换指定实参类型
    func( static_cast<double(*)>(&min) );
}
```

138. 用函数实参的类型来决定模板实参的类型和值的过程叫做[模板实参推演](#)。

139. 模板实参推演期间决定模板实参的类型时，编译器不考虑函数模板实例的返回类型。

```
double da[8] = { 10.3, 7.2, 14.0, 3.8, 25.7, 6.4, 5.5, 16.8 };
int i1 = min( da ); //min(da)返回 double 类型，然后发生标准转换 double-int
```

140. 模板实参推演的通用算法：

1. 依次检查每个函数实参，以确定在每个函数参数的类型中出现的模板参数。
2. 如果找到模板参数，则通过检查函数实参的类型，推演出相应的模板实参。
3. 函数参数类型和函数实参类型不必完全匹配。下列类型转换可以被应用在函数实参上，以便将其转换成相应的函数参数的类型：
  - 左值转换。
  - 限定修饰转换。
  - 从派生类到基类类型的转换。假定函数参数具有形式  $T<args>$ 、 $T<args>\&$  或  $T<args>^*$ ，则这里的参数表 args 至少含有一个模板参数。
4. 如果在多个函数参数中找到同一个模板参数，则从每个相应函数实参推演出的模板实参必须相同。

(1) T(模板参数), (2) const &T(函数参数模板), (3)  
int(函数实参), (4) const &int(函数参数)

类型转换

(3)+(2) ----->(4)  
若成功，则确定(1)

141. 显式模板实参

```
// min5( unsigned int, unsigned int ) 被实例化
min5< unsigned int >( ui, 1024 );

// T1 不出现在函数模板参数表中
template <class T1, class T2, class T3>
T1 sum( T2, T3 );
unsigned int calc( char ch, unsigned int ui ) {
    // 错误: T1 不能被推演出来
    unsigned int loc1 = sum( ch, ui );
    // ok: 模板实参被显式指定
    // T1 和 T3 是 unsigned int, T2 是 char
    unsigned int loc2 = sum< unsigned int, char, unsigned int >( ch, ui );
}
```

142. 显式模板实参特化（缺省参数）：只需列出不能被隐式推演的模板实参，只能省略尾部的实参。

```
// ok: T3 是 unsigned int
// T3 从 ui 的类型中推演出来
unsigned int loc3 = sum< unsigned int, char >( ch, ui );

// ok: T2 是 char, T3 是 unsigned int
// T2 和 T3 从 pf 的类型中推演出来
unsigned int (*pf)( char, unsigned int ) = &sum< unsigned int >

// 错误: 只能省略尾部的实参
```

```
unsigned int loc4 = sum< ui_type, , unsigned int >( ch, ui );
```

143. 显式模板实参应该只被用在完全需要它来解决二义性，或模板实参不能被推演出来时。因为，我们如果修改程序的用作模板函数实参中的声明，那么可以在不改变函数模板调用的情况下成功编译。

#### 144. 模板编译模式

C++支持2种模板编译模式：包含模式和分离模式。

1. 包含模式下，模板定义在头文件中，每个模板定义被实例化的文件包含，就像内联函数。  
2. 分离模式下，模板定义在实现文件中，就像非内联函数的声明和定义组织方式。

```
// model2.h  
// 分离模式：只提供模板声明  
template <typename Type> Type min( Type t1, Type t2 );  
  
// model2.C  
// the template definition  
export template <typename Type>  
Type min( Type t1, Type t2 ) { /* ... */ }
```

关键字 `export` 告诉编译器在生成被其他文件使用的模板实例时可能需要该模板定义。有些编译器可能支持语言扩展而不要求用关键字 `export`。但是无论如何，`export` 是标准 C++ 做法。

`export` 也可以出现在函数模板的声明中，但不是必须的。  
但是，并不是所有的编译器都支持分离模式，即使支持也未必总能支持的很好。  
经测试，VC6.0 和.NET2008 都不支持分离模式编译。有关编译器和链接器的详细工作情况，见文章《为什么 c++ 编译器不能支持对模板的分离式编译》。

145. 显式实例化声明（只针对包含编译模式）：虽然说对特定的实参，无论在多少个文件中调用函数模板，函数模板只做一次实例化。但具体实例化的时间和调用点是不确定的，而且在实际中，有些编译器会做多次实例化，从它们中选择一个作为最终的实例，其他的都被忽略掉。这不会影响程序的结果，但是会影响程序的编译时间。为解决这个问题，标准 C++ 提供了显式实例化声明，可以帮助程序员控制模板实例化发生的时间。

```
// 显式实例化声明  
template int* sum< int* >( int*, int );
```

对于一个函数模板实例，显式实例化声明在一个程序中只能出现一次。

在显式实例化声明所在的文件中，函数模板定义必须被给出。

显式实例化声明是和编译选项联合使用的：当一个显式实例化声明在程序的文本文件中出现时，为了告诉编译器，一个显式实例化声明已经在另一个文件中出现，而在程序其他文件中使用该模板函数时候不能再对它实例化，我们需要设置编译器的编译选项。该选项压制了程序中模板的隐式实例化，一般为/`ft`-

#### 146. 模板显式特化，为一些特殊的类型参数提供特殊的实现。

```
template <class T1, class T2, class T3>  
T1 sum( T2 op1, T3 op2 );
```

```
// 显式特化声明错误: T1 的模板实参不能被推演出来, 它必须显式指定  
template<> double sum( float, float );
```

```
// ok: T1 的实参被显式指定  
// T2 和 T3 可以从 float 推演出来  
template<> double sum<double>( float, float );
```

```
// ok: 所有实参都显式指定  
template<> int sum<int,char,char>( char , char );
```

模板显示特化定义和普通函数定义的区别: 只能用那 3 种类型转换把函数模板实例的实参转换成相应的函数参数类型。

函数模板特化之前需要该函数模板的声明, 编译器必须知道它是个模板才能对其特化。

一个程序不能对相同的模板实参集的同一模板同时有一个显式特化和一个通用实例化。

```
#include <iostream>
```

```
#include <cstring>
```

```
// 通用模板定义
```

```
template <class T>  
T max( T t1, T t2 ) { /* ... */ }
```

```
int main()
```

```
    // const char* max<const char*>( const char*, const char* ) 的实例
```

```
    // 使用通用模板定义
```

```
    const char *p = max( "hello", "world" );
```

```
    cout << " p: " << p << endl;
```

```
    return 0;
```

```
}
```

```
typedef const char *PCC;
```

// 编译出错

```
template<> PCC max< PCC >( PCC s1, PCC s2 ) { /* ... */ }
```

为了防止有的文件中使用通用模板, 有的文件中使用模板显式特化, 应将模板显式特化的声明放在通用模板的头文件中 (实现也可能在其中)。

## 147. 重载函数模板

```
// 类模板 Array 的定义  
// (introduced in Section 2.4)
```

```
template <typename Type>  
class Array{ /* ... */ };
```

```
// min() 的三个函数模板声明
```

```
template <typename Type>
```

```

Type min( const Array<Type>&, int ); // #1

template <typename Type>
Type min( const Type*, int );           // #2

template <typename Type>
Type min( Type, Type );    // #3

#include <cmath>

int main()
{
    Array<int> iA(1024); // 类实例
    int ia[1024];

    // Type == int; min( const Array<int>&, int )
    int ival0 = min( iA, 1024 );

    // Type == int; min( const int*, int )
    int ival1 = min( ia, 1024 );

    // Type == double; min( double, double )
    double dval0 = min( sqrt( iA[0] ), sqrt( ia[0] ) );

    return 0;
}

```

```

template <typename T>
int min5( T, T );
template <typename T, typename U>
int min5( T, U );

int i;
// 错误: 二义性: 来自 min5( T, T ) 和 min5( T, U ) 的两个可能的实例
min5( 1024, i );

```

要指明哪个函数模板比较好、并消除二义性的唯一方法是显式指定模板实参

// ok: 从 min5( T,U ) 实例化

min5<int, int>( 1024, i );

尽管重载是可能的，但是我们在设计重载函数时，仍然必须小心确保重载是必需的。如上例，因为 min5(T,U) 处理的调用集是由 min5(T,T) 处理的超集，所以应该只提供 min5(T,U) 的声明，而 min5(T,T) 应该被删除。

```

template <typename Type>
Type sum( Type*, int );    //更特化的

```

```

template <typename Type>

```

```

Type sum( Type, int );

int ia[1024];

// Type == int ; sum<int>( int*, int ); or
// Type == int*; sum<int*>( int*, int ); ??
int ival1 = sum( ia, 1024 );//没有二义性

```

为该实例选择的模板函数是**最特化的**。

一个模板要比另一个更特化，两个模板必须有相同的名字、相同的参数个数，对于不同类型的相应函数参数，如上面的 T\* 和 T，一个参数能接受的函数实参，是另一个模板中相应参数能接受的实参的超集。接受更有限的实参集合的模板被称为更特化的。

#### 148. 考虑模板函数实例的重载解析

在构造候选函数集的时候，加入能通过模板实参推演过程的模板实例。
若模板实参推演成功，而被推演的模板实参被显式特化，则进入候选函数集合的是显式特化。
普通函数比模板函数的优先级高，若都是精确匹配或自定义匹配（函数模板实例化后要么是精确匹配，要么是自定义转换），重载解析过程为该调用选择普通函数。
<pre> // 函数模板声明 template &lt;class T&gt; T min( T, T );  // 普通函数声明 int min( int, int ) { }  int ai[4] = { 22, 33, 44, 55 }; int main() {     // 调用普通函数 min( int, int )     min( ai[0], 99 ); } </pre>

重载解析发生在编译阶段，一旦某个调用被函数重载解析过程解析为一个普通函数，如果该程序不包含该函数的定义，也不能回头了。

<pre> // 函数模板 template &lt;class T&gt; T min( T, T ) { .... }  // 这个普通函数在该程序中没有被定义 int min( int ,int ); int ai[4] = { 22, 33, 44, 55 }; int main() {     // 链接错误: min(int, int) 被调用    但是没有被定义     min( ai[0], 99 ); } </pre>
---

总结考虑普通函数和函数模板的函数重载解析步骤：

1. 生成候选函数集

考虑与函数调用同名的函数模板，如果对于该函数调用的实参，模板实参推演能够成功，则实例化一个函数模板，或者对于推演出来的模板实参存在一个模板显式特化，则该模板显式特化就是一个候选函数。

### 2.生成可行函数集，9.3 节所描述

只保留候选函数集中函数参数可以从调用实参转换来

### 3.对类型转换划分等级，9.3 节中描述

- a 如果只选择了一个函数，则调用该函数
- b 如果该调用是二义的，则从可行函数集中去掉函数模板实例，只考虑可行函数集中的普通函数，完成重载解析过程 如 9.3 节中描述
- a 如果只选择了一个函数，则调用该函数
- b 否则，该调用是二义的

## 149. 模板定义中的名字解析

```
template<typename Type>
Type min( Type* array, int size )
{
    Type min_val = array[0];

    for ( int i = 1; i < size; ++i )
        if ( array[i] < min_val )
            min_val = array[i];
    print( "Minimum value found: " ); //不依赖于模板参数的名字 printf
    print( min_val ); //依赖于模板参数的名字 printf
    return min_val;
}
```

模板定义中的名字解析分 2 步：首先，不依赖于模板参数的名字在模板定义时被解析；其次，依赖于模板参数的名字在模板被实例化时被解析。

依赖于模板参数的名字必须在模板实例化点前被声明，而函数实例化点是不确定的，它在一个文本文件中对一个模板参数可能有多个实例化点（前面有讲到，编译器会自动选择一个作为有效的实例化）。所以，**应该在模板的任何一个实例使用之前**，声明依赖于模板参数的名字。

## 150. 名字空间和函数模板

```
// ---- primer.h ----
namespace cplusplus_primer {
    // 模板定义被隐藏在名字空间中
    template<class Type>
        Type min( Type* array, int size ) { /* ... */ }
}

// ---- user.C ----
#include <primer.h>
```

```

int ai[4] = { 12, 8, 73, 45 };
int main() {
    int size = sizeof(ai) / sizeof(ai[0]);
    // 错误:没有找到函数 min()
    min( &ai[0], size );
    using cplusplus_primer::min; // using 声明

    // ok: 指向名字空间 cplusplus_primer 中的 min()
    min( &ai[0], size );
}

// ---- primer.h ----
namespace cplusplus_primer {
    // 模板定义被隐藏在名字空间中
    template<class Type>
        Type min( Type* array, int size ) { /* ... */ }
}

// ---- user.h ----
class SmallInt { /* ... */ };
void print( const SmallInt & );
bool compareLess( const SmallInt &, const SmallInt & );

// ---- user.C ----
#include <primer.h>
#include "user.h"
// 错误: 不是 cplusplus_primer::min() 的特化
template<> SmallInt min<SmallInt>( SmallInt* array, int size ) { /* ... */ }

//正确
namespace cplusplus_primer {
// cplusplus_primer::min() 的特化
template<> SmallInt min<SmallInt>( SmallInt* array, int size ) { /* ... */ }
}

//或者
// cplusplus_primer::min() 的特化
// 此特化的名字被限定修饰
template<> SmallInt cplusplus_primer:: min<SmallInt>( SmallInt* array, int size ) { /* ... */ }

```

## 第 11 章 异常处理

151. 在 C++ 中，异常往往用类来实现

```
// stackExcp.h
class popOnEmpty { /* ... */ };
class popOnFull { /* ... */ };

#include "stackExcp.h"
void iStack::pop( int &top_value )
{
    if( empty() )
        throw popOnEmpty();
    top_value = _stack[ --_top ];
    cout << "iStack::pop(): " << top_value << endl;
}

void iStack::push( int value )
{
    cout << "iStack::push( " << value << " ) \n";
    if( full() )
        throw pushOnFull();
    stack[ _top++ ] = value;
}
```

152. 一个 try 块引入一个局部域，在 try 块内声明的变量不能在 try 块外被引用，包括 catch 子句中。最好声明整个包含在 try 块中的函数，这种组织结构把正常处理代码和异常处理代码分离得最清楚。函数 try 块对类构造函数尤其有用。

```
int main()
try {
    iStack stack( 32 );
    stack.display();
    for( int ix = 1; ix < 51; ++ix )
    {
        // 与以前相同
    }
    return 0;
}
catch (pushOnFull) {
    // 这里不能引用 stack
}
catch ( popOnEmpty ) {
    // 这里不能引用 stack
}
```

153. 在 catch 子句完成它的工作后，程序的执行将在 catch 子句列表的最后子句之后继续进行。catch 子句括号中必须是单个类型声明或单个对象声明。

```
catch ( popOnEmpty ) {      //不能操作 throw 表达式所创建的异常对象
    cerr << "trying to pop a value on an empty stack\n";
}

catch ( pushOnFull eObj ) {   //可以操纵 throw 表达式所创建的异常对象
    cerr << "trying to push the value " << eObj.value()
        << " on a full stack\n";
}
```

154. 异常对象总是在抛出点被创建，即使 throw 表达式不是一个构造函数调用，或者它没有表现出要创建一个异常对象，情况也是如此。

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };
enum EHstate state = noErr;

int mathFunc( int i ) {
    if ( i == 0 ) {
        state = zeroOp;
        throw state; // 创建异常对象，类型为 EHstate，用全局对象 state 初始化
    }
}
```

155. 在查找用来处理被抛出异常的 catch 子句时，因为异常而退出复合语句和函数定义，这个过程被称作**栈展开**。随着栈的展开，在退出的复合语句和函数定义中声明的局部变量的生命期也结束了，在结束之前，C++能保证，它们的析构函数会被调用。

156. 因为编译器无法确定在一个 try 中哪种类型的异常会被抛出，所以它有运行时刻类型检查的要求，否则无法找到 catch 语句。在找不到 catch 语句时，程序会调用 C++ 标准库中定义的函数 terminate()，terminate() 的缺省行为是调用 abort()，之所以不直接用 abort()，就是因为 terminate() 支持运行时机制：当没有处理代码能匹配被抛出的异常时，由它通知用户。

157. 有时候 catch 语句不能完全处理异常，它需要重新抛出，让更上一级的 catch 子句完成异常处理任务。

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };

void calculate( int op ) {
    try {
        // 被 mathFunc() 抛出的异常的值为 zeroOp
        mathFunc( op );
    }
```

```

    catch ( Ehstate eObj ) {
        // 做某些修正
        // 试图修改异常对象
        eObj = severeErr;

        // 希望重新抛出值为 severeErr 的异常对象
        // 但是, 实际抛出的是原始的、被 throw 创建的对象
        throw;
    }
}

```

158. 即使一个函数不能处理被抛出的异常,但是它也可能希望在带着异常退出之前执行一些动作,如释放一个锁,释放一些内存。这时候用 `catch(...)`。`catch(...)`与其他 `catch` 子句联合使用的时候,必须被放在最后,否则会产生一个编译时刻错误。
159. **异常规范**提供一种方案,它能够随着函数声明列出该函数可能抛出的异常。它保证函数不会抛出任何其他类型的异常。这种规范之要求“逃离”函数的异常类型复合规范就行了。空的异常规范表示函数不会抛出任何异常。没指定异常规范表示函数可以抛出任何异常。

```

class iStack {
public:
    // ...
    void pop( int &value ) throw(popOnEmpty);
    void push( int value ) throw(pushOnFull);
};

void recoup( int op1, int op2 ) throw(ExceptionType)
{
    try {
        // ...
        throw string("we're in control");
    }
    // 处理抛出的异常
    catch ( string ) {
        // 做一些必要的工作
    }
} //OK, 正确的程序

```

160. 如果函数声明指定了一个异常规范,则同一函数的重复声明必须指定同一类型的异常规范。

```

// 同一函数的两个声明
extern int foo( int = 0 ) throw(string);

```

```
// 错误: 异常规范被省略  
extern int foo( int parm );
```

161. 如果函数抛出了一个没有被列在异常规范中的异常，程序只有在遇到这种异常被抛出时才能检测出来这种错误，编译阶段不会发现。检测出来之后系统会调用 C++ 标准库中定义的函数 unexpected(), unexpected() 的缺省行为是调用 terminate()。编译阶段只会产生代码，保证违反异常规范的异常被抛出时，调用 unexpected()。
162. 在某些条件下，有必要改变 unexpected() 执行动作，C++ 标准库提供了一种机制，可让我们改变 unexpected() 的缺省行为；156 所说的 terminate() 的缺省行为也可以改变。
163. 在被抛出的异常类型与异常规范中指定的类型之间不允许类型转换。
164. 函数指针的声明也可以给出异常规范。和函数声明一样，同一指针的不同异常规范必须相同。
165. 在函数指针被赋值时，用作右值的指针的异常规范必须比左值的异常规范更严格或一样严格。throw(越少越严格)。

## 第 12 章 泛型算法

```
#include <vector>  
#include <string>  
#include <algorithm>  
#include <iterator>  
  
// 标准 C++ 之前的 <iostream> 语法  
#include <iostream.h>  
  
class GreaterThan {  
public:  
    GreaterThan( int sz = 6 ) : _size( sz ){}  
    int size() { return _size; }  
    bool operator()( const string &s1 ) { return s1.size() > _size; }  
private:  
    int _size;  
};  
  
class PrintElem {  
public:  
    PrintElem( int lineLen = 8 ) : _line_length( lineLen ), _cnt( 0 ){}
```

```

void operator()( const string &elem )
{
    ++_cnt;
    if ( _cnt % _line_length == 0 )
        { cout << '\n'; }
    cout << elem << " ";
}
private:
    int _line_length;
    int _cnt;
};

class LessThan {
public:
    bool operator()( const string & s1, const string & s2 ) { return s1.size() < s2.size(); }
};

typedef vector<string, allocator> textwords;

void process_vocab( vector<textwords, allocator>*&pvec )
{
    if ( ! pvec ) {
        // 给出警告消息
        return;
    }
    vector< string, allocator > texts;
    vector<textwords, allocator>::iterator iter;

    for ( iter = pvec->begin(); iter != pvec->end(); ++iter )
        copy( (*iter).begin(), (*iter).end(), back_inserter( texts ) );

    // 排序 texts 的元素
    sort( texts.begin(), texts.end() );

    // ok, 我们来看一看我们有什么
    for_each( texts.begin(), texts.end(), PrintElem() );
    cout << "\n\n"; // 只是分隔显示输出

    // 删除重复元素
    vector<string, allocator>::iterator it;
    it = unique( texts.begin(), texts.end() );
    texts.erase( it, texts.end() );

    // ok, 让我们来看一看现在我们有什么了
}

```

```
for_each( texts.begin(), texts.end(), PrintElem() );
cout << "\n\n";

// 根据缺省的长度 6 排序元素
// stable_sort() 保留相等元素的顺序
stable_sort( texts.begin(), texts.end(), LessThan() );
for_each( texts.begin(), texts.end(), PrintElem() );
cout << "\n\n";

// 计数长度大于 6 的字符串的个数
int cnt = 0;

cnt = count_if( texts.begin(), texts.end(), GreaterThan() );
cout << "Number of words greater than length six are " << cnt << endl;

static string rw[] = { "and", "if", "or", "but", "the" };
vector<string,allocator> remove_words( rw, rw+5 );
vector<string, allocator>::iterator it2 = remove_words.begin();

for ( ; it2 != remove_words.end(); ++it2 )
{
    int cnt = 0;
    cnt = count( texts.begin(), texts.end(), *it2 );
    cout << cnt << " instances removed: " << (*it2) << endl;
    texts.erase( remove(texts.begin(),texts.end(),*it2), texts.end() );
}

cout << "\n\n";
for_each( texts.begin(), texts.end(), PrintElem() );
}

// difference_type 类型能够包含一个容器的两个 iterator 的减法结果
typedef vector<string,allocator>::difference_type diff_type;

// 标准 C++ 之前的头文件语法
#include <fstream.h>
main()
{
    vector<textwords, allocator> sample;
    vector<string,allocator> t1, t2;
    string t1fn, t2fn;

    // 要求用户输入文件
    // 实际中的程序应该做错误检查
```

```

cout << "text file #1: "; cin >> t1fn;
cout << "text file #2: "; cin >> t2fn;

// 打开文件
ifstream infile1( t1fn.c_str());
ifstream infile2( t2fn.c_str());

// iterator 的特殊形式
// 通常, diff_type 被缺省提供
istream_iterator< string, diff_type > input_set1( infile1 ), eos;
istream_iterator< string, diff_type > input_set2( infile2 );

// iterator 的特殊形式
copy( input_set1, eos, back_inserter( t1 ) );
copy( input_set2, eos, back_inserter( t2 ) );

sample.push_back( t1 );
sample.push_back( t2 );
process_vocab( &sample );
}

```

166. 函数对象与函数指针相比较,有2方面的优点:首先,如果被重载的调用操作符()是 inline 函数,则编译器能够执行内联编译,提供可能的性能好处;其次,函数对象可以拥有任意数目额外数据。

```

template < typename Type, bool (*Comp)(const Type&, const Type&) >
const Type& min( const Type *p, int size, Comp comp )
{
    int minIndex = 0;
    for ( int ix = 1; ix < size; ++ix )
        if ( Comp( p[ ix ], p[ minIndex ] ) )
            minIndex = ix;
    return p[ minIndex ];
}

//-----

template < typename Type, typename Comp >
const Type& min( const Type *p, int size, Comp comp )
{
    int minIndex = 0;
    for ( int ix = 1; ix < size; ++ix )
        if ( Comp( p[ ix ], p[ minIndex ] ) )
            minIndex = ix;
    return p[ minIndex ];
}

```

}

167. 泛型算法支持内置(可能被重载的)操作符, 函数指针或函数对象。函数对象有 3 种来源:

1. 标准库预定义的一组算术、关系和逻辑函数对象;
2. 一组预定义的函数适配器, 允许我们对预定义的函数对象(甚至于任何函数对象)进行特殊化或扩展;
3. 自定义的函数对象。

168. 使用预定义函数对象必须包含`#include<functional>`

STL 预定义算术函数对象

```
plus<Type>
minus<Type>
multiplies<Type>
divides<Type>
modulus<Type>
negate<Type>
```

STL 预定义关系函数对象

```
equal_to<Type>
not_equal_to<Type>
greater<Type>
greater_equal<Type>
less<Type>
less_equal<Type>
```

STL 预定义逻辑函数对象

```
logical_and<Type>
logical_or<Type>
logical_not<Type>
```

STL 预定义的函数对象适配器

bind1st: 绑定一个特殊值到二元函数对象的第一个实参上  
bind2nd: 绑定一个特殊值到二元函数对象的第二个实参上  
not1: 翻转一元函数对象的真值  
not2: 翻转二元函数对象的真值

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

函数对象的实现

一种做法

```
class less_equal_value {
public:
    less_equal_value( int val ) : _val( val ) {}
    bool operator() ( int val ) { return val <= _val; }
private:
    int _val;
};
```

另一种做法

```
template < int _val >
class less_equal_value {
public:
    bool operator() ( int val ) { return val <= _val; }
};
```

169. const 容器只能绑定在 const\_iterator 上。当然,给一个 const\_iterator 赋一个非 const iterator 总是可以的。

170. 插入 iterator

```
int ia[] = { 0, 1, 1, 2, 3, 5, 5, 8 };
vector< int > ivec( ia, ia+8 ), vres;
// 导致未定义的运行时刻行为
unique_copy( ivec.begin(), ivec.end(), vres.begin() );
```

back\_inserter(): 它使用容器的 push\_back() 插入操作代替赋值操作符。

```
unique_copy( ivec.begin(), ivec.end(), back_inserter( vres ) );
```

front\_inserter(): 它使用容器的 push\_front() 插入操作代替赋值操作符。

```
// 哟! 错误
// vector 不支持 push_front() 操作
// 使用 deque 或 list
unique_copy( ivec.begin(), ivec.end(), front_inserter( vres ) );
```

inserter(): 它调用容器的 insert() 插入操作代替赋值操作符。它的第二个实参指示插入的位置，插入的位置并不保持不变，而是随着每个被插入的元素而递增。

```
unique_copy( ivec.begin(), ivec.end(), inserter( vres, vres.begin() ) );
//就好象
vector< int >::iterator iter = vres.begin(), iter2 = ivec.begin();
for ( ; iter2 != ivec.end(); ++iter, ++iter2 )
    vres.insert( iter, *iter2 );
```

171. 反向 iterator

```
vector< int >::reverse_iterator r_iter0 = vec0.rbegin();
vector< int >::const_reverse_iterator r_iter1 = vec1.rbegin();
```

reverse\_iterator 的++实际上 是递减，这有利于让程序员透明的为一个算法传递一对反向 iterator。

```
// 以升序排列 vector
sort( vec0.begin(), vec0.end() );
```

```
// 以降序排列 vector
```

```
sort( vec0.rbegin(), vec0.rend() );
```

### 172. iostream iterator

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
#include <functional>

/*
 * 输入:
 * 23 109 45 89 6 34 12 90 34 23 56 23 8 89 23
 *
 * 输出:
 * 109 90 89 56 45 34 23 12 8 6
 */

int main()
{
    istream_iterator< int > input( cin );
    istream_iterator< int > end_of_stream; //调用缺省构造函数

    vector<int> vec;
    copy ( input, end_of_stream, inserter( vec, vec.begin() ) );

    sort( vec.begin(), vec.end(), greater<int>() );

    ostream_iterator< int > output( cout, " " );
    unique_copy( vec.begin(), vec.end(), output );
}
```

istream\_iterator<Type> identifier( istream& );

应用在 istream\_iterator 对象上的每个递增操作符，都用 operator>>() 读入流的下一个元素。

ostream\_iterator<Type> identifier( ostream& )

ostream\_iterator<Type> identifier( ostream&, char\* delimiter )

应用在 ostream\_iterator 对象上的每个递增操作符，都用 operator<<() 读入流动下一个元素。

### 173. 五种 iterator

1. InputIterator 可以被用来读取容器中的元素，但是不保证支持向容器的写入操作。要求这个层次上提供支持的泛型算法包括 find()、accumulate() 和 equal()。任何一个算法如果要求 InputIterator，那么我们也可以向其传递 3、4、5 项列出的 iterator。

2. OutputIterator 可以被认为与 InputIterator 功能相反的 iterator，它可以被用来向容器中写入元素，但是不保证支持读取容器的内容。如，copy() 取 OutputIterator 作为第三个实参。任

任何一个算法如果要求 OutputIterator，那么我们也可以向其传递 3、4、5 项列出的 iterator。

3. ForwardIterator 可以被用来以某一遍历方向向容器读或写。用到它的有， adjacent\_find()、 swap\_range() 和 replace()。任何要求 ForwardIterator 支持的算法都可以向其传递 4 和 5 中的 iterator。

4. BidirectionalIterator 从两个方向读写一个容器。用到它的算法有 inplace\_merge()、 next\_permutation() 和 reverse()。

5. RandomAccessIterator，除了支持 BidirectionalIterator 所有的功能外，还提供了“在常数时间内访问容器的任意位置”的支持。用到它的算法包括 binary\_search、 sort\_heap()、 nth\_element()。

174. 所有泛型算法前 2 个实参都是一对 iterator，通常被称为 first 和 last。每个算法的声明指示了它所要求支持的 iterator 的最小类别。向其传递一个无效的 iterator 类别引起的错误，不保证会在编译时刻被捕捉到，因为 iterator 类别不是实际的类型，它是被传递给函数模板的类型参数。

#### 175. 何时不用泛型算法

不允许在关联容器上应用重新排序的泛型算法，如 sort() 和 partition()。

因为 list 容器不支持随机访问，所以 merge()、 remove()、 reverse()、 sort() 和 unique() 泛型算法最好不要用在 list 对象上，尽管这些算法都没有显式的要求一个 RandomAccessIterator。标准库为每个算法都提供了专门的 list 函数成员。如下

```
int array1[ 10 ] = { 34, 0, 8, 3, 1, 13, 2, 5, 21, 1 };
int array2[ 5 ] = { 377, 89, 233, 55, 144 };

list< int > ilist1( array1, array1 + 10 );
list< int > ilist2( array2, array2+5 );
```

```
template <class Compare>
```

```
void list::merge( list rhs, Compare comp );
```

```
// merge 要求两个 list 已经排序，调用 merge 后，rhs 的元素被移到 list 中，rhs 变空
ilist1.sort(); ilist2.sort();
ilist1.merge( ilist2 );
```

```
void list::remove( const elemType &value );
```

```
//remove()操作删除指定值的全部实例
ilist1.remove( 1 );
```

```

template < class Predicate >
void list::remove_if( Predicate pred );

class Even {
public:
    bool operator()( int elem ) { return !( elem % 2 ); }
};

ilist1.remove_if( Even() );

void list::reverse();

ilist1.reverse();

template <class Compare>
void list::sort( Compare comp );

list1.sort( greater<int>() );

//把 rhs 全部转移(删掉源)到 list 的 pos
void list::splice( iterator pos, list rhs );
//把 rhs 的 ix 位置的元素转移到 list 的 pos
void list::splice( iterator pos, list rhs, iterator ix );
//把 rhs 的从 first 到 last 转移到 list 的 pos
void list::splice( iterator pos, list rhs, iterator first, iterator last );

template <class BinaryPredicate>
void list::unique( BinaryPredicate pred );

//unique()操作去掉重复的连续拷贝。省情况下,使用底层类型的等于操作符。
ilist.sort();

class EvenPair {
public:
    bool operator()( int val1, int val2 )
        { return !( val2 % val1 ); }
};

ilist.unique( EvenPair() );

```

泛型算法全面介绍见附录

# 第四篇 基于对象的程序设计

## 第 13 章 类

176. 引入类类型后，我们可以以 2 种方式引用它

```
class First obj1;      //第一种，是从 C 中借用的，这种方式在 C++中也有用
Second obj2 = obj1;    //第二种， C++引入的
```

177. 数据成员不能在类体中被显式地初始化，除非是有序类型（如 int）的 static const 数据成员。

```
// 头文件
class Account {
    // ...
private:
    static const int nameSize = 16;    //这是个特例。nameSize 被看作是一个常量表达式
    static const char name[nameSize];
};

// 文本文件
//在类体内初始化一个 static const 时，该成员仍然必须被定在在类定义之外，
const int Account::nameSize;

//静态数据成员如同类成员函数一样，可以引用类的私有成员。静态数据成员的定义是在它的类的域内，当限定修饰名 Account::name 被看到后，它就可以引用 Account 的私有数据成员。
const char Account::name[nameSize] = "Savings Account";
```

178. 友元声明以关键字 friend 开头，它只能出现在类的声明中。

```
class Screen {
    friend istream& operator>>( istream&, Screen& );
    friend ostream& operator<<( ostream&, const Screen& );
public:
    // ... Screen 类的其他部分
};
```

179. 一个类只有完全被定义后，才能定义类类型的对象，因为编译器需要分配存储空间。但是，类类型的指针或引用可以在类定义之前定义。所以一个类不能有自身类型的数据成员，但是，当一个类的类头被看到时，它就被视为已经被声明了，所以一个类可以有指向自身类型的指针或引用作为数据成员。

180. 每个类对象都有自己的类数据成员拷贝，但是每个类成员函数的拷贝只有一份。

181. const 类对象只能调用 const 成员函数(构造函数和析构函数除外，一个 const 类对象“从构造完成时刻到析构开始时刻”这段时间内被认为是 const)，const 成员函数必须在定义和声明中同时指定关键字 const。

Const 成员函数中不能修改类的成员，但可以修改类的成员指向的对象。

```
#include <cstring>
class Text {
public:
    void bad( const string &parm ) const;
private:
    char *_text;
};

void Text::bad( const string &parm ) const
{
    _text = parm.c_str();           // 错误: 不能修改 _text
    for ( int ix = 0; ix < parm.size(); ++ix )
        _text[ix] = parm[ix];      // 不好的风格, 但不是错误的
}
```

Const 成员函数可以被相同参数表的非 const 成员函数重载

```
class Screen {
public:
    char get(int x, int y);
    char get(int x, int y) const;
    // ...
};

char get(int x, int y);    // 调用 const 成员
char get(int x, int y);    // 调用非 const 成员
```

在这种情况下，类对象的常量性决定了调用哪个函数

```
int main() {
    const Screen cs;
    Screen s;

    char ch = cs.get(0,0);    // 调用 const 成员
    ch = s.get(0,0);          // 调用非 const 成员
}
```

182. 同样，对于一个 volatile 类对象，只有 volatile 成员函数、构造函数和析构函数可被调用。

183. 为了允许修改 const 对象的数据成员，可以把该成员声明为 mutable。Mutable 成员即使在一个 const 成员函数中也是可以更新的。

```
inline void Screen::move( int r, int c ) const
{
    // ok: const 成员函数可以修改 mutable 成员
    _cursor = row + c - 1;
    // ...
}
```

184. 静态数据成员的“唯一性”使它与非 static 成员有以下不同：

静态数据成员的类型可以是其所属类，而非 static 数据成员只能被声明为该类的指针或引用。

```
class Bar {
public: 529
    // ...
private:
    static Bar mem1; // ok
    Bar *mem2;     // ok
    Bar mem3;      // 错误
};
```

静态数据成员可以被作为类成员函数的缺省实参，而非 static 成员不能

```
extern int var;
class Foo {
private:
    int var;
    static int stcvar;
public:
    // 错误: 被解析为非 static 的 Foo::var
    // 没有相关的类对象
    int mem1( int = var );

    // ok: 解析为 static 的 Foo::stcvar
    // 无需相关的类对象
    int mem2( int = stcvar );

    // ok: int var 的全局实例
    int mem3( int = ::var );
};
```

185. 静态成员函数

#### 不能将静态成员函数声明为 const 或 volatile

静态成员函数没有 this 指针，因此在静态成员函数中隐式或显式的引用这个指针都将导致编译错误

186. 指向类成员的指针，它们通过特定的对象或指向该类类型的对象的指针来访问。

```
Short Screen::*ps_Screen = &Screen::_height; //指向数据成员  
int (Screen::* pmf)() = &Screen::height; //不能写成&(Screen::height)
```

```
if( (myScreen.*pmf)() == (bufScreen->*pmf)() )  
    (bufScreen->*pmf)(myScreen);
```

```
Screen& Screen::repeat( Action op, int time s )  
{  
    for ( int i = 0; i < times; ++i )  
        (this->*op)();  
  
    return *this;  
}
```

类的静态成员的指针

```
double *pd = &Account::_interestRate;  
double (*pf)() = &Account::interest;
```

```
double daily = *pd / 365 * unit._amount;  
double daily = pf () / 365 * unit.amount();
```

187. union 不能有静态数据成员或引用成员，它是一种节省空间的类。如果一个类类型定义了构造函数、析构函数或拷贝赋值操作符，则它不能成为 union 的成员类型。

```
union illegal_members {  
    Screen s; // 错误: 有构造函数  
    Screen *ps; // ok  
    static int is; // 错误: 静态成员  
    int &rfi; // 错误: 引用成员  
};
```

188. 可以为 union 定义成员函数，包括构造函数和析构函数。Union 成员也可以被声明为共有，私有或者保护的。

```
union TokenValue {  
public:  
    TokenValue(int ix) : _ival(ix) { }  
    TokenValue(char ch) : _cval(ch) { }  
    // ...  
    int ival() { return _ival; }  
    char cval() { return _cval; }  
private:  
    int _ival;  
    char _cval;  
};
```

```
int main() {
    TokenValue tp(10);
    int ix = tp.ival();
}
```

### 189. 使用 union

```
enum TokenKind { ID, Constant /* 及其他语法单元 */ };
class Token {
public:
    TokenKind tok;

    // union 类型名被省略
    union {
        char    _eval;
        int     _ival;
        char    *_sval;
        double  _dval;
    } val;
};

int lex() {
    Token curToken;
    char *curString;
    int curIval;

    // ...
    case ID: // 标识符
        curToken.tok = ID;
        curToken.val._sval = curString;
        break;
    case Constant: // 整数常量
        curToken.tok = Constant;
        curToken.val._ival = curIval;
        break;

    // ... etc.
}
```

匿名 **union** 去掉了一层成员访问操作符，**union** 的成员名可以像 **Token** 类的成员一样被访问。所以匿名的 **union** 不能有私有或保护的成员，也不能定义成员函数。在全局域中定义的匿名 **union** 必须被声明在未命名的名字空间中（或者被声明为 **static**）。

```
class Token {
public:
    TokenKind tok;
    // 匿名 union
    union {
```

```

    char    _cval;
    int     _ival;
    char   *_sval;
    double  _dval;
};

};

int lex() {
    Token curToken;
    char *curString;
    int curIval;

    // ... 确定语法单元
    // ... 设置 curToken
    case ID:
        curToken.tok = ID;
        curToken._sval = curString;
        break;
    case Constant: // 整数常量
        curToken.tok = Constant;
        curToken._ival = curIval;
        break;
    // ... etc.
}

```

190. 位域，尽量用 `bitset` 类模板来代替位域。去地址操作符`&`不能被应用在位域上，所以也没有能指向类的位域的指针。位域也不能是类的静态成员。

```

typedef unsigned int Bit;
class File {
public:

    enum { READ = 01, WRITE = 02 };// 文件模式

    //定义成员 isRead()和 isWrite()
    inline int File::isRead() { return mode & READ; }
    inline int File::isWrite() { return mode & WRITE; }

private:
    unsigned int mode: 2;           //位域成员，“：“后必须是常量表达式
    unsigned int modified: 1;        //位域成员
    unsigned int prot_owner: 3;
    unsigned int prot_group: 3;
    unsigned int prot_world: 3;
    // ...
};

```

191. 在类定义中用到的名字必须在使用前首先被声明，这个规则有 2 种例外：

- 对于被用在 inline 成员函数定义中的名字

```
class String {  
public:  
    typedef int index_type;  
  
    // 成员函数的参数 index_type 在名字解析的第一步检查，所以名字 index_type 必须在  
    // 成员 operator[]( ) 定义之前被声明  
    char& operator[]( index_type elem )  
        { return _string[ elem ]; } // 函数体中 _string 的名字解析是名字解析的第二步，在  
                                // 这一步前，类的所有成员已经被声明了  
private:  
    char * _string;  
};
```

- 对于被用作缺省实参的名字

```
class Screen {  
public:
```

```
    // bkground 指向在类定义中后来声明的静态成员  
    Screen& clear( char = bkground ); // 同上
```

```
private:
```

```
    static const char bkground = '#';
```

```
};
```

非静态数据成员在它的值被程序使用之前，必须先被绑定到该类类型的对象上，或绑定到指向该类类型的对象的指针上，把非静态数据成员用作缺省实参违背了这个限制。

```
class Screen {
```

```
public:
```

```
    // ...
```

```
    // 错误：bkground 是一个非 static 成员  
    Screen& clear( char = bkground );
```

```
private:
```

```
    const char bkground;
```

```
};
```

192. 一般地，如果类成员的定义出现在类体之外，则跟在被定义的成员名后面的程序，直到该成员定义结束，都被认为是在类域之中。

```
class String {  
public:  
    typedef int index_type;  
    char& operator[]( index_type );  
private:  
    char * _string;  
};
```

```
// operator[]( ) 访问 index_type 和 _string
inline char& String::operator[]( index_type elem )
{
    return _string[ elem ];
}
```

193. 在类定义之外（文本文件中）定义的静态成员名之后的程序文本，直到成员定义结束，也都被认为是在类域中。

```
class Account {
    // ...
private:
    static double _interestRate;
    static double initInterest();
    static const int nameSize = 16;
    static const char name[nameSize];
};

// 引用 Account::initInterest()
double Account::_interestRate = initInterest();
// nameSize 没有被 Account 限定修饰
const char Account::name[nameSize] = "Savings Account";
```

194. 在类体之外的类成员定义中，在被定义的成员名字之前的程序文本，不在该类的域内。

```
class Account {
    typedef double Money;
    // ...
private:
    static Money _interestRate;
    static Money initInterest();
};
// Money 必须用 Account:: 限定修饰
Account::Money Account::_interestRate = initInterest();
```

195. 类域中的名字解析

用在类定义中的名字（除了在 **inline** 成员函数定义中的名字和缺省实参的名字）其解析过程如下：

1. 考虑在名字使用之前出现的类成员的声明；
2. 如果步骤 1 的解析没有成功，则在类定义之前的名字空间域中出现的声明应予以考虑。记住，全局域也是一个名字空间域。

被用在类成员函数定义中的名字的解析过程如下：

1. 考虑在成员函数局部域中的声明；
2. 如果在步骤 1 中的解析不成功，则考虑所有的类成员声明；
3. 如果在步骤 2 中的解析不成功，则考虑在成员函数定义之前的名字空间域中出现的声明。

```

class Screen {
public:
    // ...
    /*无法解析 verify
    inline void setHeight(int var){
        _height = verify( var );
    }*/
    void setHeight( int );
private:
    short _height;
};

int verify(int);

void Screen::setHeight( int var ) {
    // var: 指向参数
    // _height: 指向类成员
    // verify: 指向全局函数
    _height = verify( var );
}

```

用在类静态成员定义中的名字解析过程如下：

1. 考虑所有类成员的声明
2. 如果第 1 步失败，则考虑在静态成员定义之前的名字空间域中出现的声明

196. 嵌套类——嵌套类（B）一般被用来实现辅助外围类（A）的实现，所以可以将 B 的构造函数和析构函数声明为 private，然后把 A 指定为 B 的友元。而另一种更优雅的策略是，将 B 声明为 A 的私有成员，让 B 的所有成员成为 public，这时没必要将 A 指定为 B 的友元了，并且 A 和 A 的友元都可以访问 B 的成员。

外围类不能访问嵌套类的私有成员，除非外围类被声明为嵌套类的友元。

同样，嵌套类也不能访问外围类的私有成员，除非嵌套类被声明为外围类的友元。

嵌套类也可以被定义在其外围类之外。有时候我们不想让 List 类的用户看到 ListItem 的细节，因此，我们在含有 List 类及其成员实现的文本文件中给出嵌套类 ListItem 的定义。

```

//头文件
class List {
public:
    // ...
private:
    // 这个声明是必需的
    class ListItem;
    ListItem *list;
    ListItem *at_end;
};

//实现文件

```

```
// 用外围类名限定修饰嵌套类名
class List::ListItem {
public:
    ListItem( int val = 0 );
    ListItem *next;
    int value;
};
```

嵌套类的定义被看到之前，只能声明嵌套类的指针和引用。

```
class List {
public:
    // ...
private:
    class ListItem;
    ListItem *list;
    ListItem at_end; // 错误：未定义嵌套类 ListItem
};
```

嵌套类可以先被声明，然后再在外围类体中被定义，这允许多个嵌套类具有互相引用的成员

```
class List {
public:
    // ...
private:
    // List::ListItem 的声明
    class ListItem;
    class Ref {
        ListItem *pli; // pli 类型为: List::ListItem*
    };

    // List::ListItem 的定义
    class ListItem {
        Ref *pref; // pref 的类型为: List::Ref*
    };
};
```

嵌套类不能直接访问其外围类的非静态成员，即使这些成员是公有的。任何对外围类的非静态成员的访问都要求通过外围类的指针、引用或对象来完成。

```
class List {
public:
    int init( int );
private:
    class ListItem {
public:
    ListItem( int val = 0 );
    void mf( const List & );
    int value;
    int memb;
```

```

    };
}

List::ListItem::ListItem( int val )
{
    // List::init() 是类 List 的非静态成员，必须通过 List 类型的对象或指针来使用
    value = init( val );    // 错误：非法使用 init
}

void List::ListItem::mf( const List &il ) {
    memb = il.init(); // ok：通过引用调用 init()
}

```

嵌套类可以直接访问外围类的公有的静态成员、类型名、枚举值。

```

class List {
public:
    typedef int (*pFunc)();
    enum ListStatus { Good, Empty, Corrupted };
    // ...
private:
    class ListItem {
public:
    void check_status();
    ListStatus status; // ok
    pFunc action; // ok
    // ...
};
// ...
};

```

在 ListItem 的域中，这些成员可以不加限定修饰地被引用：

```

void List::ListItem::check_status()
{
    ListStatus s = status;
    switch ( s ) {
        case Empty: ...
        case Corrupted: ...
        case Good: ...
    }
}

```

在 ListItem 的域之外，以及在外围类 List 域之外引用外围类的静态成员、类型名和枚举名都要求域解析操作符

```

List::pFunc myAction; // ok
List::ListStatus stat = List::Empty; // ok

```

当引用一个枚举值时，我们不能写

**List::ListStatus::Empty**

枚举值可以在定义枚举的域内被直接访问，因为枚举定义并不像类定义一样维护了自己的相关域

## 197. 在嵌套类域中的名字解析

被用在嵌套类的定义中的名字(除了 inline 成员函数定义中的名字和缺省实参的名字之外), 其解析过程如下:

1. 考虑出现在名字使用点之前的嵌套类的成员声明;
2. 如果第 1 步没有成功, 则考虑出现在名字使用点之前的外围类的成员声明;
3. 如果第 2 步没有成功, 则考虑出现在嵌套类定义之前的名字空间域中的声明。

如果在全局域中, 在外围域 List 之外定义嵌套类 ListItem, 则 List 类的所有成员都已经被声明完毕, 因而编译器将考虑其所有声明。

被用在嵌套类的成员函数定义中的名字, 其解析过程如下:

1. 首先考虑在成员函数局部域中的声明;
2. 如果第 1 步没有成功, 则考虑所有嵌套类成员的声明;
3. 如果第 2 步没有成功, 则考虑所有外围类成员的声明;
4. 如果第 3 步没有成功, 则考虑在成员函数定义之前的名字空间域中出现的声明。

```
class List {  
public:  
    enum ListStatus { Good, Empty, Corrupted };  
    // ...  
private:  
    class ListItem {  
        public:  
            //void check_status() { int value = ::list; } 错误: 没有可见的 ::list 声明  
            void check_status();  
            ListStatus status;      // ok  
            // ...  
        };  
        ListItem *list;  
        // ...  
    };  
    int list = 0;  
    void List::ListItem::check_status()  
    {  
        int value = list;          // 引用 List:: list, 产生错误  
    }  
}
```

198. 局部类——定义在函数体内的类。很明显, 在局部类的局部域外, 没有语法能引用局部类的成员, 所以不允许局部类声明静态数据成员。

199. 局部类中的嵌套类可以在局部类外被定义, 但是该定义必须在局部域内

## 第 14 章 类的初始化、赋值和析构

200. C 语言习惯的显式初始化表，类似于用在初始化数组上的初始化表。

`Data local2={ 1024, "Anna livia plurabelle" };`

在某些应用中，通过显式初始化表，用常量值初始化大型数据结构比较有效。显式初始化可以在装载时刻完成，从而节省了构造函数的启动开销（即使它被定义为 `inline`），尤其是对全局对象。

**201. 一个类只有当没有构造函数或声明了缺省构造函数时，我们才能不指定实参集来定义类对象。**在实际应用中，容器类或者动态数组要求他们的元素或者提供缺省构造函数，或者不提供构造函数，所以在实践中，**如果定义了其他构造函数，则也有必要提供一个缺省构造函数。**

202. 构造函数不能用 `const` 或 `volatile` 关键字来声明。`explicit` 只能被应用在构造函数上（单参数的构造函数；或者有多个参数的构造函数，除第一个参数外，其他都有缺省参数）。

203. 缺省构造函数

```
// 每个都是缺省构造函数
Account::Account() { ... }
iStack::iStack( int size = 0 ) { ... }
Complex::Complex(double re=0.0,double im=0.0) { ... }

当我们写
int main()
{
    Account acct;
}
```

编译器首先检查 `Account` 类是否定义了缺省构造函数，以下情况之一会发生：

1. 定义了缺省构造函数，它被应用到 `acct` 上；
2. 定义了缺省构造函数，但它不是公有的 `acct` 的定义被标记为编译时刻错误 `main()` 没有访问权限；
3. 没有定义缺省构造函数，但是定义了一个或者多个要求实参的构造函数 `acct` 的定义被标记为编译时刻错误，实参太少；
4. 没有定义缺省构造函数，也没有定义其他构造函数，该定义是合法的，**acct 没有被初始化，没有调用任何构造函数**。在 4 中，`acct` 的数据成员的值是不定的；但是，如果 `acct` 是全局或静态对象，其数据成员值为 0.

204. 非公有构造函数的主要用处：

- 1) 防止用一个类的对象向该类另一个对象做拷贝（即非公有拷贝构造函数）；
- 2) 指出只有当一个类在继承层次中被用作基类，而不用直接被应用程序操纵时，构造函数才能被调用。

```
class Account {
    friend class vector< Account >;
```

```

public:
    explicit Account( const char*, double = 0.0 );
    // ...
private:
    Account();
    // ...
};

```

一般程序只能用关联名或账户名和开户余额定义 Account 对象，Account 的成员函数及其友元 vector，可以用任何一个构造函数来定义 Account 对象。

## 205. 拷贝构造函数

当我们写：

```
Account acct2( acct1 );
```

编译器判断是否为 Account 类声明了一个显式的拷贝构造函数，如果声明了拷贝构造函数并且是可以访问的，则调用它，如果声明了拷贝构造函数但是不可访问，则 acct2 的定义就是一个编译时刻错误，如果没有声明拷贝构造函数，则执行缺省的按成员初始化。

## 206. 堆类类型数组的初始化

缺省情况下是调用类的缺省构造函数，如果有特殊要求，解决方法如下：

```

#include <utility>
#include <vector>
#include <new>
#include <cstddef>
#include "Accounts.h"
typedef pair<char*,double> value_pair;
/* init_heap_array(),
 * 被声明为静态成员函数
 * 提供类对象堆数组的分配和初始化
 *
 * init_values: 数组元素的初始值对
 * elem_count: 数组元素个数
 * 如果为 0, 数组大小与 init_values vector 一样
 */
Account* Account::init_heap_array(vector<value_pair> &init_values,
                                    vector<value_pair>::size_type elem_count = 0 )
{
    vector<value_pair>::size_type vec_size = init_values.size();
    if ( vec_size == 0 && elem_count == 0 )
        return 0;
    // 分配的数组大小是 elem_count
    // 或者, 若 elem_count 为 0 则为 vector 的大小
    size_t elems = elem_count ? elem_count : vec_size;

```

```

// 找到一块不用的内存来保存数组
char *p = new char[sizeof(Account)*elems];

// 每个元素的独立初始化
int offset = sizeof( Account );
for ( int ix = 0; ix < elems; ++ix )
{
    // 偏移到第 ix 个元素,如果提供了一个初始值对,把该对传递给构造函数,
    // 否则, 调用缺省构造函数
    if ( ix < vec_size )
        new( p+offset*ix ) Account( init_values[ix].first, init_values[ix].second );
    else   new( p+offset*ix ) Account;
}

// ok: 元素被分配并初始化,返回第一个元素的指针
return (Account*)p;
}

void Account::dealloc_heap_array( Account *ps, size_t elems )
{
    for ( int ix = 0; ix < elems; ++ix )
        ps[ix].Account::~Account();

    delete [] reinterpret_cast<char*>(ps);
}

```

## 207. 类对象的 vector

当我们定义一个含有五个类对象的 vector 时, 如

```
vector< Point > vec( 5 );
```

元素的初始化过程如下:

1. 创建一个底层类类型的临时对象, 在其上应用该类的缺省构造函数;
2. 在 vector 的每个元素上依次应用拷贝构造函数, 用临时类对象的拷贝初始化每一个类对象;
3. 删除临时类对象。

构造函数是 **explicit vector (size\_type n, const T& value=T(),const Allocator&=Allocator())**;

尽管最终结果等同于定义五个类对象的数组, 比如:

```
Point pa[ 5 ];
```

但是, 初始化 vector 代价比较大, 临时对象的构造和析构, 以及拷贝构造函数往往比缺省构造函数计算上更复杂

作为一般的设计原则, 类对象的 vector 最适合元素的插入操作

```
vector< Point > cvs; // 空
```

```
int cv_cnt = calc_control_vertices();
```

```
// 预留内存以便存放 cv_cnt 个 Point 对象, cvs 仍然是空的...
```

```
cvs.reserve( cv_cnt );  
  
// 打开一个文件并准备迭代它  
ifstream infile( "spriteModel" );  
istream_iterator<Point> cvfile( infile ),eos;  
  
// ok, 现在插入元素  
copy( cvfile, eos, inserter( cvs, cvs.begin() ) );
```

## 208. 按成员初始化

在许多情况下，一个方案最难的部分就是要意识到它的必要性“

除了提供拷贝构造函数，另一种替代的方案是完全不允许按成员初始化，这可以通过下列两个步骤实现：

1. 把拷贝构造函数声明为私有的，这可以防止按成员初始化发生在程序的任何一个地方，除了类的成员函数和友元之外
2. 通过有意不提供一个定义，但是，我们仍然需要第 1 步中的声明，可以防止在类的成员函数和友元中出现按成员初始化。

# 第 15 章 重载操作符和用户定义的转换

209. C++ 要求，=、[]、()、-> 操作符必须被定义为类成员操作符。::、.\*、.? 不能重载。

210. 对于内置类型的操作符，它的预定义意义不能被改变，程序员也不能为内置数据类型定义其他的操作符。**程序员只能为类类型或枚举类型的操作数定义重载操作符。**

211. 除了对 operator() 外，对其他重载操作符提供缺省实参都是非法的。

212. 因为友元不是授权类的成员，所以它不受其所在类的 public、private 和 protected 的影响。

213. 在某些情况下，一个名字空间函数、**另一个在此之前被定义的类的成员函数**，或者一个完整的类必须声明为友元。

214. 如果一个函数操纵 2 个不同类型的对象，而且该函数需要访问者 2 个类的非公有成员，则这个函数可被声明为这 2 个类的友元，或者作为一个类的成员函数，并声明为另一个类的友元。

215. 只有当一个类的定义已经被看到时，它的成员函数才能被声明为另一个类的友元。这并不是总能做到，例如，如果 Screen 类必须把 Window 类的成员函数声明为友元，而 Window 类必须把 Screen 类的成员函数声明为友元，该怎么办？在这种情况下，只能把整个 Window 类声明为 Screen 的友元，把 Screen 类声明为 Window 的友元。

216. 重载的->的返回类型必须是一个类类型的指针，或则是“定义该->的类”的一个对象。如果返回类新是一个类类型的指针，则内置->的语义被应用在返回值上，如果返回值是另外一个类的对象或引用，则递归应用该过程，直到返回的是指针类型或语句错误。

```
class ScreenPtr {  
public:  
    Screen& operator*() { return *ptr; }  
    Screen* operator->() { return ptr; }  
};  
ps->move( 2, 3 );
```

因为成员访问操作符箭头的左操作数的类型是 ScreenPtr，所以使用该类的重载操作符。该操作符返回一个指向 Screen 类对象的指针，内置成员访问操作符箭头被依次应用在这个返回值上，以调用 Screen 类的成员函数 move()。

217. 为区分前置与后置操作符，重载的++和—后置操作符的声明有一个额外的 int 类型的参数。

```
class ScreenPtr {  
public:  
    Screen& operator++();      // 前置操作符  
    Screen& operator- -();  
    Screen& operator++(int);   // 后置操作符  
    Screen& operator- -(int);  
    // ...  
};  
  
class ScreenPtr {  
    // 非成员声明  
    friend Screen& operator++( ScreenPtr & );      // 前置  
    friend Screen& operator- - ( ScreenPtr & );  
    friend Screen& operator++( ScreenPtr &, int );  // 后置  
    friend Screen& operator- - ( ScreenPtr &, int );  
public:  
    // 成员定义  
};
```

218. 重载 new 和 delete

new()的返回类型必须是 void\*，并且有一个 size\_t 的参数。size\_t 参数在调用 new()时用类型大小初始化。

delete()的返回类型必须是 void，一般有 2 个参数，void\* 和 size\_t。在调用 delete()时，delete

表达式的指针值自动初始化 void\*参数，指针指向对象的大小自动初始化 size\_t 参数。

New 和 delete 操作符都自动做成静态成员函数，而无需程序员显式的把它们声明为静态的。

New()和 delete()的定义

```
class Screen {
public:
    void *operator new( size_t );
    void operator delete( void *, size_t );
    // ...
private:
    Screen *next;
    static Screen *freeStore;
    static const int screenChunk;
};

#include "Screen.h"
#include <cstddef>

// 静态成员初始化
Screen *Screen::freeStore = 0;
const int Screen::screenChunk = 24;

void *Screen::operator new( size_t size )
{
    Screen *p;
    if( !freeStore ) {
        // 链表空: 抓取一块存储区
        // 这里调用全局 new
        size_t chunk = screenChunk * size;
        freeStore = p = reinterpret_cast<Screen*>( new char[ chunk ] );

        // 现在把已经分配的内存串起来
        for( ; p != &freeStore[ screenChunk - 1 ]; ++p )
            p->next = p+1;
        p->next = 0;
    }
    p = freeStore;
    freeStore = freeStore ->next;
    return p;
}

void Screen::operator delete( void *p, size_t )
{
    // 将被删除的对象插入到空闲链表头
```

```

    static_cast< Screen*>( p ) ->next = freeStore;
    freeStore = static_cast< Screen*>( p );
}

```

New()和 delete()的使用

```
Screen *ptr = new Screen( 10, 20 );
```

与下列双语句序列等价

// C++伪码

```
ptr = Screen::operator new( sizeof( Screen ) );
```

```
Screen::Screen( ptr, 10, 20 );
```

**delete** ptr;

与下列双语句序列等价

// C++伪码

```
Screen::~Screen( ptr );
```

```
Screen::operator delete( ptr, sizeof( *ptr ) );
```

## 219. new[]()和 delete[]()

```

class Screen {
public:
    void *operator new[]( size_t );
    void operator delete[]( void*, size_t );
}

```

其中 size\_t 参数被自动初始化，其值等于存放 n 个 Screen 对象的数组所需内存的字节数。

使用

```
Screen *ps = new Screen[10];
```

```
delete[] ps;
```

创建数组的 new 表达式首先调用类操作符 new[]() 来分配存贮区，然后再调用缺省构造函数依次初始化数组的每一个元素。如果这类定义了构造函数，但是没有缺省构造函数，则相应的 new 表达式就是错误的。

删除数组的 delete 表达式先调用类的析构函数依次销毁数组的每一个元素，然后再调用类操作符 delete[]() 来释放内存。

## 220. 定位操作符 new()和 delete()

```

class Screen {
public:
    void *operator new( size_t );
    void *operator new( size_t, Screen* );
    void operator delete( void*, size_t );
//这个被重载的 delete() 操作符不能用 delete 表达式显式调用
//他的用途是：当执行 new(start) Screen 时，调用完 new(size_t, Screen*) 后，会执行
//Screen 的构造函数，如果此时该构造函数抛出异常，编译器会自动找到跟 new(size_t,
//Screen*) 相匹配的 delete() (即 delete( void*, Screen* )), 来执行释放内存的操作
}

```

```
void operator delete( void*, Screen* );
};
```

221. 转换函数，其一般形式是 operator type(), 这里不允许 type 表示数组或函数类型。转换函数必须是成员函数，它的声明不能指定返回类型和参数表。

```
#include "SmallInt.h"

typedef char *tName;
class Token {
public:
    Token( char*, int );
    operator SmallInt() { return val; }
    operator tName() { return name; }
    operator int() { return val; }
    // 其他公有成员
private:
    SmallInt val;
    char *name;
};
```

222. 用户定义的转换之后只允许标准转换序列，不允许出现第二个用户自定义的转换。

223. 构造函数声明前使用 explicit 可以避免通过构造函数进行的隐式转换。但是，通过强制类型转换，该转换仍然可以实现。

224. 选择哪个转换

**用户定义的转换序列**是用户定义的转换与“需要用来把值变成转换目标类型的标准转换”的组合，用户定义的转换序列形式如下：

**标准转换序列**——

**用户定义的转换**——

**标准转换序列**——

这里，用户定义的转换或者调用转换函数，或者调用构造函数。

```
class Number {
public:
    operator float();
    operator int();
    // ...
};

Number num;
float ff = num; // 哪一个转换函数? operator float()
```

1. operator float()——>精确匹配
2. operator int()——>标准转换

```

class SmallInt {
public:
    SmallInt( int ival ) : value( ival ) { }
    SmallInt( double dval ) : value( static_cast< int >( dval ) ) { }
};

extern void manip( const SmallInt & );
int main() {
    double dobj;
    manip( dobj ); //ok: SmallInt( double )
}

```

1. 精确匹配—>SmallInt(double);  
2. 标准转换—>SmallInt(int)

```

class SmallInt {
public:
    SmallInt( const Number & );
    // ...
};

class Number {
public:
    operator SmallInt();
    // ...
};

extern void compute( SmallInt );
extern Number num;
compute( num ); // 错误: 两个可能的转换

// ok: 显式调用以便解决二义性
compute( num.operator SmallInt() );

compute( SmallInt( num ) ); // 错误: 仍然是二义的

```

## 225. 考虑类重载解析

**候选函数:** 如果一个函数调用的实参是一个类类型的对象、类类型的指针、类类型的引用或者是指向类成员的指针，则候选函数是以下各个函数集合的并集：在调用点可见的函数、在定义该类的名字空间中声明的函数、以及在类成员表中声明为友元的函数。

重载解析(候选函数) ->名字解析(排除部分候选函数) ->重载解析(可行函数)->重载解析(最佳函数)

更多详细信息见《C++ Primer 3》 15.10、15.11、15.12

## 第 16 章 类模板

226. 模板参数的名字的作用域是整个模板声明或定义

```
typedef double Type;

template <class Type>
class QueueItem {
public:
    // ...
private:
    Type item; // item 不是 double 类型, ::Type 被隐藏掉
    QueueItem *next;
};

template <class Type>
class QueueItem {
public:
    // ...
private:
    // 错误: 成员名不能与模板参数 Type 同名
    typedef double Type;
    Type item;
    QueueItem *next;
};
```

227. 在类模板的前向声明和类模板定义中, 模板参数的名字可以不同

```
// 所有三个 QueueItem 声明都引用同一个类模板
// 模板的声明
template <class T> class QueueItem;
template <class U> class QueueItem;

// 模板的真正定义
template <class Type>
class QueueItem { ... };
```

228. 类模板的缺省参数同函数缺省参数的规则一样

229. 注意, 标准 C++ 之前的编译器并不支持模板参数的缺省实参。

230. 在类模板定义中, 类模板的名字可以被用作一个类型指示符, 凡是可以使用非模板类名的地方都可以用它。

```
template <class Type>
class QueueItem {
```

```

public:
    QueueItem( const Type & );
private:
    Type item;
    QueueItem *next;           /*这种简写形式只能用在类模板 QueueItem 自己的定义中，在其他地方用作一个类型指示符时，必须指定完整的模板参数表。*/
};

template <class Type>
void display( QueueItem<Type> &qi )
{
    QueueItem<Type> *pqi = &qi;
}

```

231. 与函数模板不同的是，类模板的实例必须总是显示的指定模板实参。根据类模板实例被使用的上下文环境，编译器无法推断出类模板实例的模板实参。

232. 只有当代码使用了类模板的一个实例的名字，并且上下文环境要求必须存在类的定义时，这个类模板才被实例化。例如，只是声明一个类的指针和引用，就没有必要知道类的定义

```

// Queue<int> 没有为其在 foo() 中的使用实例化
void foo( Queue<int> &qi )
{
    Queue<int> *pqi = &qi;
}

class Matrix;
Matrix obj1; // 错误: Matrix 没有被定义

class Matrix { ... };
Matrix obj2; // ok

```

所以，如果一个对象的类型是一个类模板的实例，那么当对象被定义时，类模板也被实例化

```

void foo( Queue<int> &qi )
{
    Queue<int> *pqi = &qi;

    // 因为成员函数被调用，所以 Queue<int> 被实例化
    pqi->add( 255 );
}

```

233. 绑定给模板非类型参数的表达式必须是一个常量表达式。

```
template <int *ptr> class BufPtr { ... };
```

```

// 错误: 模板实参不能在编译时刻被计算出来
BufPtr< new int[24] > bp;
template <int size> Buf{ ... };
template <int *ptr> class BufPtr { ... };

int size_val = 1024;
const int c_size_val = 1024;

Buf< 1024 > buf0; // ok
Buf< c_size_val > buf1; // ok
Buf< sizeof(size_val) > buf2; // ok: sizeof(int)
BufPtr< &size_val > bp0; // ok

// 错误: 不能在编译时刻被计算出来
Buf< size_val > buf3;

```

234. 在模板实参的类型和非类型参数的类型之间允许进行一些转换，能被允许的转换集是“函数实参上被允许的转换”的子集：

1. 左值转换，包括从左值到右值的转换、从数组到指针的转换、以及从函数到指针的转换
2. 限定修饰转换
3. 提升
4. 整值转换

235. 被定义在类模板定义之外的成员函数必须使用特殊的语法，来指明它是一个类模板的成员。

```

template <class Type>
class Queue {
public:
    Queue( );
private:
    // ...
};

template <class Type>
inline Queue<Type>::Queue( ) { front = back = 0; }

```

236. 类模板的成员函数本身也是一个模板（不管它有没有使用模板参数），标准 C++ 要求这样的成员函数只有在被调用或取地址时，才被实例化。（标准 C++ 之前有些编译器在实例化类模板时，就实例化类模板的成员函数）。类模板的成员函数被实例化的时间会影响到“在类模板成员函数定义中名字的解析”，以及“可以声明一个成员函数特化的时间”。

### 237. 类模板中友元的声明：

#### 1. 非模板友元类和友元函数

在类模板 QueueItem 把 foobar 类和函数 foo() 声明为友元之前，它们不必在全局域中被声明或定义。但是，在 QueueItem 类把 Foo 类的一个成员声明为友元之前，Foo 类必须已经被定义。

```
class Foo {  
    void bar();  
};  
  
template <class T>  
class QueueItem {  
    friend class foobar;  
    friend void foo();  
    friend void Foo::bar();  
    // ...  
};
```

#### 2. 绑定的友元类模板和函数模板

在一个模板可以被用在一个类模板的友元声明中之前，它的声明或定义必须先被给出。

```
template <class Type>  
class foobar{ ... };  
  
template <class Type>  
void foo( QueueItem<Type> );  
  
template <class Type>  
class Queue {  
    void bar();  
};  
  
template <class Type>  
class QueueItem {  
    friend class foobar<Type>;  
    friend void foo<Type>( QueueItem<Type> ); /*注意，这里如果写成 friend void  
foo( QueueItem<Type> ); 则  
表示该友元是一个普通函数  
而非一个函数模板 */  
    friend void Queue<Type>::bar();  
    // ...  
};
```

#### 3. 非绑定的类模板和函数模板

```
template <class Type>  
class QueueItem {
```

```

template <class T> friend class foobar;

template <class T> friend void foo( QueueItem<T> );

template <class T> friend void Queue<T>::bar();
// ...
};


```

### 238. 类模板的静态数据成员

```

#include <cstddef>
template <class Type>
class QueueItem {
    // ...
private:
    void *operator new( size_t );
    void operator delete( void *, size_t );
    // ...
    static QueueItem *free_list;
    static const unsigned QueueItem_chunk;
    // ...
};

template <class Type> void*
QueueItem<Type>::operator new( size_t size )
{
    QueueItem<Type> *p;

    if( ! free_list )
    {
        size_t chunk = QueueItem_chunk * size;
        free_list = p =
            reinterpret_cast<QueueItem<Type>*>
                ( new char[chunk] );
        for( ; p != &free_list[ QueueItem_chunk - 1 ]; ++p )
            p->next = p + 1;
        p->next = 0;
    }
    p = free_list;
    free_list = free_list->next;

    return p;
}

template <class Type>

```

```

void QueueItem<Type>::
    operator delete( void *p, size_t )
{
    static_cast< QueueItem<Type>*>( p )->next = free_list;
    free_list = static_cast< QueueItem<Type>*>( p );
}

// 为每个 QueueItem 实例生成相关的 free_list，并把它初始化为 0
template <class T> QueueItem<T> *QueueItem<T>::free_list = 0;

// 为每个 QueueItem 实例生成相关的 QueueItem_chunk，并把它初始化为 24
template <class T> const unsigned int QueueItem<T>::QueueItem_chunk = 24;

```

由上面代码知道，类模板的静态数据成员本身就是一个模板，它的定义不会引起内存分配。要引用一个类模板的静态数据成员，必须通过一个特定的模板实例

```

// 错误: QueueItem 不是一个真正的实例
int ival0 = QueueItem::QueueItem_chunk;

int ival1 = QueueItem<string>::QueueItem_chunk; // ok
int ival2 = QueueItem<int>::QueueItem_chunk; // ok

```

239. 当外围类模板实例化时，它的嵌套类模板不会自动实例化。只有当上下文中确实需要嵌套类模板的实例时，嵌套类模板才会被实例化。
240. 程序只能引用嵌套类型的一个实例，引用的时候，嵌套类的名字前必须加上外围类模板实例的名字。即使嵌套类型并没有使用外围类模板的参数，这条规则也同样适用。

```

template <class Type, int size>
class Buffer {
public:
    enum Buf_vals { last = size -1, Buf_size };
    typedef Type BufType;
    BufType array[ size ];
    enum QA { empty, full };      // 不变量
    // ...
};

Buffer::Buf_vals bfv0;           // 错误: Buffer 的哪一个实例?
Buffer<int,512>::Buf_vals bfv1; // ok
qd.status = Buffer::empty;       // 错误: Q 的哪一个实例?
qd.status = Buffer<double,1>::empty; // ok

```

241. 成员模板——类模板的成员是一个独立的模板，类模板和这些成员模板是一对多的关系。

```
template <class T>
```

```

class Queue {
private:
    // 类成员模板
    template <class Type>
    class CL
    {
        Type member;
        T mem;
    };
    // ...
public:
    // 函数成员模板
    template <class Iter>
    void assign( Iter first, Iter last )
    {
        while ( !is_empty() )
            remove(); // calls Queue<T>::remove()
        for ( ; first != last; ++first )
            add( *first ); // calls Queue<T>::add( const T & )
    }
};

```

Assign()很好的说明了成员模板的作用，一个 Queue<int>现在可以用 Vector<X>的迭代器来赋值了，只要元素类型 X 能转换为 int (定义了转换函数)。

注意成员模板定义在外围的语法：

```

template<class T>
class Queue {
private:
    template<class Type> class CL;
    // ...
public:
    template<class Iter>
    void assign( Iter first, Iter last );
    // ...
};

template<class T> template<class Type>
class Queue<T>::CL<Type>
{
    Type member;
    T mem;
};

template<class T> template<class Iter>
void Queue<T>::assign( Iter first, Iter last )

```

```

{
    while ( ! is_empty() )
        remove();

    for ( ; first != last; ++first )
        add( *first );
}

```

#### 242. 显式实例化声明——用于控制实例化发生的时间，能提高编译效率。

显式实例化类模板时，它的所有成员也被实例化。所以在显式实例化声明的地方，不单要看到类模板的定义，而且要能看到其成员的定义

```

template <class Type>
class Queue;

// 错误: 没有定义模板 Queue 及其成员
template class Queue<int>;

```

同函数模板一样，显式实例化需要开启编译器“抑制模板隐式实例化”选项。这样，编译器会在遇到需要实例化的地方，查找程序中相应的显式实例化，而不会隐式实例化它们。

#### 243. 类模板特化——针对个别类型参数定义单独的类实现或成员函数实现。理解：特化的对象实际上是一个特定的类或成员函数，不是模板了。

只有当通用的类模板被声明（不一定被定义之后）它的显式特化才可以被定义。

```

// QueueLD.h: 定义类的特化 Queue<LongDouble>
#include "Queue.h"

template<> class Queue<LongDouble> {
    Queue<LongDouble>();
    ~Queue<LongDouble>();
    LongDouble& remove();
    void add( const LongDouble & );
    bool is_empty() const;
    LongDouble min();
    LongDouble max();
private:
    // 某些特殊的实现
};

//QueueLD.cpp
// 定义类模板特化的成员函数 min()
LongDouble Queue<LongDouble>::min( ) { }      //这里不需要加 template<>了

//Queue.h
template <class Type>
class Queue {

```

```

// ...
public:
    Type min();
    Type max();
    // ...
};

// 函数模板显式特化声明
template<> LongDouble Queue<LongDouble>::min( );
template<> LongDouble Queue<LongDouble>::max( );

//Queue.cpp
// 显式特化定义
// explicit specialization definitions
template<> LongDouble Queue<LongDouble>::min( )
{
    assert( !is_empty() );
    LongDouble min_val = front->item;
    for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
        if ( pq->item.compareLess( min_val ) )
            min_val = pq->item;
    return min_val;
}
template<> LongDouble Queue<LongDouble>::max( )
{
    assert( !is_empty() );
    LongDouble max_val = front->item;
    for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
        if ( max_val.compareLess( pq->item ) )
            max_val = pq->item;
    return max_val;
}

```

同函数模板一样，一个程序中不能既有“类模板的某个类型参数的”通用实例化，又有它的特化。下面这个程序时错误的：

```

// ---- File1.C ----
#include "Queue.h"
void ReadIn( Queue<LongDouble> *pq ) {
    // pq->add() 的使用引起 Queue<LongDouble> 被实例化
}

// ---- File2.C ----
#include "QueueLD.h"

void ReadIn( Queue<LongDouble> * );
int main() {

```

```
// 使用 Queue<LongDouble> 特化定义
Queue<LongDouble> *qld = new Queue<LongDouble>;
ReadIn( qld );
}
```

编译器通常诊断不出这样的错误，但为防止此类错误，我们应该在每个使用 Queue<LongDouble> 的文件中，在它被第一次使用之前包含头文件 QueueLD.h。

244. 类模板的部分特化——类模板的部分模板实参被特定的类型(类型参数)或值(非类型参数)所取代。

```
template <int hi, int wid>
class Screen {
    // ...
};

// 类模板 Screen 的部分特化
template <int hi>
class Screen<hi, 80> {
public:
    Screen();
    // ...
private:
    string _screen;
    string::size_type _cursor;
    short _height;
    // 为 80 列的屏幕使用特殊的算法
};
```

245. 如果定义了整个类被特化或者被部分特化，则必须定义相关的成员函数和静态数据成员，模板成员的通用定义不会被用来实例化类模板特化/部分特化的成员。

246. 名字空间对类模板的影响

类模板或类模板成员的特化声明必须与“通用模板定义”在一个名字空间中

# 第五篇 面向对象的程序设计

## 第 17 章 类的继承和子类型

### 17.3 基类成员访问

247. 基类的成员函数不能被派生类重载。一个名字的重载候选函数必须出现在同一个域中。

248. 如果派生类需要重载基类的成员函数，有 2 种方法，方法 1 可以写一个 `inline` 存根函数

```
class Shy : public Diffident {  
public:  
    // ok: 方法之一: 为基类和派生类的成员  
    // 提供一个重载函数集合  
    void mumble( string whatYaSay );  
    void mumble( int softness ) {  
        Diffident::mumble( softness ); }  
    // ...  
};
```

方法 2 是用 `using` 声明：

```
class Shy : public Diffident {  
public:  
    // ok: 在标准 C++ 下 通过 using 声明  
    // 创建了基类和派生类成员的重载集合  
    void mumble( string whatYaSay );  
    using Diffident::mumble;  
  
    // ...  
};
```

这种方法把基类的所有重载函数都引入派生类域，通过这种方式无法实现指定重载基类的某一个成员函数。

249. 对 `protected` 的常见误解：

派生类可以访问 `protected` 子对象，但是不能访问一个独立基类的 `protected` 成员。

```
class Query {  
public:  
    const vector<location>* locations() const { return &_loc; }  
    // ...  
protected:  
    vector<location> _loc;  
private:
```

```

    // ...
};

bool NameQuery:: compare( const Query *pquery )
{
    // ok: 自己的 Query 子对象的 protected 成员
    int myMatches = _loc.size();

    // 错误: 没有 "直接访问另一个独立的 Query 对象的 protected 成员" 的权利
    int itsMatches = pquery->_loc.size();
    return myMatches == itsMatches;
}

```

派生类和基类成员访问问题，通常都可以通过把操作（成员函数）移到“不可访问的成员所属的类中”来解决，例如把成员函数 compare 从 NameQuery 移到 Query 中

```

bool Query:: compare( const Query *pname )
{
    int myMatches = _loc.size();           // ok
    int itsMatches = pname->_loc.size();   // ok as well

    return myMatches == itsMatches;
}

```

派生类可以直接访问该类其他对象的 protected 和 private 成员

```

bool NameQuery:: compare( const NameQuery *pname )
{
    int myMatches = _loc.size();           // ok
    int itsMatches = pname->_loc.size();   // ok as well

    return myMatches == itsMatches;
}

Class A
{
public:
    A(const A& aa){ a = aa.a;}           //ok
private:
    int a;
}

```

250. 派生类没有成为“向它的基类授权友谊的类”的友元。如果这个派生类要求这种友元关系，则它必须被相应的类显式地授权。

## 17.4 基类和派生类的构造

251. 构造函数的调用顺序

- 1) 基类构造函数，如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序而不是它们在成员初始化表中的顺序；
- 2) 成员类对象构造函数，如果有多个成员类对象，则构造函数的调用顺序是对象在类中被声明的顺序，而不是它们出现在成员初始化表中的顺序；
- 3) 派生类构造函数。

### 17.4.1 基类构造函数

252. 如果基类只想在程序中作为“其派生子类型对象中的子对象”而存在，我们应该将构造函数声明为 `protected` 而不是 `public`。

```
class Query {  
public:  
    // ...  
protected:  
    Query();  
    // ...  
};
```

253. 如果基类 `Query` 不但应该只构造一个其派生类的子对象，而且它还应该只构造一个 `NameQuery` 对象的 `Query` 子对象，我们可以通过把这第二个构造函数声明为 `private`，并把 `NameQuery` 声明为 `Query` 类的友元来保证这一点。

```
class Query {  
    friend class NameQuery;  
public:  
    // ...  
protected:  
    Query();  
    // ...  
private:  
    explicit Query( const vector<location>& );  
};
```

254. 派生类构造函数只能调用其直接基类的构造函数。

255. 注意以下代码

```
inline Query::~Query(){ delete _solution; }
```

因为 `new` 出来的 `_solution` 不一定是在堆上，所以我们还需要进一步的保证，程序层次上的有效策略是“为层次结构中的类重载 `new` 和 `delete` 操作符”，一种可能的程序层次上的策略如下：`new` 操作符把对象标记为“在堆中分配”然后再用 `new` 表达式分配该对象，而 `delete` 操作符检查这个标记是否存在，如果存在，则对操作数应用 `delete` 表达式。

#### 17.4.4 迟缓型错误检测（Lazy error detection）

256. 如果一个错误在声明点上被标记出来，则我们不能继续编译程序，直到错误被改正为止。但是，如果发生冲突的声明是库的一部分，而我们不能访问它的源码，则解决这样的冲突可能真的不是小事一桩，因为错误只能在使用点上才被标记出来。而且，我们可能永远不会有办法触发应用程序中的错误，以至于虽然这个声明代表了一个潜在的错误，但是该错误却从不会在我们的代码中被识别出来。
257. 另一方面，如果直到一个使用点上该错误才被标记出来，则我们的代码可能充满了各种未触发的语言错误，而不小心的程序员可能随时激活它们，在这种策略下，成功编译代码并不能确保它没有语义错误，只能保证程序没有违反语言的语义规则。
258. 这种在使用点上产生一个错误消息是一种“迟缓型计算”的形式，是提高程序性能的常见设计策略，它常常被应用在昂贵资源的初始化和分配上，直到真正需要这些资源时才分配或者初始化。如果这些资源永远也没有被真正用到，则我们就可以节省不必要的运行开销，如果需要这些资源，但不是一次需要全部资源，则我们可以分散程序的初始化开销。
259. 在 C++ 中，在处理重载函数、模板及类层次结构时，通常会产生一种潜在的组合错误，这种错误，往往会在使用点而不是声明点上被揭示出来。无论你是否相信这是一个正确的策略（实践中，我们认为它是正确的。在组合多个组件时，要想解决每一个潜在的错误，并不符合生产规律），它确实是正在使用的策略。这意味着我们必须小心谨慎地测试自己的代码，以便找到并解决潜在的错误。在组合两个或多个大型组件时，少量潜在的错误是可以被接受的。但是，在单个组件中，如一个类层次结构中，则往往是不可接受的。

#### 17.4.5 析构函数

260. 如果希望 `delete pBase` 调用指针所指的实际对象类型的析构函数，我们必须把 `Query` 基类的析构函数声明为虚拟的。

## 17.5 基类和派生类虚拟函数

261. 虚拟函数是可以定义为内联的，编译器会根据“能否确定虚表入口”来决定内联展开或是忽略该内联声明。
262. 静态函数不能声明为 `virtual`。
263. 在一个类层次上，派生类的虚拟函数的参数必须和基类中的一致；返回类型也要一致，但是，如果基类中虚拟函数的返回值是 `BASE`（或 `BASE*`、`BASE&`），则派生类中对应的虚拟函数返回类型可以是 `DEVIDE`（或 `DEVIDE*`、`DEVIDE&`）。

### 17.5.3 虚拟函数的静态调用

264. A 是 B 和 C 的直接基类，A 的 `display()` 虚拟函数可以定义为 B 和 C 的公共操作，并在 B 和 C 内被静态调用

```
B::display()
{
    A::display();
    //...
}

C::display()
{
    A::display();
    //....
}
```

265. 纯虚拟函数也可以有自己的定义。如果我们引入这样一个矛盾：一方面，我们认为有必要把 `print()` 声明为纯虚拟函数，以告诉编译器 `BinaryQuery` 是一个抽象基类，这样可以保证在应用程序中不会定义 `BinaryQuery` 的独立对象。另一方面，我们必须定义 `BinaryQuery` 的虚拟 `print()` 实例，并通过 `AndQuery` 和 `OrQuery` 类的对象调用它。与许多矛盾一样，我们忽略了一个重要的信息，纯虚拟函数和虚拟函数一样，虽然它可以通过虚拟机制被调用，但也可以被静态调用。

```
//A.h
class A{
    virtual void display()=0;
}
inline void A::display()
{
    //...
}
```

```

//B.h
inline void B::display()
{
    A::display();
    //..
}

//C.h
inline void C::display()
{
    C::display();
    //..
}

```

#### 17.5.4 虚拟函数和缺省实参

266. 缺省实参的值是在编译时刻确定的，而虚拟函数入口是动态确定的。考虑下面简单的类层次结构：

```

#include <iostream>

class base {
public:
    virtual int foo( int ival = 1024 ) {
        cout < "base::foo() -- ival: " < ival < endl;
        return ival;
    }
    // ...
};

class derived : public base {
public:
    virtual int foo( int ival = 2048 ) {
        cout << "derived::foo() -- ival: " << ival << endl;
        return ival;
    }
    // ...
};

int main()
{
    derived *pd = new derived;
    base *pb = pd;
}

```

```

int val = pb->foo();
cout << "main() : val through base: "
    << val << endl;

val = pd->foo();
cout << "main() : val through derived: "
    << val << endl;
}

```

编译并运行它 程序产生下列输出

```

derived::foo() -- ival: 1024          //注意：调用派生类，但输出却是 1024
main() : val through base: 1024

derived::foo() -- ival: 2048
main() : val through derived: 2048

```

267. 如果我们确实希望传递给 `foo()` 的实际缺省实参是根据被调用函数的实际实例而决定的，那么该怎么办呢？不幸的是，虚拟机制不直接支持这种行为。一种程序设计的方案是，指定一个缺省实参，它可以被用来指示“用户并没有传递相应的值”，而真正的缺省实参被声明为函数中的局部值，如果没有传递进来显式的值，则使用该局部值，例如：

```

void base:: foo( int ival = base_default_value )
{
    int real_default_value = 1024;

    if ( ival == base_default_value )      //如果没有指定实参
        ival = real_default_value;
    // ...
}

void derived:: foo( int ival = base_default_value )
{
    int real_default_value = 2048;

    if ( ival == base_default_value )      //如果没有指定实参
        ival = real_default_value;
    // ...
}

```

这里的 `base_default_value` 对于整个层次结构都是一致的，如果它出现了，则表示用户没有提供一个显式的值。

## 17.5.5 虚拟析构函数

268. 虚拟函数承接了“调用者所属类类型”的访问级别。不像基类的构造函数，一般地，基类的析构函数不应该是 protected。

```
class Query {  
public: // ...  
  
protected:  
    virtual ~Query();  
  
    // ...  
};  
  
class NotQuery : public Query {  
public:  
    ~NotQuery();  
  
    // ...  
};  
  
int main()  
{  
    Query *pq = new NotQuery;  
  
    // 非法：试图通过 Query* 类型调用 Query 的 protected 成员  
    delete pq;  
}
```

## 17.5.7 虚拟 new 操作符

```
NotQuery *pnq;  
// set pnq ...  
  
// new 表达式调用 NotQuery 的拷贝构造函数  
NotQuery *pnq2 = new NotQuery( *pnq );  
  
// Query 是一个抽象类  
Query* pq = foo();  
const Query *pq2 = pq->?; // 怎样复制 pq?
```

如果我们能够声明一个操作符 new 的虚拟实例，则问题就解决了。正确的 new 操作符实例将被自动调用。不幸的是，new 操作符不能被声明为虚拟的，因为它是一个静态成员

函数。

在构造类对象之前被应用到未使用的内存上。

虽然我们不能把 new 操作符声明为虚拟的，但是我们可以提供一个代理 new 操作符来把我们的对象分配并拷贝到空闲存储区中，这个代理通常被称为 clone()。

```
class Query {  
public:  
    virtual Query *clone() = 0;  
  
    // ...  
};
```

下面是 NameQuery 实例的一种实现方式

```
class NameQuery : public Query {  
public:  
    virtual Query *clone()  
        // 调用 NameQuery 的拷贝构造函数  
        { return new NameQuery( *this ); }  
  
        // ...  
};
```

当目标指针的类型是 Query\* 时，这是完全正确的。如

```
Query *pq = new NameQuery( "Valery" );  
Query *pq2 = pq->clone();
```

当目标指针类型是实际的 NameQuery\* 时，它工作得就有些不太好了，在这种情况下，要求我们提供一个向下转换把 Query\* 指针转换回 NameQuery\*。

```
NameQuery *pnq = new NameQuery( "Rilke" );  
NameQuery *pnq2 = static_cast<NameQuery*>( pnq->clone() );
```

对于虚拟函数，要求派生类的返回类型必须与其基类实例的返回类型完全匹配。但是如果虚拟函数的基类实例返回一个类类型（或指向类类型的指针或引用）则派生类实例可以返回一个“从基类实例返回的类有派生出来”的类（或指向类类型的指针或引用）。见 263)

```
class NameQuery : public Query {  
public:  
    virtual NameQuery *clone()  
        { return new NameQuery( *this ); }  
        // ...  
};
```

现在 pq2 和 pnq2 的初始化都可以实现，而无需显式强制转换

```
// Query *pq = new NameQuery( "Broch" );  
Query *pq2 = pq->clone();           // ok  
  
// NameQuery *pnq = new NameQuery( "Rilke" );  
NameQuery *pnq2 = pnq->clone();     // ok
```

## 17.5.8 虚拟函数、构造函数和析构函数

269. 派生类对象中构造函数的调用顺序是，先调用基类的构造函数，然后是派生类的构造函数。如：

```
NameQuery poet( "Orlen" );
```

构造函数的调用顺序是先 Query，然后 NameQuery。

当执行 Query 基类的构造函数时 poet 的 NameQuery 部分还没有被初始化。实际上 poet 还不是一个 NameQuery，只有它的 Query 子对象被构造了。如果在基类的构造函数中调用了一个虚拟函数。而基类和派生类都定义了该函数的实例，将会怎么样？应该调用哪一个函数实例？如果可以调用虚拟函数的派生类实例，并且它访问任意的派生类成员，那么调用的结果在逻辑上是未定义的，而程序可能会崩溃。为了防止这种事情发生，虚拟函数与构造函数/析构函数的关系：**虚拟函数的多态性只发生在对象的构造函数调用之后和析构函数调用之前这段时间**。在基类构造函数中调用的虚拟实例总是在基类中活动的虚拟实例，在基类析构函数中也是如此（派生类部分已被销毁）。

270. 按成员初始化和赋值

```
NameQuery folk( "folk" );
    //用 folk 对 music 进行初始化
```

```
NameQuery music = folk;
```

会导致以下事情发生：

1. 编译器检查 NameQuery 是否定义了一个显式的拷贝构造函数实例，答案是没有，所以，编译器准备应用缺省的按成员初始化；
2. 编译器接下来检查 NameQuery 类是否含有基类子对象，是的，它含有 Query 基类子对象；
3. 编译器检查 Query 基类是否定义了显式的拷贝构造函数实例，答案也是没有，所以编译器准备应用缺省的按成员初始化；
4. 编译器检查 Query 类是否含有基类子对象，没有；
5. 编译器以声明的顺序检查 Query 的每个非静态成员，如果成员是非类对象，如 \_paren 和 \_solution，则它用 folk 的成员值初始化 music 对象的成员，如果成员是类对象，如 \_loc，则它递归地应用步骤 1。是的，vector 类定义了一个显式的拷贝构造函数实例，该拷贝构造函数被调用，用 folk.\_loc 初始化 music.\_loc；
6. 然后编译器按声明的顺序检查 NameQuery 类型的每个非静态成员 string 成员类对象被识别出来，它有一个显式的拷贝构造函数，于是调用该拷贝构造函数，用 folk.\_name 初始化 music.\_name。

拷贝构造函数可以通过 DEVIDE(const DEVIDE&rhs):Base(rhs) 来调用基类的拷贝构造函数，来初始化基类子类型部分，而拷贝赋值函数没有这个功能，所以必须在函数体内显示的调用基类的拷贝赋值函数：

```
this->BASE::operator=( rhs );
```

或

```
(*static_cast<Query*>(this)) = rhs;
```

## 第 18 章 多继承和虚拟继承

### 18.2 多继承

- 271. 与单继承一样，只有当一个类的定义已经出现后，它才能被列在多继承的基类表中。
- 272. 在多继承下，派生类可以从两个或者更多个基类中继承同名的成员，然而在这种情况下，直接访问是二义的，将导致编译时刻错误。

### 18.3 public、private 和 protected 继承

- 273. public 派生被称为类型继承 (type inheritance)；private 派生被称为实现继承 (implementation inheritance)，派生类和基类不是 is-a 的关系，而是为了重用基类的实现(数据和函数)，这种情况下，基类的共有接口不是派生类的一部分，所以应该用 private 将其屏蔽。

#### 18.3.1 继承与组合 (composition)

- 274. 实现继承通常是一种 has-a 关系，下面给出了一个关于“在包含 has-a 关系的类设计中”是否使用组合或私有继承的广泛建议：
  - 1) 如果我们希望改写一个类的虚拟函数，则必须使用私有继承
  - 2) 如果我们希望一个类能够引用“一个包含多种可能类型的层次结构”中的一个类（例如 iswitch 与 SPAS 和 XMS），那么就必须通过引用使用组合
  - 3) 和 PeekbackStack 类一样，如果只是希望简单地重用实现，则按值组合比继承更好
  - 4) 如果希望对象的迟缓型分配，则按引用(使用一个指针)组合通常是一个不错的设计选择。

#### 18.3.2 免除 (exempting) 个别成员的私有继承影响

- 275. 类的设计者可以针对基类的个别成员，使其免除非公有派生的影响。但是只能将继承得到的成员恢复到原来的访问级别，该访问级别不能比基类中原来指定的级别更严格或更不严格。

```
class PeekbackStack : private IntArray {
public:
    // 维持公有访问级别
    using IntArray::size;
    // ...
};
```

## 18.4 继承下的类域

276. 在继承下,派生类的域被嵌套在直接基类的域中。如果一个名字在派生类中没被解析出来, 则编译器会尝试在外围基类域中查找该名字。

### 18.4.1 多继承下的类域

277. 名字解析在查找标识符时, 首先从出现该引用的直接域中开始。在多继承下, 查找过程对每个基类的继承子树同时进行检查。如果只在其中一个基类子树中找到了声明, 则该标识将被解析, 查找算法结束。如果在两个或多个基类子树中都找到了声明, 则表示这个引用是二义的, 会产生一个编译时刻错误消息。

## 18.5 虚拟继承

278. 一般地, 除非虚拟继承为一个眼前的设计问题提供了解决方案, 否则建议不要使用它。

```
// 关键字 public 和 virtual 的顺序不重要
class Bear : public virtual ZooAnimal { ... };
class Raccoon : virtual public ZooAnimal { ... };
```

### 18.5.2 特殊的初始化语义

279. 在非虚拟派生中, 派生类只能显式初始化其直接基类。所以, 在多继承中, 一个派生类对象的声明可能引起多次基类构造函数的执行。然而, 在虚拟派生中, 这个基类的构造函数无论如何都只执行一次, 虚拟基类的初始化变成了最终派生类的责任。这个最终派生类是由每个特定类对象的声明来决定的。

```
Panda::Panda( string name, bool onExhibit=true ) : ZooAnimal( name, onExhibit, "Panda" ),
    Bear( name, onExhibit ),
    Raccoon( name, onExhibit ),
    Endangered( Endangered::environment, Endangered::critical ),
    _sleeping( false )
{}
```

280. 作为中间派生类, 所有对虚拟基类构造函数的调用都被自动抑制了, 向中间类构造函数传递的实参可能是不必要的。避免这种不必要的参数传递的解决方案是: 提供一个显式的构造函数, 用于“当它被作为中间派生类时”的情形。

```
class Bear : public virtual ZooAnimal {
public:
    // 当作为最终派生类时
    Bear( string name, bool onExhibit=true )
```

```

    : ZooAnimal( name, onExhibit, "Bear" ),
      _dance( two_left_feet )
  {}

  // ... rest the same

protected:
  // 当作为一个中间派生类时
  Bear() : _dance( two_left_feet ) {}

  // ... rest the same
};


```

281. 虚拟继承下构造函数和析构函数的调用顺序：被虚拟继承的基类无论出现在哪个位置上，它们都是在非虚拟继承基类之前被构造。

282. 虚拟继承的基类的成员的可视性

1. ZooAnimal 虚拟基类实例，如 name() 和 family\_name()，它们没有被 Bear 和 Raccoon 改写；
2. 继承自 Raccoon，属于 ZooAnimal 虚拟基类的 onExhibit() 实例，以及 Bear 定义的被改写了的 onExhibit() 实例；
3. 继承自 ZooAnimal，分别被 Bear 和 Raccoon 改写了的 print() 实例。

哪些在最终派生类 Pada 中无二义性？被在非虚拟继承下，以上 3 种情况都有二义性。在虚拟继承下：

- (1) 没有二义性，调用 ZooAnimal 中的成员函数；
- (2) 也没有二义性，调用 Bear 中被改写的 onExhibit()，因为特化的派生类实例的优先级高于“共享的虚拟继承基类”的实例的优先级；
- (3) 有二义性，因为 Bear::print() 和 Raccoon::print() 的优先级都高于 ZooAnimal::print()。

## 第 19 章 C++ 中继承的用法

### 19.1 RTTI

283. 对于要获得的派生类类型的信息 dynamic\_cast 和 typeid 操作符的操作数的类型必须是带有一个或多个虚拟函数的类类型。

#### 19.1.1 dynamic\_cast 操作符

284. `dynamic_cast` 操作符, 它允许在运行时刻进行类型转换从而使程序能够在一个类层次结构中安全地转换类型。把基类指针转换成派生类指针, 或把指向基类的左值转换成派生类的引用。它常常被称为安全的向下转换 `downcasting`。

```
// dynamic_cast 和测试在同一条件表达式中, 所以不可能在 dynamic_cast 和测试之间错
//误地插入代码,从而不可能在测试之前使用 pm
if( programmer *pm = dynamic_cast<programmer*>( pe ) ) {
    // 使用 pm 调用 programmer::bonus()
}
else {                                //指针通过返回值报错
    // 使用 employee 的成员函数
}

try {
    programmer &rm = dynamic_cast<programmer &>( re );
    // 用 rm 调用 programmer::bonus()
}
catch ( std::bad_cast ) {           //引用通过异常报错
    // 使用 employee 的成员函数
}
```

## 19.1.2 typeid 操作符

285. 使用 `typeid` 操作符时, 程序文本文件必须包含 C++ 标准库中定义的头文件`<typeinfo>`。

286. `typeid` 操作符必须与表达式或类型名一起使用, 例如, 内置类型的表达式和常量可以用作 `typeid` 的操作数当操作数不是类类型时 `typeid` 操作符会指出操作数的类型。

```
int iobj;

cout << typeid( iobj ).name() << endl;    // 打印: int
cout << typeid( 8.16 ).name() << endl;    // 打印: double
```

287. 当 `typeid` 操作符的操作数是类类型, 但不是带有虚拟函数的类类型时 `typeid` 操作符会指出操作数的类型, 而不是底层对象的类型。

```
class Base { /* 没有虚拟函数 */ };
class Derived : public Base { /* 没有虚拟函数 */ };

Derived dobj;
Base *pb = &dobj;

cout << typeid( *pb ).name() << endl; // 打印: Base
```

288. typeid(pe)与虚拟函数调用机制不同，为了要获取到派生类类型，typeid 的操作数必须是一个类类型。

```
employee *pe = new manager;
employee& re = *pe;
typeid( *pe ) == typeid( manager )      // true
typeid( *pe ) == typeid( employee )     // false
typeid( re ) == typeid( manager )       // true
typeid( re ) == typeid( employee )      // false
typeid( &re ) == typeid( employee* )    // true
typeid( &re ) == typeid( manager* )     // false
```

289. typeid 操作符实际上返回一个类型为 type\_info 的类对象。

### 19.1.3 type\_info 类

```
class type_info {
    // 依赖于编译器的实现
private:
    type_info( const type_info& );
    type_info& operator=( const type_info& );
public:
    virtual ~type_info();

    int operator==( const type_info& ) const;
    int operator!=( const type_info& ) const;

    const char * name() const;
};
```

290. 类型名是惟一保证被所有 C++ 编译器实现提供的信息，可通过 type\_info 成员函数 name() 获得。正如在本节开始提到的，对 RTTI 的支持是与编译器实现相关的，而且，某些编译器可能为类 type\_info 提供了其他成员函数，而没有在上面列出来。你应该查询编译器手册来找到确切的 RTTI 支持。

## 19.2 异常和继承

```
void iStack::push( int value )
{
    if( full() )
        // value 被存储在异常对象中.
```

```
    throw pushOnFull( value );
    // ...
}
```

执行该 throw 表达式会发生许多个步骤：

- 1 throw 表达式通过调用类类型 pushOnFull 的构造函数创建一个该类的临时对象。
  - 2 创建一个 pushOnFull 类型的异常对象，并传递给异常处理代码，该异常对象是第 1 步 throw 表达式创建的临时对象的拷贝，它通过调用 pushOnFull 类的拷贝构造函数而创建。
  - 3 在开始查找异常处理代码之前，在第 1 步中由 throw 表达式创建的临时对象被销毁。
- 在某些情况下，编译器可能能够直接创建异常对象，而不需要创建第 1 步的临时对象。但是，消除该临时对象并不是 C++ 标准的要求，而且并不是总能做到的。

291. 对于一个异常对象，直到该异常的最后一个 catch 子句退出时，它才调用析构函数销毁。

292. 再论异常规范

异常规范跟在函数声明的 const 和 volatile 限定修饰符之后：

```
class bad_alloc : public exception {
    // ...
public:
    bad_alloc() throw();
    bad_alloc( const bad_alloc & ) throw();
    bad_alloc & operator=( const bad_alloc & ) throw();
    virtual ~bad_alloc() throw();
    virtual const char* what() const throw();
};
```

同一个函数所有声明中的异常规范都必须指定相同的类型。对于成员函数，如果该函数被定义在类定义之外，则其定义所指定的异常规范，必须与“类定义中该成员函数声明中的”异常规范相同。

基类中虚拟函数的异常规范可以与派生类改写的成员函数的异常规范不同，但是，派生类虚拟函数的异常规范必须与基类虚拟函数的异常规范一样或者更严格。

293. 标准库中定义的异常类分为逻辑错误和运行时错误：

```
namespace std {
    class logic_error : public exception {
        public:
            explicit logic_error( const string &what_arg );
    };
    class invalid_argument : public logic_error {
        public:
            explicit invalid_argument( const string &what_arg );
    };
    class out_of_range : public logic_error {
        public:
```

```

        explicit out_of_range( const string &what_arg );
    };
    class length_error : public logic_error {
    public:
        explicit length_error( const string &what_arg );
    };
    class domain_error : public logic_error {
    public:
        explicit domain_error( const string &what_arg );
    };
}
}

namespace std {
    class runtime_error : public exception {
    public:
        explicit runtime_error( const string &what_arg );
    };
    class range_error : public runtime_error {
    public:
        explicit range_error( const string &what_arg );
    };
    class overflow_error : public runtime_error {
    public:
        explicit overflow_error( const string &what_arg );
    };
    class underflow_error : public runtime_error {
    public:
        explicit underflow_error( const string &what_arg );
    };
}

```

## 第 20 章 iostream 库

294. 标准库的三种流：

<iostream>	——	istream/ostream/iostream wistream/wostream/wiostream	//wchar_t 版本
<fstream>	——	ifstream/ofstream/fstream wifstream/wofstream/wfstream	
<sstream>	——	istringstream/ostringstream/stringstream wistringstream/wostringstream/wstringstream	

```
#include <sstream>
```

```

string program_name( "our_program" );
string version( "0.01" );
string mumble( int *array, int size )
{
    if ( ! array ) {
        ostringstream out_message;

        out_message << "error: "<< program_name << "--" << version
        << ":" << __FILE__ << ":" << __LINE__ << " -- ptr is set to 0; "
        << " must address some array.\n";

        // 返回底层 string 对象
        return out_message.str();
    }
    // ...
}

```

295. 输出一个 c 字符串的地址:

```

const char *pstr="abc";
cout<<static_cast<void*>(const_cast<char*>(pstr));

```

296. 操作符

endl	插入一个换行符到输出流，并刷新缓冲区
boolalpha	
noskipws	

297. 缺省情况下，输入操作符>>丢弃任何的中间空白。要读入空白字符，有 2 种方法：

1. 使用 istream 中的 get(), 它一般与 ostream 中的 put() 配合使用

```

// 获取每个字符 包括空白字符
while ( cin.get( ch ) )
    cout.put( ch );

```

2. 用 noskipws 操作符

298. cin>>i 会在 2 种情况下返回 false: 1)读到文件结束; 2)遇到一个无效值

以下是一种健壮的做法：

```

cin >> item_number;
if ( ! cin )
    cerr << "error: invalid item_number type entered!\n";
它明显不支持链式表达，所以链式表达只能在确实没有错误机会的情况下才使用
// ok: 但更容易出错
cin >> item_number >> item_name >> item_price;

```

## 299. get()的三种形式

```
while(cin.get(ch){}
while(ch==cin.get()!=EOF){}
while(cin.get(szBuf,sizeof(szBuf),'\n')){} //注意：遇到分隔符'\n'后，该分隔符仍留在 cin 中
    while ( cin.get( line, max_line ) )
    {
        // 最大读取数量 max_line - 1, 也可以为 null
        int get_count = cin.gcount();
        cout << "characters actually read: "
        << get_count << endl;

        // 处理每一行，如果遇到换行符，在读下一行之前去掉它
        if( get_count < max_line-1 )
            cin.ignore();
    }
```

ignore 从 istream 中读入并丢弃 length 个字符，或遇到 dilm 时将开始到 dilm(包含 dilm)的字符个数，或直到文件尾。

```
istream& ignore( streamsize length = 1, int delim = traits::eof )
```

相比较，getline(char \*sink, streamsize size, char delimiter='\n')更好用，因为它自动将输入流中的 delimiter 丢弃。

## 300.

# 附录 A：VC 编译器详解

## 1. 编译参数的设置

主要通过 VC 的菜单项 Project->Settings->C/C++ 页来完成。我们可以看到这一页的最下面 Project Options 中的内容，一般如下：

```
/nologo /MDd /W3 /Gm /GX /ZI /Od /D "WIN32" /D "_DEBUG" /D "_WINDOWS" /D  
"_AFXDLL" /D "_MBCS" /Fp"Debug/WritingDlgTest.pch" /Yu"stdafx.h" /Fo"Debug/"  
/Fd"Debug/" /FD /GZ /c
```

各个参数代表的意义在不同的 VC 版本不同，可以参考对应的 Msdn 为准。可以通过 Msdn 查询相关 options 参数查询详细说明。

General				Vs 2008 对应项
项	含义	默认值	相关 options 参数	
Warning level	警告信息阀门，最高级别 none，其次是 level 1，level 2，level 3，level 4。Level 1 是最严重警告。	3, /W3	/W0 , /W2, /W3, /W4	C/C++——常规—— 警告等级
Warnings as errors	将警告信息当做错误信息处理	否	否, /WX	C/C++——常规—— 将警告视为错误
Optimizations	代码优化，可以在 Optimizations 页进行详细设置	否, /Od	/Od, /O1, /O2, /Ox	C/C++——优化—— 优化
Generate browse info	用以生成.sbr 文件，用以记录类、变量等符号信息，可以在 listing files 页进行更多设置。.bsc 文件是利用.sbr 生成的	否	否, /FR, /Fr	C/C++——浏览信息——启用浏览信息
Debug info	生成调试信息：None，不产生任何调试信息（编译比较快； Line Numbers Only，仅生成全局的和外部符号的调试信息到.OBJ 文件或.EXE 文件，减小目标文件的尺寸； C 7.0- Compatible，记录调试器用到的所有符号信息到.OBJ 文件和.EXE 文件； Program Database，创建.PDB 文件记录所有调试信息； Program Database for "Edit & Continue"，创建.PDB 文件记录所有调试信息，并且支持调试时编辑。	Program Database for "Edit & Continue", /ZI	禁用 /Z7 .Zi .ZI	C/C++——常规—— 调试信息格式
C++ Language				
pointer_to_member_representation	设置类定义/引用的先后关系。一般为 Best-Case Always 表示在引用类之前该类肯定已经定义了；在定义类之前，请使用 General Popurse Always 来声明指向类成员的指针。当在两个互相引用的不同类中定义成员时可能有这种需要。对这种互相引用的类，其中一个类必须在定义前引	Best-Case Always, /vmb	/vmb, /vmg	C/C++——命令行——附加选项

	用。			
Enable Exception Handling	启用同步异常处理，并假定 <b>extern C</b> 函数从不引发异常。	是, /GX /GX-	/GX, /GX-	C/C++——代码生成——启用 C++ 异常 /EH{s a}[c][-]
Enable Run-Time Type Information	当 <b>/GR</b> 启用时，编译器将定义 <b>_CPPRTTI</b> 预处理器宏。在 Visual C++ 2005 中， <b>/GR</b> 默认处于启用状态。 <b>/GR-</b> 将禁用运行时类型信息。 如果代码使用 <b>dynamic_cast</b> Operator 或 <b>typeid</b> ，请使用 <b>/GR</b> 。但是， <b>/GR</b> 确实会导致映像的 <b>.rdata</b> 节增大。如果您的代码不使用 <b>dynamic_cast</b> 或 <b>typeid</b> ，则使用 <b>/GR-</b> 可能会生成较小的映像。	/GR-	/GR, /GR-	C/C++——语言——启动运行时类型信息
Disable Construction Displacements	支持 Visual C++ 6.0 的早期版本中不正确的行为，现在已不再需要。		/vdn	已移除
<b>Code Generation</b>				
Processor	表示代码指令优化，可以为 80386、80486、Pentium、Pentium Pro，或者 Blend 表示混合以上各种优化.VC2005 已移除该项		/G3, /G4, /G5, /G6, /G7, /GB	已移除
Use run-time library	用以指定程序运行时使用的运行时库。有一个原则就是，一个进程不要同时使用几个版本的运行时库。 Single-Threaded ( LIBC.LIB )； Debug Single-Threaded ( LIBCD.LIB )； 单线程的没有动态库。 Multithreaded ( LIBCMT.LIB )； Debug Multithreaded ( LIBCMTD.LIB )； Multithreaded DLL( MSVCRT.DLL, MSVCRT.LIB )； Debug Multithreaded DLL( MSVCRTD.DLL, MSVCRTD.LIB )。连接了单线程库就不支持多线程调用，连接了多线程库就要求创建多线程的应用程序。	/ML	/ML, /MLd, /MT, /MTd, /MD, /MDd	C/C++——代码生成——运行库
Calling convention	设定调用约定。有三种： <b>__cdecl</b> 、 <b>__fastcall</b> 和 <b>__stdcall</b> 。各种调用约定的主要区别在于，函数调用时，函数的参数是从左到右压入堆栈还是从右到左压入堆栈；在函数返回时，由函数的调用者来清理压入堆栈的参数还是由函数本身来清理；以及在编译时对函数名进行的命名修饰(可以通过 Listing Files 看到各种命名修饰方式)。	/Gd	/Gd, /Gr, /Gz	C/C++——高级——调用约定
Struct member alignment	注意：不同的对齐方式存取数据的速度不一样	/Zp8	/Zp1——16	C/C++——代码生成——结构成员对齐
<b>Customize</b>				
Disable	是否禁用微软为标准 C 做的扩展	/Ze	/Ze,	C/C++——语言——

Language Extensions			/Za	禁用语言扩展
Eliminate Duplicate Strings	为执行过程中程序映像和内存中的相同字符串创建单个副本，从而得到较小的程序，这种优化称为字符串池。在下列代码中， <code>s</code> 和 <code>t</code> 用相同字符串初始化。字符串池使它们指向相同的内存： <code>char *s = "This is a character buffer"; char *t = "This is a character buffer";</code>	否	否, /GF	C/C++——代码生成 ——启用字符串池
Enable Function-Level Linking	<p>此选项允许编译器以封装函数 (COMDAT) 的形式将各个函数打包。链接器要求单独打包为 COMDAT 的函数在 DLL 或 .exe 文件中排除或安排各个函数。</p> <p>可以使用链接器选项 <a href="#">/OPT (优化)</a> 从 .exe 文件中排除未引用的封装函数。</p> <p>可以使用链接器选项 <a href="#">/ORDER (按顺序放置函数)</a> 按指定顺序将封装函数放在 .exe 文件中。</p> <p>如果内联函数实例化为调用（例如，当关闭内联或获取函数地址时出现这种情况），则始终打包内联函数。另外，在类声明内部定义的 C++ 成员函数会自动打包；其他函数不会如此，所以需要选择此选项以便将它们作为封装函数编译。</p>	/Gf[-]	/Gf[-]	C/C++——代码生成 ——启用函数级链接
Enables minimal rebuild	启用最小重新生成。在首次编译期间，编译器在项目的 .idb 文件中存储源文件和类定义之间的依赖项信息。（依赖项信息表明每个源文件所依赖的类定义以及该定义位于哪个.h 文件中。）后面的编译使用存储在 .idb 文件中的信息确定是否需要编译某个源文件（即使它包含已修改的.h 文件）。	否,	否, /Gm	C/C++——代码生成 ——启用最小重新生成
Enable Incremental Compilation	同样通过.IDB 文件保存的信息，只重编译最新改动过的函数。	否	否, /Gi	已移除
Suppress Startup Banner and Information Messages	用以控制参数是否在 output 窗口输出。取消编译器启动时的登录版权标志显示和编译期间的信息性消息显示。	/nologo	否, /nologo	C/C++——常规 ——取消显示启动版权标志
Listing Files				
Generate browse info	同 Geneal 的 Generate browse info			
Exclude Local Variables from Browse Info	是否不将局部变量的信息加入.sbr 文件中	/FR	/FR, /Fr	C/C++——浏览信息 ——启用浏览信息
Listing file type	设置生成的列表信息文件的内容：Assembly-Only Listing 仅生成汇编代码文件 (.ASM 扩展名)；Assembly With Machine Code 生成机器代码和汇编代码文件 (.COD 扩展	/Fa	/FA /Fa	C/C++——输出文件 ——汇编输出

	名 ; Assembly With Source Code 生成源代码和汇编代码文件 (.ASM 扩展名); Assembly, Machine Code, and Source 生成机器码、源代码和汇编代码文件 (.COD 扩展名)。			
<b>Optimizations</b>				
Inline function expansion	内联函数扩展的三种优化 /Ob{0 1 2}: 0, 禁用内联展开(默认情况下是打开的)。1, 只展开标记为 <code>inline</code> , <code>__inline</code> , <code>__forceinline</code> 或 <code>__inline</code> 的函数, 或在类声明内定义的 C++ 成员函数中的函数。2, 展开标记为 <code>inline</code> 或 <code>__inline</code> 的函数和编译器选择的任何其他函数(由编译器自行进行展开, 通常称作自动内联)。在使用 <a href="#">/O1、/O2 (最小化大小、最大化速度)</a> 或 <a href="#">/Ox (完全优化)</a> 时, <code>/Ob2</code> 有效。此选项要求使用 <code>/O1</code> 、 <code>/O2</code> 、 <code>/Ox</code> 或 <code>/Og</code> 来启用优化。	/Ob{0 1 2}	/Ob{0 1 2}	C/C++——优化——内联函数展开
<b>Precompiled Headers</b>				
	预编译头文件的设置。使用预编译可以提高重复编译的速度。VC 一般将一些公共的、不大变动的头文件(比如 <code>afxwin.h</code> 等)集中放到 <code>stdafx.h</code> 中, 这一部分代码就不必每次都重新编译(除非是 Rebuild All)。	/Yc, /Yu		C/C++——预编译头
<b>Preprocessor</b>				
Additional include directories	指定额外的包含目录, 一般是相对于本项目的目录, 如 ..\Include。	/I[path]		C/C++——常规——附加包含目录

## 2.链接参数的设置

General				Vs 2008 对应项
项	含义	默认值	相关 options 参数	
Generate debug info	生成 Debug 信息到 .PDB 文件(具体格式可以在 Category->Debug 中设置);	/DEBUG	否, /DEBUG	链接器——调试——生成调试信息
Ignore All Default Libraries	通知链接器将一个或多个默认库从链接器解析外部引用时所搜索的库列表中移除。	否	否, /NODEFAULTLIB[ :library]	链接器——输入——忽略所有默认库
Link Incrementally	增量链接。控制链接器如何处理增量链接。默认情况下, 链接器以增量模式运行。若要重写默认增量链接, 请指定 <code>/INCREMENTAL:NO</code> 。增量链接的程序在功能上等效于非增量链接的程序。不过, 因为它是为后面的增量链接而准备的, 所以增量链接的可执行 (.exe) 文件或动态链接库 (DLL): 大于非增量链接的程序, 因为有代码和数据的填充。(填充允许链接器增加函数和数据的大小而不用重新创建 .exe 文件。) 可以包含跳转 thunk 以处理函数	/INCREMENTAL	/INCREMENTAL, /INCREMENTAL:N O	链接器——常规——启用增量链接

	重定位到新地址。为了确保最终发布版本不包含填充或 thunk，请非增量链接您的程序。			
Generate Mapfile	<p>默认情况下，链接器用程序的基名称和扩展名.map 命名映射文件。可选的 <i>filename</i> 使您得以重写映射文件的默认名称。</p> <p>映射文件是一个文本文件，包含有关被链接程序的下列信息：</p> <ul style="list-style-type: none"> <li>模块名称，为文件的基名称；</li> <li>时间戳，来自程序文件头（不是来自文件系统）；</li> <li>程序中的组列表，包括每个组的起始地址（节:偏移量的形式）、长度、组名和类；</li> <li>公共符号的列表，包括每个地址（节:偏移量的形式）、符号名称、平直地址和包含符号定义的 .obj 文件；</li> <li>入口点（节:偏移量的形式）；</li> </ul>	/MAP[:filename]	/MAP[:filename]	链接器——调试——生成映射文件
Enable Profiling	生成一个可与“性能工具”探查器结合使用的输出文件。 /PROFILE 使链接器在程序映像中生成一个重定位节。重定位节允许分析器转换程序映像以获取配置文件数据。	/PROFILE	/PROFILE	链接器——高级——探查
<b>Customize</b>				
Force File Output	<p>即使引用了符号但未定义或多次定义符号，/FORCE 选项也通知链接器创建有效的 .exe 文件或 DLL。不带参数的 /FORCE 意味着多个定义和无法解析。</p> <p>/FORCE 选项可以带一个可选参数：</p> <ul style="list-style-type: none"> <li>不论 LINK 是否找到符号的一个以上的定义，均使用 /FORCE:MULTIPLE 创建输出文件。</li> <li>不论 LINK 是否找到未定义的符号，均使用 /FORCE:UNRESOLVED 创建输出文件。如果入口点符号无法解析，则/FORCE:UNRESOLVED 将被忽略。</li> </ul>	/FORCE:[MULTIPLE UNRESOLVED]	/FORCE:[MULTIPLIED UNRESOLVED]	链接器——命令行——附加选项
Print Progress Messages	链接器将有关链接会话进度的信息发送到“输出”窗口。在命令行上，信息被发送到标准输出，并可以重定向到文件。	/VERBOSE[:ICF LIB REF SAFESEH]	/VERBOSE[:ICF LIB REF SAFESEH]	链接器——命令行——附加选项
<b>Debug</b>				
	设置是否生成调试信息，以及调试信息的格式。格式可以有 Microsoft Format、COFF Format（Common Object File Format）和 Both Formats 三种选择	/debugtype[:COFF BOT H]		
Separate Types	表示将 Debug 格式信息以独立的.PDB 文件存放，还是直接放在各个源文件的.PDB 文件中。选中的话，表示采用后者的方式，这种方式调试启动比较快。	/pdptype:sept	否，/pdptype:sept	
<b>Input</b>				链接器——输入
<b>Output</b>				
Base Address	以改变程序默认的基地址（EXE 文件默认为 0x400000，DLL 默认为 x10000000），操作系统装载一个程序时总是	/BASE[:address filename]	/BASE[:address file]	链接器——高级——基址

	试着先从这个基地址开始	e.key]	name,key]	
Entry-Point Symbol	可以指定程序的入口地址，一般为一个函数名（且必须采用__stdcall 调用约定）。一般 Win32 的程序，EXE 的入口为 WinMain，DLL 的入口为 DllEntryPoint；最好让连接器自动设置程序的入口点。默认情况下，通过一个 C 的运行时库函数来实现：控制台程序采用 mainCRTStartup(或 wmainCRTStartup)去调用程序的 main(或 wmain)函数；Windows 程序采用 WinMainCRTStartup( 或 wWinMainCRTStartup) 调用程序的 WinMain( 或 wWinMain，必须采用__stdcall 调用约定)；DLL 采用 _DllMainCRTStartup 调用 DllMain 函数(必须采用__stdcall 调用约定)。	/ENTRY[:function]	/ENTRY[:function]	链接器——高级——入口点
Stack allocations	用以设置程序使用的堆栈大小（请使用十进制），默认为 1 兆字节。reserve 值指定虚拟内存中的总的堆栈分配。对于 x86 和 x64 计算机，默认堆栈大小为 1MB。在 Itanium 芯片组上，默认大小为 4MB。 commit 取决于操作系统所作的解释。在 WindowsNT 和 Windows2000 中，它指定一次分配的物理内存量。提交的虚拟内存导致空间被保留在页面文件中。更高的 commit 值在应用程序需要更多堆栈空间时可节省时间，但会增加内存需求并有可能延长启动时间。对于 x86 和 x64 计算机，默认提交值为 4KB。在 Itanium 芯片组上，默认值为 16KB。	/STACK:reserve[,commit]	/STACK:reserve[,commit]	链接器——系统——堆栈提交大小，堆栈保留大小
Version Information	告诉连接器在 EXE 或 DLL 文件的开始部分放上版本号。	/VERSION:major[.minor]	/VERSIO N:major[.minor]	链接器——常规——版本

### 3.VC 编译语法解释

-优化-

/O1 最小化空间 minimize space  
/Op[-] 改善浮点数一致性 improve floating-pt consistency  
/O2 最大化速度 maximize speed  
/Os 优选代码空间 favor code space  
/Oa 假设没有别名 assume no aliasing  
/Ot 优选代码速度 favor code speed  
/Ob 内联展开（默认 n=0） inline expansion (default n=0)  
/Ow 假设交叉函数别名 assume cross-function aliasing  
/Od 禁用优化（默认值） disable optimizations (default)  
/Ox 最大化选项。 (/Ogityb2 /Gs) maximum opts. (/Ogityb1 /Gs)  
/Og 启用全局优化 enable global optimization  
/Oy[-] 启用框架指针省略 enable frame pointer omission

/Oi 启用内建函数 enable intrinsic functions  
-代码生成-  
/G3 为 80386 进行优化 optimize for 80386  
/G4 为 80486 进行优化 optimize for 80486  
/GR[-] 启用 C++ RTTI enable C++ RTTI  
/G5 为 Pentium 进行优化 optimize for Pentium  
/G6 为 Pentium Pro 进行优化 optimize for Pentium Pro  
/GX[-] 启用 C++ 异常处理（与 /EHsc 相同） enable C++ EH (same as /EHsc)  
/EHs 启用同步 C++ 异常处理 enable synchronous C++ EH  
/GD 为 Windows DLL 进行优化 optimize for Windows DLL  
/GB 为混合模型进行优化（默认） optimize for blended model (default)  
/EHa 启用异步 C++ 异常处理 enable asynchronous C++ EH  
/Gd \_\_cdecl 调用约定 \_\_cdecl calling convention  
/EHc extern“C”默认为 nothrow extern "C" defaults to nothrow  
/Gr \_\_fastcall 调用约定 \_\_fastcall calling convention  
/Gi[-] 启用增量编译 enable incremental compilation  
/Gz \_\_stdcall 调用约定 \_\_stdcall calling convention  
/Gm[-] 启用最小重新生成 enable minimal rebuild  
/GA 为 Windows 应用程序进行优化 optimize for Windows Application  
/Gf 启用字符串池 enable string pooling  
/QIfdiv[-] 启用 Pentium FDIV 修复 enable Pentium FDIV fix  
/GF 启用只读字符串池 enable read-only string pooling  
/QI0f[-] 启用 Pentium 0x0f 修复 enable Pentium 0x0f fix  
/Gy 分隔链接器函数 separate functions for linker  
/GZ 启用运行时调试检查 enable runtime debug checks  
/Gh 启用钩子函数调用 enable hook function call  
/Ge 对所有函数强制堆栈检查 force stack checking for all funcs  
/Gs[num] 禁用堆栈检查调用 disable stack checking calls  
-输出文件-  
/Fa[file] 命名程序集列表文件 name assembly listing file  
/Fo 命名对象文件 name object file  
/FA[sc] 配置程序集列表 configure assembly listing  
/Fp 命名预编译头文件 name precompiled header file  
/Fd[file] 命名 .PDB 文件 name .PDB file  
/Fr[file] 命名源浏览器文件 name source browser file  
/Fe 命名可执行文件 name executable file  
/FR[file] 命名扩展 .SBR 文件 name extended .SBR file  
/Fm[file] 命名映射文件 name map file  
-预处理器-  
/FI 命名强制包含文件 name forced include file  
/C 不吸取注释 don't strip comments  
/U 移除预定义宏 remove predefined macro  
/D{=#} 定义宏 define macro  
/u 移除所有预定义宏 remove all predefined macros

/E 将预处理定向到标准输出 preprocess to stdout  
/I 添加到包含文件的搜索路径 add to include search path  
/EP 将预处理定向到标准输出，不要带行号 preprocess to stdout, no #line  
/X 忽略“标准位置” ignore "standard places"  
/P 预处理到文件 preprocess to file  
-语言-  
/Zi 启用调试信息 enable debugging information  
/Zl 忽略 .OBJ 中的默认库名 omit default library name in .OBJ  
/ZI 启用调试信息的“编辑并继续”功能 enable Edit and Continue debug info  
/Zg 生成函数原型 generate function prototypes  
/Z7 启用旧式调试信息 enable old-style debug info  
/Zs 只进行语法检查 syntax check only  
/Zd 仅要行号调试信息 line number debugging info only  
/vd{0|1} 禁用/启用 vtordisp disable/enable vtordisp  
/Zp[n] 在 n 字节边界上包装结构 pack structs on n-byte boundary  
/vm 指向成员的指针类型 type of pointers to members  
/Za 禁用扩展（暗指 /Op） disable extensions (implies /Op)  
/noBool 禁用“bool”关键字 disable "bool" keyword  
/Ze 启用扩展（默认） enable extensions (default)  
- 杂项 -  
/?, /help 打印此帮助消息 print this help message  
/c 只编译，不链接 compile only, no link  
/W 设置警告等级（默认 n=1） set warning level (default n=1)  
/H 最大化外部名称长度 max external name length  
/J 默认 char 类型是 unsigned default char type is unsigned  
/nologo 取消显示版权消息 suppress copyright message  
/WX 将警告视为错误 treat warnings as errors  
/Tc 将文件编译为 .c compile file as .c  
/Yc[file] 创建 .PCH 文件 create .PCH file  
/Tp 将文件编译为 .cpp compile file as .cpp  
/Yd 将调试信息放在每个 .OBJ 中 put debug info in every .OBJ  
/TC 将所有文件编译为 .c compile all files as .c  
/TP 将所有文件编译为 .cpp compile all files as .cpp  
/Yu[file] 使用 .PCH 文件 use .PCH file  
/V 设置版本字符串 set version string  
/YX[file] 自动的 .PCH 文件 automatic .PCH  
/w 禁用所有警告 disable all warnings  
/Zm 最大内存分配（默认为 %） max memory alloc (% of default)

## 4.VS 生成文件说明及一些小窍门

.ilk——在增量链接时，LINK 更新在第一次增量链接期间创建的.ilk 状态文件。该文件和.exe 文件或.dll 文件具有相同的基名称，并具有扩展名.ilk。

在后面的增量链接期间，LINK 更新.ilk 文件。如果缺少.ilk 文件，则 LINK 执行完全链接并创建新的.ilk 文件。如果.ilk 文件无法使用，则 LINK 执行非增量链接。

.pch——预编译头文件

.sbr——BSCMAKE 的输入文件是.sbr 文件。编译器为编译的每个对象文件(.obj)创建一个.sbr 文件。生成或更新浏览信息文件时，项目的所有.sbr 文件在磁盘上都必须可用。

若要创建包含所有可能信息的.sbr 文件，请指定/FR。若要创建不包含本地符号的.sbr 文件，请指定/FR。如果.sbr 文件包含本地符号，仍然可以通过使用 BSCMAKE 的/EI 选项从.bsc 文件中省略这些符号。

不用执行完全编译便可以创建.sbr 文件。例如，如果指定/FR 或/FR，则可以指定编译器的/Zs 选项来执行语法检查并且仍然生成.sbr 文件。

如果首先压缩.sbr 文件以移除未引用的定义，则生成过程的效率会更高。编译器自动压缩.sbr 文件。

.pdb——程序数据库(PDB)文件保存着调试和项目状态信息，使用这些信息可以对程序的调试配置进行增量链接。当以/ZI 或/Zi (用于 C/C++) 生成时，将创建一个 PDB 文件。

在 VisualC++ 中，/Fd 选项用于命名由编译器创建的 PDB 文件。当使用向导在 VisualStudio 中创建项目时，/Fd 选项被设置为创建一个名为 project.PDB 的 PDB。

如果使用生成文件创建 C/C++ 应用程序，并指定/ZI 或/Zi 而不指定/Fd 时，则最终将生成两个 PDB 文件：

每当创建 OBJ 文件时，C/C++ 编译器都将调试信息合并到 VCx0.PDB 中。插入的信息包括类型信息，但不包括函数定义等符号信息。因此，即使每个源文件都包含公共头文件（如 <windows.h>），这些头文件中的 typedef 也只存储一次，而不是在每个 OBJ 文件中都存在。链接器将创建 project.PDB，它包含项目的 EXE 文件的调试信息。project.PDB 文件包含完整的调试信息（包括函数原型），而不仅仅是在 VCx0.PDB 中找到的类型信息。这两个 PDB 文件都允许增量更新。链接器还在其创建的.exe 或.dll 文件中嵌入.pdb 文件的路径。

.IDB——在首次编译期间，编译器在项目的.idb 文件中存储源文件和类定义之间的依赖项信息。（依赖项信息表明每个源文件所依赖的类定义以及该定义位于哪个.h 文件中。）后面的编译使用存储在.idb 文件中的信息确定是否需要编译某个源文件（即使它包含已修改的.h 文件）。

最小重新生成依赖于类定义不会在包含文件之间更改。类定义对于项目必须是全局的（对于给定类应只有一个定义），因为.idb 文件中的依赖项信息是为整个项目创建的。如果项目中的某个类有多个定义，请禁用最小重新生成。

.map——/MAP 选项通知链接器创建映射文件。

默认情况下，链接器用程序的基名称和扩展名.map 命名映射文件。可选的 filename 使您得以重写映射文件的默认名称。

映射文件是一个文本文件，包含有关被链接程序的下列信息：

模块名称，为文件的基名称

时间戳，来自程序文件头（不是来自文件系统）

程序中的组列表，包括每个组的起始地址（节：偏移量的形式）、长度、组名和类

公共符号的列表，包括每个地址（节：偏移量的形式）、符号名称、平直地址和包含符号定义

的.obj 文件

入口点（节:偏移量的形式）

/MAPINFO 选项指定要包括在映射文件中的附加信息。

.ncb——VisualC++通过.ncb 文件提供有关“类视图”、“对象浏览器”和 IntelliSense 的实时符号信息。自动为每个解决方案创建.ncb 文件，创建的依据是相应解决方案的项目中的源文件。

### 一些小窍门

1)如果你想与别人共享你的源代码项目，但是把整个项目做拷贝又太大。你完全可以删掉以下文件：dsw、.ncb、.opt、.aps、.clw、.plg 文件以及 Debug、Release 目录下的所有文件。

2)当你的 Workspace 中包含多个 Project 的时候，你可能不能直观地、一眼看出来哪个是当前项目。可以如下设置：Tools->Options->Format，然后在 Category 中选择 Workspacewindow，改变其默认的字体（比如设成 Fixedsys）就行了。

3)如何给已有的 Project 改名字？将该 Project 关掉。然后以文本格式打开.dsp 文件，替换原来的 Project 名字即可。

4)VC6 对类成员的智能提示功能很有用，但有时候会失灵。你可以先关掉项目，将.clw 和.ncb 删掉，然后重新打开项目，点击菜单项 View->ClassWizard，在弹出的对话框中按一下“AddAll”按钮；重新 RebuildAll。应该可以解决问题。