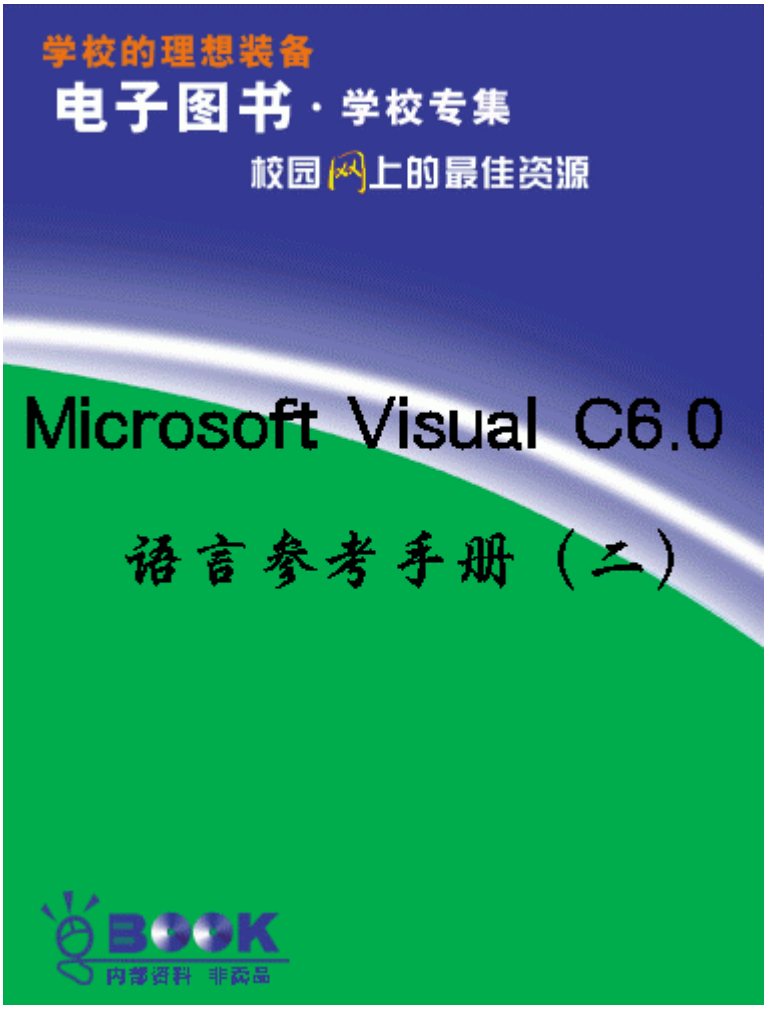

学校的理想装备
电子图书·学校专集
校园网上的最佳资源
Microsoft Visual C6.0
语言参考手册（二）
BOOK
内部资料 非卖品

返回总目录

Microsoft Visual C++ 6.0
语言参考手册

第二部分目录

引言　C++语言参考手册的组织

第 1 章,“词法规定”,介绍编译器可识别的 C++程序的基本元素。这些元素被称为“词法元素”,用于构成语句、定义、说明等,它们用于构造完整的程序。
第 2 章,“基本概念”,解释以下概念:范围、连接、程序的开始和终止、存储类和类型。这些概念是理解 C++的关键。同时还介绍了本书中所用的术语。
第 3 章,“标准转换”描述编译执行时内部的、或“基本的”类型之间的类型转换。本章还说明了编译器在指针、引用及成员指针类型之间是如何执行转换的。
第 4 章,“表达式”描述 C++表达式,即用于计算值、设计对象或函数、或产生其它副作用的运算符和操作数的序列。
第 5 章,“语句”,解释用于控制程序如何执行以及按什么顺序执行的 C++程序的组成元素,包括表达式语句、空语句、复合语句、选择语句、循环语句、跳转语句以及说明语句。
第 6 章,“说明”,是讨论用完整的说明如何构成说明语句的三章内容之一。本章介绍以下问题:存储类指示符、函数定义、初始化、枚举、类、结构和联合的说明,以及 typedef 说明,相关的信息可在第 7 章“说明符”及附录 B“Microsoft 特殊修饰符”中找到。
第 7 章,“说明符”,解释说明语句中用于命名对象、类型或函数的部分。
第 8 章,“类”,介绍 C++类,C++把用 class、struct 或 union 关键字说明的对象作为一个类类型,本章说明如何使用这些类类型。

第 9 章,“派生类”,包括继承的详细内容:通过继承方法可定义一个新的类型,拥有已存在类型的全部属性,再加上添加的任何新的属性。
第 10 章,“成员访问控制”,解释你如何控制访问类的成员,使用访问控制指示符可有助于产生更强健的代码,因为你可以限制一个对象状态可变更方式的数目。
第 11 章,“特殊成员函数”,描述对类类型而言是特有的特殊函数。这些特殊函数执行初始化(构造函数)、清除(析构函数)以及转换。本章还描述用于动态存储器分配的 new 和 delete 运算符。
第 12 章,“重载”,解释 C++的一个特征,即允许用相同的名称但不同的参量定义一组函数。调用该组中的哪个函数取决于实际函数调用时的参量表。此外,本章还包括重载运算符,即用 C++运算符定义你自己的行为的一种机制。
附录 A,“语法总结”,是包含 Microsoft 扩充的 C++语法的总结。语法的某些部分贯穿于本手册的“语法”一节。
附录 B,“Microsoft 特殊修饰符”,描述了 Microsoft C++的特殊修饰符,这些修饰符用于控制存储器寻址、转换调用等。
附录 C,“COM 支持类的编译”,是用于支持某些部件对象模型类型的四种 Microsoft 特殊类的参考。
附录 D,“图表”,包括以下图表:ASCII 字符代码、ASCII 多种语言代码、ANSI 字符代码和键盘代码。

本手册的范围

C++和 C 语言一样,是一种极大地依赖于提供以下功能的丰富的库函数集的一种语言,这些功能是:

- 可移植的操作系统接口(文件及屏幕 I/O)
- 字符串及缓冲区操作
- 浮点数学变换
- 其它支持功能

有关运行库和输入输出流类的信息,可参见“ Microsoft Visual C++ 6.0 运行库参考手册 ”。
关于 Microsoft 的基类信息,可参看两卷“ Microsoft Visual C++ 6.0 MFC 类库参考手册 ”。
这三卷书都是“ Microsoft Visual C++ 6.0参考库手册 ”的一部分。

本手册中的特定术语

本手册中,术语“ 参量 ”指传递给一个函数的实体。在某些情况下,它被指定为“ 实际的 ”或“ 形式的 ”,即分别指在函数调用时指定变量表达式以及在函数定义时指定变量说明的两种情况。
术语“ 变量 ”指简单的 C 语言类型数据对象,术语“ 对象 ”既指 C++对象,也指变量,是一个相容的术语。有关术语的更多的信息,可参见第 2 章的“ 术语 ”以

及“基本概念”。

第 1 章　词 法 规 定

本章介绍 C++程序的基本元素,你使用这些被称为“词法元素”或“符号”的元素去构造语句、定义、说明等,并使用它们构造完整的程序。本章讨论以下词法元素:

- 语言符号
- 注释
- 标识符
- 关键字
- 标点符号
- 运算符
- 文字

本章还包括表 1.1,该表中给出了 C++运算符的优先级和结合律(优先级从最高到最低排列)。有关运算符的完整讨论参见第 4 章“表达式”。

文件翻译概述

C++程序和 C 程序一样,由一个或多个文件组成,每个文件按如下概念顺序被翻译(实际的顺序遵循“as if”规则:只要遵照这些步骤则翻译必然发生):

1．词法符号化:此翻译阶段执行字符映射及三字母处理、行分割及符号化。
2．预处理:此翻译阶段调入由#include 命令引用的辅助源文件,处理“字符串化”和“字符化”命令,执行符号传递和宏扩展(若需要更多信息则可见本册后面“预处理器参考”中的“预处理器命令”),预处理的结果是一组有序的符号,合在一起定义了一个“转换单元”。
预处理器命令总是以数字符号(#)开头(即,一行的第一个非空白字符必须是一个数字符号)。一行只能出现一条预处理器命令。例如:

```
#include <iostream.h>  //在转换单元包含 iostream.h 文本。
#define NDEBUG       //定义 NDEBUG(NDEBUG 包含空文本字符串)。
```

3．产生代码:此翻译阶段是用预处理阶段生成的符号来生成目标代码。
在此阶段,执行源代码的语法及语义检查。
若需要更多的信息可参阅本手册后面“预处理器参考”中的“翻译阶段”。
C++预处理器是 ANSI C 预处理器的严格的超集,但是 C++预处理器在某些实例上有所不同。
以下所列的是 ANSI C 与 C++预处理器的几处不同点:

- 支持单行注释,详见“注释”
- 一种预定义宏__cplusplus 仅为 C++所定义。详见本卷后面“预处理器参考”中的“预定义宏”。
- C 预处理器不能识别 C++的.*.->*和::运算符。有关运算符详见本章后面的“运算符”以及第 4 章“表达式”。

语言符号

一个语言符号是 C++程序中对编译器有意义的最小单元。C++语法分析程序识别以下这些种类的语言符号:标识符、关键字、文字、运算符、标点符号及其它分隔符。这一串符号构成一个转换单元。
语言符号通常由"空白"分开,空白可以是一个或多个:

- 空格
- 水平或垂直制表符
- 换行符
- 换页符
- 注释

语法
符号:
关键字
标识符
常量
运算符
标点符号
预处理符号:
头名称
标识符
pp 数

字符常量
字符串文字
运算符
标点符号
每个非空白字符不能是以上中的一种。

语法分析程序通过对输入字符从左到右扫描,尽可能地从输入流中创建最长的语言符号集来分割语言符号。考虑以下代码段:

```
a=i+++j;
```

写此代码的程序员可能有以下两种语句目的:

```
a=i+(++j)

a=(i++)+j
```

由于语法分析程序从输入流尽可能创建最长的符号集,所以它选择第二种解释,得到符号 i++、+和 j。

注　释

注释是编译器忽略的文本,但对程序设计者而言却非常有用。注释通常对代码进行标注以便以后参考。编译器把它们作为空白对待。在调试时你可以使用注释使得特定行代码不运行;但是,#if#endif 预处理命令在这方面更好,因为你可以括起包含注释在内的代码,但是你不能嵌套注释。

一个 C++注释按以下的一种方式书写:

- /*(斜杠,星号)字符,后面跟随任意字符序列(包括新行),再跟上*/字符。这种语法与 ANSI C 的一样。
- //(两个斜杠)字符,后面跟随任意字符序列。一个不是紧随后面反斜杠的新行将结束此种形式的注释。因此,它通常被称为“单行注释”。

注释字符(/*,*/和//)在字符常量:字符串文字或注释内部没有任何特殊意义。因此,使用第一种语法的注释不能被嵌套使用,考虑以下例子:

```
/* Intent: Comment out this block of code.
  Problem:Nested comments on each line of code are illegal.
  FileName=String("hello.dat"); /*Initialize file string*/
  cout << "File:" << FileName << "\n";/*Print status message*/
*/
```

执行代码不能编译,因为编译器扫描输入流,从第一个/*到第一个*/,则认为它是一个注释。在这种情况下,第一个*/出现在注释 Initialize file string 的末尾。那么对最后一个*/,则再没有一个/*与其配对了。

注意:注释的单行形式(//)后面跟随一个续行符(/)会产生出人意料的结果。考虑以下代码:

```
#include<stdio.h>

void main()
{
   printf("This is a number %d",//\
```

```
          5);
}
```

在预处理之后,前面的代码有错并显示如下:

```
#indude<stdio.h>

void main()
{
   printf("This is a number %d",
}
```

标识符

一个标识符是一个用于对以下内容之一进行编码的字符序列:

- 对象或变量名称
- 类、结构或联合名称
- 枚举的类型名称
- 类、结构、联合或枚举的成员
- 函数或类成员函数
- typedef 名称
- 标号名称
- 宏名称
- 宏参数

语法
标识符:
    非数字
    标识符 非数字
    标识符 数字
非数字:如下之一
    - a b c d e f g h i j k l m
    n o p q r s t u v w x y z
    A B C D E F G H I J K L M
    N O P Q R S T U V W X Y Z
数字:如下之一
    0 1 2 3 4 5 6 7 8 9
Microsoft 特殊处→
仅 Microsoft C++标识符的前 247 个字符是有意义的。由于用户定义类型的名称由编译器“修饰”以保存类型信息的事实而使得这一限制变得复杂。结果名称,包括类型信息,不能超过 247 个字符长度。(详见联机“Microsoft Visual C++6.0 程序员指南”中的“修饰名称”)。能够影响修饰的标识符长度的因素有:

- 不论标识符表示一个用户定义类型的对象还是表示用户定义类型的派生类型的对象。
- 不论标识符表示一个函数还是表示函数派生的类型。
- 一个函数的参量的个数。

Microsoft 特殊处结束
一个标识符的首字符必须是字母,不论大写或小写,或一个下划线(_)。由于 C++标识符对大小写敏感,所以 fileName 和 FileName 是不同的。
标识符不能与关键字使用一样的拼写和大小写方式。标识符中含有关键字是合法的,例如,pint 是一个合法的标识符,尽管它包含了关键字 int。
在一个标识符的开头使用连续的两个下划线(__)或一个下划线打头后跟一个大写字母,则在所有范围中为 C++的实现而保留,你应当避免使用一个打头的下划线带一个小写字母作为有文件范围的名称,因为它可能与现在或将来保存的标识符相冲突。

关键字

关键字是有特殊含义的预定义的保留标识符。它们不能被用作你程序中的标识符。以下是 C++保留的关键字:
语法
关键字:如下之一

asm*	auto	bad_cast	bad_typeid
bool	break	case	catch
char	class	const	const_cast
continue	default	delete	do
double	Dynamic_cast	else	enum
except	explicit	extern	false

续表

Finally	float	for	int
goto	if	inline	
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	type_info
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	while		

*为与其它 C++实现相兼容而保留的,但没有实现。使用__asm。

Microsoft 特殊处→

在 Microsoft C++中,由两个下划线开头的标识符是为编译器的实现而保留的。因此,Microsoft 规定是在 Microsoft 特定关键字前加上双下划线,这些字不可用作标识符名。

allocate[3]	__inline	property[3]
__asm[1]	__int8	selectany[3]
__based[2]	__int16	__single_inheritance
__cdecl	__int32	__stdcall
__declspec	__int64	thread[3]
dllexport[3]	__leave	__try
dllimport[3]	__multiple_inheritance	uuid[3]
__except	naked[3]	__uuidof
__fastcall	nothrow[3]	__virtual_inheritance
__finally		

1. 替代 C++ asm 语法
2. __based 关键字被限于 32 位目标编译使用
3. 这些当与__declspec 一起使用时是特殊的标识符,在其它上下文中的应用则没有限制。

Microsoft 扩充部分在缺省状态下是允许的,为确保你的程序完全可移植,你可以通过指定 ANSI 可兼容性/Za 命令行选项,使得编译期间 Microsoft 扩充部分不可用。当你这样做时,Microsoft 特定关键字是不可用的。
当 Microsoft 扩充部分使能时,你可以在你的程序里使用前面罗列的关键字。
对 ANSI 应用,这些关键字被冠以双下划线。为向后兼容,所有关键字除__except、__finally、__leave 和__try 外,其单下划线版本均被支持,此外,__cdecl 不带前面下划线的也可用。
Microsoft 特殊处结束

标点符号

C++中的标点符号对编译器而言具有语法及语义意义,但其自身并不指定产生数值的某种操作。某些标点符号,不论是单独使用或以组合方式使用,均可作为 C++ 的运算符或对预处理器有重要意义。

语法

标点符号:如下之一

```
! % ^ & * ( ) - + = { } | ~
[ ] \ ; ' : " < > ? , . / #
```

标点符号[]、()和{ }在翻译第 4 阶段后必须成对出现。

运算符

运算符指定执行以下一种求值运算:

- 一个操作数(单目运算符)
- 两个操作数(双目运算符)
- 三个操作数(三目运算符)

C++语言包含了 C 中的所有运算符,还添加了几个新的运算符,表 1.1 列出了 Microsoft C++中可用的运算符。

运算符按照严格的优先级来确定包含这些运算符的表达式的运算顺序。运算符要么与其左边的表达式结合,要么与其右边的表达式结合,这被称为“结合律”。

同组中的运算符具有相同的优先级,在表达式中从左向右运算,除非用括号()明确指定其顺序,表 1.1 给出了 C++运算符的优先级及结合律(优先级从高到低)。

表 1.1　C++运算符优先级与结合性

运算符	名称或含义	结合律
::	范围分辨	无
::	全局的	无
[]	数组下标	从左到右
()	函数调用	从左到右
()	类型转换	无
.	成员选择(对象)	从左到右
->	成员选择(指针)	从左到右
++	后缀增 1	无
--	后缀减 1	无
new	分配对象	无
delete	撤消对象分配	无
delete[]	撤消对象分配	无
++	前缀增 1	无
--	前缀减 1	无
*	取消关联	无
&	取地址	无
+	单目运算符加	无

续表

-	算术非运算(单目运算符)	无
!	逻辑非	无
~	按位求补	无
sizeof	对象的大小	无
sizeof()	类型的大小	无
typeid()	类型名	无
(类型)	类型强制(转换)	从右到左
const-cast	类型强制(转换)	无
dynamic-cast	类型强制(转换)	无
reinterpret-cast	类型强制(转换)	无
static-cast	类型强制(转换)	无
.*	利用指针到达类成员(对象)	从左到右
->*	取消关联类成员指针	从左到右
*	乘	从左到右
/	除	从左到右
%	余数(取模数)	从左到右
+	加	从左到右
-	减	从左到右
<<	左移	从左到右

续 表

>>	右 移	从 左 到 右
<	小 于	从 左 到 右
>	大 于	从 左 到 右
<=	小 于 等 于	从 左 到 右
>=	大 于 等 于	从 左 到 右
==	等 于	从 左 到 右
!=	不 等	从 左 到 右
&	按 位 与	从 左 到 右
^	按 位 异 或	从 左 到 右
|	按 位 或	从 左 到 右
&&	逻 辑 与	从 左 到 右
||	逻 辑 或	从 左 到 右
e1?e2:e3	条 件	从 右 到 左
=	赋 值	从 右 到 左
*=	乘 后 赋 值	从 右 到 左
/=	除 后 赋 值	从 右 到 左
%=	取 模 后 赋 值	从 右 到 左
+=	加 后 赋 值	从 右 到 左
-=	减 后 赋 值	从 右 到 左
<<=	左 移 后 赋 值	从 右 到 左
>>=	右 移 后 赋 值	从 右 到 左

续表

&=	按位与后赋值	从右到左
|=	按位或后赋值	从右到左
^=	按位异或后赋值	从右到左
,	逗号	从左到右

文　字

非变元的程序元素被称为“文字”或“常量”。在这里术语“文字”和“常量”可互换使用。
文字有四个主要类别:整型、字符型、浮点型和字符串文字。
语法
文字:
整型常量
字符常量
浮点常量
字符串文字

整型常量

整型常量是没有小数部分或指数的常量数据元素,总是以数字开头。你可以按十进制、八进制或十六进制形式指定整型常量。可指定为有符号的或无符号类型

以及长型或短型。

语法

整型常量：

十进制常量 整数后缀 opt

八进制常量 整数后缀 opt

十六进制常量 整数后缀 opt

‘c字符序列'

十进制常量：

非0数字

十进制常量 数字

八进制常量：

0

八进制常量 八进制数字

十六进制常量：

0x 十六进制数字

0X 十六进制数字

十六进制常量 十六进制数字

非0数字：如下之一

1 2 3 4 5 6 7 8 9

八进制数字：如下之一

0 1 2 3 4 5 6 7

十六进制数字：如下之一

0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

整数后缀：

无符号后缀 长型后缀 opt

长型后缀 无符号后缀 opt

无符号后缀:如下之一

u U

长型后缀:如下之一

l L

64 位整数后缀

i64

用八进制或十六进制计数法指定整型常量时,使用一个前缀指明基数。要指定一给定整型类型的整型常量,则使用一个后缀指定类型。

指定一个十进制常量,须以一个非 0 数字开头。例如：

```
int i=157;      //十进制常量
int j=0198;     //不是一个十进制数,是一个错误的八进制常量
int k=0365;     //打头的 0 指定的是八进制常量,而不是十进制数
```

指定一个八进制常量,则以 0 开头,后面跟随从 0-7 范围内的数字序列。在指定一个八进制常量时,数字 8 和 9 是错误的。例如：

```
int i=0377;   //八进制常量
int j=0397;   //错误:9 不是一个八进制数符
```

指定一个十六进制常量,则以 0x 或 0X(X 的大小写无关)开始,后面跟随 0-9 和 a(或 A)-f(或 F)范围内的数符序列。十六进制数符 a(或 A)到 f(或 F)代表的数值范围是 10-15。例如:

```
int i=0x3fff;     //十六进制常量
int j=0X3FFF;  //与 i 相等
```

指定为无符号类型,则使用 u 或 U 作后缀指定为长型,则用 l 或 L 作后缀。例如:

```
unsigned uVal=328u;                  //无符号数
long lVal=0x7FFFFFL;               //长型数值作为十六进制常量
unsigned long ulVal=0776745ul;  //无符号长型数值
```

字符常量

字符常量是“源字符集”中的一个或几个成员,源字符集是一个程序中使用的字符集,字符常量由单引号(')括起。它们被用于表示“执行字符集”即程序执行所用机器的字符集中的字符。

Microsoft 特殊处→

对 Microsoft C++而言,源字符集与执行字符集都是 ASCII 码。

Microsoft 特殊处结束

有三种字符常量:

- 普通字符常量
- 多字符常量
- 宽字符常量

注:使用宽字符常量替代多字符常量可确保可移植性字符常量被指定为单引号括

起的一个或多个字符,例如:

```
char ch='x';  //指定普通字符常量
int mbch='ab';//指定依赖于系统的多字符常量
wchar_t wcch=L'ab'; //指定宽字符常量
```

注意 mbch 的类型是 int,如果它被说明为 char 类型,则第二个字节将被保留起来。一个多字符常量有四个有意义的字符,若指定的字符数超过四则将产生错误信息。

语法

字符常量:

    c‘字符序列'
    L‘c字符序列'

c字符序列:

    c字符
    c字符序列 c字符

c字符:

    源字符集中除单引号(')、反向斜杠(\)或换行符外的任何字符
    转义序列

转义序列:

    简单转义序列
    八进制转义序列
    十六进制转义序列

简单转义序列:如下之一

\' \" \? \\
\a \b \f \n \r \t \v
八进制转义序列:
\八进制数字
\八进制数字 八进制数字
\八进制数字 八进制数字 八进制数字
十六进制转义序列:
\x 十六进制数字
十六进制转义序列 十六进制数字
Microsoft C++支持普通字符、多字符和宽字符常量,使用宽字符常量去指定扩充执行字符集(例如要支持国际应用)的成员。普通字符常量为类型 char,多字符常量类型为 int,宽字符常量类型为 wchar_t。(wchar_t 类型定义在标准头文件 STDDEF.H, STDLIB.H 和 STRING.H 中,但是宽字符函数的原型仅在 STDLIB.H 中)。
在指定普通的和宽字符常量时唯一的不同之处是宽字符常量字母以 L 打头。例如:

```
char schar='x';//普通字符常量
wchar_t wchar=L'\x81\x19'; //宽字符常量
```

表 1.2 给出的是依赖于系统的或不允许在字符常量中使用的保留的或非显示的字符,这些字符须由转义序列表示。

表 1.2　C++保留的或非显示的字符

字符	ASCII 表示	ASCII 值	转义序列
换行	NL(LF)	10 或 0x0a	\n
水平制表符	HT	9	\t
垂直制表符	VT	11 或 0x0b	\v
退格键	BS	8	\b
回车	CR	13 或 0x0d	\r
换页符	FF	12 或 0x0c	\f
报警	BEL	7	\a
反斜杠	\	92 或 0x5c	\\
问号	?	63 或 0x3f	\?
单引号	'	39 或 0x27	\'
双引号	"	34 或 0x22	\"
八进制数	*ooo*	—	\ooo
十六进制数	*hhh*	—	\xhhh
空字符	NUL	0	\0

如果反向斜杠后面的字符不是指定的一个合法转义序列,则结果在实现时确定。Microsoft C++中,跟在反向斜杠后的字符作为文字被接收,尽管转义不存在,而且出现层 1 的警告(“无法识别的字符转义序列”)。

以*ooo* 形式指定的八进制转义序列包括一个反向斜杠和一个、两个或三个八进制字符;以\x*hhh* 形式指定的十六进制转义序列包括字符\x 和其后跟随的十六进制数字序列;与八进制转义序列不同的是在十六进制转义序列中对十六进制数字

的数目没有限制。
八进制转义序列由第一个非八进制数字的字符结束或当三个字符可见时,例如:

```
wchar_t och=L'\076a';   //序列在 a 处结束
char ch='\233';         //序列在 3 个字符后结束
```

相似地,十六进制转义序列在第一个非十六进制数字处结束。由于十六进制数字包括字母 a～f(和 A～F),所以要确定转义序列在所想要的数字处结束。
由于单引号(')用于括住字符常量,所以用转义序列\'来表示被包含的单引号。双引号(")可以不用转义序列表示。反向斜杠(\)被放在行尾时是作为续行符的,如果你想要一个反向斜杠出现在一个字符常量中,你必须在一行里打上两个反斜杠(关于行继续的更多信息可参阅本卷后面“预处理器参考”中的“翻译阶段”)。

浮点常量

浮点常量指定的数值必须带有小数部分,这些数值包含小数点(.),且可以包含指数。
语法
浮点数量:
    小数常量 指数部分 $_{opt}$ 浮点后缀 $_{opt}$
    数字序列 指数部分 浮点后缀 $_{opt}$
小数常量:
    数字序列 $_{opt}$.数字序列
    数字序列

指数部分：
    e 符号 $_{opt}$ 数字序列
    E 符号 $_{opt}$ 数字序列
符号:如下之一
    + -
数字序列：
    数字
    数字序列 数字
浮点后缀:如下之一
    f l F L
浮点常量有一个“尾数”指定数的值,一个“指数”指定数的量值,以及一个可选后缀指定常量的类型。尾数指定为一个字符序列后带小数点,带上表示数的小数部分的可选的数字序列。例如：
18.46
38.
如果出现指数,则是用作 10 为底的幂次来指定数的量值,如下例所示：
18.46e0 //18.46
18.46e1 //184.6
如果出现指数,尾部的小数点可以不要,整个数像 18E0 一样。
浮点常量的缺省类型是 double。相应地使用后缀 f 或 l(或 F 或 L,后缀不是大小写敏感的),常量可分别被指定为 float 或 long double。
尽管 long double 和 double 有着同样的表示,但它们是不同的类型。例如,你可

以重载函数：
```
void func(double);
```
和
```
void func(long double);
```

字符串文字

字符串文字是由双引号括起的源字符集中的 0 个或多个字符组成，一个字符串文字表示一个字符序列，合在一起构成一个以空格结尾的字符串。
语法
字符串文字：
    "s 字符序列 $_{opt}$"
    L"s 字符序列 $_{opt}$"
s 字符序列：
    s 字符
    s 字符序列 s 字符
s 字符：
    除双引号(")、反斜杠(\)或换行符之外的源字符集中的任何成员
    转义序列
C++字符串有这些类型：

- char[*n*]数组，n 为字符串(以字符方式)的长度加 1，因为用结束符'\0'标识字符串结尾。
- wchar_t 数组，为宽字符串。

修改字符串常量的结果是不确定的,例如:

```
char *szStr="1234";
szStr[2]= 'A'; //结果不确定
```

Microsoft 特殊处→

在某些情况下,相同的字符串文字可以被“合并”以节省执行文件的空间,在字符串文字合并时,编译器使所有对特定字符串文字的引用都指向存储器中相同的位置,而不是让每个引用指向各自的字符串文字实例。/GF 编译选项使能字符串合并。

Microsoft 特殊处结束

当指定字符串文字时,相邻的字符串将被连接起来。因此,如下说明:

```
char szStr[]="12" "34";
```

与下面的说明完全相同:

```
char szStr[]="1234";
```

这种相邻字符串的连接使得可以很容易地通过多个行指定长字符串。

```
cout <<"Four score and seven years"
       "ago, our forefathers brought forth"
       "upon this continent a new nation.";
```

在前一个例子中,整个字符串"Four score and seven years ago ,our forefathers brought forth upon this continent a new nation."一起被分割处理。这个字符串还可以按如下方式用行分割符来指定:

```
cout<<"Four score and seven years\
ago, our forefathers brought forth\
```

upon this continent a new nation.";
常量中所有相邻的字符串被连接后,NULL 字符'\0'被添加,为 C 字符串处理函数提供字符串结束标记。
当第一个字符串包含转义字符时,字符串的连接可能产生出人意料的结果,考虑以下两个说明:

```
char szStr1[]="\01" "23";
char szStr2[]="\0123";
```

尽管很自然地认为 szStr1 和 szStr2 包含相同的值,但它们实际所包含的值由图 1.1 给出。

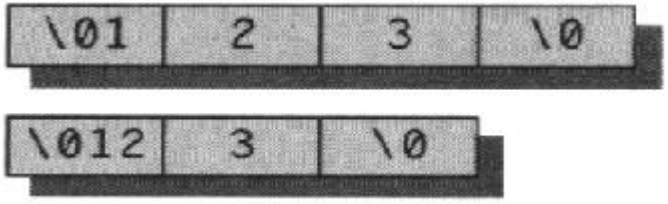


图 1.1　转义和字符串的连接

Microsoft 特殊处→
一个字符串文字的最大长度约为 2048 个字节,这个限制适用于 char[]和 wchar_t[]类型的字符串。如果一个字符串文字包含被双引号括起的部分,则预处理器将这个部分连接成一个单字符串,而且对每行连接,在总字节数上加上一个额外的字节。
例如假定一个字符串包含 40 行,每行 50 个字符(2000 个字符),以及有一行为 7 个字符,且每行均由双引号括起,合计为 2007 个字节,再加上一个字节的结束空

字符,总共 2008 个字节。在字符串进行连接时,前面的 40 行每行添加一个额外的字符到总数中,这样得到总共 2048 个字节。(额外的字符并非真的写到字符串中)。但是,注意,如果用续行符(\)代替双引号,则预处理器不会为每行添加一个额外的字符。

Microsoft 特殊处结束

通过计数字符个数再为终止符'\0'加 1 或为 wchar_t 类型加 2,则可确定字符串对象的大小。

因为双引号(")用于括住字符串,所以对被括起的双引号本身则使用转义序列(\")来表示。单引号(')可以不用转义序列表示。反向斜杠被置于行尾时是一个续行符,如果你想要反向斜杠出现在一个字符串中,则你必须打上两个反斜杠(\\)(关于续行问题参见本书后面“预处理器引用”中的“翻译阶段”部分)。

为了指定一个类型为宽字符(wchar_t[])的字符串,则用带字符 L 的开双引号开头。例如:

```
wchar_t wszStr[]=L"lalg";
```

所有在“字符常量”中列出的普通转义代码在字符串常量中都是合法的,例如:

```
cout<<"First line\nSecond line";
cout<<"Error! Take corrective action\a";
```

由于转义代码在第一个非十六进制数字的字符处终止,所以用嵌有十六进制转义代码来指定字符串常量可能造成意想不到的结果,以下例子意在创建一个包含 ASCII 5 的字符串文字,后面跟随字符 five:

```
\x05five"
```

实际的结果是十六进制的 5F,即 ASCII 代码的下划线,后面跟随字符 ive。以下

例子生成的是所希望的结果：

```
"\005five"//使用八进制常量
"\x05" "five"//使用字符串分割
```

第 2 章　基 本 概 念

本章意在解释对理解 C++至关重要的一些概念,C 程序员将对这些概念中的很多内容非常熟悉,但存在一些细微的差别可能导致意想不到的程序结果,包含了以下论题:

- 术语
- 说明和定义
- 范围
- 程序和连接
- 开始和结束
- 存储类
- 类型

附加的论题包括 l 值、r 值和数值范围界限。

术　语

本书所使用的 C++术语在表 2.1 中给出了定义:

表 2.1　C++术语

术语	含义
说明	说明向一个程序引入名称及它们的类型,而不必定义相关联的对象或函数。但很多说明被作为定义
定义	定义提供信息而允许编译器为对象分配存储器或为函数产生代码
生存期	一个对象的生存期是对象存在包括创建和析构的时间阶段
连接	名称可以有外部连接、内部连接或无连接,在一个程序内(一套转换单元),只有带外部连接的名称才表示相同的对象或函数。在一个转换单元内,带有内部或外部连接的名称均可表示相同的对象或函数(当函数重载时除外)。(关于转换单元的更多信息,参阅本书后面“预处理器参考”中“翻译阶段”)。无连接的名称只表示唯一的对象或函数
名称	名称表示一个对象、函数、重载的函数集、枚举元、类型、类成员、模板、值或标号、C++程序用名称指出与它们相关联的语言元素。名称可以是类型名称或标识符

续表

对象	一个对象是用户定义类型(一个类类型)的一个实例(一个数据项)。一个对象和一个变量之间的区别是变量保留状态信息,而对象还可以拥有动作。本手册给出的对象和变量之间的区别是:“对象”指用户定义类型的实例,而“变量”指基本类型的实例。在对象或变量均可用情况下,术语“对象”被用作为包容性术语,意为“对象或变量”
范围	名称只能在程序上下文的特定区域内使用,这些区域被称为名称的范围
存储类	被命名的对象的存储类指其生存期、初始化及在某特定情况下指其连接
类型	带有相关类型的名称指的是存储在一个对象中或由一个函数返回的值的含义
变量	一个变量是一个基本类型(如 int、float 或 double)的数据项,变量存储状态信息但不为信息如何被处理而定义动作。看前面列出的“对象”项来获得有关术语“变量”和“对象”是如何在本文档中被运用的

说明和定义

说明告诉编译器一个程序元素或名称存在,定义指定名称描述的是什么代码或数据,一个名称必须先说明而后才能使用。

说　明

一个说明向一个程序中引入一个或多个名称,说明在一个程序可出现不止一次,因此,可以为每个编译单元说明类、结构、枚举类型和其它用户定义类型。多说明的限制是所有说明必须可标识。说明还可作为定义,除了以下说明:

- 是一个函数原型(没有函数体的函数说明)
- 有 extern 指示符但没有初始表达式(对象和变量)或函数体(函数),这意味着该定义在当前转换单元里不是必须的,且给出了名称外部连接。
- 是类说明内部的静态数据成员。

由于静态类数据成员是由类的所有成员共享的离散变量,它们必须在类说明外部加以定义和初始化(关于类和类成员的更多的信息参见第 8 章“类”)。

- 是一个不带定义的类名说明,例如 class T;
- 是 typedef 语句

说明同时也是定义的例子有:

```
//说明和定义整型变量 i 和 j
int i;
int j=10;

//说明枚举 suits
enum suits {Spades=1,Clubs,Hearts,Diamonds};
```

```
//说明类 CheckBox
class CheckBox:public Control
{
public:
      Boolean IsChecked();
   virtual int ChangeState()=0;
}
```

有些说明不是定义:

```
extern int i;
char *strchr(const char *Str,const char Target);
```

定　义

一个定义是一个对象或变量、函数、类或枚举的唯一规格,由于定义必须唯一,所以一个程序对于一给定的程序元素只能包含一个定义。

说明和定义之间可以是多对一的关系。有两种情况下一个程序元素可被说明但不被定义:

- 函数被说明但从来没有用函数调用或带有函数地址的表达式引用过。
- 一个类仅用于其定义不要求被知道的情况,但是,类必须被说明,以下代码描述了这种情况:

```
class WindowCounter;//前向引用,没有定义
```

```
class Window
{
  static WindowCounter WindowCounter;//WindowCounter 定义不被要求
}
```

范　围

C++的名称仅能用于一个程序的特定区域,这个区域被称为名称的“范围”。范围确定一个不表示静态范围的对象的名称的生存期,范围还指出了当类构造和析构函数被调用时,以及对范围而言是局部的变量被初始化时一个名称的可见性(详见第 11 章“特殊成员函数”中“构造”和“析构”部分)。

有五种范围:

- 局部范围,在一个块内说明的名称仅在此块中以及包含在此块中的块内是可访问的,且仅从说明后开始。一个函数在函数最外层块范围内的普通参量的名称具有局部范围,就好象它们在包含函数体的块内被说明一样,考虑以下代码段:

```
{
   int i;
}
```

由于 i 的说明包含在花括号括起的块内,所以 i 有局部范围而且永远不能被访问,因为在闭花括号前没有代码对其进行访问操作。

- 函数范围,标号是唯一具有函数范围的名称,它们可在一个函数内的

任意位置被使用,但在函数外部则不能访问。

- 文件范围,任何在所有块或类的外部说明的名称具有文件范围,在其说明后,在转换单元的任何位置均可访问。带有文件范围的不说明静态对象的名称通常被称为“全局”文字。
- 类范围,类成员名具有类范围。类成员函数仅能用对象上的成员选择运算符(.或->)或成员指针运算符(.*或->*)或指向那个类的对象的指针进行访问,非静态类成员数据被认为相对于那个类的对象是局部的。

考虑以下类说明:

```
class Point
{
    int x ;
    int y ;
}
```

类成员 x 和 y 被认为是在类 Point 范围内。

- 原型范围,在函数原型中说明的名称仅到了原型结尾处才是可见的。以下原型说明了两个名称(szDest,szSource);这些名称在原型结尾处便超出范围了:

```
char *strcpy(char *szDest,const char *szSource);
```

说明点

一个名称被认为就在其说明符后在其(可选的)初始化之前马上说明。(有关说明

符的更多信息参见第 7 章“说明符”)。一个枚举器被认为应在命名它的标识符后,但在其(可选的)初始化之前立即说明。
考虑这个例子:

```
double dVar=7.0;
{
   void main()
{
  double dVar=dVar;
}
```

如果说明点在初始化之后,则局部 dVar 将被初始化为 7.0,即全局变量 dVar 的值。但是,实际并非如此,所以 dVar 被初始化为一个未定义的值。
枚举遵循同样的规则,但是,枚举器被输出到枚举的包含范围外。在下例中枚举元素 Spades、Clubs、Hearts 和 Diamonds 被说明,因为枚举器被输出到封闭的范围外,它们被认为具有全局范围。例中的标识符已经在全局范围被定义。
考虑以下代码:

```
const int Spades=1,Clubs=2,Hearts=3,Diamonds=4;

enum Suits
{
  Spades=Spades, //错误
  Clubs,         //错误
  Hearts,        //错误
```

```
  Diamonds          //错误
}
```

由于在前面代码中标识符已经在全局范围内被定义,所以产生出错信息。

注意:用相同的名称指定不止一个的程序元素,例如一个枚举器和一个对象,这都被认为是很糟的编程方式,应当避免使用。在前例中,这种方式导致出错。

隐藏名称

你可以通过在一个包围的块内说明名称来隐藏该名称,在图 2.1 中,i 在更内层的块内又被说明了,而隐藏了在较外层块范围内与 i 相关的变量。

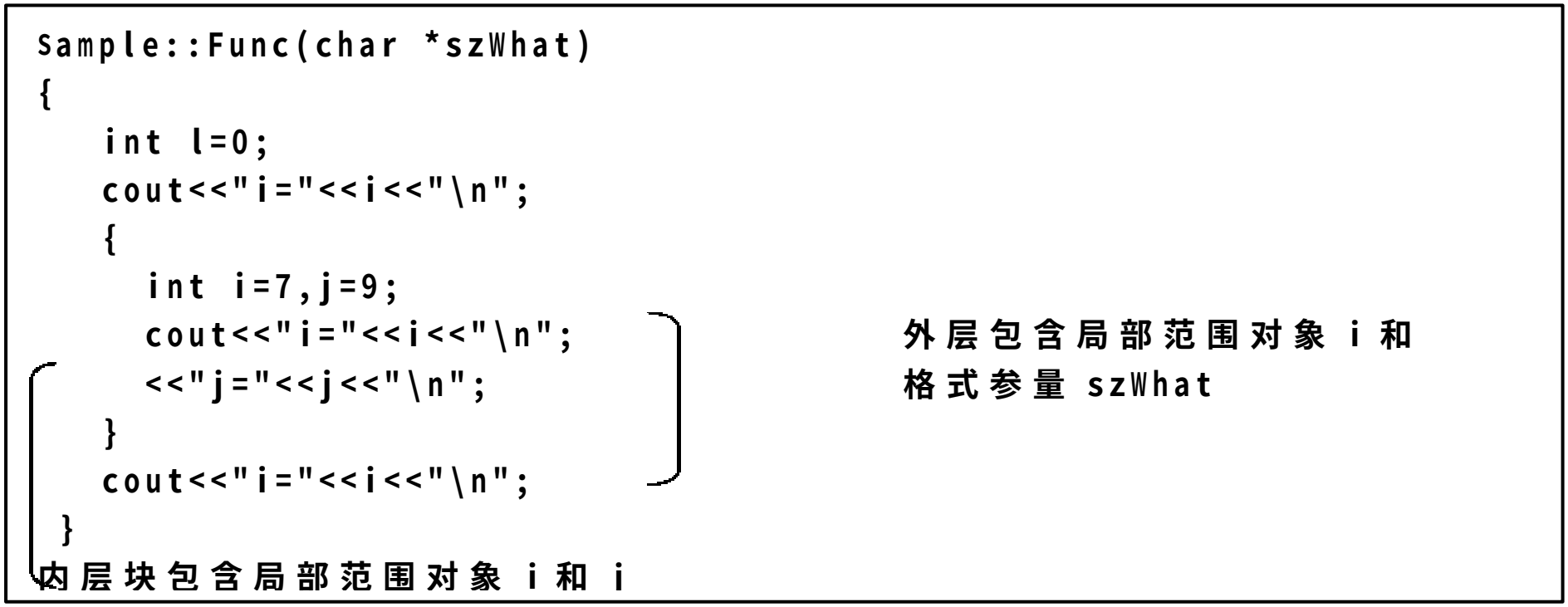


图 2.1 块范围和名称隐藏

图 2.1 中给出的程序输出为：

```
i=0
i=7
j=9
i=0
```

注意：参量 szWhat 被认为在函数范围内，因此，它被看成好象是在函数的最外层

块内被说明的。

具有文件范围的隐藏名称

你可以通过在块范围内明确地说明相同的名称而隐藏带文件范围的名称。但是文件范围名称可以使用范围分辨运算符(::)进行访问。例如:

```
#include <iostream.h>

int i=7;    //i 具有文件范围即在所有的块外部说明的
void main(int argc, char *argv[])
{
    int i=5;    //i 具有块范围,隐藏了具有文件范围的 i
    cout << " Block_scoped i has the value : " << i << "\n" ;
    cout << " File_scoped i has the value : " << ::i << "\n" ;
}
```

前面代码的结果是:

```
Block_scoped i has the value : 5
File_scoped i has the value : 7
```

隐藏类名

你可以通过在相同的范围内说明一个函数、对象或变量、或者枚举器来隐藏类名。

当然,类名还可以通过使用关键字 class 前缀进行访问。

```
//在文件范围说明类 Account
class Account
{
public:
   Account(double InitialBalance)
     { balance = InitialBalance;}
   double GetBalance()
     { return balance;}
private:
     double balance;
};
double Account=15.37 //隐藏类名 Account

void main()
{
    class Account checking(Account); //限定 Acconut 为类名

    cout << "Opening account with balance of:"
       << checking.GetBalance() << "\n";
}
```

注意，类名(Account)在任何位置被调用时,都要用关键字 class 将其与文件域变量 Account 区别开来,当类名出现在范围分辩运算符(::)的左侧时不用此规

则。范围分辨运算符左侧的名称总被认为是类名,以下例子说明如果用关键字 class 来说明一个类型 Account 的对象的指针:

```
class Account * Checking = new class Account(Account);
```

上述语句中的 Account 在初始化器中(在圆括号内)具有文件范围,类型为 double。

注意:在这个例子中,标识符名称重新使用认为是糟糕的程序设计风格。

有关指针的更多的信息参见本章后面的“派生类”。有关类对象的说明和初始化详见第 8 章“类”。有关使用 new 和 delete 自由存储运算符的更多信息,参见第 11 章“特殊的成员函数”。

函数的形式参量的范围

函数的形式参量(在函数定义内指定的参量)被认为是在函数体的最外层块的范围内。

程序和连接

一个程序由连接在一起的一个或多个转换单元组成,从包含 main 函数的转换单元开始执行(概念上)(有关转换单元的更多信息,参见本书后面“预处理器参考”的“翻译阶段”。

有关 main 函数,详见本章后面的“程序开始:main 函数”)。

连接的类型

在转换单元之间共享对象或函数的名称的方式被称为“连接”。这些名称有：

- 内部连接，即它们仅指自己的转换单元中的程序元素，不与其它转换单元共享。

在其它转换单元中的相同的名称可以指不同的对象或不同的类。具有内部连接的名称有时被认为相对于它们的转换单元是局部的。

具有内部连接的名称的一个说明例子如下：

```
static int i //static 关键字确保是内部连接
```

- 外部连接，即指程序中的任意一个转换单元中的程序元素，在转换单元的之间共享。

  在另一个转换单元中的相同名称被认定为指示同一个对象或类，具有外部连接的名称有时被认为是全局的 。

  具有外部连接的名称的一个说明例子如下：

```
extern int i;
```

- 无连接，即指唯一实体，在另一个范围中的同样的名称可能不是相同的对象，例如一个枚举(但注意，你可以用无连接来向一个对象传递一个指针，这使得在其它转换单元中该对象是可访问的。)

具有文件范围的名称的连接

以下的连接规则适用于具有文件范围的名称(不是 typedef 及枚举器名称)：

- 如果一个名称被显式说明为 static，则它具有内部连接，标识在转换

单元内的一个程序元素。

- 枚举器名称及 typedef 名称是无连接
- 具有文件范围的所有其它的名称都是外部连接

Microsoft 特殊处→

- 如果一个具有文件范围的函数名称被显式说明为 inline,如果它被实例化或其地址被引用时,它具有外部连接。因此,对于一个具有文件范围的函数,可能有内部或外部的连接。

Microsoft 特殊处结束

一个类有内部连接,如果:

- 不使用 C++功能(例如成员访问控制、成员函数、构造函数、析构函数等)
- 不用于说明具有外部连接的另一个名称。此限制意味着,类类型对象传递给具有外部连接的函数导致类具有外部连接。

具有类范围的名称的连接

以下连接规则适用于具有类范围的名称:

- 静态类成员具有外部连接
- 类成员函数具有外部连接
- 枚举器及 typedef 名称是无连接

Microsoft 特殊处→

- 被说明为 friend 函数的函数必须具有外部连接。把一个静态函数说明为 friend 会出错。

Microsoft 特殊处结束

具有块范围的名称的连接

以下连接规则适用于具有块范围的名称(局部名称):

- 说明为 extern 的名称具有外部连接,除非它们在前面说明为 static
- 具有块范围的所有其它名称是无连接

具有无连接的名称

具有无连接的名称仅仅有:

- 函数参量
- 没有被说明为 extern 或 static 的块范围名称
- 枚举器
- 在 typedef 语句中说明的名称,而当 typedef 语句用于为一个未命名的类类型提供一个名称时是一个例外。如果类具有外部连接,则该名称可能具有外部连接。以下例子给出了具有外部连接的 typedef 名称的情形:

```
typedef struct
 {
   short x;
   short y;
 } POINT
```

```
extern int MoveTo(POINT pt);
```
typedef 名称 POINT 成了未命名的结构的类名,然后它被用于说明一个具有外部连接的函数。
因为 typedef 名称为无连接,所以在转换单元之间它们的定义可以不同,因为编译是独立地进行的,所以编译器无法检查出这些不同处,结果是,这种类型的错误要到链接时才能被查出。
考虑以下情形:
```
//转换单元 1
typedef int INT

INT myInt;
...

//转换单元 2
typedef short INT

extern INT myInt
...
```
前面的代码在链接时产生一个“未分解的外部”错误。
C++函数仅能在文件或类范围内定义,以下例子描述了如何定义函数并给出了一个错误的函数定义:
```
#include <iostream.h>
```

```
void ShowChar(char ch);   //说明函数 ShowChar

void ShowChar(chat ch)    //定义函数 ShowChar
{                         //函数具有文件范围
  cout << ch;
}

struct Char               //定义类 Char
{
  char Show();           //说明 Show 函数
  char Get();            //说明 Get 函数
  char ch;
 };

char Char::Show()     // 定义具有类范围的 Show 函数
{
  cout << ch;
  return ch
}

void GoodFuncDef(char ch)  // 定义具有文件范围的 GoodFuncDef
```

```
{
    int BadFuncDef(int i)  // 错误地嵌套函数
    {
       return i * 7;
    }
    for (int i = 0;i < BadFuncDef(2); ++i)
        cout << ch;
        cout << "\ n" ;
}
```

连接到非 C++函数：

只有在前面说明为具有 C 连接的 C 函数和数据才能被访问。可是，它们必须定义在分开的编译过的转换单元中。

语法

连接规格：

```
    extern 字符串文字{说明表 opt}
    extern 字符串文字 说明
```

说明表：

```
   说明
   说明表 说明
```

Microsoft C++支持字符串文字域中的字符串“C”和“C++”。以下例子给出了可选择的具有 C 连接的名称的说明方法：

```
//说明 printf 具有 C 连接
extern "C" int printf (const char *fmt,...);

//使头文件"cinclude.h 中的所有内容具有 C 连接
extern "C"
{
#include <cinclude.h>
}

//说明具有 C 连接的两个函数 ShowChar 和 GetChar
extern "C"
{
char ShowChar(char ch);
char GetChar(void);
}

//定义具有 C 连接的两个函数 ShowChar 和 GetChar
extern "C" char ShowChar(char ch)
{
    putchar(ch);
    return ch;
}
```

```
extern "C" char GetChar(void)
{
    char ch;

    ch = getchar();
    return ch
}

//说明一个具有 C 连接的全局变量 errno
extern "C" int errno;
```

启动和结束

通过使用两个函数:main 和 exit,使程序方便地启动和结束。其它的启动和结束代码可能被执行。

程序启动:main 函数

一个被称为 main 的特殊函数是所有 C++程序的入口点,此函数并未被编译器预定义,而是必须在程序文本中提供。如果你正在编写依附于单代码程序设计模式的代码,你可以使用 main 的宽字符版本 wmain。main 的说明语法是:

```
int main();
```
或,可选地:
```
int main(int argc[,char *argv[] [,char* envp[]]]);
```
wmain 的说明语法如下:
```
int wmain();
```
或,可选地:
```
int wmain(int argc[,wchar_t *argv[][,wchar_t *envp[]]]);
```
另外,main 和 wmain 函数还可被说明为返回值是 void(无返回值)。如要你将 main 或 wmain 说明为返回值 void,则用 return 语句无法向父进程或操作系统返回退出代码。当 main 或 wmain 被说明为 void 时,若要返回退出代码,则必须使用 exit 函数。

使用 wmain 代替 main

在单代码程序设计模式中,你可以定义 main 函数的宽字符版本。如果你想编写可移植的依附于单代码规格的代码,则使用 wmain 代替 main。

你用与 main 相似的格式说明 wmain 的形式参量,而后你可以向程序传递宽字符参量及可选地宽字符环境指针。wmain 的 argv 和 envp 参量是 wchar_t *类型。

如果你的程序使用 main 函数,则在程序开始处由操作系统创建了一个多字节字符环境,环境的一个宽字符拷贝仅在需要时创建。(例如,通过对_wgetenv 或_wputenv 函数的调用)。在第一次调用_wputenv 或第一次调用_wgetenv 时,如果已经存在 MBCS 环境,则一个相应的宽字符字符串环境被创建,且由一个_wenviron 全局变量指向,它是_environ 全局变量的宽字符版本。在这点上,环

境的两份拷贝(MBCS 和单代码)同时存在,且在程序的整个生存期中由操作系统获得。
同样地,如果你的程序使用 wmain 函数,则在第一次调用_putenv 或 getenv 时创建一个 MBCS(ASCII)环境,且由全局变量_environ 指向。
有关 MBCS 环境的更多信息,参见“Microsoft Visual C++ 6.0 参考库”的“Microsoft Visual C++ 6.0 运行库参考”中的第 2 章“单字节和多字节字符集”。

参量变义

在原型:

```
int main(int argc[,char *argv[][,char *envp[]]]);
```

或

```
int wmain(int argc[,wchar_t *argv[][,wchart_t *envp[]]]);
```

中的参量允许进行方便的参量命令行语法分析,且可选择地访问环境变量。参量定义如下:

argc

包含跟随在 *argv* 后面的参量个数的整数。argc 参量总是大于或等于 1。

argv

是一个以空格结尾字符串的数组,表示由程序的用户在命令行输入的参量。通过转换,*argv*[0]是调用程序的命令,argv[1]是第一个命令行参量,如此等等,直到 argv[*argc*],它总是为 NULL。有关截取命令行处理的更多信息,参见本章后面“定制 C++命令行处理”。

第一个命令行参量总是 argv[1],而最后一个是 *argv*[argc-1]。
Microsoft 特殊处→
envp
envp 数组用于 Microsoft C++中,它也是很多 UNIX 系统的常见的扩充。它是一个字符串数组,用于表示在用户环境设置的变量,此数组由一个 NULL 项结尾。关于截取环境处理的信息,参见本章后面的“定制 C++命令行处理”。该参量在 C 中是 ANSI 兼容的,但在 C++中却不是。
Microsoft 特殊处结束
以下例子展示如何使用 main 中的 *argc*、*argv* 和 *envp* 参量:

```
#include <iostream.h>
#include <string.h>

void main(int argc, char *argv[], char *envp[])
{
   int iNumberLines = 0;  //缺省情况是没有行号

   //如果不只提供了.EXE 文件名,而且指定了命令行选项/n
   //则环境变量的列表是按行编号的
   if (argc==2 && srticmp(argv[1],"/n")==0)
      iNumberLines=1;

   //通过字符串表直到遇到 NULL
```

```
    for (int i=0; envp[i]!=NULL;++i)
    {
        if (iNumberLines)
            cout << i << ": "<< envp[i]<<"\n"
    }
}
```

通配符扩展

Microsoft 特殊处→
你可以在命令行使用通配符问号(?)和星号(*),指定文件名和路径变量。命令行参量由被称为_setargv 的例行程序处理。缺省地,_setargv 将通配符扩展为单独字符串进入 argv 字符串数组中,如果通配符变量未找到匹配,则参量作为文字传送。
Microsoft 特殊处结束

分析 C++命令行变量

Microsoft 特殊处→
Microsoft C/C++开始代码使用以下规则解释操作系统命令行给出的参量:

- 参量由空白定界,可以是一个空格或一个制表符
- 插入字符(^)不被认为是一个转义字符或定界符。在传递给程序中的 argv 数组前,字符由操作系统中命令行语法分析程序进行完全处理。

- 由双引号包围的字符串(“字符串”)被解释为单个参量,而不管其中是否包含空白。被引号括起的字符串可以嵌入参量中。
- 由反斜杠引导的一个双引号(\")被解释为一个文字双引号字符(")
- 反斜杠被作为文字解释,除非它们后面紧跟一个双引号。
- 如果偶数个反斜杠后面跟随一个双引号,则为每对反斜杠放一个反斜杠到 argv 数组中,双引号被解释为一个字符串定界符。
- 如果奇数个反斜杠后面跟随一个双引号,为每对反斜杠放一个反斜杠到 argv 数组中,双引号则被剩下的反斜杠进行“转义”而使得一个文字双引号(")放入到 argv 中。

以下例子说明命令行参量是如何被传递的:

```
include <iostream.h>

void main(int argc, //数组 argv 中的字符串个数
          char *argv[], //命令行参量字符串数组
          char *envp[] )//环境变量字符串数组
{
   int count;

  //显示每个命令行参量
  cout << "\nCommand-line arguments:\n";
  for ( count = 0;count < argc;count++)
      cout << " argv["<<count<<"]   "
```

```
        << argv[count]<< "\n";
}
```

表 2.2 给出例子输入和预期的输出以说明前面列出的规则。

表 2.2 语法分析命令行的结果

命令行输入	argv[1]	argv[2]	argv[3]
"abc"d e	abc	d	e
a\\\b d"e f"g h	a\\\b	de fg	h
a\\\"b c d	a\"b	c	d
a\\\"b c" d e	a\\b c	d	e

Microsoft 特殊处结束

定制 C++命令行处理

Microsoft 特殊处→

如果你的程序不采用命令行参量,你可以通过截取执行命令行处理的库例行程序的使用而节省一小部分空间。这个例行程序被称为_setargv 且在“通配符扩展”中描述。为了截取它的使用,在包含 main 函数的文件中定义一个例程,什么也不做,但要命名为_setargv;然后对_setargv 的调用将被你定义的_setargv 去满足,则没有加载其库版本。

类似地,如果你从未通过 *envp* 参量访问环境表,但可以提供你自己的空例程取代环境处理例程_setenvp。就如同使用_setargv 函数,_setenvp 必须说明为 extern “C”。

你的程序可能调用 C 运行时库中的 spawn 或 exec 例程簇。如果是这种情况,你不能取消环境处理例程,因为这个例程用于从父进程向子进程传递一个环境。

Microsoft 特殊处结束

main 函数限制

main 函数有几条限制不适用任何其它的 C++函数。main 函数:

- 不能重载(参见第 12 章“重载”)
- 不能说明为 inline
- 不能说明为 static
- 不能取其地址
- 不能被调用

程序结束

在 C++中,有几种方式退出程序;

- 调用 exit 函数
- 调用 abort 函数
- 从 main 执行一条 return 语句

exit 函数

在标准包括文件 STDLIB.H 中说明的 exit 函数用于终止一个 C++程序。
提供作为 exit 一个参量的值将作为程序返回码或退出码返回给操作系统。通过约定,0 返回码意味着程序成功地完成了。

注意:你可以使用定义在 STDLIB.H 中的常量 EXIT_FAILURE 和 EXIT_SUCCESS 来指示你的程序的成功或失败。
从 main 函数发出一个 return 语句等价于用其返回值作参量去调用 exit 函数。有关更多的信息参见“Microsoft Visual C++ 6.0 运行库参考”中的“exit”。

abort 函数

也在标准包括文件 STDLIB.H 中说明的 abort 函数用于终止一个 C++程序。exit 和 abort 之间的区别是:exit 允许 C++发生运行终止处理(调用全局变量析构函数),而 abort 则是立即终止程序。有关更多的信息参见“Microsoft Visual C++6.0 运行库参考”中的“abort”。

return 语句

从 main 发出一条 return 语句在功能上等价于调用 exit 函数。考虑以下例子:

```
int main
{
   exit (3);
   return 3;
}
```

在上例中 exit 和 return 语句功能上是完全相同的,可是 C++要求在返回一个数值时函数应有返回类型而不是 void。return 语句允许从 main 返回一个值。

额外的启动考虑

在 C++中,对象的构造函数和析构函数可以包含执行用户代码,因此,了解进入 main 之前发生了什么初始化以及从 main 退出以后什么析构函数被调用是很重要的(有关对象的构造函数和析构函数,参见第 11 章“特殊成员函数”中的“构造函数”和“析构函数”)在进入 main 之前发生了以下初始化:

- 静态数据缺省初始化为 0。没有明确初始化的所有静态数据在执行任何其它代码包括运行初始化前均被设定为 0。静态数据成员还必须加以显式地定义。
- 在一个转换单元中的全局静态对象的初始化。它的发生可能出现在进入 main 之前或在对象的转换单元中的任何函数或对象的第一次使用之前。

Microsoft 特殊处→
在 Microsoft C++中,全局静态对象在进入 main 之前初始化。
Microsoft 特殊处结束
全局静态对象是彼此相互依赖的,但在不同的转换单元中可能产生不正确的动作。

额外的终止考虑

你可以用 exit、return 或 abort 终止一个 C++程序。可以使用 atexit 函数增加退出处理,这些在下节中讨论。

使用 exit 或 return

当你从 main 中调用 exit 或执行一条 return 语句时,静态对象按照与初始化相反的顺序被销毁,下面的例子显示了如何初始化和清除工作的:

```
#include <stdio.h>
class ShowData
{
public:
    //构造函数打开一个文件
    ShowData(const char *szDev)
    {
       OutputDev=fopen(szDev,"w");
}

    //析构函数关闭一个文件
    ~ShowData() { fclose(OutputDev); }

    //Disp 函数在输出设备上显示一字符串
    void Disp(char *szData)
   {
       fputs(szData,OutputDev);
    }
```

```
private:
    FILE *OutputDev;
};

//定义一个类型为 ShowData 的静态对象,选取的输出设备是"CON"即标准输出设备
   ShowData sdl="CON";
//定义另一个类型为 ShowData 的静态对象,直接输出到文件"HELLO.DAT"中
ShowData sd2="CON",

int main()
{
    sd1 Disp("hello to default device\n");
    sd2.Disp("hello to file hello.dat\n");

    return 0;
}
```

在上面的例子中,静态对象 sd1 和 sd2 在进入 main 之前被创建并被初始化,在使用 return 语句终止该程序后,先销毁 sd2 而后是 sd1。ShowData 类的析构函数关闭这些静态对象相关的文件(有关初始化、构造函数、析构函数的更多信息,参见第 11 章“特殊成员函数”)。

编写这个代码的另一种方法是说明为块范围对象 ShowData,且允许它们在超出

范围时被销毁：

```
int main()
{
   ShowData sd1,sd2("hello.dat");

    sd1.Disp("hello to default device\n");
    sd2.Disp("hello to file hello.dat\n");

    return 0;
}
```

使用 atexit

使用 atexit 函数,你可以指定一个退出处理函数在程序终止前执行。没有一个在 atexit 调用之前初始化的全局静态对象在退出该处理函数执行前被销毁的。

使用 abort

调用abort函数立即引起终止,它越过初始化的全局静态对象的正常的析构过程,还越过了使用 atexit 函数指定的任何特殊处理。

存储类

存储类管理 C++中的对象和变量的生存期、连接和处理。一个给定的对象只能

有一个存储类。本节讨论数据的 C++存储类：

- 自动的
- 静态的
- 寄存器的
- 外部的

自动的

自动存储的对象和变量对于一个块的给定实例是局部的。在递归或多线程代码中，自动对象和变量被保证为在一个块的不同实例中有不同的存储。Microsoft C++在程序的栈中存储自动的对象和变量。
在一个块内定义的对象和变量为 auto 存储，除非用 extern 或 static 关键字加以指定。自动的对象和变量可以用 auto 关键字加以指定，但没有必要显式指明 auto。自动的对象和变量是无连接的。
自动的对象和变量仅仅持续到它们被说明的所在块的结束。

静态的

被说明为 static 的对象和变量在程序执行期间保留它们的值。在递归代码中，一个静态对象或变量在一个块代码的不同实例中确保有相同的状态。
在所有块的外部定义的对象和变量缺省地具有静态生存期和外部连接。被显式地说明为 static 的一个全局对象或变量具有内部连接。
静态对象和变量在程序执行的期间是持续的。

寄存器的

只有函数的参量和局部变量可以说明为寄存器存储类。
象自动变量一样,寄存器的变量仅仅持续到它们被说明的所在块的结束。
编译器不允准用户对寄存器变量的要求,相反地,当全局优化在使用时,编译器作出自己的寄存器选择,可是,编译器允准所有 register 关键字的相关语义。

外部的

被说明为 extern 的对象和变量说明一个定义在另一个转换单元中的对象或一个具有外部连接的封闭的范围中的对象。
具有 extern 存储类的 const 变量的说明强制变量具有外部连接,一个 extern const 变量的初始化允许定义在转换单元中。在一个转换单元中初始化而不是定义在该转换单元中,这将产生不确定的结果。
以下例子给出了两个 extern 说明:DefinedElsewhere(指向定义在一个不同的转换单元中的名称)和 DefinedHere(指向定义在一个封闭的范围中的名称):

```
extern int DefinedElsewhere;//定义在另一个转换单元中

void main()
{
    int DefinedHere;
    {
      extern int DefinedHere;//指向封闭的范围中的 DefinedHere
```

```
    }
}
```

对象的初始化

一个局部的自动的对象或变量在控制流程每次到达它的定义时都被初始化。一个局部的静态对象或变量在控制流程第一次到达它的定义时被初始化。考虑以下例子,例子中定义了一个类,它历经对象的初始化和析构,而后定义了三个对象 I1、I2 和 I3:

```
#include <iostream.h>
#include <string.h>

//定义了一个类,它历经初始化和析构
class InitDemo
{
public:
    InitDemo(const char *szWhat);
    ~InitDemo();
private:
     char *szObjName;
};

//类 InitDemo 的析构函数
```

```
InitDemo::InitDemo (const char * szWhat)
{
    if ( szWhat!=0  && strlen(szWhat)>0 )
    {
       //为 szObjName 分配存储,然后将 szWhat 初始值拷贝到 szObjName
       szObjName=new char[strlen(szWhat)+1];
       strcpy (szObjName,szWhat);
          cout << "Initializing" << szObjNmae << "\n";
   }
   else
      szObjName=0;
}

//InitDemo 的析构函数
InitDemo::~InitDemo()
{
    if (szObjName!=0)
    {
      cout << "Destroying: " << szObjName << "\n";
      delete szObjName;
    }
}
```

```
//输入 main 函数
void main()
{
    InitDemo I1("Auto I1")
    {
      cout << "In block.\n";
      InitDemo I2("Auto I2");
      static InitDemo I3("Static I3");
    }
    cout << "Exited block.\n";
}
```

上面的代码描述了对象 I1、I2 和 I3 是如何以及何时被初始化的和何时被销毁的。该程序的输出为：

```
Initializing: Auto I1
In block.
Initializing: Auto I2
Initializing: Static I3
Destroying: Auto I2
Exited block.
Destroying: Auto I1
Destrying: Static I3
```

关于程序有几点要注意的:
第一,I1 和 I2 在控制流退出定义它们的块时被自动销毁。
第二,在 C++中,没有必要在一个块的开始去说明对象或变量。此外,这些对象仅当控制流到达它们的定义时才被初始化(I2 和 I3 是这种定义的例子),输出确切地显示了他们是何时被初始化的。
最后,静态局部变量例如 I3 在程序的持续期间保留它们的值,但是在程序结束时被销毁。

类　型

C++支持三种对象类型:

- 基本类型是建立在语言中的(如 int、float 或 double)。这些基本类型的实例通常被称为“变量”。
- 派生类是从内部类型派生的新类型。
- 类类型是通过现存类型组合创建的新类型,这些在第 8 章“类”中讨论。

基本类型

C++中的基本类型分为三类:“整型的”、“浮点”和“void”。整型能够处理全部的数;浮点类型能够指定可能有小数部分的值。
void 类型描述值的空集。不能指定 void 类型的变量,它基本上用于说明没有返

回值的函数或说明无类型或任意类型的数据的“通用”指针。任何表达式可显式地转换或造型转换为类型 void,可是,这些表达式限于以下应用:

- 一个表达式语句(参见第 4 章“表达式”)。
- 逗号运算符的左侧操作数(参见第 4 章“表达式”中的“逗号运算符”)。
- 条件运算符的(?:)的第二或第三个操作数(参见第 4 章“表达式”中的“带条件运算符的表达式”)。

表 2.3 解释了类型尺寸的限制,这些限制是独立于 Microsoft 实现的。

表 2.3　C++语言的基本类型

目录	类型	内容
整型	char	类型 char 是整型,通常包含执行字符集的成员,在 Microsoft C++中,这是 ASCII 码 C++编译器把类型为 char、signed char 和 unsigned char 的变量作为具有不同的类型对待。类型为 char 的变量被提升为 int,就好象它们缺省为 signed char,除非使用/J 编译选项。在这种情况下,它们被作为类型 unsigned char,且被提升为不带符号扩展的 int
	short	类型 short int(或简称 short)是一个整型,大于或等于 char 类型的尺寸,小于或等于 int 类型的尺寸 short 类型的对象可被说明为 signed short 或 unsigned short signed short 是 short 的同义词
	int	类型 int 是整型,它大于或等于 short int 类型的尺寸,小于或等于类型 long 的尺寸 int 类型的对象可被说明为 signed int 或 unsigned int。signed int 是 int 的同义词

续表

	__int*n*	指定整型变量的尺寸,这里的 n 是以位为单位给出的尺寸,n 的值可 以是 8、16、32 或 64
	long	类型 long(或 long int)是大于或等于类型 int 的尺寸的整型 long 类型的对象可被说明为 signed long 或 unsigned long signed long 是 long 的同义词
浮点	float	类型 float 是最小的浮点类型
	double	类型 double 是一个浮点类型,它大于或等于 float 类型的尺寸,但小于或等于 long double 类型的尺寸
	long double*	类型 long double 是一个等于类型 double 的浮点类型

*long double 和 double 的表示是完全相同的,可是 long double 和 double 是两种类型。

Microsoft 特殊处→

表 2.4 列出了 Microsoft C++中基本类型所需的存储空间大小。

表 2.4　基本类型的尺寸

类型	尺寸
char,unsigned char,signed char	1 个字节
short,unsigned short	2 个字节
int,unsigned int	4 个字节
long,unsigned long	4 个字节
float	4 个字节
double	8 个字节
long double*	8 个字节

*long double 和 double 的表示是完全相同的,但 long double 和 double 是两种类型。

关于类型转换的更多信息,参见第 3 章“标准转换”。
Microsoft 特殊处结束

指定尺寸的整型

Microsoft C++还支持指定尺寸的整型,你可以使用 __int*n* 类型指示符来说明 8、16、32 或 64 位的整数变量,其中 *n* 是整型变量的尺寸,*n* 值可为 8，16，32，或 64。以下例子为每个这些指定尺寸的整型数说明一个变量:

```
__int8  nSmall ;   //说明 8 位整数
__int16 nMedium ;  //说明 16 位整数
```

```
_ _int32 nLarge ;    //说明 32 位整数
_ _int64 nHuge ;//说明 64 位整数
```

类型__int8、_ _int16 和_ _int32 是具有同样尺寸的 ANSI 类型的同义词,对编写在跨多平台中行为相同的可移植代码很有用。注意,__int8 数据类型是类型 char 的同义词,_ _int16 是类型 short 的同义词,__int32 是类型 int 的同义词,__int64 数据类型在 ANSI 中没有等价类。

由于 __int8、__int16 和 __int32 被编译器认为是对应的同义词,所以在使用这些类型作参量去重载函数调用时,应加以小心。例如,以下 C++代码将产生一个编译错误。

```
void MyFunc(_ _int8) {}
void MyFunc(char) {}

void main()
{
    _ _int8 newVal;
    char MyChar;
    MyFunc(MyChar); //不明确的函数调用;
    MyFunc(newVal); //char 与_ _int8 是同义词。
}
```

派生类型

派生类型是可被用于程序中,可包含直接派生类型和复合派生类型的新类型。

直接派生的类型

从已存在类型直接派生的新类型是指向、指示或(在函数情形)传输类型数据以返回一个新类型的类型。

- 变量或对象数组
- 函数
- 一个给定类型的指针
- 对象的引用
- 常量
- 类成员指针

变量或对象数组

变量或对象数组可以包含指定数目的特定类型,例如,从整型派生的数组是一个类型为 int 的数组。以下代码例子说明并定义了一个 10 个 int 变量的数组和一个含有类 SampleClass 的 5 个对象的数组:

```
int    ArrayOfInt[10];
SampleClass aSampleClass[5];
```

函数

函数有 0 个或多个给定类型的参量以及返回指定类型的对象(或什么也不返回,如果函数具有 void 返回类型)。

一个给定类型的指针

变量或对象的指针选择存储器中的一个对象。对象可以是全局的、局部的(或栈结构)或动态分配的。给定类型的函数指针允许程序在一个特定对象或多个对象上的函数延迟选择直到运行时。以下例子给出了一个指向类型为 char 的变量的

指针定义：

```
char *SzPathStr;
```

对象的引用

对象的引用为通过引用访问对象提供了一种便利的方法,但与使用通过值访问对象所要求的语法相同。以下例子描述了如何使用作为函数参量和作为函数返回类型的引用：

```
BigClassType &func(BigClassType &objname)
{
    objname.DoSomething();//注意,使用了成员运算符(.)

    objname.SomeData=7;//通过非 const 引用传递的数据是可修改的。

    return objname;
}
```

关于通过引用向函数传递对象的几个要点是：

- 访问 class、struct 和 union 对象的成员的语法是相同的,就如同它们通过值:成员运算符(.)传递一样。
- 在函数调用前未拷贝对象,传递它们的地址,这可以减少函数调用的系统开销。

另外,返回一个引用的函数仅需要接收它们所指的对象的地址,而不是整个对象的拷贝。

尽管上面的例子仅仅是用函数进行上下文通信中的引用,但引用并不限于此种应

用。例如,考虑一个函数需要 l 值的情况——重载运算符的普通要求:

```
class Vector
{
public:
    Point &operator[](int nSubscript);  //函数返回一个引用类型

...
};
```

上面的说明为类 Vector 指定了一个用户定义的下标运算符。在一个赋值语句中,出现两种可能的状态;

```
Vector vl;
int i;
Point p;
vl[7]=p;  //Vector 用作一个 l 值
p=vl[7];  //Vector 用作一个 r 值
```

后一种情况,即 vl[7]用作一个 r 值,可以不用引用去实现。可是,前一种情况,即 vl[7]用作一个 l 值,如果不用引用类型的函数就不能方便地实现。从概念上讲,前面例子中最后两条语句翻译为以下代码:

```
vl.operator[](7)=3;  //Vector 用作一个 l 值
i=vl.operator[](7);  //Vector 用作一个 r 值
```

当用这种方式去看时,容易看到第一个语句必定为一个 l 值,在赋值语句的左侧且语义正确。

关于重载以及特殊情况的重载运算符的更多信息,参见第 12 章“重载”中的“重载运算符”。
你还可以使用引用去说明一个变量或对象的一个 const 引用。说明为 const 的一个引用保留了通过引用传递参量的效能,而防止被调用函数修改最初对象。考虑以下代码:

```
//Invalue 是一个 const 引用。
void PrintInt(const int &IntValue)
{
    printf("%d\n",IntValue);
}
```

引用初始化不同于对一个引用类型的变量的赋值,考虑以下代码;

```
int i=7;
int j=5;

//引用初始化
int &ri=i;  // 初始化 ri 指向 i。
int &rj=j;  // 初始化 rj 指向 j。

//赋值
ri=3;       //i 现在等于 3
rj=12;      //j 现在等于 12
ri=rj;       //i 现在等于(12)。
```

常量

关于 C++中允许的各种常量的更多信息参见第 1 章“词法规定”中的“文字”。

类成员指针

这些指针定义一个类型指向一个特殊类型的类成员,这样的一个指针可以被类类型的任何对象或类类型派生类型的任何对象所使用。

使用类成员指针增强了 C++语言的类型安全性,如表 2.5 所列的对成员指针可以使用三种新运算符和结构。

表 2.5 和成员一起使用的运算符和结构

运算符或结构	语法	使用
::*	*type::*ptr-name*	成员指针说明。type 指定类名,*ptr_name* 指定成员指针名。成员指针可被初始化,例如: MyType::*pMyType=&MyType::i
.*	*obj-name.*ptr-name*	使用一个对象或对象引用间接引用一个成员指针,例如: int j=Object.*pMyType
->*	*obj_ptr->*ptr_name*	使用一个对象指针间接引用一个成员指针,例如: int j=pObject->*pMyType

考虑以下例子,该例子中定义了一个类 AClass 和一个派生类型 pDAT,它指向成

员 I1:

```
#include <iostream.h>

//定义类 AClass
{
public:
    int I1;
    Show() { cout << I1 << "\n"; }
};

//定义一个派生类型 pDAT,指向类型 AClass 的对象的 I1 成员
int AClass::*pDAT=&AClass::I1;

void main()
{
  AClass aClass;   //定义类型 AClass 的一个对象。
  AClass *paClass=&aClass; //定义对象的一个指针。

  int i;
  aClass.*pDAT=7777;  //用 .*运算符对 aClass::I1 赋值
  aClass.Show();
```

```
  i=paClass->*pDAT;   //*运算符间接引用一个指针

  cout << i << "\n";
}
```

成员 pDAT 的指针是从类 AClass 派生的一个新类型,它是一个比 int 的“普通”指针更强壮的类型,因为它仅指向类 AClass 的 int 成员(在此情况为 I1)。静态成员的指针是普通指针而不是类成员指针,考虑以下例子:

```
class HasStaticMember
{
public:
  static int SMember;
};
int  HasStaticMember::SMember=0;

int *pSMember=&HasStaticMember::SMember;
```

注意,该指针的类型是“int 指针”,而不是“HasStaticMember::int 的指针”。

成员指针可以和成员数据一样指向成员函数,考虑以下代码:

```
#include <stdio.h>
//用虚函数 Identify 说明一个基类 A,(注意在此上下文中,struct 与类是相同的)
struct A
{
```

```
    virtual void Identify()=0;  //类 A 没有定义
};

//说明 Identify 成员函数一个指针
void (A::*pIdentify)()=&A::Identify;

//说明从类 A 派生的类 B
struct B : public A
{
    void Identify();
};

//为类 B 定义 Identify 函数
void B:Identify()
{
    printf("Identification is B::Identify\n");
}

void main()
{
    B Bobject; //说明类型 B 的对象
    A *pA ;//说明类型 A 的指针
```

```
    PA=&Bobject;//使 PA 指向类型 B 的一个对象
   (PA->*pIdentify)(); //通过成员 pIdentify 的指针调用 Identify 函数
}
```

该程序的输出是:

```
Identification is B::Identify
```

该函数是通过类型 A 的一个指针调用的。可是,由于函数是一个虚函数,pA 所指对象的正确函数被调用。

复合派生类型

本节描述以下复合派生类型:

- 类
- 结构
- 联合

有关集合类型和集合类型的初始化可参见第 7 章“说明符”中的“集合初始化”。

类

类是组合在一起的一组成员对象、操作这些成员的函数以及(可选的)成员对象和函数的访问控制规格。

通过将类中对象和函数进行分组,C++允许程序员创建派生类型,而不仅是定义数据,还可以定义对象的行为。

类成员在缺省情况下是私有访问和私有继承的。第 8 章“类”中介绍类。访问控制包含在第 10 章“成员访问控制”中。

结构

C++的结构与类相同,不同的是所有成员数据和函数缺省的为公共访问,且继承性缺省的为公共继承。

有关访问控制的更多信息,参见第 10 章“成员访问控制”。

联合

联合允许程序员在相同的存储器空间中定义能够包含不同变量的类型。以下代码给出了如何使用一个联合存储几个不同类型的变量:

```
//说明可以拥有类型为 char、int 和 char *的数据的联合
union ToPrint
{
    char chVar;
    int iVar;
    char *szVar;
};

//说明一个枚举类型,用于描述打印的类型
enum PrintType{ CHAR_T,INT_T,STRING_T };

void Print( ToPrint Var,PrintType Type)
{
    switch(Type)
    {
```

```
    case CHAR_T:
        printf("%c",Var.chVar);
        break;
    case INT_T:
        printf("%d",Var.iVar);
        break;
    case STRING_T:
        printf("%s",Var.szVar);
        break;
  }
}
```

类型名称

基本的和派生的类型的同义词均可使用 typedef 关键字来定义。以下代码描述了 typedef 的使用：

```
typedef unsigned char BYTE;       //8 位无符号实体
typedef BYTE *PBYTE;              //指向 BYTE 的指针

BYTE Ch;                          //说明一个类型 BYTE 的变量
PBYTE pbCh;                       //说明一个指向 BYTE 变量的指针
```

上面的例子给出了基本类型 unsigned char 和其派生类型 unsigned char *的统一的说明语法。typedef 结构对简化说明也很有用。一个 typedef 说明定义

一个同义词,而不是一个新的独立类型。以下例子说明了一个表示指向返回类型为 void 的函数的指针的类型名(PVFN)。这种说明的好处是在后面的程序中这些指针的数组说明起来非常简单。

```
//两个函数原型
void func1();
void func2();

//定义 PVFN 表示一个指向返回类型为 void 的函数的指针
typedef void (*PVFN)();

...

//说明一个函数指针数组
PVFN pvfn[]={ func1,func2 };

//调用一个函数
(*pvfn[1])();
```

l 值和 r 值

C++中表达式可以求值为“l 值”或“r 值”。l 值是一个表达式,求值为某个非 void 的类型,且指定一个变量。

l 值出现在赋值语句的左侧,(因此用"l"表示),通常为 l 值的变量可以用 const 关键字使其不可修改,这样不能出现在赋值语句的左侧。引用类型总是 l 值。
术语 r 值有时用于描述一个表达式的值,且与一个 l 值区分开来,所有的 l 值都是 r 值,但 r 值不都是 l 值。
一些正确及错误使用的例子;

```
i=7;        //正确   变量名 i 是一个 l 值
7=i;        //错误   常量 7 是一个 r 值
j*4=7;      //错误   表达式 j*4 产生一个 r 值
*p=i;       //正确   一个间接引用指针是一个 l 值
const int ci=7; //说明一个 const 变量
ci=9        //ci 是一个不可修改的 l 值,所以赋值导致一条错误信息的产生
((i<3) ? i:j )=7; //正确,条件运算符(?:)返回一个 l 值
```

注意:本节的例子描述了运算符未重载时的正确的和错误的用法,通过重载运算符,你可以让一个表达式,如 j*4 成为一个 l 值。

数的限制

两个标准包括文件 LIMITS.H 和 FLOAT.H 定义了“数的限制”或一个给定类型变量所能取的最小和最大值,这些最小值和最大值确保与 ANSI C 使用相同数据表示的任意 C++编译器都是可移植的。LIMITS.H 包含文件为整型定义数的限制,FLOAT.H 为浮点类型定义数的限制。

整数限制

Microsoft 特殊处→
整数类型的限制列在表 2.6 中,这些限制同时定义在标准头文件 LIMITS.H 中。

表 2.6　整型常量的限制常量

常量	含义	值
CHAR_BIT	不是位域的最小变量的位数	8
SCHAR_MIN	signed char 类型变量的最小值	-128
SCHAR_MAX	signed char 类型变量的最大值	127
UCHAR_MAX	unsigned char 类型变量的最大值	255(0xff)
CHAR_MIN	char 类型变量的最小值	-128;如果用/J 选项为 0
CHAR_MAX	char 类型变量的最大值	127;如果用/J 选项为 255
MB_LEN_MAX	在多字符常量中最大字节数	2
SHRT_MIN	short 类型变量的最小值	-32768
SHRT_MAX	short 类型变量的最大值	32767
USHRT_MAX	unsigned short 类型变量的最大值	65535(0xffff)
INT_MIN	类型变量的最小值	-2147483647-1
INT_MAX	类型变量的最大值	2147483647
UINT_MAX	unsigned int 类型变量的最大值	4294967295(0xffffffff)
LONG_MIN	long 类型变量的最小值	-2147483647-1

续表

LONG_MAX	Long 类型变量的最大值	2147483647
ULONG_MAX	unsigned long 类型变量的最大值	424967295(0xffffffff)

如果数值超过了最大的整数表示范围,则 Microsoft 编译器产生错误。
Microsoft 特殊处结束

浮点数限制

Microsoft 特殊处→
表 2.7 列出了浮点常量值的限制,这些限制同时定义在标准头文件 FLOAT.H 中。
表 2.7 浮点常量限制

常量	含义	值
FLT_DIG	数字个数 q,即具有 q 个十进制	6
DBL_DIG	数字的浮点数可舍入为一个浮	15
LDBL_DIG	点表示,而且返过来不损失精度	15
FLT_EPSILON	最小的正数 x,如 x+1.0 不等于 1.0	1.19202896e-07F
DBL_EPSILON		2.220446049250313l e-016
LDBL_EPSILON		2.220446049250313l e-016
FLT_GUARD		0

续表

FLT_MANT_DIG	在浮点有效数中由 FLT_RADIX 指定的	24
DBL_MANT_DIG	基数中的数字位数。基数为 2;	53
LDBL_MANT_DIG	因此这些值指定位数	53
FLT_MAX	最大的可表示的浮点数	3.402823466e+38f
DBL_MAX		1.7976931348623158e+308
LDBL_MAX		1.7976931348623158e+308
FLT_MAX_10_EXP	最大整数,10 的这个数的乘方	38
DBL_MAX_10_EXP	是可表示的浮点数	308
LDBL_MAX_10_EXP		308
FLT_MAX_EXP	最大整数,FLT_RADOX 的这个数的乘方	128
DBL_MAX_EXP	是可表示的浮点数	1024
LDBL_MAX_EXP		1024
FLT_MIN	最小的正值	1.175494351e-38F
DBL_MIN		2.2250738585072014e-308
LDBL_MIN		2.2250738585072014e-308

续表

FLT_MIN_10_EXP	最小负整数,10 的这个数的乘方	-37
DBL_MIN_10_EXP	是可表示的浮点数	-307
LDBL_MIN_10_EXP		-307
FLT_MIN_EXP	最小负整数,FLT_RADIX 这个数的乘方	-125
DBT_MIN_EXP	是可表达的浮点数	-1021
LDBT_MIN_EXP		-1021
FLT_NORMALIZE		0
FLT_RADIX	指数表示的基数	2
_DBL_RADIX		2
_LDBL_RADIX		2
FLT_ROUNDS	浮点加法的舍入模式	1(近)
_DBL_ROUNDS		1(近)
_LDBL_ROUNDS		1(近)

注意，表 2.7 中的信息可能在产品的未来版本中有所不同。

Microsoft 特殊处结束

第 3 章　标 准 转 换

C++语言定义了其基本类型之间的转换,还定义了指针、引用和成员指针派生类型的转换。这些转换被称为“标准转换”(有关类型、标准类型和派生类型的更多信息,参见第 2 章“基本概念”中的“类型”。)

本章讨论以下标准转换:

- 整型提升
- 整型转换
- 浮点转换
- 浮点和整型转换
- 算术转换
- 指针转换
- 引用转换
- 成员指针转换

注意:用户定义类型可指定它们自己的转换,用户定义类型的转换包含在第 11 章“特殊成员函数”的“构造函数”和“转换”中。

以下代码产生转换(在此例子中为整型提升):

```
long lnum1,lnum2;
int inum;
```

```
//inum 在赋值前提升为类型 long
lnum1=inum;

//inum 在乘法之前提升为类型 long
lnum2=inum*lnum2;
```

注意:仅当转换产生一个引用类型时,转换的结果是一个 l 值。例如,一个用户定义的转换说明为:

```
operator int&()
```

返回一个引用,它是一个 l 值。可是转换说明为:

```
operator int()
```

返回一个对象且不是一个 l 值。

整型提升

一个整型对象可转换为另一个宽整型(即,一种可以表示更大数值集的类型)。这种拓宽类型的转换被称为“整型提升”。使用整型提升,你可以在另一个整型类型的表达式的任意位置使用如下内容:

- char 和 short int 类型的对象、文字和常量
- 枚举类型
- int 位域
- 枚举器

C++提升是“值保留的”,即提升后的值确保与提升前的值一样。在值保留的提

升中,如果 int 可表示源类型的所有范围,较短整型对象(如位域或 char 类型对象)被提升为 int 类型。如果 int 不能表示值的所有范围,则对象被提升为 unsigned int 类型。尽管这种策略与 ANSI C 中使用的是一样的,值保留转换并不保留对象的“符号性”。

值保留提升和保留符号性的提升通常产生相同的结果,可是,如果被提升的对象是以下之一,则它们产生不同的结果:

- /、%、/=、%=、<、<=、>或>=的操作数之一。

这些运算符确定结果依赖于符号,因此,值保留和符号保留提升在使用这些操作数时产生不同的结果。

- >>或>>=的左操作数

这个运算符在执行移位操作时对待有符号量和无符号量是不同的。对有符号量,右移导致符号位进入空位;对于无符号量,空位填 0。

- 一个重载函数的一个参量或一个依赖于参量匹配操作数的类型的符号的重载运算符(有关定义重载的运算符的更多信息,参见第 12 章“重载”中的“重载的运算符”)。

整型转换

整型转换在整数类型之间进行,整数类型为 char、int 和 long(以及这些类型的 short、signed 和 unsigned 版本)。

本节描述整型转换的以下类型:

- 有符号的转为无符号的

- 无符号的转为有符号的
- 标准转换

有符号的转为无符号的

有符号的整型对象可被转换为相应的无符号类型,当这些转换发生时,实际的位模式并没有改变,但是数的解释改变了,考虑以下代码:

```
#include <iostream.h>

void main()
{
    short i=-3;
    unsigned short u;

    cout << (u=I) << "\n";
}
```

其输出结果如下:

```
65533
```

在上面例子中,定义了一个 signed short 变量 i,且初始化为一个负数,表达式(u=i)使 i 在赋给 u 之前转换为 unsigned short 类型。

无符号的转为有符号的

无符号的整型对象可被转换为相应的有符号类型。但是如果无符号对象的值的范围超出了有符号类型可表示范围,则这种转换会导致数据的误解释,如下例所示:

```
#include <iostream.h>

viod main()
{
   short I;
   unsigned short u=65533;

   cout << (I=u) << " \n"
}
```

其输出结果如下:

```
-3
```

在上面例子中,u 是一个 unsigned short 整型对象,必须转换为一个有符号量与表达式 I=u 的求值相符,由于它的值在 signed short 中不能正常表示,所以数据被误解释为上述所示内容。

标准转换

整型对象可以转换为更短的有符号或无符号整数类型,这种转换被称为“标准转换”。
如果源对象的值超出了用更短的类型所表示的范围,则转换会导致数据丢失。
注意:当向更短类型的转换发生时,编译器给出一个高级警告。

浮点转换

一个浮点类型的对象可以被安全地转换为一个更精确的浮点类型,也就是,该转换不造成损失有效位。例如,从 float 到 double 或从 double 到 long double 的转换是安全的,而且值没有变化。
一个浮点类型对象还可以转换为一个低精度的类型,只要它在那个类型可表示的范围内(有关浮点类型的范围参见第 2 章“基本概念”中的“浮点数限制”)。如果源数值不能被精确地表示,则它可被转换为邻近的更高或更低的可表示的值。如果这样的值不存在,结果是不确定的,考虑以下例子:

```
cout << (float)1E300 << endl;
```

float 类型可表示的最大值为 3.402823466E38,一个比 1E300 小得多的数。因此,该数被转换为无穷大,且结果为 1.#INF。

浮点和整型的转换

某些表达式能够导致浮点类型对象转换为整型,或反之亦然。
本节叙述以下类型的浮点和整型转换:

- 浮点到整型
- 整型到浮点

浮点到整型

当浮点类型的一个对象转换为一个整数类型时,小数部分被截除,在转换过程中不发生舍入,截除意味着一个像 1.3 这样的数转换为 1,而 -1.3 转换为 -1。

整型到浮点

当整型对象转换为一浮点类型且源值不能被准确表示时,结果是邻近的更高或更低的可表示的值。

算术转换

很多二进制运算符(在第 4 章“表达式”中的“带二进制运算符的表达式”中讨论)导致操作数的转换并以同样的方式产生结果,这些运算导致转换的方式被称为“常用的算术转换”。

不同类型的操作数的算术转换的执行如表 3.1 所示。

表 3.1　类型转换的条件

满足的条件	转换
两个操作数之一为类型 longdouble	另一个操作数被转换为类型 longdouble
前面的条件不满足,其中一个操作数是 double 型	另一个操作数被转换为类型 double
前面条件不满足，其中一个操作数是浮点型	另一个操作数被转换为类型 float
前面条件不满足(没有一个操作数是 float 类型)	对操作数的整型提升按如下方式执行： ● 如果操作数之一为类型 unsigned long，另一个操作数被转换为类型 unsigned long ● 如果前面的条件不满足,且如果操作数之一 为类型 long 而另一个为类型 unsigned int，两个操作数都被转换为类型 unsigned long。 ● 如果前面的两个条件不满足,且如果操作数之一为类型 long,则另一个操作数被转换为类型 long。

续表

- 如果前面的三个条件不满足,且如果操作数之一为类型 unsigned int,则另一个操作数被转换为类型 unsigned int。
- 如果前面的条件没有一个满足的,则两个操作数都被转换为类型 int。

以下代码说明了表 3.1 中描述的转换规则:

```
float fVal;
double dVal;
int iVal;
unsigned long ulVal;

dVal=iVal*ulVal;    //iVal 转换为 unsigned long 类型;乘法的结果转换为 double 类型。
dVal=ulVal+fVal;    //ulVal 转换为 float 类型;加法的结果转换为 double 类型。
```

上面例子中第一条语句给出了两个整数类型 iVal 和 ulVal 的乘法。满足的条件是没有一个操作数是浮点类型且其中一个操作数类型为 unsigned int,因此,另一个操作数 iVal 被转换为类型 unsigned int。结果赋给 dVal。满足的条件是一个操作数类型为 double,因此,乘法的 unsigned int 结果被转换为类型 double。

上面例子中的第二条语句给出了一个 float 和一个整数类型即 fVal 和 ulVal 的加法,ulVal 变量被转换为类型 float(表 3.1 中的第三个条件),加法的结果被转换为类型 double(表 3.1 中的第二个条件),且赋给 dVal。

指针转换

在赋值、初始化、比较和其它表达式期间指针可以被转换,本节叙述以下指针转换:

- 空指针
- void 类型指针
- 对象指针
- 函数指针
- 类指针
- 表达式
- 用 const 或 volatile 修饰的指针

空指针

等于 0 的一个整型常量表达式或造型转换为类型 void *的表达式,被转换为一个指针,且称为“空指针”。此指针确保与任意有效对象或函数的指针相比不相等(除了基本对象的指针,这种指针可以具有相同的偏移量但指向不同的对象)。

void 类型指针

void 类型指针可被转换为任何其它类型的指针,但必须用显式的类型造型(不像 C 语言)(有关类型造型的更多信息参见第 4 章“表达式”中的“带显式类型转换的表达式”)。任意类型的指针可被隐含地转换为 void 类型指针。
一个类型的不完整对象的一个指针可被转换为 void 指针(隐性地)并可以反回来(显式地),这种转换的结果等于源指针的值。一个对象如果被说明,但没有足够的信息来确定其尺寸或基类,则被认为是不完整的。

对象指针

任何不为 const 或 volatile 的对象的指针可被隐式地转换为 void *类型指针。

函数指针

一个函数指针可被转换为类型为 void *,如果类型 void *足够大可以保持那个指针。

类指针

在两种情况下,一个类指针可被转换为一个基类指针。
第一种情况是当指定的基类是可访问的,且转换不是模糊的(有关模糊的基类引用的更多信息参见第 9 章“派生类”中的“多重基类”)
一个基类是否可访问取决于派生中继承的种类,考虑图 3.1 中描述的继承性:

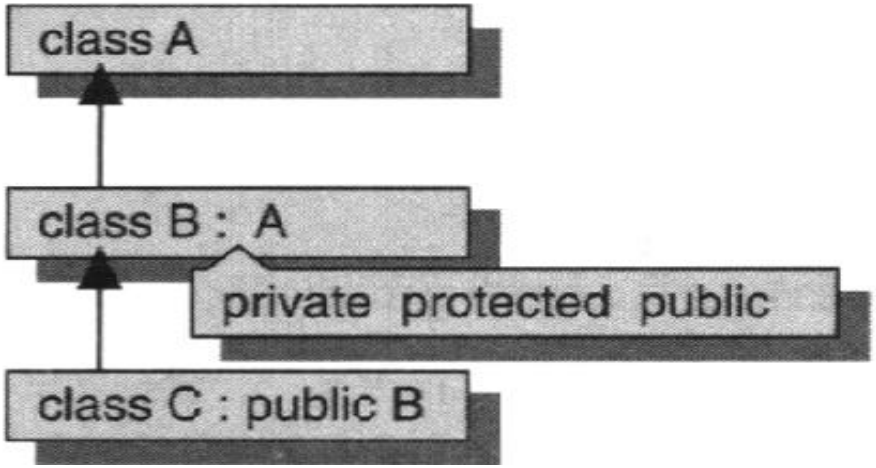


图 3.1　基类访问性描述的继承图

表 3.2 给出了图 3.1 描述的情形的基类访问性。

表 3.2　基类访问性

函数类型	派生	从 B*到 A*的转换是否合法
外部(非类范围)函数	私有的	否
	保护的	否
	公共的	是
B 成员函数(在 B 范围内)	私有的	是
	保护的	是
	公共的	是
C 成员函数(在 C 范围)	私有的	否
	保护的	是
	公共的	是

一个类指针可被转换为一个基类指针的第二种情况是当你使用显式类型转换的情形(有关显式类型转换的更多信息,参见第 4 章“表达式”中的“带显式类型转换的表达式”)。
这种转换的结果是一个“子对象”的指针,该对象部分完全由基类描述。
以下代码定义两个类 A 和 B,其中 B 由 A 派生而来(有关继承,见第 9 章“派生类”)然后定义了类型 B 的一个对象 bObject,和指向对象的两个指针(pA 和 pB)。

```
class A
{
public:
    int AComponent;
    int AMemberFunc();
};

Class B:public A
{
public:
   int BComponent;
   int BMemberFunc();
};
B bObject;
A *pA=&bObject;
B *pB=&bObject;
```

```
pA->AMemberFunc();  //在类 A 中可行
pB->AMemberFunc();  //可行:从类 A 中继承而来
pA->BMemberFunc();  //错误:不在类 A 中
```

指针 pA 为类型 A*,可被解释为“指向类型 A 的一个对象的指针”,bObject 成员(例如 BComponent 和 BMemberFunc)是类型 B 所唯一的,因此不能通过 pA 进行访问,pA 指针仅仅允许访问类 A 定义的对象的特征(成员函数和数据)。

指针表达式

任何带数组类型的表达式可被转换为同类型的一个指针,转换的结果是指向第一个数组元素的指针。以下例子说明了这种转换:

```
char szPath[_MAX_PATH];    //char 类型的数组
char *pszPath=szPath;      // 等于&szPath[0]
```

一个表达式结果是返回一特定类型的函数,可被转换为返回那个类型的函数的一个指针,除了:

- 该表达式被用作取地址运算符(&)的一个操作数。
- 该表达式被用作函数调用运算符的一个操作数。

用 const 或 volatile 修饰的指针

C++不提供 const 或 volatile 类型到非 const 或 volatile 的一个标准转换,但是任何类型的转换可以使用显式类型造型(包括不安全的转换)加以指定。

注意 C++的成员指针,除了静态成员指针之外,与普通指针不同且没有同样的标准转换。
静态成员指针是普通指针且有与普通指针一样的转换(有关更多的信息参见第 2 章“基本概念”中的“类成员指针”)。

引用转换

一个类的引用在下列情况下可被转换为一个基类的引用:

- 指定的基类是可访问的(正如本章前面的“类指针”中定义的)
- 转换是非模糊的(有关模糊基类引用的更多信息参见第 9 章“派生类”中的“多重基类”)。

转换的结果是表示基类的子对象的一个指针。
有关引用的更多信息,参见第 2 章“基本概念”中的“对象引用”。

成员指针转换

类成员指针在赋值、初始化、比较和其它表达式期间可被转换,本节描述了以下成员指针转换:

- 整型常量表达式
- 基类成员指针

整型常量表达式

等于 0 的一个整型常量表达式被转换为一个被称为“空指针”的指针。该指针确保与任意有效的对象或函数相比不相等(除了基本对象指针之外,这种指针可以具有相同的偏移量但指向不同的对象)。
以下代码给出了类 A 的成员 i 的一个指针的定义。pai 指针被初始化为 0,即空指针:

```
class A
{
public:
  int i:
};

int A::*pai=0;
```

基类成员指针

当以下条件被满足时,基类成员指针可被转换为从该类派生的类的成员指针:

- 逆转换,从派生类指针到基类指针,是可访问的。
- 派生类不从基类虚拟地继承。

当左操作数是一个成员指针时,右操作数必须为成员指针类型或为等于 0 的常量表达式。
这种赋值仅在以下情况下是有效的:

- 右操作数是与左操作数相同类的成员指针。
- 左操作数是从右操作数类公共地非模糊地派生的类成员的一个指针。

第 4 章　表　达　式

本章描述 C++的表达式,表达式是用于一个或多个以下目的的运算符和操作数序列:

- 从操作数计算出一个值
- 设计对象或函数
- 产生“副作用”(副作用是非表达式求值的任何动作,例如,修改一个对象的值)。

在 C++中,运算符可被重载而且它们的含义可由用户定义,但是它们的优先级以及所带操作数的个数不能被修改。本章描述该语言中所提供的而非重载的运算符的语法和语义,包括以下主题:

- 表达式的类型
- 表达式的语义
- 造型转换

(有关重载的运算符的更多信息参见第 12 章“重载”中的“重载的运算符”)。

注意:内部类型的运算符不能被重载,它们的行为是预先定义好的。

表达式的类型

C++表达式分为以下几类：

- 基本表达式，这些是构成所有其它表达式的基础。
- 后缀表达式，这些是基本表达式后面跟随一个运算符，例如，数组下标或后缀增 1 运算符。
- 由单目运算符构成的表达式，单目运算符只能作用表达式中的一个操作数。
- 由双目运算符构成的表达式，双目运算符作用于表达式中的两个操作数。
- 带条件运算符的表达式，条件运算符是一个三目运算符，在 C++语言中只有这种运算符带三个操作数。
- 常量表达式，常量表达式完全由常量数据构成。
- 带显式类型转换的表达式，显式类型转换或“造型转换”可以用在表达式中。
- 带成员指针运算符的表达式。
- 造型转换，类型安全的“造型转换”可以用于表达式中。
- 运行类型信息，指出程序执行期间对象的类型。

基本表达式

基本表达式是构造更复杂表达式的基础。它们是文字、名称和范围分辨运算符(::)

限定的名称。
语法
基本表达式:
    文字
    this
    ::标识符
    ::运算符函数名称
    ::限定名称
    (表达式)
    名称
文字是一个常量基本表达式,其类型依赖于其说明的形式,有关说明文字的全部信息参见第 1 章“词法规定”中的“文字”。
this 关键字是一个类对象指针,它在非静态成员函数内可以使用,且指向被调用函数的类的实例,this 关键字不能用在类成员函数体之外。
this 指针的类型是函数内的 *type* *const(其中 type 是类名),它特别地不修改 this 指针。以下例子给出了成员函数说明以及 this 类型 :

```
class Example
{
public:
  void Func();          //*const this
  void Func() const;  //const * const this
  void Func() volatile; //volatile *const this
```

}
有关修改 this 指针的类型的更多信息参见第 8 章“类”中的“this 指针的类型”。
范围分辨运算符(::)后面跟随一个标识符、运算符函数名称或限定名称构成一个基本表达式。这种表达式的类型通过标识符、运算符函数名称或名称的说明加以指定。如果说明的名称值为 1（1-value）,则它也是值为 1(1-value)。范围分辨运算符允许指示一个全局名称,即使那个名称被隐藏在当前范围内。有关如何使用范围分辨运算符的例子参见第 2 章“基本概念”中的“范围”。
包含在圆括号内的表达式是一个基本表达式,其类型和值与那些非括号括起的表达式的是完全相同的。如果非括号括起的表达式是 l 值的,则它也是 l 值的。

名称

在 C++基本表达式的语法中,一个名称是一个基本表达式，它仅可出现在成员选择运算符(.或->)之后并命名一个类的成员。
语法
名称：
    标识符
    运算符函数名称
    转换函数名称
    ~类名称
    限定名称
已被说明的任何标识符是一个名称。

一个运算符函数名称是一个说明为如下形式的名称:
operator 运算符名称(参量 1 [,参量 2]);
有关运算符函数名称的更多信息,参见第 12 章“重载”中的“重载的运算符”。
一个转换函数名称是一个说明为如下形式的名称:
运算符 类型名称()
注意:在说明一个转换函数时,你可以用一个派生的类型名称如 char 替代类型名称。
转换函数提供向用户定义类型以及从用户定义类型的转换。有关用户提供的转换的更多信息,参见第 11 章“特殊成员函数”中的“转换函数”。
被说明为类名称的名称被作为一个类类型对象的“析构函数”,该析构函数在一个对象生存期的结束地方执行清除操作。有关析构函数的信息,参见第 11 章“特殊成员函数”中的“析构函数”。

限定名称

语法
限定名称:
    限定类名称::名称
如果一个限定类名称后面跟随范围分辨运算符(::),再后是那个类或那个类的基类的成员名称,那么范围分辨运算符被认为是一个限定名称,一个限定名称的类型与成员的类型相同,而且限定名称表达式的结果是成员,如果成员是 l 值,则限定名称也是 l 值。有关说明限定类名称的信息,参见第 6 章“说明”中的“类型指示符”或第 8 章“类”中的“类名称”。

一个限定类名称的类名称部分可通过在当前或包围的范围中相同名称的重新说明而加以隐藏,类名称仍能够被找到和使用。有关如何使用一个限定类名称以访问一个隐藏类名称的例子参见第 2 章“基本概念”中的“范围”。
注意:类名称::类名称和类名称::~类名称的形式的类构造函数和析构函数分别必须指相同的类名称。
带有多于一个限定的名称,如下面所示,指定了一个嵌套类的一个成员:
类名称::类名称::名称

后缀表达式

后缀表达式构成基本表达式或后缀运算符跟在一个基本表达式后的表达式。后缀运算符列在表 4.1 中。

表 4.1 后缀运算符

运算符名称	运算符符号
下标运算符	[ ]
函数调用运算符	( )
显式类型转换运算符类型名	*type-name*( )
成员选择运算符	.或 ->
后缀增 1 运算符	++
后缀减 1 运算符	--

语法
后缀表达式:

基本表达式
后缀表达式[表达式]
后缀表达式(表达式表 $_{opt}$)
简单类型名(表达式表 $_{opt}$)
后缀表达式.名称
后缀表达式->名称
后缀表达式++
后缀表达式--

表达式表:
赋值表达式
表达式表,赋值表达式

下标运算符

一个后缀表达式后面跟随着下标运算符[],用于指定数组索引。表达式之一必须为指针或数组类型,即必须已被说明为 *type* *或 type[]。其它表达式必须为一个整数类型(包括枚举类型)。在通常用法中,括在方括号中的表达式是整型之一,但并没有严格的要求,考虑以下例子:

```
MyType  m[10];  //说明一个用户定义类型的数组

MyType  n1=m[2]; //选择数组的第三元素
MyType  n2=2[m]; //选择数组的第三元素
```

在前面例子中,表达式 m[2]与 2[m]完全相同。尽管 m 不是一个整数类型,但效果

是相同的。m[2]与 2[m]之所以相等是因为下标表达式 *e1*[*e2*]的结果由
*((*e2*)+(*e1*))
给定。由表达式产生的地址不是距离地址 *e1* 的 *e2* 字节,而是地址被定位以产生数组 *e2* 中的下一个对象,例如:

```
double aDb1[2];
```

aDb[0]和 aDb[1]的地址分别是 8 个字节,即一个 double 类型的对象的尺寸,这样按对象类型确定尺寸是由 C++语言自动完成的,且在本章后面的“附加的运算符”中定义,其中还讨论了指针类型操作数的加法和减法。

正和负下标

数组的第一个元素是元素 0,C++数组的范围是从 *array*[0]到 *array*[尺寸-1],但 C++支持正和负下标。负下标必须落在数组界限内或结果是不可预料的。以下例子说明了这个概念:

```
#include <iostream.h>

void main()
{
  int iNumberArray[1024];
  int *iNumberLine = &iNumberArray[512];

  cout << iNumberArray[-256] << "\n";    //不可预料的
  cout << iNumberLine[-256] << "\n";      //可行的
```

```
}
```

在 iNumberArray 中的负下标可能产生运行错误,因为它产生一个比原数组在存储器中低 256 个字节的一个地址。iNumberLine 对象被初始化为 iNumberArray 的中间量,因此对它既可以使用正数组索引也可以使用负数组索引。数组下标错误不产生编译错误,但它们产生不可预料的结果。

下标运算符是可以互换的,因此,只要下标运算符不被重载,则表达式 *array*[*index*] 和 *index*[*array*]被认为是相等的。(参见第 12 章“重载”中的“重载的运算符”)。第一种格式是最常见的编码用法,不过两者都行。

函数调用运算符

一个后缀表达式后面跟着函数调用运算符(),则指定一函数调用,函数调用运算符的参量是 0 个或多个表达式,函数的实际参量由逗号隔开。

后缀表达式必须是以下类型之一:

- 函数返回类型 T,一个说明例子为:

```
T func(int i)
```

- 函数返回类型 T 的指针,一个说明例子为:

```
T (*func)(int i)
```

- 函数返回类型 T 的引用,一个说明例子为:

```
T (&func)(int i)
```

- 成员指针函数间接引用返回类型 T,函数调用例子为:

```
(pObject->*pmf)();
(Object.*pmf)();
```

形式参量与实际参量

调用程序以“实际参量”形式向被调用函数传递信息,被调用函数使用相应的“形式参量”访问信息。

当一个函数被调用时,执行以下任务:

- 所有的实际参量(那些由调用者提供)被求值,这些变量被求值时没有隐式地顺序,但在进入函数之前所有的变量被求值,且所有的副作用被完成。
- 每个形式参量用其表达式表中相应的实际参量进行初始化(一个形式参量是一个在函数头部已作说明,并用于函数体内的参量),转换被完成好象通过初始化,即标准的和用户定义的转换均被执行以将实际参量转换为正确的类型。初始化的执行由以下例子进行概念性地说明:

```
void func(int i);//函数原型
...
Func(7);          //执行函数调用
```

调用前的概念性初始化为:

```
int Temp_i=7;
Func(Temp_i);
```

注意,初始化被执行就好象是使用等号语法而不是括号语法,i 的一个拷贝在向函数传递值之前被作好(有关更多的信息,参见第 7 章“说明符”中的“初始化表达式”,第 11 章“特殊成员函数”中的“转换”、“使用特殊成员函数的初始化”和“显式初始化”)。

因此,如果函数原型(说明)调用一个 long 类型参量,而如果调用程序提供一个 int 类型的实际参量,则实际参量用标准类型转换提升为 long 类型(参见第 3 章“标准转换”)。
提供一个实际参量而没有标准的或用户定义的转换将其转换为形式参量的类型则是一个错误。
对于类类型的实际参量,形式参量通过调用类构造函数被初始化(有关这些特殊类成员函数的更多信息,参见第 11 章“特殊成员函数”中的“构造函数”)。

- 函数调用被执行。

以下程序段说明了一个函数调用:

```
void fanc(long param1,double param2);

void main()
{
   int i,j;

   //用实际参量 i 和 j 调用 func
   func(i,j);
   ...
}

//用形式参量 param1 和 param2 定义 func
void func(long param1,double param2)
```

```
{
    ...
}
```

当从 main 调用 func 时,形式参量 param1 用 i 值(使用标准转换将 i 转换为 long 类型相对应的正确类型)进行初始化,形式参量 param2 用 j 值(使用标准转换将 j 转换为 double 类型)进行初始化。

参量类型的处理

被说明为常量类型的形式参量不能在函数体内进行改变,函数可以改变任何非 const 类型的变量。但是改变相对函数是局部的,不会影响实际参量的值,除非实际参量是一个非 const 类型对象的一个引用。

以下函数说明了一部分这些概念:

```
int func1(const int i,int j,char *c)
{
    i=7;          //错误:i 是常量
   j=1;       //可以, 但 j 值在返回值丢失
   *c='a'+j;  //可以:在调用函数中改变 c 的值

   return i;
}

double& func2(double& d,coust char *c)
```

```
{
    d=14.387;   //可以:在调用函数中改变 d 的值
    *c='a';   //错误:c 是一个指向常量对象的指针

    return d;
}
```

省略号和缺省参量

函数可以说明为接受比函数定义中指定参量更少的情况,这是通过使用以下两种方法之一实现的:省略号(...)或缺省参量。

省略号意味着参量可能被要求,但说明中没有指定数目和类型。这通常是糟糕的C++编程方法,因为它使 C++的益处之一即类型安全性失败。不同的转换用于使用省略号说明的函数而不是那些形式和实际参量类型已知的函数:

- 如果实际参量为 float 类型,则它在函数调用前被提升为 double 类型。
- 任何有符号的或无符号的 char、short、枚举类型或位域使用整型提升被转换为有符号的或无符号的 int。
- 任何作为一个数据结构的值传递的类类型参量;拷贝是通过二进制拷贝创建的而不是通用调用类的拷贝构造函数(如果存在的话)实现的。

省略号,如果使用,必须在参量表的最后进行说明。有关传递可变个数参量的更多信息,参见“Microsoft Visual C++6.0 参考库”中的“Microsoft Visual C++6.0

运行库参考”中的 va_arg、va_start 和 va_list 的讨论。
缺省参量能使你能够指定这样的一个参量的值,假设它在函数调用中不提供。以下代码段给出了缺省参量的工作情况,有关指定缺省参量的限制的更多信息见第7章中的“缺省参量”。

```
#include <iostream.h>

//说明一个 print 函数打印字符串及一个终止符。
void print (const char *string, const char *terminator="\n");

void main()
{
    print("hello,");
    print("World!");

    print("good morning",",");
    print("sunshine.");
}

//定义 print
void print(char *string, char *terminator)
{
    if (string!=NULL)
```

```
        cont << string;

    if (terminator!=NULL)
        cout << terminator;
}
```

前面的程序说明了一个函数 print,它带两个参数,但第二个参量 terminator 有一个缺省值“\n”。在 main 中,开头两次调用 print 允许缺省的第二个参量提供一个换行终止打印的字符串,第三个调用为第二个参量指定了明确的值,程序的输出为:

```
hello,
world!
good morning, sunsline.
```

函数调用结果

一个函数调用求值为一个 r 值,除非函数被说明为一个引用类型。带引用的函数返回类型求值为 l 值,而且可以用于一个赋值语句的左侧,如下所示:

```
#include <iostream.h>

class Point
{
public:
    //定义“accessor”函数为引用类型
```

```
    unsigned & x() { return _x;}
    unsigned & y() { return _y;}
private:
    unsigned _x;
    unsigned _y;
};

void main()
{
     Point ThePoint;

     ThePoint.x( )=7;              //使用 x()作为一个 l 值
     unsigned y=ThePoint.y( ); //使用 y()作为一个 r 值

    //使用 x()和 y()作为 r 值
    cout << "x=" << ThePoint.x() << "\n"
         << "y=" << ThePoint.y() << "\n";
}
```

000000

前面的代码定义了一个类调用 Point,包含表示 x 和 y 坐标的私有数据对象,这些数据对象必须被修改且它们的值被检索。该程序只是这样的一个类的几种设计之一,使用 GetX 和 SetX 或者 GetY 和 SetY 函数是另一种可能的设计。

返回类类型、类类型指针或类类型引用的函数可被用作成员选择运算符的左操作数,因此,以下代码是合法的:

```
class A:
{
public:
    int SetA(int i){ return(I=i); }
    int GetA() { return I; }

private:
    int I;
};

//说明三个函数
//func1,返回类型 A
//func2,返回类型 A 的一个指针
//func3,返回类型 A 的一个引用
A func1();
A* func2();
A& func3();

int iResult=func1().GetA();
func2()->SetA(3);
```

```
func3().SetA(7);
```

函数可被递归地调用,有关函数说明的更多信息参见第 6 章“说明中的函数指示符”,第 8 章“类”中的“指示符”和“成员函数”。相关的材料在第 2 章“基本概念”中的“程序和连接”中。

成员选择运算符

一个后缀表达式后面跟随着一个成员选择运算符(.)和一个名称,是一个后缀表达式的另一种例子。成员选择运算符的第一个操作数必须有类或类引用类型,第二个操作数必须标识那个类的一个成员。

表达式的结果是成员的值,而且如果被命名的成员值为 1,则它也值为 1。

一个后缀表达式后面跟随着一个成员选择运算符(->)和一个名称,是一个后缀表达式。成员选择运算符的第一个操作数必须有一个类对象的类型指针(说明为 class、struct 或 union 类型的一个对象);第二个操作数必须指定那个类的一个成员。

表达式的结果是成员的值,而且如果被命名的成员值为 1,则它也是值为 1。

->运算符间接引用指针,因此,表达式 *e->member* 和(**e*).*member*(其中 *e* 表示一个表达式)产生完全相同的结果(除了运算符->或*被重载时之外)。

当一个值通过一个联合的某成员被存储,但通过另一个成员被检索时,没有转换被执行。以下程序将数据作为 int 存入对象 U,但检索数据却作为 char 类型的两个分开的字节:

```
#include <iostream.h>
```

```
void main()
{
    struct ch
    {
        char b1;
        char b2;
    };

    union u
    {
         struct ch uch;
         short i;
    };

    u U;

    U.i=0x6361;  //“ac”的位模式
    cout << U.uch.b1 << U.uch.b2 << "\n";
}
```

后缀增 1 及减 1 运算符

C++提供前缀及后缀增 1 和减 1 运算符;本节仅描述后缀增 1 和减 1 运算符。(更

多的信息参见第 12 章“重载”中的“增 1 和减 1”)。两者的区别在于,在后缀表示法中,运算符出现在后缀表达式之后,而在前缀表示法中,运算符出现在表达式前。以下例子给出了一个后缀增 1 运算符:

```
i++
```

使用后缀增 1 或“后增 1”运算符(++)的作用是操作数增加相应类型的一个单位量。同样地,使用后缀减 1 或“后减 1”运算符(--)的作用是操作数减少相应类型的一个单位量。

例如,使用后缀增 1 运算符于一个 long 类型对象数组的一个指针,实际上指针内部表示法中增加了 4。此动作使先前指向数组第 n 个元素的指针指向第($n+1$)个元素。

后缀增 1 和后缀减 1 运算符的操作数必须是算术或指针类型的可修改(非 const)的 l 值,后缀增 1 或后缀减 1 表达式的结果是使用增 1 运算符之前的后缀表达式的值,结果的类型与后缀表达式的相同,但不再是一个 l 值。

以下代码说明了后缀增 1 运算符:

```
if (var++>0)
    *p++=*q++;
```

在此例中,变量 var 与 0 相比,而后增 1。如果 var 在加 1 之前为正数,则执行下一条语句。首先,由 q 指向的对象的值赋给由 p 指向的对象,然后,q 和 p 都增 1。

后增 1 和后减 1,当被用于枚举类型时,产生整型值。因此,以下代码是非法的:

```
enum Days {
    Sunday=1,
    Monday,
```

```
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};

    void main()
    {
        Days Today = Tuesday;
      Days SaveToday;

        SaveToday = Today++;    //错误
}
```

这段代码的目的是保存 today 的星期号,然后移到下一天,但是结果是 today++ 表达式产生一个 int 即在赋给枚举类型 Days 的对象时出现一个错误。

带单目运算符的表达式

单目运算符仅作用于表达式中的一个操作数,单目运算符有:

- 间接运算符(*)
- 取地址运算符(&)
- 单目加运算符(+)

- 单目负运算符(-)
- 逻辑非运算符(!)
- “1”的求补运算符(~)
- 前缀增 1 运算符(++)
- 前缀减 1 运算符(--)
- sizeof 运算符
- new 运算符
- delete 运算符

这些运算符具有从右向左的结合律。

语法

单目表达式:

后缀表达式

++单目表达式

--单目表达式

单目运算符 造型转换表达式

sizeof.单目表达式

sizeof(类型名称)

分配表达式

撤消分配表达式

单目运算符:以下之一

* & + - ! ~

间接运算符(*)

单目间接运算符(*)“间接引用”一个指针,即它将一个指针值转换为一个 l 值。间接运算符的操作数必须是一个类型的指针。间接表达式的结果是指针类型从其派生的类型。在此上下文中,*运算符的使用不同于它作为双目运算符即乘法的含义。

如果操作数指向一个函数,结果是一个函数指示符。如果它指向一个存储位置,则结果是指定该存储位置的一个 l 值。

如果指针值是无效的,则结果是不确定的。以下表包括一些最常见的条件使指针值无效:

- 指针是一个空指针。
- 指针指定一个局部项的地址,在引用时是不可见的。
- 指针指定一个地址不适当地与该对象指向的类型对齐
- 指针指定一个地址不被执行程序使用。

取地址运算符(&)

单目取地址运算符(&)取其操作数的地址,取地址运算仅能用于以下情况:

- 函数(尽管取函数地址的用法是不需要的)
- l 值
- 限定名称

在上面所列的前面两种情况下,表达式的结果是从操作数类型派生出的一个指针类型(一个 r 值)。例如,如果操作数为 char 类型,则表达式的结果是 char 指针类型。取地址运算符被用于 const 或 volatile 对象则等于 const *type**或

volatile *type**,其中 *type* 为源对象的类型。
第三种情况将取地址运算符应用在一个限定名称上，产生的结果取决于限定名称是否指定一个静态成员,如果是,结果是指向成员说明中指定类型的一个指针;如果成员不是静态的,则结果是由限定类名称指示的类成员名称的一个指针(有关限定类名称的更多信息,参见本章前面的“基本表达式”)。以下代码段给出:根据成员是否是静态的其结果是如何不同的:

```
class PTM
{
public:
      int iValue;
  static float fValue;
};

int   PTM::*piValue = &PTM::iValue;    //可行:非静态的
float PTM::*pfValue = &PTM::fValue;    //错误:静态的
float *spfValue = &PTM::fValue;          //可行
```

在此例中,表达式&PTM::fValue 产生 float*类型而不是 float PTM::*类型,因为 fValue 是一个静态成员。
一个重载的函数的地址仅在清楚地知道引用的是函数的哪个版本时才能被获取。关于如何获取一个特定的重载的函数的地址的信息,参见第 12 章“重载”中的“重载的函数的地址”。
对一个引用类型使用取地址运算符与对一个引用限定的对象上使用该运算符给

出相同的结果。以下程序说明了这个概念：

```
#include <iostream.h>

void main()
{
    double d;         //定义 double 类型的一个对象
    double& rd=d;     //定义该对象的一个引用
    //比较对象的地址和对象引用的地址
    if (&d==&rd)
        cout << "&d equals &rd" << "\n";
   else
        cout << "&d is not equal to &rd" << "\n";
}
```

程序的输出总是:&d equals &rd。

单目加运算符(+)

单目加运算符(+)的结果是其操作数的值。单目加运算符的操作数必须是一个算术类型。

在整型操作数上执行整型提升。结果类型是操作数被提升的类型。因而,表达式+ch,其中 ch 为 char 类型，结果为 int 类型；值未修改。关于提升是如何完成的信息,参见第 3 章“标准转换”中的“整型提升”。

单目负运算符(-)

单目负运算符(-)对其操作数求反。单目负运算符的操作数必须为一个算术类型。
在整型操作数上执行整型提升，结果类型是操作数被提升的类型。有关提升是如可完成的信息,参见第 3 章“标准转换”的“整型提升”。
Microsoft 特殊处→
无符号量单目求反的执行是从 2^n 中减去操作数的值,其中 n 是给定无符号类型的对象的位数(Microsoft C++运行在对 2 的补码运算的处理器中,在其它处理器中求反算法可能有所不同)。
Microsoft 特殊处结束

逻辑非运算符(!)

如果操作数为非 0 值,则逻辑非运算符(!)的结果为 0;仅当操作数为 0 时结果方为 1。操作数必须为算术或指针类型,结果是 int 类型。
对一个表达式 *e*,单目表达式!*e* 等于表达式(*e*==0),除非包含了重载的运算符。
以下例子说明了逻辑非运算符(!):

```
if ( !(x<y) )
```

如果 x 大于等于 y,表达式的结果为 1(真);如果 x 小于 y,结果为 0(假)。
指针的单目算术运算是非法的。

“1”的求补运算符(~)

“1”求补运算符(~),有时被称为“按位补”运算符,产生其操作数的按位的“1”

的补,即在操作数中每一位为 1 的在结果中为 0;相反地,操作数中为 0 的每位在结果中为 1。“1”求补运算符的操作数必须为整型。

```
unsigned short y=0xAAAA;
    y=~y;
```

在此例中,赋给 y 的新值是无符号值 0xAAAA 的“1”求补或 0x5555。

在整型操作数上执行整型提升,结果类型是操作数被提升的类型。关于升级是如何完成的的更多信息,参见第 3 章“标准转换”中的“整型提升”。

增 1 和减 1 运算符(++,--)

前缀增 1 运算符(++),亦被称为“前增 1”运算符,在其操作数上加 1,这个加 1 后的值是表达式的结果。操作数必须是一个非 const 类型的 l 值,结果是与操作数类型相同的一个 l 值。

前缀减 1 运算符(--),亦被称为“前减 1”运算符,与前增 1 运算符相似,除了操作数是减 1 且结果是这个减 1 后的值。

前缀和后缀增 1 及减 1 运算符均影响其操作数,关键的不同点是表达式的值何时发生增 1 或减 1(有关更多的信息,参见本章前面的“后缀增 1 和减 1 运算符”)。在前缀形式中,增 1 或减 1 发生在值被用于表达式求值之前,所以表达式的值与操作数的值不同;在后缀形式中,增 1 或减 1 发生在位被用于表达式求值之后,所以表达式的值与操作数的值相同。

整型或浮点类型的操作数按整数值 1 进行增或减,结果的类型与操作数类型相同;指针类型操作数按它所指对象的尺寸增 1 或减 1,被增 1 的指针指向下一个对象,被减 1 的指针指向前一个对象。

这个例子说明了单目减 1 运算符：

```
if (line [--i]!=‘ \n ’ )
    return;
```

在此例中，变量 i 在用作 line 下标之前减 1。

因为增 1 和减 1 运算符有副作用，在宏中使用带增 1 或减 1 运算符的表达式可产生意料不到的结果(有关宏的更多信息，参见本部分后面“预处理器引用”中的“宏”)。考虑以下例子：

```
#define max(a,b) ((a)<(b)) ? (b) : (a)

...

int i,j,k;

k=max(++i,j);
```

宏扩展为：

```
k=((++i)<(j)) ? (j) : (++i);
```

如果 i 大于等于 j，它将增 1 两次。

注意：在很多情况下，C++的内联函数(inline functions)比宏更好，因为它们消除了这里描述的副作用，而允许语言执行更完整的类型检测。

sizeof 运算符

sizeof 运算符产生关于 char 类型尺寸的操作数尺寸，sizeof 运算符的结果是

size_t 类型，它是定义在包括文件 STDDEF.H 中的一个整型。sizeof 操作数可以是以下之一：

- 一个类型名。要对类型名用 sizeof,则名称必须用圆括号括起来。
- 一个表达式。当用于表达式时,sizeof 可以使用或不使用括号指定，该表达式不求值。

当 sizeof 运算符用于 char 类型的对象时,它结果为 1;当 sizeof 运算符用于一个数组时,它结果为那个数组中的字节总数。例如：

```
#include <iostream.h>

void main()
{
    char szHello[]="Hello, world!";

    cout << "The size of the type of " << szHello <<" is:"
        << sizeof(char) << "\n";
    cout <<"The length of " << szHello << "is:"
        << sizeof szHello << "\n";
}
```

程序输出为：

```
The size of the type of Hello, world! is:1
The length of Hello, world! is:14
```

当 sizeof 运算符用于 class、struct 或 union 类型时,结果是该 class、struct

或 union 类型的一个对象的字节数加上字边界上任何添加的对齐成员的填充字节。(/Zp[紧凑结构成员]编译器选项和 pack 编译指示影响成员的对齐边界)。
sizeof 运算符从不产生 0,即使对一个空类亦如此。
sizeof 运算符不能用于以下操作数:

- 函数(但是 sizeof 可应用于函数指针)
- 位域
- 未定义的类
- void 类型
- 不完整类型
- 括号括起的不完整类型的名称

当 sizeof 运算符用于一个引用时,结果就如同 sizeof 用于对象自身一样。sizeof 运算符经常被用于计算一个数组的元素个数,使用如下形式的表达式:
sizeof array/sizeof array[0]

new 运算符

new 运算符试图动态地分配(在运行时)类型名称的一个或多个对象。new 运算符不能用于分配一个函数,但可以用于分配一个函数指针。
语法
分配表达式:

::$_{opt}$ new 位置 $_{opt}$ 新类型名称 新初始化器 $_{opt}$
::$_{opt}$ new 位置 $_{opt}$(类型名称) 新初始化器 $_{opt}$

位置:

(表达式表)
新类型名称:
类型指示符表 新说明符$_{opt}$
new 运算符被用于分配对象及对象数组,new 运算符从被称为“自由存储”的程序存储器区域中进行分配。在 C 中,自由存储区经常被指示为“堆”。
当 new 被用于分配一个单个对象时,它产生一个指向那个对象的指针;结果类型为新类型名称*或类型名称*;当 new 被用于分配一个单维对象数组时,它产生一个指向数组第一个元素的指针,结果类型为新类型名称*或类型名称*;当 new 被用于分配一个多维对象数组时,它产生一个指向数组第一个元素的指针,而且结果类型保留除最左边数组维数外的所有的尺寸,例如:
new float[10][25][10]
产生类型 float(*)[25][10]。因此,以下代码不能工作,因为它试图用类型 float 的指针的维数[25][10]赋给一个 float 数组的指针:
```
float  *fp;
fp=new float [10][25][10];
```
正确的表达式应是:
```
float (*cp)[25][10]
cp=new float[10][25][10];
```
cp 的定义用维数[25][10]分配一个 float 类型数组的指针,它不分配指针数组。除最左边数组维数外的所有维数必须为等于正数的常量表达式;最左边数组维数可以为等于正数的任意表达式,当用 new 运算符分配一个数组时,第一个维数可以为 0,new 运算符返回一个唯一的指针。

类型指示符表不能包含 const、volatile、类说明或枚举说明。因此,以下表达式是非法的:

```
volatile char *vch = new volatile char[20];
```

new 运算符不分配引用类型,因为它们不是对象。

如果没有足够的存储器满足分配需求,缺省地 operator new 返回 NULL。你可以通过编写定制的异常处理例程并以你的函数名称作为参量调用 _set_new_handler 运行库函数来改变这种缺省行为。有关这种恢复模式详细内容,参见第 11 章“特殊成员函数”中的“运算符 new 功能”。

用 new 分配的对象的生存期

用 new 运算符分配的对象当它们从定义的所在范围退出时不被销毁。因为 new 运算符返回它分配的对象的指针,程序必须用合适的范围定义一个指针以访问那些对象。例如:

```
void main()
{
    //使用 new 运算符分配一个 20 个字符的数组
    char *AnArray = new char[20];

    for (int i=0;i<20;++i)
    {
        //在循环的第一次迭代时,分配另一个 20 个字符的数组
        if (i==0)
```

```
        {
        char *AnotherArray = new char[20];
        }
       ...
    }
 delete AnotherArray;  //错误:指针超出范围
  delete AnArray;   //可以:指针还在范围内
}
```

在例子中一旦指针 AnotherArray 超出了范围,对象再也不能被删除了。

初始化用 new 分配的对象

选项 new 初始化器域包含在 new 运算符的语法内,这允许使用用户定义的构造函数去初始化 new 对象。关于初始化是如何完成的更多信息,参见第 7 章“说明符”中“初始化器”。

以下例子说明如何使用带 new 运算符的初始化表达式:

```
#include <iostream.h>

class Acct
{
public:
   //定义缺省构造函数和构造函数接受初始的 balance
   Acct() { balance=0.0; }
```

```
    Acct( double init_balance ) { balance = init_balance; }
private:
    double balance;
};
void main()
{
    Acct *CheckingAcct = new Acct;
    Acct *SavingAcct = new Acct(34.98);
    double *HowMuch = new double(43.0);
    ...
}
```

在此例中,对象 CheckingAcct 使用 new 运算符分配,但没有指定缺省的初始化,因此,调用类的缺省构造函数 Acct();然后以同样的方式分配对象 SavingAcct,但被显式地初始化为 34.98。由于 34.98 是 double 类型,调用以该类型为参量的构造函数进行初始化处理。最后,非类类型 HowMuch 初始化为 43.0。

如果一个对象是一个类类型的,而且那个类有构造函数(如前面例子那样),仅在以下条件之一被满足时对象才能被 new 运算符初始化:

- 在初始化器中提供的参量与构造函数中的那些参量相符。
- 类有一个缺省的构造函数(一个可以不带参量调用的构造函数)。

访问控制和模糊控制在 operator new 上和在按照第 9 章“派生类中的模糊性”及第 11 章“特殊成员函数”中的“使用特殊成员函数初始化”中提出的规则集的构造函数上执行。

当使用new运算符分配数组时没有显式地为每个元素执行初始化;如果存在的话,只有缺省构造函数被调用。有关更多的信息,参见第 7 章“说明符”中的“缺省参量”。
如果存储器分配失败(operator new 返回一个 0 值),则没有初始化被执行,这阻止了对不存在的数据进行初始化。
由于用函数调用,初始化表达式被求值的顺序未定义,而且,你不能依靠这些表达式在存储器分配执行之前完全被求值。如果存储器分配失败,new 运算符返回 0,在初始化器中的某些表达式可能不能被完全求值。

new 是如何工作的

分配表达式即包含 new 运算符的表达式做三件事:

- 为对象或待分配的对象定位并保留存储器,当这步完成时,分配正确的存储单元数,但还不是一个对象。
- 初始化对象,一旦初始化完成,则有足够的信息为该对象分配存储。
- 返回一个指向由 *new-type-name* 或 *type-name* 派生的指针类型对象的一个指针,程序使用该指针访问新分配的对象。

new 运算符调用函数 operator new。对于任何类型的数组及非 class、struct 或 union 类型的对象,全局函数::operator new 被调用去分配存储器,类类型对象可以在每个类基础上定义自己的 operator new 静态成员。
当编译器遇到 new 运算符去分配一个 *type* 类型对象时,它调用 *type*::operator new (sizeof (*type*));或者若是没有用户定义的 operator new 被定义,则调用::operator new (sizeof (*type*))。因此,new 运算符可以为对象分配正确的

存储器数目。
注意:operator new 的参量是 size_t 类型,这个类型定义在 DIRECT.H、MALLOC.H、MEMORY.H、SEARCH.H、STDDEF.H、STDIO.H、STDLIB.H、STRING.H 和 TIME.H 中。
语法中的一个选项允许位置的规格(参见 new 运算符的语法)。该位置参量仅能在 operator new 用户定义的实现中被使用;它允许将额外的信息传递给 operator new,带位置域的表达式如:

```
T *TObject = new(0x0040) T;
```

被翻译为:

```
T *TObject = T::operator new(sizeof(T),0x0040);
```

位置域最初的目的是允许依赖于硬件的对象在用户指定的地址处分配。
注意:尽管前面的例子给出的是在位置域仅有一个参量,但对这种方式有多少个额外的参量可被传递给 operator new 并没有限制。
即使当 operator new 已为一个类类型定义,全局运算符还可以通过下面例子中的形式来使用:

```
T *TObject = ::new TObject:
```

范围分辩运算符(::)强制使用全局 new 运算符。

delete 运算符

delete 运算符撤消由 new 运算符创建的对象分配。delete 运算符有一个 void 类型的结果,因此不返回值,delete 的操作数必须是由 new 运算符返回的一个指针。

对不是用 new 分配的对象指针使用 delete 会产生意料不到的结果,但是你可以对具有 0 值的指针使用 delete。因为在失败的情况下 new 总是返回 0,所以这种措施意味着删除一个失败的 new 操作的结果是无害的。

语法

撤消分配表达式:

::$_{opt}$ delete 造型转换表达式

::$_{opt}$ delete [] 造型转换表达式

对一个对象使用 delete 运算符将撤消其存储器分配,一个程序在对象被删除后间接引用一个指针则会产生意料不到的结果或崩溃。

如果 delete 运算符的操作数是一个可修改的 l 值,则在对象被删除后其值是未定义的。

常量对象指针不能用 delete 运算符撤消分配。

delete 是如何工作的

delete 运算符调用函数运算符 delete,对类类型(class(类)、struct(结构)和 union(联合))对象,delete 运算符在撤消存储器分配(如果指针非空)之前调用对象的析构函数。对非类类型对象,全局 delete 运算符被调用。对类类型对象,delete 运算符可被定义在一个每个类基础上。如果对给定类没有这样的定义,则全局运算符被调用。

使用 delete

对 delete 运算符有两个语法变型:一个针对单个对象的,另一个是对对象数组

的。以下代码段给出这些的不同：

```
void main()
{
      //使用new运算符在自由存储区分配一个用户定义对象UDObject和double
类型对象
      UDType *UDObject = new UDType;
      double *dObject = new double;
      ...
      //删除两个对象
      delete UDObject;
      delete dObject;
      ...
      //使用new运算符在自由存储区分配一个用户定义对象数组
      UDType (*UDArr)[7] = new UDType[5][7];
      ...
      //使用数组语法删除对象数组
      delete [] UDArr;
}
```

这两种情况产生未定义的结果：对一个对象使用 delete 的数组形式(delete[])和对一个数组使用 delete 的非数组形式。

带双目运算符的表达式

双目运算符作用于一个表达式中的两个操作数。双目运算符有：

乘法运算符

- 乘法(*)
- 除法(/)
- 取模(%)

加法运算符

- 加法(+)
- 减法(-)

位移运算符

- 右移(>>)
- 左移(<<)

关系及相等性运算符

- 小于(<)
- 大于(>)
- 小于等于(<=)
- 大于等于(>=)
- 等于(==)
- 不等于(!=)

按位运算符

- 按位与(&)

按位异或(^)
按位或(|)
逻辑与(&&)
逻辑或(||)

乘法运算符

乘法运算符有:

- 乘法(*)
- 除法(/)
- 取模或“除法取余”(%)

这些双目运算符具有从左到右的结合律。

语法

乘法表达式:

pm表达式
乘法表达式 * pm表达式
乘法表达式 / pm表达式
乘法表达式 % pm表达式

乘法表达式采用算术类型操作数,取模运算符(%)有一个较为严格的要求,即其操作数必须为整型(要得到浮点除法的余数,使用运行函数 fmod)。第 3 章“标准转换”中的“算术转换”中包含的转换适用于操作数,且结果是转换后的类型。

乘法运算符产生的是第一个操作数被第二个相乘的结果。

除法运算符产生的是第一个操作数被第二个相除的结果。

取模运算符产生由表达式 *e*1-(*e*1/*e*2)***e*2 给出的余数,其中 *e*1 是第一个操作数,*e*2 是第二个操作数,其中两个操作数均为整型。
在除法或取模数表达式中被 0 除都是未定义的,会产生一个运行错误。因此,以下表达式产生未定义的错误的结果:
i % 0
f / 0.0
如果乘法、除法或取模表达式的两个操作数具有相同的符号,则结果为正;否则,结果为负,取模操作的符号结果是定义的实现。
Microsoft 特殊处→
在 Microsoft C++中,取模表达式的结果总是与第一个操作数的符号相同。
Microsoft 特殊处结束
如果两个整数计算的除法是不精确的,而且仅一个操作数为负,结果是比除法产生的准确值小的最大整数(在量上,不考虑符号)。例如,-11/3 计算的结果为-3.666666666,则整型除法的结果为-3。
乘法运算符之间的关系由以下标识给出:
(*e1*/*e2*)**e2*+*e1*%*e2*==*e1*

加法运算符

加法运算符有:

- 加法(+)
- 减法(-)

这些双目运算符具有从左到右的结合律。

语法

加法表达式：

乘法表达式

加法表达式 + 乘法表达式

加法表达式 - 乘法表达式

加法运算符采用算术或指针类型操作数。加法运算符(+)的结果是操作数求和；减法运算符(-)的结果是操作数之间的差。如果一个或两个操作数都是指针，它们必须是对象的指针，而不是函数的指针。

加法运算符的操作数为算术、整型或标量型，这些定义在表 4.2 中。

表 4.2 加法运算符使用的类型

类型	含义
算术型	整型和浮点类型统称为"算术"类型
整型	char 和所有尺寸的 int(long, short)类型和枚举都是“整数”类型
标量型	标量型操作数是算术或指针类型的操作数

这些运算符的合法组合为：

算术型 + 算术型

标量型 + 整 型

整 型 + 标量型

算术型 - 算术型

标量型 - 标量型

注意:加法和减法不是等价操作。
如果两个操作数都是算术类型,则在第 3 章“标准转换”中“算术转换”包括的转换应用于操作数,而且结果是被转换了的类型。

指针类型的加法

如果加法操作中一个操作数为指向一个对象数组的指针,则另一个必须为整型,其结果是与源指针同类型的指针,且指向另一个数组元素。以下代码段说明这个概念:

```
short IntArray[10];    //short 类型对象占两个字节
short *pIntArray=Int Array;

for (int i=0;i<10;++i)
{
    *pIntArray=i;
    cout << *pIntArray << "\n";
    pIntArray = pIntArray + 1;
}
```

尽管整型值 1 被加到 pIntArray 上,但它并不意味着“地址加 1”;而是指“调整指针指向数组中的下一个对象”,而且正好是走了两个字节(或 sizeof(int))。
注意:pIntArray=pIntArray+1 形式的代码在 C++程序中很少看到。为了执行增 1,这样形式更好:pIntArray++或 pIntArray+=1。

指针类型的减法

如果两个操作数都是指针,减法的结果是两个操作数之间的差(在数组元素中)。减法表达式产生一个类型 ptrdiff_t(定义在标准包括文件 STDDEF.H 中)的有符号的整型结果。

操作数之一可以是整型,它只能是第二个操作数。减法的结果与源指针同类型,减法的值是指向第(n-i)个数组元素的指针,其中 n 是由源指针指向的元素,i 是第二个操作数的整型值。

位移运算符

按位移位运算符有:

- 右移(>>)
- 左移(<<)

这些双目运算符具有从左到右的结合律。

语法

移位表达式:

加法表达式

移位表达式 << 加法表达式

移位表达式 >> 加法表达式

移位运算符的两个操作数都必须是整型,整型提升的执行按照第 3 章“标准转换”中的“整型提升”中指定的规则进行。结果的类型与左操作数类型相同。

右移表达式 $e1>>e2$ 的值为 $e1/2^{e2}$;左移表达式 $e1<<e2$ 的值为 $e1*2^{e2}$。

如果移位表达式的右操作数为负数或如果右操作数大于等于左操作数(提升了的)

的位数,则结果是未定义的。
左移运算符使得第一个操作数的位模式向左移动由第二个操作数指定的位数,由移位操作进行的空位填充是 0 填充,这是逻辑移位,与循环移位相反。
右移运算符使得第一个操作数的位模式向右移动由第二个操作数指定的位数。对无符号量移位操作的空位填充是 0 填充,对有符号量,符号位被传入空位。如果左操作数是一个无符号量,则移位是一个逻辑移位,否则是算术移位。
Microsoft 特殊处→
有符号的负数的右移结果是依赖于定义的实现,尽管 Microsoft C++传播最重要的位以填充空位,但不保证其它的实现也这样做。
Microsoft 特殊处结束

关系与相等运算符

关系与相等运算符指定其操作数的相等、不等或关系值,关系运算符在表 4.3 中所示。

表 4.3　关系与相等运算符

运算符	含义
==	等于
!=	不等于
<	小于
>	大于
<=	小于等于
>=	大于等于

关系运算符

双目关系运算符指定以下关系：

- 小于
- 大于
- 小于等于
- 大于等于

语法

关系表达式：

移位表达式

关系表达式 < 移位表达式

关系表达式 > 移位表达式

关系表达式 <= 移位表达式

关系表达式 <= 移位表达式

关系运算符具有从左到右的结合律。关系运算符的两个操作数必须为算术或指针类型,它们产生 int 类型值。如果表达式中关系为假则返回的值为 0;否则为 1。考虑以下代码,其中说明了几种关系表达式:

```
#include <iostream.h>

void main()
{
    cout << "The true expression 3 > 2 yields: "
         << (3 > 2) << "\n";
    cout << "The false expression 20 < 10 yields: "
         << (20 < 10) << "\n";
    cout << "The expression 10 < 20 < 5 yields: "
         << (10 < 20 < 5) << "\n";
}
```

此程序的输出为:

```
The true expression 3 > 2 yields:1
The false expression 20 < 10 yields:0
The expression 10 < 20 < 5 yields:1
```

在前面例子中的表达式必须包含在括号内,因为插入运算符(<<)比关系运算符优先级高。因此,不带括号的第一个表达式将等于:

```
 (cout << "The true expression 3 > 2 yields:" << 3) < (2 << "\n");
```

注意第三个表达式等于 1,因为关系运算符从左到右的结合律,表达式 10<20<5 的显式分组为:

(10 < 20) < 5

因此,测试执行的是:

1<5

结果为 1(或真)。

第 3 章“标准转换”中的“算术转换”包含的常用的算术转换应用于算术类型的操作数上。

使用关系运算符比较指针

当同类型对象的两个指针进行比较时,结果由指向程序地址空间的对象位置指定。指针还可以和等于 0 或 void *类型指针的常量表达式进行比较。如果指针比较相对于 void*类型指针,则另一指针被隐式地转换为 void*类型。然后进行比较。

不同类型的两个指针不能进行比较,除非:

- 一个类型是从另一类型派生而来的类类型。
- 至少一个指针被显式地转换(强制)为 void*类型(另一个指针隐式地转换为 void *类型)。

指向相同类型的相同对象的两个指针保证比较结果为相等。如果一个对象的两个非静态成员指针进行比较,使用以下规则:

- 如果类类型不是联合,而且如果两个成员不是由一个访问指示符如公共的、保护的和私有的分开,在后面说明的成员指针将比前面说

明的成员的指针大些(有关访问指示符见第 10 章“成员访问控制”中的“访问指定符”的语法部分)。

- 如果两个成员被访问指示符分开,则结果是不确定的。
- 如果类类型是一个联合,在这个联合中的不同数据的指针相比较是相等的。

如果两个指针指向同一数组中的元素或指向的某个元素超出了数组的尾界,则带有较高下标的对象的指针相比高些。指针的比较仅当指针指向同一数组对象或指向超过数组边界的某个位置时才确保是有效的,。

相等运算符

双目相等运算符比较其操作数为严格相等或不等。

语法

相等表达式:

关系表达式

相等表达式 == 关系表达式

相等表达式 != 关系表达式

相等运算符:等于(==)和不等于(!=)比关系运算符优先级低,但它们的行为是相似的。

如果两个操作数有相同的值则相等运算符(==)返回真值;否则返回假值。不等运算符(!=)当两个操作数具有不同的值时返回真值,否则返回假值。

相等运算符可以比较两个同类型成员指针,在这种比较中,执行成员指针的转换如第 3 章“标准转换”中的“成员指针转换”讨论的。成员指针还可以与等于

0 的常量表达式进行比较。

按位运算符

按位运算符有：

- 按位与(&)
- 按位异或(^)
- 按位或(|)

这些运算符返回其操作数的按位组合。

按位与运算符

按位与运算符(&)返回两个操作数的按位与结果。左右操作数中所有均为 1 的位在结果中为 1,而任一个操作数中为 0 的位在结果中为 0。

语法

与表达式：

关系表达式

与表达式 & 相等表达式

按位与运算符的两个操作数都必须为整型,包含在第 3 章“标准转换”中的“算术转换”中的常用算术转换被用于操作数上。

按位异或运算符

按位异或运算符(^)返回两个操作数的按位异或结果。在左右之一操作数中为 1 的所有位但不都是 1 的位,出现在结果中为 1;两个操作数中相同值的位(为 1 或

0)则在结果中为 0。

语法

异或表达式:

    与表达式

    异或表达式 ^ 与表达式

按位异或运算符的两个操作数必须为整型,包含在第 3 章“标准转换”中“算术转换”的常用算术转换被用于操作数上。

按位或运算符

按位或运算符返回两个操作数按位或的结果,在左或右操作数中为 1 的所有位在结果中均为 1,在两个操作数中均为 0 的位在结果中为 0。

语法

或表达式:

    异或表达式

    或表达式 | 异或表达式

按位或运算符的两个操作数都必须为整型,包含在第三章“标准转换”中“算术转换”的常用算术转换被用于操作数上。

逻辑运算符

逻辑运算符:逻辑与(&&)逻辑或(||)被用于组合使用关系或相等表达式形成的多个条件。

逻辑与运算符

逻辑与运算符在两个操作数均为非 0 时返回整型值 1;否则返回 0。逻辑与具有从左到右的结合律。

语法

逻辑与表达式:

或表达式

逻辑与表达式 && 或表达式

逻辑与运算符的操作数不要求为同一类型,但它们必须为整型或指针类型,操作数通常为关系或相等表达式。

在继续计算逻辑与表达式的值之前,第一个操作数被完全求值,且所有的副作用也被完成。

第二个操作数仅在第一个操作数被求值为真(非 0)时才被计算求值,这种求值去掉了第二个操作数在逻辑与表达式为假时不必要的求值,你可以用这种短路计算防止空指针的间接引用,正如以下例中所示:

```
char *pch=0
...
(pch) && (*pch='a');
```

如果 pch 为空(0),表达式的右端永远不被求值,因此,通过空指针的赋值是不可能的。

逻辑或运算符

逻辑或运算符在任意一个操作数为非 0 时返回整数值 1;否则返回 0,逻辑或具有

从左到右的结合律。

语法

逻辑或表达式：

    逻辑与表达式

    逻辑或表达式 || 逻辑与表达式

逻辑或运算符的操作数不要求为同一类型，但必须为整型或指针类型，操作数通常为关系或相等表达式。

在继续计算逻辑或表达式之前第一个操作数被完全求值，且所有的副作用被完成。

第二个操作数仅在第一个操作数被求值为假(0)时才被计算求值，这种求值去除了第二个操作数在逻辑或表达式为真时不必要的求值。

```
printf("%d",(x==w || x==y || x==z));
```

在此例中，如果 x 等于 w、y 或 z 中的任一值，printf 函数的第二个变量求值为真，且打印值 1。否则，它求值为假，且打印值 0，只要有一个条件求值为真，求值即终止。

赋值运算符

赋值运算符在由左操作数指定的对象中存入一个值。有两种赋值操作：“简单赋值”即将第二个操作数的值存入到由第一个操作数指定的对象中；“复合赋值”即在存储结果之前执行一个算术、移位或按位操作。表 4.4 中除了=运算符外所有的赋值运算符都是复合赋值运算符。

表 4.4　赋值运算符

运算符	含义
=	将第二个操作数的值存入第一个操作数指定的对象中(“简单赋值”)
*=	将第一个操作数值乘以第二个操作数值,将结果存入第一个操作数指定　的对象中
/=	将第一个操作数值除以第二个操作数值,将结果存入第一个操作数指定的对象中
%=	第一个操作数除以第二个操作数的模数,将结果存入第一个操作数指定　的对象中
+=	将第二个操作数的值加入第一个操作数的值中,将结果存入第一个操作数指定的对象中
-=	从第一个操作数值中减去第二个操作数的值,将结果存入第一个操作数指定的对象中
<<=	将第一个操作数的值左移第二个操作数值指定的位数,将结果存入第一个操作数指定的对象中
>>=	将第一个操作数的值右移第二个操作数的值指定的位数,将结果存入第一个操作数指定的对象中
&=	获取第一和第二个操作数按位与的结果,将结果存入第一个操作数指定的对象中
^=	获取第一和第二个操作数按位异或的结果,将结果存入第一个操作数指定的对象中

续表

|=	获取第一和第二个操作数按位或的结果,将结果存入第一个操作数指定的对象中

语法

赋值表达式:

条件表达式

单目表达式 赋值运算符 赋值表达式

赋值运算符:以下之一

= *= /= %= += -= <<= >>= &= ^= |=

赋值运算符的结果

赋值运算符返回由左操作数指定的对象赋值后的值。结果类型为左操作数类型。赋值表达式的结果总是一个 l 值。这些运算符具有从右到左的结合律。左操作数必须是一个可修改的 l 值。

注意:在 ANSI C 中,赋值表达式的结果不是一个 l 值,因此,C++的合法表达式 (a+=b)+=c 在 C 中是非法的。

简单赋值

简单赋值运算符(=)将第二个操作数的值存入第一个操作数指定的对象中。如果两个对象都是算术类型,在值存储前右操作数被转换为左操作数的类型。

const 和 volatile 类型对象可以赋给 volatile 或既不是 const 又不是 volatile 类型的 l 值。

对类类型(struct、union 及 class 类型)对象的赋值由命名为运算符=的函数来执行。此运算符函数的缺省行为是执行按位拷贝;但此行为可用重载的运算符进行修改(有关更多的信息参见第 12 章“重载”中的“重载的运算符”)。
任何从一给定基类非模糊地派生的类的对象可被赋给基类的一个对象。反之则不行,因为从派生类到基类存在隐式转换,但从基类到派生类没有。例如:

```
#include <iostream.h>

class ABase
{
public:
   ABase() { cout << "constructing ABase\n"; }
};-+

class ADerived:public ABase
{
public:
   ADerived() { cout << "constructing ADerived\n"; }
};

void main()
{
    ABase aBase;
```

```
    ADerived aDerived;

    aBase=aDerived;  //可行
    aDerived=aBase;  //错误
}
```

引用类型的赋值就好象正在对引用指向的对象进行赋值一样。

对类类型对象,赋值与初始化不同。为了说明赋值和初始化如何不同,考虑以下代码:

```
userType1 A;
UserType2 B=A;
```

上面代码给出了一个初始化器,它为 UserType1 调用构造函数,该构造函数是有一个 UserType1 类型的一个参量。给出代码:

```
UserType1 A;
UserType2 B;

B=A;
```

赋值语句:

```
B=A;
```

可以有以下的一个作用:

- 为 UserType2 调用函数运算符=,提供的运算符=用一个 UserType1 参量提供的。
- 调用显式转换函数 UserType1::operator UserType2,如果此函数存

在的话。

- 调用一个构造函数 UserType2::UserType2,假如这样的构造函数存在,而且带一个 UserType1 参量,并拷贝结果。

复合赋值

表 4.4 中给出的复合赋值运算符指定为 *e1 op=e2* 形式,其中 *e1* 是一个非常量类型的可修改的 l 值,*e2* 是以下之一:

- 一个算术类型
- 一个指针,如果 *op* 是+或-

e1 op=e2 形式与 *e1=e1 op e2* 执行相同的动作,但 e1 仅被求值一次。
对枚举类型的复合赋值将产生一个错误信息。如果左操作数是一个指针类型,右操作数必须为指针类型或必须为等于 0 的常量表达式;如果左操作数是一个整型,右操作数一定不能为指针类型。

逗号运算符

逗号运算符允许对两个语句分组,其中一个是预期的。
语法
表达式:
    赋值表达式
    表达式,赋值表达式
逗号运算符具有从左到右的结合律。由逗号分开的两个表达式从左到右求值。左操作数总是被求值的,而且所有副作用在右操作数求值之前完成。

考虑表达式：

e1,e2

该表达式的类型和值是 *e2* 的类型和值，*e1* 求值的结果被丢弃。如果右操作数是一个 l 值则结果是一个 l 值。

逗号具有特殊含义时(例如,在函数的实参中或集合初始化器中),逗号运算符及其操作数必须包括在括号中。因此,以下函数调用不是等价的：

```
//说明函数
void Func(int ,int);
void Func(int);

Func(arg1,arg2);     //调用 Func(int ,int)
Func((arg1,arg2));   //调用 Func(int)
```

该例子说明逗号运算符：

```
for (i=j=1;i+j<20;i+=i,j--);
```

在此例中,for 语句的第三个表达式的每个操作数被独立地求值,左操作数 i+=i 首先被求值;然后右操作数 j--被求值。

```
func_one(x,y+2,z);
func_two((x--,y+2),z);
```

在 func_one 函数调用中,由逗号分开的三个参量被传递:x,y+2 和 z;在 func_two 函数调用中,括号迫使编译器将第一个逗号解释为序列求值运算符。此函数调用向 func_two 传递两个参量,第一个参量是序列求值操作(x--,y+2)的结果,具有表达式 y+2 的值和类型;第二个参量是 z。

带条件运算符的表达式

条件运算符(?:)是一个三目运算符(它带三个操作数)。条件运算符的工作如下:

- 在继续之前,第一个操作数被求值,且所有的副作用被完成。
- 如果第一个操作数求值为真(一个非 0 值),第二个操作数被求值。
- 如果第一个操作数求值为假(0),第三个操作数被求值。

条件运算符的结果是被求值的操作数,即第二或第三个的结果。在条件表达式中后两个操作数只有一个被求值。

语法

条件表达式:

逻辑或表达式

逻辑或表达式 ? 表达式 : 条件表达式

条件表达式无结合律。第一个操作数必须为整型或指针类型,以下规则用于第二及第二个操作数:

- 如果两个表达式是同类型的,则结果是那个类型的。
- 如果两个表达式都是算术类型,则常用的算术转换(包含在第 3 章“标准转换”中的“算术转换”)被执行以将它们转换为公用类型。
- 如果两个表达式都是指针类型或一个是指针类型,另一个是求值为 0 的常量表达式,则指针转换被执行去将它们转换为公用类型。
- 如果两个表达式都是引用类型,则引用转换被执行以将它们转换为公用类型。
- 如果两个表达式都是 void 类型,则公用类型为 void 类型。

- 如果两个表达式是一个给定类类型,则公用类型是那个类类型。

任何未列入前面清单的第二和第三个操作数的组合都是非法的。结果的类型是公用类型,且如果第二和第三个操作数为同一类型且都是 l 值的则结果是一个 l 值。

例如:

```
(val>=0) ? val : -val
```

如果条件为真,表达式等于 val;否则,表达式等于-val。

带量表达式

C++在以下说明中要求常量表达式,即等于常量的表达式:

- 数组界限
- case 语句中的选择器
- 位域长度规格
- 枚举初始化器

语法

常量表达式:

条件表达式

在常量表达式中唯一合法的操作数是:

- 文字
- 枚举常量
- 用常量表达式初始化的说明为常量的值
- sizeof 表达式

在常量表达式中非整型常量必须转换(显式或隐式地)为合法的整型。因此,以下代码是合法的:

```
const double Size=11.0;
char chArray[(int)Size];
```

在常量表达式中显式地转换为整型是合法的,除了用作 sizeof 运算符的操作数之外。所有其它类型或派生类型都是非法的。

逗号运算符和赋值运算符不能用于常量表达式中。

带显式类型转换的表达式

正如第 3 章“标准转换”中描述的,C++提供隐式类型转换。当你需要更精确地控制应用的转换时,你还可以指定显式类型转换。

显式类型转换运算符

C++允许使用与函数调用语法类似的语法进行显式类型转换,一个简单类型名称后面跟随一个括在括号中的表达式表,构成用指定表达式指定的类型的一个对象。

以下例子给出了一个转到 int 类型的显式类型转换:

```
int i=int(d);
```

以下例子使用定义在“函数调用结果”中的 Point 类的修改后的版本:

```
#include <iostream.h>

class Point
{
```

```
public:
      //定义缺省构造函数
      Point() {_x = _y = 0;}
      //定义另一个构造函数
      Point(int X,int Y) {_x=X;_y=Y;}

      //定义作为引用类型的“accessor”函数
      unsigned& x() { return _x;}
      unsigned& y() { return _y;}
      void Show() { cout << "x=" << _x << ","
                  << "y=" << _y << "\n";}
private:
      unsigned _x;
      unsigned _y;
};

void main ()
{
     Point Point1,Point2;

     //赋给 point1 显示转换为(10,10)
     Point1=Point(10,10);
```

```
    //通过对 unsigned 类型赋一个 20 的显示转换使 x()作为一个 l 值来用
    Point1.x()=unsigned(20);
    Point1.Show();

     //赋给 Point2 缺省的 point 对象
     Point2=Point();
     Point2.Show();
}
```

此程序的输出为：

```
x=20,y=10
x=0,y=0
```

尽管前面的例子阐述的是使用常量的显式类型转换，在对象上用同样的技巧去执行这些转换。以下代码段说明这种情况：

```
int i=7;
float d:

d=float(i);
```

显示类型转换还可以用“造型转换”语法指定，前面的例子，用造型转换语法重写为：

```
d=(float)i;
```

当从单值转换时，造型转换和函数形式转换具有相同的结果。但在函数形式语法中，你可以为转换指定多个参量。这个不同点对于用户定义的类型是很重要的。

考虑一个 Point 类及其转换：

```
struct Point
{
      Point(short x, short y) { _x=x,_y=y; }
      ...
      short _x,_y;
};
       ...
Point pt=Point(3,10);
```

前面的例子，使用了函数形式转换，显示了如何将两个值(一个为 x，一个为 y)转换为用户定义的类型 Point。

重要点：使用显式类型转换要小心，因为它们覆盖了 C++编译器的内部类型检查。

语法

造型转换表达式：

单目表达式

(类型名)造型转换表达式

造型标记必须用于不具有一个简单类型名称(例如指针或引用类型)的类型的转换。可以用一个简单类型名称表示的类型的转换可以用任何两者之一形式书写。有关哪些构成一个简单类型名称的更多信息，参见第 6 章“说明”中的“类型指示符”。

在造型转换内的类型定义是非法的。

合法转换

如果转换可以用标准转换实现,则你可以从一给定类型向另一类型做显式转换,其结果是相同的。本节描述的转换是合法的,任何其它的非显式地由用户定义的转换(为一个类类型)是非法的。

如果指针足够大以便保存整型值,则一个整型值可被显式地转换为一个指针,转换到一个整型值的指针可以再转回到指针,其值是相同的。这个一致性是由以下等式给出(其中 *p* 表示任意类型的一个指针):

p==(*type* *)整型转换(*p*)

用显式转换,编译器不检查被转换的值是否适合新类型,但从指针到整型的转换除外,反之亦然。

本节描述以下转换:

- 转换指针类型
- 转换空指针
- 转换到一个前向引用类类型
- 转换到引用类型
- 在成员指针类型间转换

转换指针类型

一个对象类型的指针可被显式地转换到另一个对象类型的指针,被说明为 void * 的指针被认为是指向任何对象类型的指针。

一个基类指针可以显式地转换到一个派生类的指针,只要以下条件被满足:

- 有一个非模糊的转换。

- 在任意点,基类都未被说明为虚拟的。

由于到 void *类型的转换可以改变一个对象的表示,所以不保证转换 *type1** void* *type2** 等价于转换 *type1* type2**(仅是值的改变)。
当执行这样的转换时,结果是指向表示基类的源对象的子对象的一个指针。
关于模糊的和虚拟的基类更多信息参见第 9 章“派生的类”。
C++允许对象或函数指针显式转换到 void *类型。
如果函数指针类型有足够的位容纳对象指针类型,则对象指针类型可显式地转换到函数指针。
常量对象的指针可显式地转换到非常量类型的指针,此转换的结果指向源对象。
常量类型对象或常量类型对象的引用可造型转换到非常量类型的一个引用。结果是源对象的一个引用。源对象很可能说明为常量,因为它要在程序持续期间保留常量。因此,一个显式转换导致这种安全保护失败,而允许对这种对象的修改,在这种情况中的动作是不确定的。
volatile 类型对象的指针可以造型转换到非 volatile 类型的指针,这种转换的结果指的是源对象。类似地,volatile 类型对象可以造型转换为非 volatile 类型的引用。

转换空指针

空指针(0)被转换到它自己。

转换到一个前向引用类类型

一个类已被说明但尚未定义(一个前向引用)可被用于指针造型转换,在这种情况

下,如果类的关系已知,编译器返回一个源对象指针,而不是可能的子对象的指针。

转换到引用类型

任何其地址可被转换到一个给定指针类型的对象,也可以转换到模拟的引用类型。例如,任何其地址可被转换到类型 char *的对象也可转换到类型 char &。没有构造函数或类转换函数被调用以产生一个到引用类型的转换。
对象或值可被转换到类类型对象,仅当为此目的已特别提供了一个构造函数或转换运算符。关于这些用户定义的函数的更多信息参见第 11 章“特殊成员函数”中的“转换构造函数”。
一个基类的引用到一个派生类的引用的转换(或反之亦然)与指针的作法相同。
到引用类型的造型转换结果为一个 l 值,造型转换到其它类型的结果不是 l 值。
在指针或引用造型转换的结果上执行的操作仍在源对象上执行。

在成员指针类型之间转换

一个成员指针可以按以下规则转换为不同的成员指针类型:两个指针必须是同一个类的成员指针或必须是两个类的成员指针,且其中之一非模糊地从另一个派生而来。当转换成员指针函数时,返回的与参量的类型必须匹配。

带成员指针运算符的表达式

成员指针运算符.*和->*返回表达式左侧指定的对象的特定的类成员的值,以下例子显示了如何使用这些运算符:

```
#include <iostream.h>

class Window
{
public:
     void Paint();    //使窗口重画
     int WindowsId;
};

//定义派生的类型 pmfnPaint 和 pmWindowId
//这些类型分别是 Paint()和 WindowId 成员指针
void (Window::*pmfnPaint)()=&Window::Paint;
int Window::*pmWindowId=&Window::WindowId;

void main()
{
    Window AWindow;
    Window *pWindow=new Window;
//正常调用 Paint 函数,然后使用成员指针
    AWindow.Paint();
    (AWindow.*pmfnPaint)();
    pWindow ->Paint();
```

```
    (pWindow ->*pmfnPaint)();    //由于*没有函数调用联结紧密,所以需要括号

    int Id;
    //检索窗口编号
    Id=AWindow.*pmWindowId;
    Id=pWindow->*pmWindowId;
}
```

在上面例子中,pmfnPaint 的成员指针被用于调用成员函数 Paint。另一个成员 pmWindowId 的指针被用于访问 WindowId 成员。

语法

pm-表达式:

造型转换表达式

pm-表达式.*造型转换表达式

pm-表达式->*造型转换表达式

双目运算符.*将其第一个操作数(即必须为一个类类型对象)和第二操作数(即必须为成员指针类型)组合在一起。

双目运算符->*将其第一个操作数(即必须为类类型对象的指针)和第二个操作数(即必须为成员指针类型)组合在一起。

在一个包含.*运算符的表达式中,第一个操作数必须为在第二个操作数中指定的成员指针的类类型,且可访问该指针或非模糊地从那个类派生且可访问的一个可访问类型。

在一个包含->*运算符的表达式中,第一个操作数必须为第二个操作数指定类型的“类类型指针”类型,或必须为从那个类非模糊派生出的一个类型。
考虑以下类和程序段:

```
class BaseClass
{
publc:
    BaseClass();  //基类构造函数
    void Funcl();
};

//说明成员函数 Func1 的一个指针
void (BaseClass::*pmfnFunc1)()=&BaseClass::Func1;

class Derived : public BaseClass
{
public:
    Derived();   //派生类构造函数
    void Func2();
};

//说明成员函数 Func2 的一个指针
void (Derived::*pmfnFunc2)()=&Derived::Func2;
```

```
void main()
{
    BaseClass ABase;
    Derived ADerived;

    (ABase.*pmfnFunc1)();     //可行:为 Baseclass 定义
    (ABase.*pmfnFunc2)();     //错误:不能用基类去访问派生类成员指针
    (ADerived.*pmfnFunc1)(); //可行:Derived 是从 BaseClass 非模糊地派
生的
    (ADerived.*pmfnFunc2)(); //可行:为 Derived 定义
}
```

.*或->*成员指针运算符的结果是成员指针说明中指定的类型的一个对象或函数。因此,在前面的例子中,ADerived.*pmfnFunc1()表达式的结果是返回 void 的函数的一个指针。如果第二个操作数值为 1,则此结果也值为 1。

注意:如果成员指针运算符之一的结果是一个函数,那么结果仅能被用作函数调用运算符的一个操作数。

表达式的语义

本节解释表达式在何时以及以什么样的顺序被求值,包括以下主题:

- 求值的顺序

- 顺序点
- 模糊表达式
- 表达式中的表示

求值的顺序

本节讨论表达式被求值的顺序,但不解释这些表达式中运算符的语法或语义,本章较前面的章节提供了这些运算符每个的完整参考。
表达式按照其运算符的优先级及分组情况被求值(第 1 章“词法规定”中的表 1.1 给出了施加于表达式上的 C++运算符的关系)。考虑这个例子:

```
#include <iostream.h>

void main()
{
    int a=2,b=4,c=9;

    cout << a + b * c << "\n";
    cout << a + (b * c) << "\n";
    cout << (a + b) * c<< "\n";
}
```

前面代码的输出为:

```
38
38
```

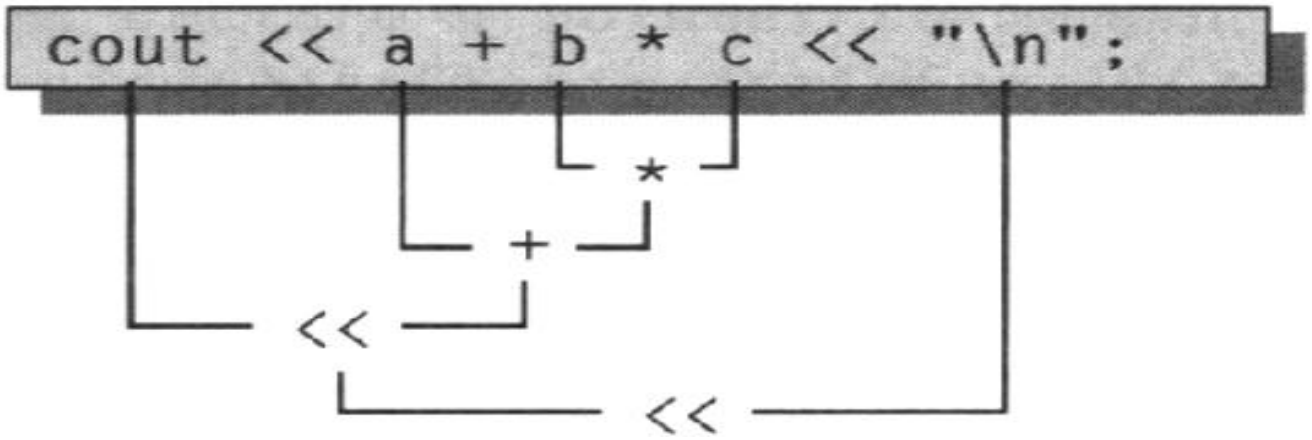


图 4.1　表达式求值顺序

图 4.1 中给出的表达式求值顺序由运算符的优先级和结合律确定：

1. 在这个表达式中乘法(*)具有最高优先级,因此,b*c 子表达式首先被求值。
2. 加法(+)具有次级优先级,所以 a 被加到 b 和 c 的积中。
3. 左移(<<)在表达式中具有最低优先级,但有两次出现,由于左移运算符从左到右分组,左子表达式首先被求值,而后是右子表达式。

当括号被用于分组子表达式时,它们改变表达式被求值的优先级以及求值顺序,如图 4.2 所示。

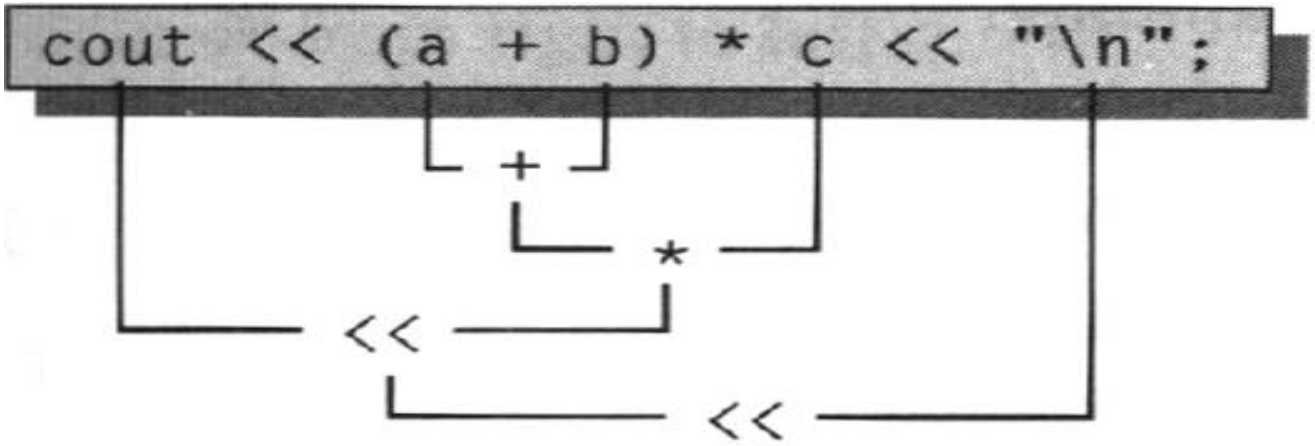


图 4.2 带括号的表达式求值顺序

像图 4.2 中那些表达式纯粹是为了副作用而求值,在这种情况下,将信息转换到标准输出设备。

注意:左移运算符被用于向一个类输出流对象中插入一个对象。当与输入输出流一起使用时,有时它被称为“插入”运算符。有关输入输出流库的更多内容,参见“Microsoft Visual C++6.0 参考库”的“Micosoft Visual C++6.0 运行库参考”卷。

顺序点

一个表达式在连续的“顺序点”之间只能改变一个对象的值一次。

Microsoft 特殊处→

C++语言定义现在不指定顺序点。对任何包含 C 运算符和不包含重载的运算符的表达式,Microsoft C++使用与 ANSI C 相同的顺序点。当运算符被重载时,语义从运算符顺序变为函数调用顺序。Microsoft C++使用以下顺序点:

- 逻辑与运算符(&&)的左操作数,在继续之前,逻辑与运算符的左操作数被完全求值,且所有的副作用被完成。不保证逻辑与运算符的右操作数将被求值。
- 逻辑或运算符(||)的左操作数,在继续之前,逻辑或运算符的左操作数被完全求值,且所有的副作用被完成。不保证逻辑或运算符的右操作数将被求值。
- 逗号运算符的左操作数,在继续之前,逗号运算符的左操作数被完全求值,且所有的副作用被完成。逗号运算符的两个操作数总是被求值的。
- 函数调用运算符,一个函数的函数调用表达式和所有的参量,包括缺省参量,在进入函数之前被求值,且所有的副作用被完成。在参量或函数调用表达式之间没有特别指定求值的顺序。
- 条件运算符的第一个操作数,在继续之前,条件运算符的第一个操作数被完全求值,且所有的副作用被完成。
- 一个完整初始化表达式的结束,如在说明语句中一个初始化的结束。
- 在表达式语句中的表达式,表达式语句由可选的表达式后跟一个分号(;)组成。表达式为其副作用而被完全求值。
- 在一个选择(if 或 switch)语句中的控制表达式,在依赖于这样的代码被执行之前,表达式被完全求值,且所有副作用被完成。
- 在一个 while 或 do 语句中的控制表达式。在 while 或 do 循环的下一次重复的任何语句执行之前,表达式被完全求值,且所有副作用被

完成。

- for 语句三个表达式中的每一个,在进入下一个表达式之前,每一个表达式被完全求值,且所有副作用被完成。
- 在返回语句中的表达式,在控制返回到调用函数之前,表达式被完全求值,且所有副作用被完成。

Microsoft 特殊处结束

模糊的表达式

某些表达式在含义上是模糊的,当同一个表达式中一个对象的值被不止一次地修改时,这些表达式最频繁地出现。这些表达式依赖于一个特定的求值顺序,而该语言中却没有定义一种求值顺序,考虑以下例子:

```
int i=7;

func(i,++i);
```

C++语言不保证在函数调用中参量的求值顺序。因此,在上例中,func 可能为其参量接收 7 和 8 或者 8 和 8,这取决于参量是从左到右还是从右到左求值的。

表达式中的表示法

C++语言在指定操作数时指定某些兼容性,表 4.5 给出了要求操作数为 type 类型的运算符可接受的操作数的类型。

表 4.5　运算符可接受的操作数类型

期望的类型	允许的类型
type	**const** *type*
	volatile *type*
	type &
	const *type &*
	volatile *type &*
	volatile const *type*
	volatile const *type &*
*type**	*type** **coust**
	*type** **volatile**
	*type** **volatile const**
const *type*	*type*
	const *type*
	const *type &*
volatile *type*	*type*
	volatile *type*
	volatile *type &*

由于前面的规则总是可以用于组合中，在期望一个指针的地方可以提供一个常量指针指向一个 volatile 对象。

造型转换

C++语言规定:如果一个从基类派生的类包含虚函数,则指向那个基类类型的指针可用于调用存在于派生类对象中的该虚函数的实现。包含虚函数的类有时被称为一个“多态类”。
由于一个派生类完全包含它所从派生的所有基类的定义,所以可安全地造型一个指针向上到类层次指向这些基类的任意一个。给定一个指向基类的指针,按层次向下造型指针可以是安全的,如果正被指向的对象实际是从基类派生的一个类型,则是安全的。在这种情况下,实际对象被说是“完整的对象”,基类指针被说是指向完整对象的“子对象”。例如,考虑图 4.3 中给出的类层次。

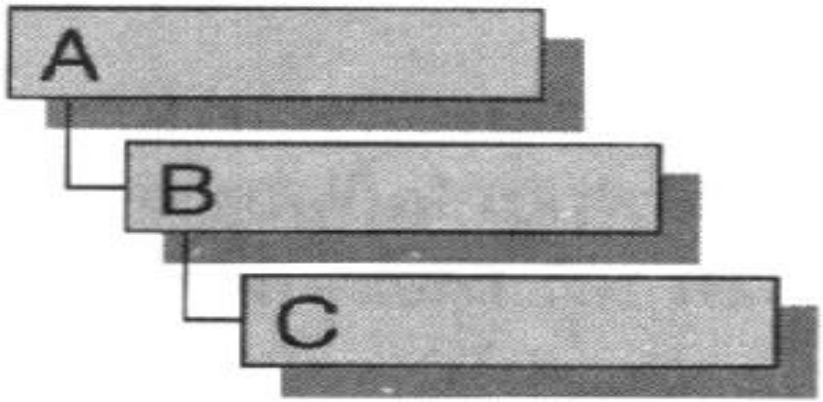


图 4.3　类层次

一个类型 C 的对象,如图 4.4 中所示被可视化。

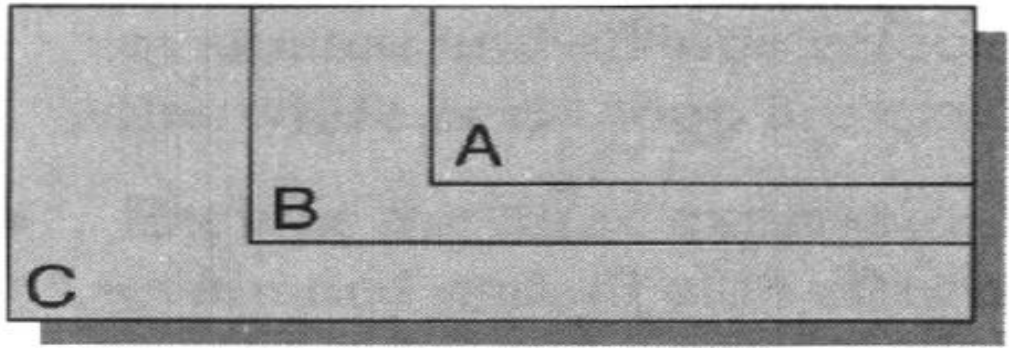


图 4.4　具有 B 子对象和 A 子对象的类 C

给定一个类 C 的实例,有一个 B 子对象和一个 A 子对象。包括 A 和 B 子对象的 C 的实例是“完整的对象”。

使用运行类型信息,可以检查一个指针是否真的指向一个完整对象,而且是否可以安全地造型指向其层次中另一个对象。dynamic_cast 运算符可用于完成这些

类型的造型转换,它还执行必要的运行检查以便使操作安全。

造型运算符

有几个造型运算符是 C++语言特定的。这些运算符旨在除去旧的 C 语言造型形式中模糊性和危险的继承。这些运算符是:

- dynamic_cast 用于多态类型的转换
- static_cast 用于非多态类型的转换
- const_cast 用于除去 const, volatile 和 __unaligned 属性
- reinterpret_cast 用于位的简单再解释

使用 const_cast 和 reinterpret_cast 作为一种最后的手段,因为这些运算符与旧的造型形式有相同的危险性。但是它们还是有必要用于完全取代旧的造型形式。

dynamic_cast 运算符

表达式 dynamic_cast<*type_id*>(expression)将 *expression* 操作数转换为一个 *type-id* 类型的对象,*type-id* 必须为以前定义的类类型或“void 指针”的一个指针或一个引用。如果 *type-id* 是一个指针,则 *expression* 的类型必须为一个指针或如果 *type-id* 是一个引用,则它为一个 l 值。

语法

dynamic_cast<*type_id*>(*expression*)

如果 *type-id* 是一个指向 *expression* 的非模糊可访问的直接或间接基类的指针,则

结果是 *type-id* 类型的唯一子对象的一个指针,例如:

```
class B{...};
class C:public B {...};
class D:public C {...};

void f(D* pd)
{
    C* pc=dynamic_cast<C*>(pd); //可行:C 是一个直接基类,pc 指向 pd 的 C 子对象
    B* pb=dynamic_cast<B*>(pd); //可行:B 是一个间接基类,pd 指向 pd 的 C 子对象
    ...
}
```

这种类型转换被称为向上造型,因为它使指针按类层次向上移,从一个派生的类到一个派生源类。向上造型是一种隐式转换。

如果 *type-id* 是 void*,则执行一种运行检查以确定 *expression* 的实际类型,结果是由 *expression* 指向的完整对象的一个指针。例如:

```
class A{...};

class B{...};

void f()
```

```
{
    A* pa=new A;
    B* pb=new B;
    void* pv=dynamic_cast<void*>(pa);
    //pv 现在指向类型 A 的一个对象
    ...
    pv=dynamic_cast<void*>(pb);
    //pv 现在指向类型 B 的一个对象
}
```

如果 *type-id* 不是 void*,则执行运行检查去查看由 *expression* 指向的对象是否可转换为 *type-id* 指向的类型。

如果 *expression* 的类型是 *type-id* 类型的一个基类,则执行运行检查去查看 *expression* 是否确实指向 *type-id* 类型的一个完整对象,如果是真的,则结果是 *type-id* 类型的一个完整对象的指针。例如:

```
class B{...};
class D : public B{...};

void f()
{
    B* pb=new D;//不清楚但是可行
    B* pb2=new B;
```

```
    D* pd=dynamic_cast<D*>(pb); //可行:pb 确实指向一个 D
    ...
    D* pd2=dynamic_cast<D*>(pb2); //错误:pb2 指向一个 B,而不是一个
D,pd2==NULL
    ...
}
```

这种类型转换被称为向下造型，因为它使指针按类层次向下移,从一个给定类到一个从它派生出的类。

在多重继承的情况下,模糊性的可能性被引入。可虑图 4.5 中给出的类层次:

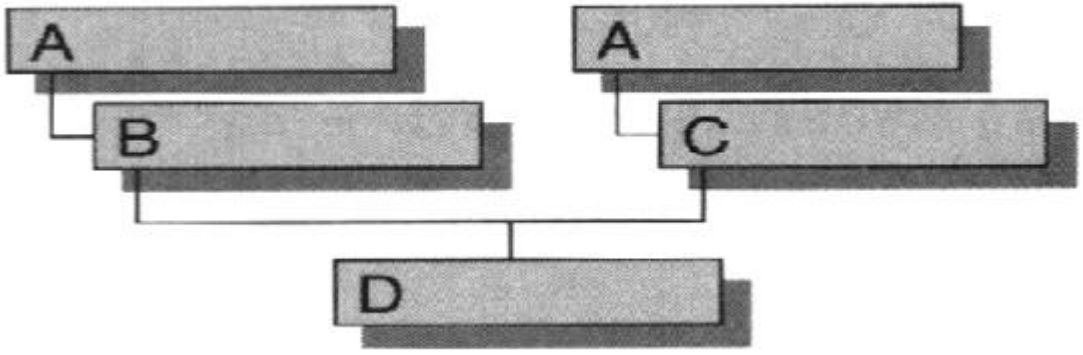


图 4.5　类层次显示多重继承

类型 D 的一个对象的指针可被安全地造型到 B 或 C。但是如果 D 被造型到指向一个 A 对象,结果将是 A 的哪个实例呢?这将导致一个模糊造型错误。为了克服这个问题,你可以执行两个非模糊的造型。例如:

```
void f()
{
```

```
    D* pd=new D;
    A* pa=dynamic_cast<A*>(pd);   //错误:模糊的
    B* pb=dynamic_cast<B*>(pd);   //首行造型到 B
    A* pa2=dynamic_cast<A*>(pb); //可行:非模糊的
}
```

当你使用虚拟基类时还会引入更进一步的模糊性。考虑图 4.6 所示的类层次:

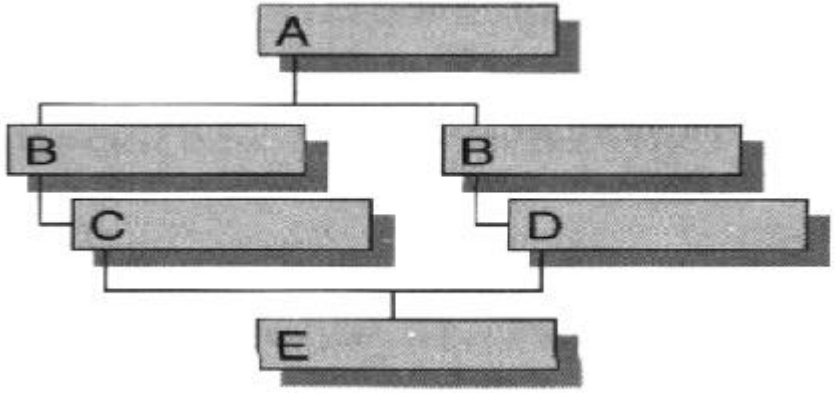


图 4.6　类层次显示虚拟基类

在此层次中,A 是一个虚拟基类,关于一个虚拟基类的定义见第 9 章“派生类”中的“虚拟基类”。给定 E 类的一个实例及指向 A 子对象的一个指针,转到指向 B 的指针的 dynamic_cast 将因为模糊性而失败。你必须先造型回到完整的 E 对象,然后沿层次向上,以一种非模糊的方式到达正确的 B 对象。

考虑图 4.7 所示的类层次。

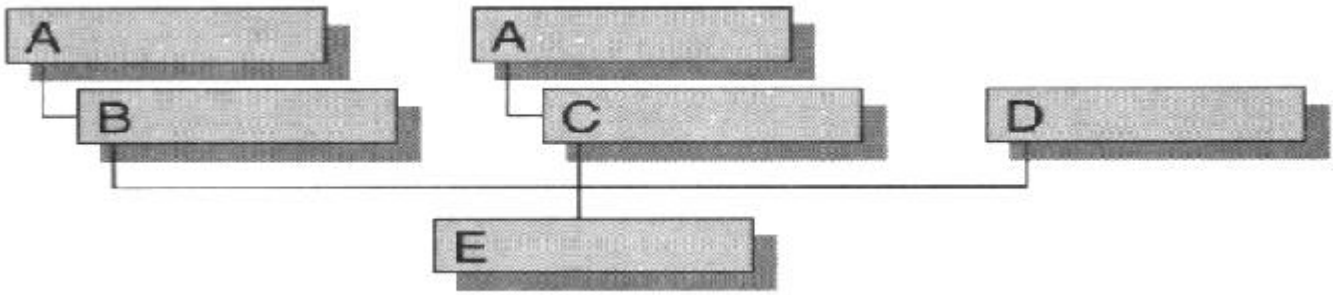


图 4.7　类层次显示重复基类

给定类型 E 的一个对象和 D 子对象的一个指针,从 D 子对象导航到最左边的 A 子对象,可有三种转换。你可以执行一个 dynamic_cast 转换从 D 指针转到 E 指针,然后一个转换(为 dynamic_cast 或隐式转换)从 E 到 B,最后一个隐式转换从 B 到 A。例如:

```
void f(D* pd)
{
E* pe=dynamic_cast<B*>(pd);
B* pb=pe      //向上造型,隐式转换
    A* pa=pb;    //向上造型,隐式转换
}
```

dynamic_cast 运算符还可以用于执行“跨越造型”。使用相同的类层次,例如可以造型一个指针从 B 子对象到 D 子对象,只要完整的对象是 E 类型的。

考虑跨越造型,确实有可能仅在两步内做转换从 D 指针到最左边的 A 子对象的指针。你可以从 D 到 B 造型转换,然后一个隐式转换从 B 到 A。例如:

```
void f(D* pd)
```

```
{
    B* pb=dynamic_cast<B*>(pd);  //跨越造型转换
    A* pa=pb;//向上造型,隐式转换
}
```

一个空指针值被 dynamic_cast 转换为目标类型的空指针值。

当你使用 dynamic_cast<*type_id*>(*expression*)时,如果 *expression* 不能被完全地转换到 *type_id* 类型,则运行检查会导致造型转换失败。例如:

```
class A{...};

class B{...};

void f()
{
    A* pb=new A;
    B* pb=dynamic_cast<B*>(pa);  //失败,不安全;B 不是从 A 派生来的
    ...
}
```

到指针类型的一个失败的造型转换的值是空指针。到引用类型的失败的造型转换丢弃了一个 bad_cast 异常。

bad_cast 异常

dynamic_cast 运算符丢弃了一个 bad_cast 异常,作为到引用类型的一个失败的

造型的值,bad_cast 的接口为:

```
class bad_cast:public logic {
public:
    bad_cast(const __exString& what_arg): logic(what_ary) {}
    void raise() {handle_raise(); throw *this;}
    //虚拟的__exString what() const; //继承的
};
```

static_cast 运算符

static_cast<*type_id*>(*expression*)表达式将 *expression* 转换到独立地基于表达式中给出的类型 type_id 类型。没有运行类型检查被执行以确保转换的安全性。

语法

static_cast<*type_id*>(*expression*)

static_cast 运算符可被用在例如转换一个基类指针到一个派生类指针的操作,这样的转换不总是安全的,例如:

```
class B{...};

class D:public B{...};

void f(B* pb,D* pd)
{
    D* pd2=static_cast<D*>(pb);  //不安全,pb 可能只指向 B
```

```
    B* pb2=static_cast<B*>(pd);  //安全转换
    ...
}
```
与 dynamic_cast 相反,在 pb 的 static_cast 转换上没有作任何运行检查。由 pb 指向的对象可能不是 D 类型的对象,使用*pd2 的情况可能是灾难性的。例如,调用一个函数是 D 类的一个成员但不是 B 类的,可能导致访问冲突。
dynamic_cast 和 static_cast 运算符可在一个类层次中移动一个指针,但 static_cast 仅依赖于造型转换语句中提供的信息,因此是不安全的。例如:
```
class B{...};
class D:public B{...};

void f(B* pb)
{
    D* pd1=dynamic_cast<D*>(pb);
    D* pd2=static_cast<D*>(pb);
}
```
如果 pb 真的指向类型 D 的一个对象,则 pd1 和 pd2 将得到相同的结果。如果 pd==0,则它们也将得到相同的结果。
如果 pb 指向类型 B 的一个对象,但不是指向完整的 D 类,则 dynamic_cast 将足以知道返回 0。但是 static_cast 依赖于程序员的坚持让 pb 指向类型 D 的一个对象而且简单地返回那个假定的 D 对象的指针。

因此,static_cast 可以实现相反的隐式转换,在这种情况下,结果是未定义的,留给程序员去确保 static_cast 转换的结果为安全的。
这些动作还适用于类类型以外的其它类型。例如,static_cast 可被用于从 int 到 char 的转换。但结果字符可能没有足够的位数保持整个整型值,又一次留给程序员去确保 static_cast 转换的结果为安全的。
static_cast 运算符还可被用于执行任何隐式转换,包括标准转换和用户定义的转换。例如:

```
typedef unsigned char BYTE

void f()
{
    char ch;
    int i=65;
    float f=2.5;
    double db1;

    ch=static_cast<char>(i); //从 int 到 char
    db1=static_cast<double>(f);  //从 float 到 double
    ...
    i=static_cast<BYTE>(ch);
    ...
}
```

static_cast 运算符可显式地将一个整型值转换到一个枚举类型。如果整型值未落在枚举值的范围内,枚举结果值是不确定的。
static_cast 运算符将一个空指针值转换到目标类型的空指针值。
任何表达式可用 static_cast 运算符显式地转换到 void 类型,目标 void 类型可有选择地包括 const、volatile 或 __unaligned 属性。
static_cast 运算符不能造型除去 const、volatile 或 __unaligned 属性。有关除去这些属性的信息参见“const_cast 运算符”。

const_cast 运算符

const_cast 运算符可被用于从一个类中除去 const、volatile 和 __unaligned 属性。
语法
const_cast<*type_id*>(*expression*)
任何对象类型的指针或一个数据成员的指针可被显式地转换到完全相同的类型,带 const, volatile 和 __unaligned 限定符除外。对指针和引用,结果将指向源对象,对数据成员指针结果和数据成员的源(非造型)指针一样指向同一成员。由于依赖于引用对象的类型,通过指针、引用或数据成员指针的求结果的写操作将产生未定义的动作。
const_cast 运算符将一个空指针值转换为目标类型的空指针值。

reinterpret_cast 运算符

reinterpret_cast 运算符允许任何指针被转换到任何其它指针类型,它还允许

任何整型转换到任意指针类型,且反之亦然。滥用 reinterpret_cast 运算符可轻易导致不安全,除非是所期望的转换是固有的低等级,否则你应使用其它造型运算符之一。
语法
reinterpret_cast<type_id>(expression)
reinterpret_cast 运算符可被用于像 char*到 int*或 One_class*到 Unrelated*这种内部的非安全的转换。
reinterpret_cast 的结果除了用于造型回到其源类型外,不能安全地用于其它任何情况,其它用法至多也是不可移植的。
reinterpret_cast 运算符不能造型除去 const、volatile 或__unaligned 属性,关于除去这些属性的信息参见 const_cast 运算符。
reinterpret_cast 运算符将一个空指针值转换到目标类型的空指针值。

运行类型信息

运行类型信息(RTTI)是一种机制,允许对象的类型在程序执行期间被确定。RTTI 加到 C++语言中,因为很多类库销售者正自己实现这个功能,这导致库之间的不兼容。因此,很明显需要在语言级支持运行类型信息。
为了明确性,RTTI 的这个讨论几乎完全限制于指针。但是讨论的概念也适用于引用。
运行类型信息有三个主要的 C++语言元素:

- dynamic_cast 运算符,用于多态类型的转换。更多的信息参见本章前面部分“ dynamic_cast 运算符 ”。

- typeid 运算符,用于指定一个对象的确切类型。
- type_info 类,用于保持由 typeid 运算符返回的类型信息。

typeid 运算符

typeid 运算符允许一个对象的类型在运行时被确定。

语法

typeid(*type_id*)

typeid(expression)

一个 typeid 表达式的结果是一个常量 type_info&。其值是一个 type_info 对象的引用。对象表示 type_id 或 expression 的类型,这依赖于 typeid 使用的是哪种形式。更多的信息参见本章后面“type_info 类”。

typeid 运算符在被用于一个多态类类型的 l 值时做运行检查,对象的真正类型不能由所提供的静态信息确定。这些情况有:

- 一个类的引用
- 一个指针,用*间接引用
- 一个下标指针(如[])(注意用带多态类型的指针的下标通常是不安全的)

如果 *expression* 指向一个基类类型,且对象实际上是从基类派生的类型,则结果是派生类的 type_info 引用。*expression* 必须指向一个多态类型,即带虚拟函数的类。否则,结果是在 *expression* 中指出的静态类的 type_info。而且,指针必须为间接引用的,因此它所指向的对象被使用。没有间接引用指针,则结果将是指针的 type_info,而不是它所指向的内容。例如:

```
class Base{...};

class Derived:public Base{...};

void f()
{
    Derived*  pd=new Derived;
    Base*  pb=pd;
    ...
    const type_info& t=typeid(pb);   //t 保持 type_info 指针
    const type_info& t1=typeid(*pb); //t1 保持派生的 info
    ...
}
```

如果 *expression* 间接引用一个指针,而且那个指针的值为 0,typeid 丢弃一个 bad_typeid 异常。如果指针不是指向一个有效的对象,异常__non_rtti_object 被丢弃。

如果 *expression* 既不是对象的基类的一个指针也不是一个引用,则结果是表示 *expression* 的静态类型的一个 type_info 引用。

bad_typeid 异常

在有些情况下,typeid 运算符丢弃一个 bad_typeid 异常,bad_typeid 的接口为:

```
class bad_typeid:public logic{
```

```
public:
    bad_typeid(const char * what_arg):logic(what_arg) {}
    void raise() { handle_raise(); throw * this; }
    //虚拟的__exString what() const;//继承的
};
```

更多的信息参见“typeid运算符”。

type_info 类

type_info 类描述由编译器在程序内产生的类型信息,该类的对象有效地存储一个指向类的名称的指针,type_info 类还存储适合于比较两个类型的相等性或整理顺序的一个编码值。类型的编码规则和整理顺序是未指定的,而且在不同程序间可以有所不同。

要使用 type_info 类必须包含 typeinfo.h 头文件。

```
class type_info {
public:
    virtual ~type_info();
    int operator==(const type_info& rhs) const;
    int operator!=(const type_info& rhs) const;
    int before(const type_info& rhs) const;
    const char* name() const;
    const char* raw_name() const;
private:
```

```
    ...
};
```
==和!=运算符可分别用于比较与其它 type_info 对象是相等还是不相等的。
在类型的整理顺序和继承性关系之间没有联系。在成员函数之前使用 type_info:去指定类型的整理顺序。不保证前面的 type_info::在不同的程序或即使是同一程序的不同运行中产生相同的结果。在这种况下,前面的 type_info::类似于取地址(&)运算符。
type_info::name 成员函数为代表类型的、人可读的名称的 0 终止字符串返回一个 char*常量。被指向的存储器是被高速缓存的,而且永远不应该被直接撤消分配。
type_inf::raw_name 成员函数为表示对象类型装饰名的 0 终止字符串返回一个 char*常量。名称实际上被存在其装饰格式内,以节省空间。因而,此函数比 type_info::name 速度快,因为它不需要非装饰名称。由 type_info::raw_name 函数返回的字符串在比较操作中很有用,但却不是可读的。如果你需要一个人可读的字符串,则用 type_info::name 函数。
只有指定/GR(使能运行类型信息)编译器选项时产生多态类的类型信息。

第 5 章　语　　句

C++语句是程序元素,用于控制对象是如何以及以怎样的顺序操作的。本章包括:

- 概述
- 标号语句
- 表达式语句,这些语句为一个表达式的副作用或其返回值而求值。
- 空语句,这些语句可被提供于 C++语法要求有语句的,但不执行任何动作的地方。
- 复合语句,这些语句是包括于花括号({})中的语句组。它们可被用于语法中任何需要单个语句的地方。
- 选择语句,这些语句执行一个测试,如果测试为真(非 0),则执行某部分代码;如果测试为假,则可执行另一部分代码。
- 迭代语句,这些语句反复执行某一块代码,直到一个指定的终止条件被满足为止。
- 跳转语句,这些语句或者将控制立即转给函数中的另一个地方或者从函数返回控制。
- 说明语句,说明向一个程序中引入名称(第 6 章“说明”对说明提供更详细的信息)。
- 异常处理语句,包括 C++异常处理(try、throw 或 catch)和结构异常处理(__try/__except、__try/__finally)。try_except 语句提供一

个方法在正常终止执行的事件出现时去获取一个程序的控制权;try_finally 和 levave 语句提供一个方法,在一块代码的执行被中断时以保证清除代码的执行。

语句概述

C++语句顺序地执行,除非表达式语句、选择语句、迭代语句或跳转语句特别修改了那个顺序。

语法

语句:

- 标号语句
- 表达式语句
- 复合语句
- 选择语句
- 迭代语句
- 跳转语句
- 说明语句
- *try-throw-catch*

在大多数情况下,C++语句的语法与 ANSI C 是完全相同的,两者之间基本的不同点是:在 C 中,说明仅在块的开始被允许;C++添加了 *declaration-statement*,有效地除去了这个限制。这使你可以在一个用于预计算初始化值的程序中引入变量。在块内说明变量还允许你对这些变量的范围和生存期施以准确的控制。

标号语句

标号语句为将程序的控制直接转给一个给定的语句,该语句必须被标号,参见下节“使用带 goto 语句的标号”和“在 case 语句中使用标号”。

语法

标号语句:

    标识符:语句
    cast 常量表达式:语句
    default:语句

使用带 goto 语句的标号

在源程序中一个标识符标号的出现说明了一个标号,只有一条 goto 语句可以将控制转给一个标识符标号。以下代码段给出了使用 goto 语句和一个标识符标号从一个紧密嵌套的循环中跳出的情况:

```
for(p=0;p<NUM_PATHS;++p)
{
    NumFiles=FillArray(pFileArray,pszFNames);
    for(i=0;i<NumFiles;++i)
    {
      if ((pFileArray[i]=fopen(pszFNames[i],"r"))==NULL)
         goto FileOpenError;
```

```
        //处理被打开的文件
    }
}

FileOpenError:
    cerr << "Fatal file open error,Processing interrupted.\n");
```

在上面例子中,如果出现了一个不知道的文件打开错误,则 goto 语句直接将控制转到打印错误信息的语句。

一个标号不能自己独立出现,而必须总是附于一条语句上。如果一个标号被它自身需要,则在标号后放一条空语句。

标号具有函数范围,不能在其函数内被再说明。但是相同的名称可在不同的函数里用作标号。

在 case 语句中使用标号

在 case 关键字后出现的标号不能在 switch 语句外面再出现(此限制还适用于 default 关键字)。以下代码段给出了 case 标号的正确使用:

```
//Microsoft 窗口信息处理循环样本
switch(msg)
{
case WM_TIMER://处理时间事件
    SetClassWord(hWnd,GCW_HICON,ahIcon[nIcon++]);
    ShowWindow(hWnd,SW_SHOWNA);
```

```
        nIcon %=14;
        Yield();
        break;
case WM_PAINT:
        //获取设备上下文的一个句柄
        //BeginPaint在合适情况发出WM_ERASEBKGND

        memset(&ps,0x00,sizeof(PAINTSTRUCT));
        hDC=BeginPaint(hWnd,&ps);

        //通知窗口绘制已完成

        EndPaint(hWnd,&ps);
        break;

case WM_CLOSE:
        //关闭此窗口和所有子窗口

        killTimer(hWnd,TIMER1);
        DestroyWindow(hWnd);
        if (hWnd==hWndMain)
            PostQuitMessage(0);  //退出应用
```

```
    break;

default:
    //没有被 case 语句特别包含的所有信息采用此选项
    return DefWindowProc(hWnd,Message,wParam,lParam);
    break;
}
```

表达式语句

表达式语句使表达式被求值。作为一条表达式语句的结果没有控制的转移或循环的发生。

语法

表达式语句:

    表达式 opt;

一条表达式语句中的所有表达式在执行下一条语句前被求值。且所有副作用都完成。最普通的表达式语句是赋值和函数调用。C++还提供一空语句。

空语句

“空语句”是一条没有表达式的表达式语句,它在语言的语法要求一条语句但没

有表达式求值时是很有用的,它由一个分号组成。
空语句通常被用作循环语句中的占位员或在复合语句或函数的末尾放置标号的语句。
以下代码段显示了结合空语句如何将一个字符串拷贝到另一个:

```
char *strcpy(char *Dest,const char *Source)
{
    char *DestStart=Dest;

    //将由 Source 所指的值赋给 Dest,直到遇到字符串的结束符
    while(*Dest++=*Source++)
        ;            //空语句
    return DestStart;
}
```

复合语句(块)

复合语句由括在花括号({})中的 0 个或多个语句组成,一个复合语句可被用于任何需要语句的地方,复合语句通常被称为“块”。
语法
复合语句:
    {语句表 $_{opt}$}
语句表:

语句
语句表 语句
以下例子使用一个复合语句作为 if 语句的语句部分(有关 if 语句的详细语法见“if 语句”):

```
if (Amount>100)
{
    cout << "Amount was too large to handle\n";
    Alert();
}
else
    Balance-=Amount;
```

注意:因为一个说明是一条语句,所以一个说明可以是语句表中语句之一。因此,在一个复合语句内说明的,但没有显式地说明为 static 的名称,具有局部范围和(对对象而言)生存期,有关具有局部范围的名称的处理的详细内容参见第 2 章“基本概念”中的“范围”。

选择语句

C++的选择语句 if 和 switch 提供一种方法有选择地执行部分代码。
语法
选择语句:
if(表达式) 语句

```
    if (表达式) 语句 else 语句
    switch(表达式) 语句
```

if 语句

if 语句对括号内的表达式求值,表达式必须为算术的或指针类型;或必须为定义了到一个算术的或指针类型的非模糊转换的一个类类型(有关转换见第 3 章“标准转换”)。
在 if 语法两种形式的任何一种形式中,如果表达式求值为一个非 0 值(真),则依赖于求值的语句被执行;否则,被跳过。
在 if...else 语法中,如果表达式求值结果为 0,则第二条语句被执行。
if...else 语句的 else 子句与最近的前面的尚未与 else 语句匹配的 if 语句相匹配,以下代码段说明这是如何工作的:

```
if (condition1==true)
    if (condition2==true)
       cout << "condition1 true; condition2 true\n";
else
       cout << "condition1 true; condition2 false\n";
else
     cout << "condition1 false\n";
```

很多程序员使用花括号({})显式地指明复杂的 if 和 else 子句的匹配,如下例中所示:

```
if (condition1==true)
```

```
{
    if (condition1==true)
        cont << "condition1 true; condition2 true\n";
    else
        cont << "condition1 true; condition2 false\n";
}
    else
        cont << "condition1 false\n";
```

尽管花括号不是严格必须的,但它们指明了 if 和 else 语句之间的配对关系。

switch 语句

C++的 switch 语句允许在多个代码节中作选择,选择依赖于表达式的值。括在括号中的表达式即“控制表达式”必须为一个整型或一个类类型且非模糊地转到整型。整型提升如第 3 章“标准转换”中“整型提升”所描述的被执行。
switch 语句导致一个非条件的跳转到、进入或跳过为“开关体”的语句,这依赖于控制表达式的值, case 标号的值和 default 标号的是否存在。开关体通常是一个复合语句(尽管这不是语法上的要求)。通常情况下,开关体中的某些语句用 case 标号或用 default 标号标记。标记语句不是语法上必须的,但没有它们 switch 语句就毫无意义。default 标号仅能出现一次。
语法

```
case 常量表达式:语句
default:语句
```

在 case 标号中的常量表达式被转换到控制表达式的类型,然后比较相等性。在一个给定的 switch 语句中,在 case 语句中的常量表达式不能有两次求值为相同的值。其行为如表 5.1 所示:

表 5.1 switch 语句行为

条件	操作
被转换的值与被提升的控制表达式相匹配	控制转到那个标号后的语句
没有一个常量与 case 标号中的常量匹配存在 default 标号	控制转到 default 标号
没有一个常量与 case 标号中的常量匹配,不存在 default 标号	控制转到 switch 语句后面的语句

switch 语句的一个内部块可以包含带初始化的定义,只要它们是可到达的,即不会被所有可能的执行路径绕过。用这些说明引入的名称具有局部范围。以下代码段显示了 switch 语句是如何工作的:

```
switch ( tolower(*argv[1]) )
{
    //错误,不可到达的说明
    char szChEntered[]="Character entered was:";
case'a':
    {
    //szChEntered 的说明是可行的,为局部范围
```

```
    char szChEntered[]="Character entered was:";
    cout << szChEntered << "a\n";
    }
    break;

case'b':
    //szChEntered 的值未定义
    cout << szChEntered << "b\n";
    break;

default:
    //szChEntered 的值未定义
    cout << szChEntered << "neither a nor b\n";
    break;
}
```

switch 语句可被嵌套。在这种情况中,case 或 default 标号与包含它们的最深层嵌套的 switch 语句相对应。例如:

```
switch(msg)
{
case WM_COMMAND://窗口命令,查找更多的
    switch(wParam)
    {
```

```
    case IDM_F_NEW:    //文件创建菜单命令
        delete wfile;
        wfile=new WinAppFile;
        break;
    case IDM_F_OPEN:    //文件打开菜单命令
        wfile->FileOpenDlg();
        break;
    ...
    }
case WM_CREATE:    //创建窗口
    ...
    break;
case WM_PAINT://窗口需要重画
    ...
    break;
default:
    return DefWindowProc(hWnd, Message, wParam, lParam) ;
}
```

上面的代码段来自 Microsoft Windows 消息循环,它说明了 switch 语句能够怎样地被嵌套。由 wParam 的值进行选择的 switch 语句仅当 msg 为 WM_COMMAND 时被执行。作为菜单选择使用的 case 标号 IDM_F_NEW 和 IDM_F_OPEN 与里层的 switch 语句相对应。

控制并没有被 case 或 default 标号阻止。要在复合语句的尾部结束执行,则插入一个 break 语句。这将控制转给 switch 语句后面的语句。此例说明控制是如何“向下通过”的,除非用一条 break 语句:

```
BOOL fClosing=FALSE;
...
switch(wParam)
{
case IDM_F_CLOSE:  //文件关闭命令
    fClosing=TRUE;
    //向下通过
case IDM_F_SAVE:  //文件保存命令
    if (document->IsDirty())
        if (document->Name()=="UNTITLED")
            FileSaveAs(document);
        else
            FileSave(document);

  if (fClosing)
        document->Close();

  break;
}
```

上面的代码说明了如何利用 case 标号不妨碍控制流程的事实。如果 switch 语句将控制转给 IDM_F_SAVE,则 fClosing 为 FALSE。因此,在文件被存储后,该文档未关闭。但是如果 switch 语句将控制转给 IDM_F_CLOSE,则 fclosing 设置为真值,保存文件的代码被执行。

迭代语句

迭代语句使得语句(或复合语句)按照某些循环终止条件执行 0 次或多次。当这些语句为复合语句时,它们被顺序地执行,除非遇到 break 语句或 continue 语句(这些语句的描述参见本章后面的“break 语句”和“continue 语句”)。
C++提供三种迭代语句即 while、do 和 for。每种都迭代到其终止表达式求值为 0(假),或直到用一个 break 语句强迫循环终止。表 5.2 归纳了这些语句和它们的作用;每一种都在随后的章节中被详细地讨论。

表 5.2　C++循环语句

语句	求值位置	初始化	增量
while	循环顶	无	无
do	循环底	无	无
for	循环顶	有	有

语法
循环语句:
    while (表达式) 语句

do 语句 while (表达式)
for (for-初始语句 表达式opt;表达式opt) 语句
for-初始语句
表达式语句
说明语句
一个迭代语句的语句部分不能是一个说明,但可以是一个复合语句,包含一个说明。

while语句

while语句重复执行一语句直到指定的终止条件(表达式)求值为 0。终止条件的测试在循环每次执行之前进行;因此,while 循环执行 0 次或多次,这取决于终止表达式的值。以下代码使用 while 循环剪切一个字符串的尾空格:

```
char *trim(char *szSource)
{
    char *pszEOS;
    //设置字符串尾部指针指向字符串尾部之前的那个字符
    pszEOS = szSource + strlen(szSource) - 1;

    while (pszEOS>=szSource && *pszEOS==' ')
        *pszEOS--='\0';

    return szSource;
```

```
}
```

终止条件在循环的顶部被求值,如果没有尾部空格,循环则永不执行。
表达式必须为一整型、指针类型或带有一个非模糊的转换到一个整型或指针类型的一个类类型。

do 语句

do 语句重复执行一个语句直到指定的终止条件(表达式)求值为 0。终止条件的测试在循环每次执行之后进行,因此,do 循环执行一次或多次,这取决于终止表达式的值。以下函数使用 do 语句去等待用户按下一个指定键:

```
void WaitKey(char ASCIICode)
{
    char chTemp;

    do
    {
        chTemp=_getch();
    }
    while(chTemp!=ASCIICode);
}
```

在前面的代码中使用一个 do 循环而不是 while 循环。使用 do 循环,_getch 函数在终止条件被求值之前被调用以获取一个击键值。此函数可以用 while 循环编写,但没有那么简洁:

```
void WaitKey(char ASCIICode)
{
    char chTemp;

    chTemp=_getch();

    while (chTemp!=ASCIICode)
    {
       chTemp=_getch();
    }
}
```

表达式必须为一整型、指针类型或带有一个非模糊的转换到一个整型或指针类型的一个类类型。

for 语句

for 语句可被分成三个独立的部分,如表 5.3 所示。

表 5.3 for 循环元素

语法名	何时被执行	内容
for-初始语句	在 for 语句或子语句的任何其它元素之前	常被用于初始化循环索引,可包含表达式或说明

续表

表达式 1	在循环的给定重复、包括第一次重复	一个表达式求值为一个整型或执行之前。 一个具有到整型的非模糊转换的类类型
表达式 2	在循环的每次重复的末尾、表达式 1 在表达式 2 被求值后被测试	通常用于增量循环索引

for 初始化语句常被用于说明及初始化循环指标变量,表达式 1 通常被用于测试循环终止标准。表达式 2 常被用于增量循环指标。

for 语句重复执行语句直到表达式 1 求值为 0,for-初始化语句、表达式 1 和表达式 2 域均是可选的。

以下 for 循环:

```
for (for-init-statement;expression1;expression2)
{
    //语句
}
```

等价于以下 while 循环:

```
for-init-statement;
while (expression1)
{
  //语句
```

```
  expression2;
}
```
使用 for 语句指定一个无穷循环的方便用法为：
```
for(;;)
{
    //待执行的语句
}
```
这等价于
```
while(1)
{
    //待执行的语句
}
```
for 循环的初始化部分可以是一个说明语句或一个表达式语句,包括空语句。初始化可以包括任意顺序的表达式和说明,用逗号隔开。在 for-初始化语句内说明的任何对象具有局部范围,就好象它在 for 语句前刚说明。尽管对象的名称可被用于相同范围内的局部不止一个的 for 循环中,但说明仅能出现一次,例如：
```
#include <iostream.h>

void main()
{
    for (int i=0;i<100;++i)
       cout << i << "\n";
```

```
  //循环指标 i 不能在这里的 for-始化语句中说明,因为它还在该范围内
  for (i=100;i>=0;--i)
       cout << i << "\n";
}
```

尽管 for 语句的三个域通常被用于初始化、终止测试和增量,但它们不限于这些用法。例如,以下代码打印从 1 到 100 的数,子语句为空语句:

```
#include <iostream.h>

void main()
{
     for (int i=0;i<100;cout << ++i << endl)
     ;
}
```

跳转语句

C++跳转语句执行一个控制的立即转移。

语法

跳转语句:

```
     break;
     continue;
```

```
    return 表达式opt;
    goto 标识符;
```

break 语句

break 语句被用于退出一个循环或 switch 语句。它将控制转到紧跟在循环子语句或 switch 语句后的语句。
break 语句只终止最紧密包含它的循环或 switch 语句。在循环中,break 被用于在终止条件等于 0 之前终止;在 switch 语句中,break 语句用于通常在一个 case 标号之前终止代码段。
以下例子说明在 for 循环中 break 语句的用法:

```
for(; ;)  //没有终止条件
{
    if(List->AtEnd())
       break;

    list->Next();
}
```

```
cout << "Control transfers to here.\n";
```

注意:还有其它简单的方法退出循环。在更复杂的循环中最好使用 break 语句,因为循环可能很难说在几条语句被执行之前是否应终止。
有关在 switch 语句体内使用 break 语句的例子见本章中前面的“ switch 语句 ”。

continue 语句

continue 语句迫使控制立即转到最小包含循环的循环继续语句。(“循环继续”是包含循环控制表达式的语句)。因此,continue 语句仅能在一个循环语句的依赖语句(尽管它可能是那语句中的唯一的语句)中出现一次。在一个 for 循环中,continue 语句的执行导致表达式 2 的求值,然后是表达式 3。
以下例子显示如何使用 continue 语句绕过代码段跳到循环的下一次迭代:

```
#include <conio.h>

//从以 0 结尾的字符串 szLegalString 获取一个字符。返回输入字符的下标

int GetLegalChar(char *szLegalString)
{
    char *pch;

   do
   {
      char ch=_getch();

 //使用 strchr 库函数确定读入的字符是否在字符串中,如果不在,就用
continue 语句
 //越过循环中的余下的语名
```

```
 if (pch=strchr(szLegalString,ch))==NULL)
    continue;

  //一个在字符串 szLegalString 中字符被输入,返回其下标
  return(pch-szLegalString);

  //continue 语句将控制转到这里
 }  while(1);

return 0;
}
```

return 语句

return 语句允许一个函数立即将控制转回到调用函数(或者如果是主函数,则控制转到操作系统)。return 语句接受一个表达式,即传回调用函数的值。void 类型函数、构造函数和析构函数不能在 return 语句中指定表达式;其它所有类型的函数都必须在 return 语句中指定一个表达式。
如果指定有,则表达式被转换到函数说明中指定的类型,如同执行一个初始化。从表达式类型到函数的 return 类型的转换会导致临时对象的创建,有关临时量怎样以及何时被创建的更多信息参见第 11 章“特殊成员函数”中的“临时对象”。当控制流退出包含函数定义的块时,其结果与执行没有表达式的 return 语句的结果是一样的。对于作为返回值说明的函数这是非法的。

一个函数可有任意数目的 return 语句。

goto 语句

goto 语句将控制无条件地转到命名的标号处,标号必须在当前函数中。
有关标号和 goto 语句的更多信息参见本章开头的“标号语句”和“使用带 goto 语句的标号”。

说明语句

说明向当前范围引入新的名称,这些名称可以是:

- 类型名(class、struct、union、enum、typedef 和成员指针)
- 对象名称
- 函数名称

语法
说明语句:
    说明
如果在一个块内的说明引入一个已经在块的外部说明的名称,则在块持续期间前面的说明被隐藏,在块结束后,前面的说明再次可见。
在同一块内对相同的名称作多个说明是非法的。
关于说明和名称隐藏的更多信息参见第 2 章“基本概念”中的“说明与定义”及“范围”。

自动对象的说明

在 C++中,对象可用 auto 或 register 关键字说明为自动存储类,如果一个局部对象(在函数内部说明的对象)没有使用任何存储类关键字,则被假定为 auto。C++对这些对象的初始化和说明与用静态存储类说明的对象是不同的。

自动对象的初始化

auto 或 register 存储类的对象的说明语句每次被执行时,则进行初始化。来自“continue 语句”中的以下例子给出了 do 循环内部的自动对象 ch 的初始化:

```
#include <conio.h>

//从以空格结尾的字符串 szLegalString 获取一个字符。返回输入字符的指标
int GetLegalChar (char *szLegalString)
{
    char *pch;
do
{
  //循环每次执行时,此说明语句被执行一次
  char ch=_getch();

  if ((pch=strchr(szLegalString, ch))==NULL)
      continue;
```

```
    //一个在字符串 szLegalString 中的字符被输入,返回其指标
    return (pch-szLegalString);
  } while(1);
}
```

对循环的每次重复(每次都遇到说明),宏_getch 被求值,且 ch 用结果进行初始化。当用 return 语句将控制转到块外部时,ch 被销毁(在此情况下,存储器被撤销分配)。有关初始化的另一个例子参见第 2 章“基本概念”中的“存储类”。

自动对象的析构

定义在循环中的对象在循环的每次重复时从块中退出时,或当控制转到说明之前的一点时,被销毁一次。在块内而不是循环中说明的对象在退出块时或当控制转到说明之前的一点时被销毁。

注意:析构函数可简单地意味着撤销对象的分配或对类类型对象调用对象的析构函数。

当跳转语句将控制转到循环或块外时,从其转出的块内说明的对象被销毁在向其转入的块内的对象没有被销毁。当控制转到说明之前的一点时,对象被销毁。

控制的转移

你可以用 goto 语句或 switch 语句中的 case 标号指定一个程序转移越过一个初始器。这样的代码是非法的,除非包含初始器的说明包括在跳转语句出现的块中。

以下例子给出了一个说明和初始化对象 total、ch 和 i 的循环,还有一条错误的

goto 语句将控制越过一个初始化器。

```
//读输入直到输入一个非数字字符
while(1)
{
    int total=0;

    char ch=_getch();

    if (ch>='0' || ch<='9')
    {
       goto label1;    //错误:移动越过了 i 的初始化器

       int i=ch-'0';
label1:
       total+=i;
   }  //如果 goto 错误未出现,i 将在此处被销毁
   else
      //Break 语句将控制转到循环外,销毁 total 和 ch
      break;
}
```

在前面的例子中,goto 语句试图将控制移动越过 i 的初始化器。但是如果 i 被说明但未被初始化,则移动将是合法的。

在为 while 语句的语句部分服务的块内说明的 total 和 ch 的对象,在使用 break 语句退出块时被销毁。

静态对象的说明

一个对象可以用 static 或 extern 关键字说明为静态存储类。局部对象必须显式地说明为 static 或 extern 以具有静态存储类。所有全局对象(在所有函数外部说明的对象)具有静态存储类,你不能在微模式程序中说明静态实例。

静态对象的初始化

全局对象在程序开始处初始化(关于全局对象的构造和析构的更多信息,参见第 2 章“基本概念”中的“额外的启动考虑”和“额外的结束考虑”)。
说明为 static 的局部对象在程序流程中第一次遇到其说明时被初始化,在第 2 章“基本概念”中介绍的以下类显示了这是如何工作的:

```
#include <iostream.h>
#include <string.h>

//定义一个类记录初始化和析构
class InitDemo
{
public:
  InitDemo(char *szWhat);
  ～InitDemo();
```

```
private:
  char *szObjName;
};

//*InitDemo 类的构造函数
InitDemo::InitDemo(char *szWhat)
{
    if (szWhat!=0 && strlen(szWhat)>0)
    {
        szObjName=new char[strlen(szWhat)+1];
        strcpy(szObjName,szWhat);
    }
    else
       szObjName=0;

    clog << "Initializing:" << szObjName << "\n";
}

//InitDemo 的析构函数
InitDemo::~InitDemo()
{
```

```
    if (szObjName !=0)
     {
        clog << "Destroying: " << szObjName << "\n";
       delete  szObjName;
     }
}

//主函数
void main(int argc,char *argv[]);
{
     if(argc<2)
     {
        cerr << "Supply a one-letter argument.\n");
        return -1;
     }

     if(*argv[1]== 'a' )
     {
        cout << "*argv[1] was an 'a' \n";

        //说明静态局部对象
        static InitDemo I1("static I1");
```

```
    }
    else
      cout << "*argv[1] was not an 'a'\n";
}
```

如果提供给此程序的命令行参量以小写字母“a”开头,则 I1 的说明被执行,发生初始化,结果为:

```
*argv[1] was an 'a'
Initializing: static I1
Destroying: static I1
```

否则,控制流绕过 I1 的说明,结果为:

```
*arg[1] was not an 'a'
```

当一个静态局部对象用一个不等于常量表达式的初始化器说明时,该对象在第一次进入块执行前一点被给定值 0(转换到合适的类型)。但是该对象是不可见的,而且直到说明的实际点处才调用构造函数。

在说明所在点处,对象的构造函数(如果对象是一个类类型的)被如期调用(静态局部对象仅在它们第一次被看见时初始化)。

静态对象的析构

局部静态对象在由 atexit 指定的终止期间被销毁。

如果一个静态对象因为程序控制流绕过了其说明而未被构造,则不要试图去销毁那个对象。

异常处理

Microsoft C++支持两种异常处理,C++异常处理(try、throw、catch)和结构异常处理(__try/__except、__try/__finally)。如果可能,应尽可能用 C++的异常处理而不用结构异常处理。
注意:在本节中,术语“结构异常处理”和“结构的异常”(或“C 异常”)仅指由 Win32 提供的结构异常处理机制,所有其它的异常处理的引用(或“C++异常”)都指的是 C++异常处理机制。
尽管结构异常处理与 C 和 C++源文件一起作用,但它不是专为 C++设计的,对 C++程序,你应当使用 C++异常处理。

try, catch 和 throw 语句

C++语言提供对处理异常情况的内部支持,异常情况即是所知道的“异常”,可能在你的程序执行期间出现。
try、throw 和 catch 语句已被加到 C++语言中去实现异常处理。有了 C++异常处理,你的程序可以向更高的执行上下文传递意想不到的事件,这些上下文能更好地从这些异常事件中恢复过来。这些异常由正常控制流外的代码进行处理。
Microsoft C++编译器朝着 C++进化中的标准去实现基于 ISO WG21/ANSI X3J16 工作文件的 C++异常处理模式。
语法
try 块:

try 复合语句 处理器表

处理器表：

处理器 处理器表 $_{opt}$

处理器：

catch(异常说明) 复合语句

异常说明：

类型指示符表 说明符

类型指示符表 抽象说明符

类型指示符表

...

throw-表达式：

throw 赋值表达式 $_{opt}$

try 子句后的复合语句是代码的保护段。throw 表达式“丢弃”(凸起)一个异常,catch 子句后的复合语句是异常处理器,“捕获”(处理)由 throw 表达式丢弃的异常。异常说明语句指示子句处理的异常的类型,类型可以是任何有效的数据类型,包括 C++的类。如果异常说明语句是一个省略号(...),catch 子句处理任何类型的异常,包括 C 的异常。这样的处理器必须是其 try 块的最后一个处理器。

throw 的操作数语法上与 return 语句的操作数相似。

注意:Microsoft C++不支持函数 throw 特征机制,如 ANSI C++草案的 15.5 节所描述的。此外,它也不支持 ANSI C++草案的 15 节中描述的 *function-try-block*。

执行过程如下：

1. 控制通过正常的顺序执行到达 try 语句,保护段(在 try 块内)被执行。
2. 如果在保护段执行期间没有引起异常,跟在 try 块后的 catch 子句不执行。从异常被丢弃的 try 块后跟随的最后一个 catch 子句后面的语句继续执行下去。
3. 如果在保护段执行期间或在保护段调用的任何例行程序中(直接或间接的调用)有异常被丢弃,则从通过 throw 操作数创建的对象中创建一个异常对象(这隐含指可能包含一个拷贝构造函数)。在此点,编译器在能够处理丢弃类型的异常的更高执行上下文中寻找一个 catch 子句(或一个能处理任何类型异常的 catch 处理器)。catch 处理程序按其在 try 块后出现的顺序被检查。如果没有找到合适的处理器,则下一个动态封闭的 try 块被检查。此处理继续下去直到最外层封闭 try 块被检查完。
4. 如果匹配的处理器未找到,或如果在不自动分行时出现异常,但在处理器得到控制之前预定义的运行函数 terminate 被调用。如果一个异常发生在丢弃异常之后,则在循环展开开始之前调用 terminate。
5. 如果一个匹配的 catch 处理器被找到,且它通过值进行捕获,则其形参通过拷贝异常对象进行初始化。如果它通过引用进行捕获,则参量被初始化为指向异常对象,在形参被初始化之后,“循环展开栈”的过程开始。这包括对那些在与 catch 处理器相对应的 try 块开始和异常丢弃地点之间创建的(但尚未析构的)所有自动对象的析构。析构以与构造相反的顺序进行。catch 处理器被执行,且程序恢复到跟随在最后的处理器之后的执行(即不是 catch 处理器的第一条语句或构造)。控制仅能通过一个丢弃的异常输入一个 catch 处理器,而永远不能通过 goto 语句或 switch 语句中的 case 标号。

以下是一个 try 块和其相应的 catch 处理器的简单例子,此例子检测使用 new 运

算符的存储器分配操作的失败。如果 new 成功了,则 catch 处理器永不执行:

```
#include <iostream.h>

int main()
{
    char *buf;
    try
    {
        buf=new char[512];
        if(buf==0)
           throw "Memory allocation failure!";
    }
    catch (char *str)
    {
        cout << "Exception raised: " << str <<'\n';
    }
    //...
    return 0;
}
```

throw 表达式的操作数指示一个 char*类型的异常正被丢弃。它由表示有捕获 char*类型的一个异常的能力的 catch 处理器进行处理。在存储器分配失败事件中,这是从前面例子得到的输出:

```
Exception raised: Memory allocation failure!
```

C++异常处理的真正能力不仅在于其处理各种不同类型的异常的能力,还在于在堆栈循环展开期间为异常丢弃前构造的所有局部对象自动调用析构函数的能力。

以下例子演示了使用带析构语义的类的 C++异常处理:

```
#include <iostream.h>

void MyFunc(void);

class CTest
{
public:
  CTest() {};
  ～CTest() {};
  const char *ShowReason() const { return "Exception in CTest class.";}
};

class CDtorDemo
{
public:
  CDtorDemo();
  ～CDtorDemo();
```

```
};

CDtorDemo::CDtorDemo()
{
  cout << "Constructing CDtorDemo.\n";
}

CDtorDemo::~CDtorDemo()
{
   cout << "Destructing CDtorDemo.\n";
}

void MyFunc()
{
   CDtorDemo D;
   cout << "In MyFunc(). Throwing CTest exception.\n";
   throw CTest();
}

int main()
{
   cout << "In main.\n";
```

```
    try
    {
        cout << "In try block, calling MyFunc().\n";
        MyFunc();
    }
    catch (CTest E)
    {
        cout << "In catch handler.\n";
        cout << "Caught CTest exception type:";
        cout << E.ShowReason() << "\n";
    }
    catch (char *str)
    {
         cout << "Canght some other exception:" << str << "\n";
    }
    cout << "Back in main. Execution resumes here.\n";
    return 0;
}
```

以下是上面例子的输出：

```
In main.
In try block, calling MyFunc().
Constructing CDtorDemo.
```

```
In MyFunc(). Throwing CTest exception.
Destructing CDtorDemo.
In catch handler.
Caught CTest exception type; Exception in CTest class.
Back in main. Execution resumes here.
```

注意在此例中,异常参量(catch 子句的参量)在两个 catch 处理器中都被说明:

```
catch(CTest E)
//...
catch(char *str)
//...
```

你不需要说明此参量;在很多情况下可能通知处理器有某个特定类型的异常已经产生就足够了。但是如果你在异常说明中没有说明一个异常对象,你将无法访问 catch 处理程序子句中的那个对象。

一个不带操作数的 throw 表达式把当前正被处理的异常再次丢弃,这样一个表达式仅仅应该出现在一个 catch 处理器中或从 catch 处理器内部被调用的函数中,再次丢弃的异常对象是源异常对象(不是拷贝)。例如:

```
try
{
    throw CSomeOtherException();
}
catch(...)    //处理所有异常
{
```

```
    //对异常作出响应(也许仅仅是部分的)
    //...

    throw;  //将异常传给某个其它处理器
}
```

未处理的异常

如果为当前的异常找不到匹配的处理器(或省略的 catch 处理器),则预定义的 terminate 函数被调用(你还可以在你的任何处理器中显式地调用 terminate)。terminate 的缺省动作是调用 abort,如果你想要 terminate 在退出应用之前调用你程序中的某个其它的函数,用待调用的函数名作为单个参量去调用 set_terminate 函数。你可在你程序中的任何地方调用 set_terminate。terminate 例程总是调用给定的最后一个函数作为 set_terminate 的一个参量。例如:

```
#include <eh.h>    //用于函数原型
//...
void term_func() {//...}
int main( )
{
    try
    {
      //...
```

```
        set_terminate(term_func);
        //...
      throw "Out of memory!";    //对此异常无 catch 处理器
    }
    catch(int)
  {
     cout << "Integer exception raised.";
    }
    return 0;
}
```

term_func 函数应该终止程序或当前的线程,理想地是通过调用 exit。如果它没有且是返回其调用者,则 abort 被调用。

有关 C++异常处理的更多信息,参见 Margaret A.Ellis 和 Bjarne Stroustrup 所著的“注释的 C++参考手册”。

结构异常处理

__try/__except 和 __try/__finally 语句是 Microsoft 对 C 语言的扩充,使得应用在将正常终止执行的事件之后获得程序的控制权。

注意:结构异常处理与 C 和 C++源文件一起工作。但是它不是特别为 C++设计的。尽管局部对象的析构函数将被调用,如果你在一个 C++程序中使用结构异常处理(如果你使用/GX 编译器选项);但你用 C++异常处理能确保你的代码具有更好的移植性。C++异常处理机制更灵活,它可以处理任何类型的异常。

更多的信息参见本卷前面的“C 语言参考”中的第 5 章“语句”中的“try-finally 语句”。

语法

try-except 语句：

```
__try 复合语句
__except (表达式) 复合语句
```

try-finally 语句：

```
__try 复合语句
__finally 复合语句
```

如果你有使用结构化异常处理的 C 模式，则它们可与使用 C++异常处理的 C++模式混用。

当一个 C(结构化的)异常产生时，它可以由 C 处理器处理，或由 C++的 catch 处理器捕获，要看哪个与异常上下文更动态地接近。两种模式的一个主要区别是：当 C 异常产生时，它总是无符号整型，而一个 C++异常可以是任何类型的。即 C 异常由一个无符号整型值标识，而 C++丢弃异常则由数据类型标识。但是当一个 C++的 catch 处理器能捕获一个 C 异常时(例如，通过一个“省略号的”catch 处理器)，一个 C 异常也可以通过使用 C 异常绕接类作为一个有类型的异常来处理。通过从此类派生，每个 C 异常可归属于一个特定的派生类。

为了使用 C 异常 wrapper 类，你可安装一个定制的 C 异常转换器函数，该函数在每次产生 C 异常时由内部异常处理机制调用。在你的转换器函数内，你可以丢弃任何有类型的异常，它们可由对应的匹配 C++的 catch 处理器捕获。要指定一个定制翻译函数，用你的翻译函数名作为单个参量去调用_set_se_translator 函

数。

第 6 章　说　　明

说明向一个程序中引入新的名称,本章的主题包括:

- 说明符
- 枚举说明
- 连接规格
- 模板规格
- 名称空间

除引入一个新名称外,一个说明还指定一个标识符是如何被编译器解释的。说明并不自动地保留标识符相关的存储,保留存储由定义完成。

注意:大多数的说明也是定义。

语法

说明:

　　说明指示符 $_{opt}$ 说明符表 $_{opt}$;
　　函数定义
　　连接规格
　　模板规格

在说明符表中的说明符包含正被说明的名称,尽管说明符表是作为可选项给出的,但它也只能是在函数的说明或定义中被省略。

注意:一个函数的说明通常被称为一个“原型”。此说明提供参量的类型信息和

函数的返回类型,从而允许编译器执行正确的转换以及确保类型的安全性。
一个说明的说明指示符部分也是作为可选项给出的,但它只能在类类型或枚举的说明中被省略。
说明出现在一个范围中,这用于控制被说明名称的可见性和被定义对象(如果有)的持续时间。关于范围规则与说明是如何交互的更多信息见第 2 章“基本概念”中的范围。
一个对象的说明也是一个定义,除非它包含下一节存储类指示符中所描述的 extern 存储类指示符。一个函数说明也是一个定义,除非它是一个原型——不带定义函数体的一个函数头。一个对象的定义导致存储分配和那个对象的合适的初始化。

指示符

本节解释说明中的说明指示符部分(说明的语法在本章开头已给出)。
语法
多个说明指示符
    多个说明指示符 $_{opt}$ 说明指示符
说明指示符:
    存储类指示符
    类型指示符
    函数指示符
    friend

typedef
__declspec(扩充的说明修饰符序列)

Microsoft 特殊关键字__declspec 在附录 B“Microsoft 特殊修饰符”中的“扩充属性语法”中讨论。

说明的说明指示符部分是可被构造为一个类型名称的最长的说明指示符序列,说明的剩余部分是引入的名称,以下表中的例子说明了此概念:

说明	说明指示	名称
char *lpszAppName;	char*	lpszAppName
typedef char * LPSTR;	char*	LPSTR
LPSTR strcyp(LPSTR,LRSTR);	LPSTR	strcpy
volatile void *pvvObj;	volatilevoid *	pvvObj

因为 signed、unsigned、long 和 short 都隐含了 int,所以跟随这些关键字之一的一个 typedef 名称被作为说明符表中的一个成员,而不是作为说明指示符。

注意:因为一个名称可以被重新说明,所以其解释应对应当前范围中最近的说明。重说明能影响名称是如何被编译器解释的,特别是 typedef 名称。

存储类指示符

C++存储类指示符告诉编译器一个对象应存储的地方,以及他们说明的对象或函数的持续时间和可见性。

语法

存储类指示符:

auto
register
static
extern

自动存储类指示符

auto 和 register 存储类指示符仅能被用于说明被用于块内的名称或说明函数的形参。术语“自动”来源于这些对象的存储是在运行时自动分配的(通常在程序的堆栈上)事实。

auto 关键字

极少有程序员在说明中使用 auto 关键字。因为所有具有块范围的对象在没有显示地说明为另一个存储类的都隐含指为自动的。因此,以下两个说明是等价的:

```
{
auto int i; //显式说明为 auto
int j;  //隐含为 auto
}
```

register 关键字

Microsoft 特殊处→
编译器不接受用户对寄存器变量的要求,而是在全局寄存器分配优化(/Oe 选项)打开时作出其自己的寄存器选择。但是与 register 关键字相关的所有其它语义

是可用的。

Microsoft 特殊处结束

ANSI C 不允许取一个寄存器对象的地址,这个限制不用于 C++中。但如果取地址运算符(&)用于一个对象,则编译器必须把对象放在一个可用一个地址表示的地方,即在实践中,这意味着在存储器中而不是在一个寄存器中。

静态存储类指示符

静态存储类指示符 static 和 extren 可被用于对象和函数,表 6.1 给出了 static 和 extern 关键字可以使用的地方和不能使用的地方。

表 6.1 static 和 extern 的使用

构造	static 可用否?	extern 可用否?
在一个块内的函数说明	否	是
函数的形参	否	否
块内的对象	是	是
块外的对象	是	是
函数	是	是
类成员函数	是	否
类成员数据	是	否
typedef 名称	否	否

使用 static 关键字指定的名称具有内部连接,除了一个类的静态成员是具有外部连接,即名称在当前转换单元外面是不可见的。使用 extern 关键字指定的名

称具有外部连接,除非以前定义为具有内部连接。关于名称的可见性的更多信息参见第 2 章“基本概念”中的“范围”和“程序和连接”。
注意:说明为 inline 而且不是类成员函数的函数与说明为 static 的函数给定的是相同的连接特征。
说明尚未被编译器碰到的一个类名可被用在 extern 说明中。具有此种说明而引入的名称直到遇到了类说明才能被使用。

不带存储类指示符的名称

没有显式的存储类指示符的文件范围名称具有外部连接,除非他们:

- 使用 const 关键字说明。
- 在前面说明具有内部连接。

函数指示符

在函数说明中,你可以使用 inline 和 virtual 关键字作为指示符。这种 virtual 的用法有别于其在一个类定义的基类指示符中的使用。

inline 指示符

inline 指示符指示编译器用函数体代码去替换函数调用,这种替换叫“联编扩展”(有时称为“联编”)联编扩展在更大的代码的潜在代价上减少函数调用的费用。
inline 关键字告知编译器选用联编扩展,但是编译器能够创建函数的分开的实例(实例化),以及创建标准的调用连接,而不是联编插入代码。这能够发生的两

种情况是：

- 递归函数。
- 在转换单元的别处通过指针指向的函数。

注意被考虑作为联编候选者的函数必须使用新风格的函数定义。被说明为 inline 且不是类成员函数的函数具有内部连接，除非有其它的指定。

Microsoft 特殊处→

__inline 关键字等价于 inline。

__forceinline 关键字指示编译器进行联编函数而不执行任何代价/效益分析。程序员在使用此关键字中必须有好的判断能力，不减少 __forceinline 的使用会导致更大的有时甚至是更慢的代码。既使用 __forceinline，编译器也不能在所有的情况中都联编代码。编译器不能联编一个函数，如果：

- 函数或其调用者是用/ob0 编译的(调试模块的缺省选项)。
- 函数及其调用者使用不同类型的异常处理(一个是 C++异常处理，另一个是结构化的异常处理)。
- 函数有一个可变的参量表。
- 函数使用联编集合，除非用/og、/Ox、/O1 或/O2/编译。
- 函数用值返回一个非展开的对象，当用/GX、/EHs 或/EHa 编译时。
- 函数接到一个由值传递的拷贝构成的对象，当用/GX、/EHs 或/EHa 编译时。
- 函数是递归的而且没有伴随#pragma(inline_recursion 为 on)。带有 inline__recursion 编译指示，递归函数可被联编到 8 次调用的深度，或由 inline_depth 编译指示确定的深度(见下面内容)。

- 是虚函数。
- 程序取函数的地址。

如果编译器不能联编一个用__forceinline 说明的函数,它产生一个 1 级警告(4714)。

Microsoft 特殊处结束

如果用正常的函数,一个联编函数的参量没有定义一个求值顺序,事实上,它可以与使用正常函数调用拓扑进行传递时的参量求值顺序不同。

Microsoft 特殊处→

递归函数可以被联编替换到由 inline_depth 编译指示指定的深度。在那个深度之后,递归函数调用被当作是对函数的一个实例的调用,inline_recursion 编译指示控制在当前扩展之下的函数的联编扩展。相关的信息参见联编函数扩展(/Ob)编译器选项。

Microsoft 特殊处结束

联编类成员函数

在一个类说明体内定义的函数是一个联编函数,考虑以下类说明:

```
class Account
{
public:
  Account (double initial_balance){balance=initial_balance;}
  double GetBalance();
  double Deposit(Double Amount);
```

```
  double Withdraw(double Amount;
private;
  double balance;
};
```

Account 构造函数是一个联编函数,成员函数 GetBalance、Deposit 和 Withdraw 没有被指定为 inline,但可以用如下这样的代码作为联编函数去实现:

```
inline double Acount::GetBalance
{
    return balance;
}

inline double Account::Deposit(double Amount)
{
    return (balance+=Amount);
}

inline double Account::Withdraw(double Amount)
{
    return (balance-=Amount);
}
```

注意:在类说明中,函数没有用 inline 关键字进行说明。inline 关键字可在类说明中被指定,结果是一样的。

一个给定的联编成员函数,在每个编译单元中必须以同样的方式被说明,此约束使得联编函数表现得如同是被实例化的函数,此外,一个联编函数必须只有一个定义。
一个类成员函数缺省为外部连接,除非那个函数的定义包含 inline 说明符。前面的例子显示了这些函数不需要用 inline 说明符显式地说明,在函数定义中使用 inline 使其成为一个联编函数。但是在对那个函数调用之后再用 inline 重新说明该函数则是非法的。

联编函数和宏

尽管联编函数与宏相似(因为函数代码在编译时在调用点处被扩展)。但联编函数由编译器进行语义分析,而宏则通过预处理器进行扩展。结果是存在以下几个重要的不同点:

- 联编函数遵从强制在普通函数上的所有的类型安全性协议。
- 联编函数的说明与任何其它函数一样使用相同的语法,除了在函数说明中它们包括 inline 关键字。

作为参量传给联编函数的表达式只求值一次。在某些情况下,作为参量传给宏的表达式可求值不止一次。以下列子给出了一个宏,完成从小写字母到大写字母的转换:

```
#include <stdio.h>
#include <conio.h>

#define toupper(a) ((a)>='a' && ((a))<="z") ? ((a)-('a'-'A')) : (a))
```

```
void main()
{
    char ch = toupper(_getch());
    printf("%c",ch);
}
```

表达式 toupper(_getch())的意图是一个字符应从控制台设备(stdin)读入,而且如果需要则转成大写。

由于实现_getch 执行一次以确定字符是否大于或等于“a”,且一次确定是否小于或等于“z”。

如果在那个范围内,_getch 再执行一次将字符转成大写。这意味着当理想情况只应等待一个字符时,程序等待两个或三个字符。

联编函数纠正了这个问题:

```
#include <stdio.h>
#include <conio.h>

inline char toupper(char a)
{
    return ((a>='a' && a<='z') ? a-('a'-'A') : a);
}

void main()
```

```
{
    char ch=toupper(_getch());
    printf("%c",ch);
}
```

何时使用联编函数

联编函数常用于象访问私有的数据成员这样的小函数。这些一行或两行“访问器”函数的主要目的是返回对象的状态信息,短函数对函数调用的一般代价很敏感。长些的函数在调用或返回顺序上按比例花费少些时间,且从联编获益少些。在第 4 章“表达式”中的“函数调用结果”中介绍的 Point 类可以优化为如下所示:

```
class Point
{
public:
  //定义“访问器”(accessor)函数为引用类型
  unsigned& x();
  unsigned& y();
private:
  unsigned _x;
  unsigned _y;
};
```

```
inline unsigned& Point::x()
{
  return _x;
}

inline unsigned& Point::y()
{
  return _y;
}
```

假定坐标操作在这样一个类的一个客户中是相对普通的操作,用 inline 指定两个访问器函数(在前面例子中的 x 和 y)典型地节省了如下开销:

- 函数调用(包括参量传递和在堆栈中放置对象的地址)
- 保持调用者的堆栈框架
- 新的堆栈框架的设置
- 返回值传递
- 旧的堆栈框架恢复
- 返回

virtual 指示符

virtual 关键字仅被用于非静态类成员函数。它意味着将调用与函数结合在一起要推迟到运行时,更多的信息见第 9 章“派生类”中的“虚函数”。

typedef 指示符

typedef 指示符定义一个名称,可被用作一个类型或派生类型的同义词。你不能在一个函数定义内部使用 typedef 指示符。

语法

typedef 名称:
    标识符

一个 typedef 说明引进一个名称,在其范围内成为由说明的说明指示符部分给定的类型的同义词。与 class、struct、union 和 enum 说明不同,typedef 说明没有引入新类型,它们引进的是已存在类型的新名称。

typedef 说明的用法之一是使说明更为统一、紧凑,例如:

```
typedef char CHAR;//字符类型
typedef CHAR * PSTR; //指向字符串(char*)的指针
...
PSTR strchr(PSTR source,CHAR target);
```

由上面说明引入的名称是以下的同义词:

名称	同义类型
CHAR	char
PSTR	char *

上面例子的代码说明了一个类型名称 CHAR,然后用于定义派生类型名称 PSTR(指向一个字符串的指针)。最后,该名称用于说明函数 strchr。要看 typedef 关键

字如何用于区分说明的,把前面 strchr 的说明与以下说明比较:

```
char *srtchr(char * source,char target);
```

要使用 typedef 在相同的说明中指定基本的和派生的类型,你可以用逗号分开说明符。

例如:

```
typedef char CHAR,*PSTR;
```

typedef 的特别复杂的用法是为一个“指向返回类型为 T 的函数的指针”定义一个同义词。例如,一个含义为“指向不带参量且返回类型为 void 的函数的指针”的 typedef 说明使用以下代码:

```
typedef void (*PVFN)();
```

同义词在说明通过指针调用的函数数组时是方便的:

```
#include <iostream.h>
#include <stdlib.h>

extern void func1();  //说明 4 个函数
extern void func2();  //这些函数假定在其它的位置定义的
extern void func3();
extern void func4();

typedef void (*PVFN)(); //为指向不带参量且返回类型为 void 的函数
                        //的指针说明同义词
```

```
void main(int argc,char * argv[])
{
    //说明一个函数指针数组
    PVFN pvfn1[]={func1,func2,func3,func4];

    //调用在命令行指定的函数
    if(argc>0 && *argv[1]>'0' && *argv[1]<='4')
    (*pvfn1[atoi(argv[1])-1])();
}
```

typedef 名称的重说明

typedef 说明可被用于重说明相同的名称以指向相同的类型。例如：

```
//FILE1.H
typedef char CHAR;

//FILE2.H
typedef char CHAR;

//PROG.CPP
#include "file1.h"
#include "file2.h"  ///可行
...
```

程序 PROG.CPP 包括两个头文件,均包含 CHAR 名称的 typedef 说明。只要两个说明指向相同的类型,这样的重说明是可接受的。
一个 typedef 不能对前面已说明为一个不同类型的名称重定义。因此,如果 FILE2.H 包含:

```
//FILE2.H
typedef int CHAR;  //错误
```

编译器产生一个错误,因为试图重新说明 CHAR 名称以指向一个不同的类型,这种扩充去构造如下内容:

```
typedef char CHAR;
typedef char CHAR;  //可行,重新说明作为同一类型

typedef union REGS  //可行,REGS 名称由具有相同含义的 typedef 名称重新说明
{
    struct wordregs x;
    struct byteregs h;
}  REGS;
```

带类类型的 typedef 的使用

带类类型的 typedef 指示符的使用被极大地支持,因为 ANSI C 提供 typedef 说明中无名的结构的说明。例如,很多 C 程序员使用如下方法:

```
typedef struct //说明一个无名的结构并给它一个 typedef 名称 POINT
```

```
{
    unsigned x;
    unsigned y;
}  POINT;
```
这种说明的好处是它允许象：
```
POINT ptOrigin;
```
的说明代替：
```
struct point_t ptOrigin;
```
在 C++中,typedef 名称和实际类型(用 class、struct、union 和 enum 关键字说明的)的区别越来越明显。尽管在一个 typedef 语句中说明一个无名的结构的 C 的实践还在工作,但它并没有提供在 C 中那样的值得称道的效益。
在以下代码中,POINT 函数不是一个类型构造函数,它被解释为具有一个 int 返回类型的函数说明符。
```
typedef struct
{
    POINT();    //不是一个构造函数
    unsigned x:
    unsigned y:
} POINT;
```
上面的例子使用未命名的类 typedef 语法说明了一个名为 POINT 的类。POINT 被看作是一个类名;但,以下限制适用于以此方式引进的名称：

- 名称(同义词)不能出现在一个 class、struct 或 union 前缀的后面。

- 名称在一个类说明中不能被用作构造函数或析构函数名称。

总之,此语法并未提供任何继承、构造函数或析构函数的机制。

typedef 名称的名称空间

使用 typedef 说明的名称象其它标识符(除了语句标号)一样占据相同的名称空间。因此,它们不能使用相同的标识符作为一个前面已说明了的名称,除了在一个类类型说明中。

考虑以下例子:

```
typedef unsigned ling UL ;//说明一个 typedef 名称 UL
int UL;    //错误,重定义
```

附属于其它标识符的名称隐藏规则也同样管理使用 typedef 说明的名称的可见性,因此,以下例子在 C++中是合法的:

```
typedef unsigned long UL;  //说明一个 typedef 名称 UL
...
long Beep
{
    unsigned int UL; //重说明隐藏 typedef 名称
}
//typedef 名称在这里“未隐藏”。
```

友元指示符

friend 指示符用于设计与类成员函数具有相同的访问权限的函数或类。友元函

数和类在第 10 章“成员访问控制”中的“友元”部分有详细的介绍。

类型指示符

类型指示指定说明的名称的类型。
语法
类型指示符:
    简单类型名称
    类指示符
    枚举指示符
    详细阐述的指示符
    ::类名称
    const
    volatile
以下章节讨论简单类型名称,详细阐述了类型指示符和嵌套的类型名称。

简单类型名称

一个简单类型名称是一个完整类型的名称。
语法
简单类型名称:
    完整类名称
    限定类型名称
    char

short
int
long
signed
unsigned
float
double
void

表 6.2 给出了简单类型名称如何在一起使用的。

表 6.2 类型名称组合

类型	可与以下一起出现	注释
int	long 或 short,但不是两个均出现	int 类型隐含 long int 类型
long	int 或 double	long 类型隐含 long int 类型
short	int	short 类型隐含 short int 类型
signed	char、short、int 或 long	signed 类型隐含 signed int。signed char 类型的对象的最重要的位或有符号整型的位域被作为符号位。

续表

unsigned	char、short、int 或 long	unsigned 类型隐含 unsigned int。unsigred char 类型的对象的最重要的位和无符号整型的位域不作为符号位。

详细阐述的类型指示符

详细阐述的类型指示符被用于说明用户定义的类型,这些可以是类或枚举类型。

语法

详细阐述的类型指示符;

    类关键字 类名称
    类关键字 标识符
    enum 枚举名称

类关键字:

    class
    struct
    union

如果指定了标识符,则它被作为一个类名称,例如:

```
class Window;
```

该语句说明 Window 标识符为一个类名称,此语法用于类的前面说明。关于类名称的更多信息参见第 8 章“类”中的“类名称”。

如果一个名称用 union 关键字说明,则它必须还使用 union 关键字进行定义,使

用 class 关键字定义的名称可使用 struct 关键字进行说明(或反之),因此,以下代码例子是合法的:

```
//合法例 1
struct A; // A 的前向说明

class A  //定义 A
{
public:
    int i;
};

//合法例 2
class A;//A 的前向说明

struct A //定义 A
{
private:
   int i;
};

//合法例 3
union A;  //A 的前向说明
```

```
union A//定义 A
{
    int i;
    char ch[2];
};
但以下这些例子是非法的:
//非合法例 1
union A;   // A 的前向说明

struct A   //定义 A
{
    int i;
};

//非法例 2
union A;  //A 的前向说明

class A   //定义 A
{
public:
    int i;
```

```
};

//非法例 3
struct A;  //A 的前向说明

union  A  //定义 A
{
    int i;
    char ch[2];
};
```

嵌套的类型名称

Microsoft C++支持嵌套的类型的说明,包括命名的和无名的。

语法

限定的类型名称:

    typedef 名称

    类名称::限定的类型名称

完整的类名称:

    限定的类名称

    ::限定的类名称

限定的类名称:

    类名称

类名称::限定的类名称

在某些编程情形中,需要定义嵌套的类型。这些类型仅对它们被定义所在的类类型的成员函数是可见的,它们还可以通过用范围分辨运算符(::)构造一个限定的类型名而变成可见的。

注意:一个使用嵌套类型的最常用的类层次是输入输出流。在输入输出流头文件中,类 ios 的定义包括了一系列枚举的类型,封装起来仅与输入输出流库一同使用。

以下例子定义了嵌套的类:

```
class WinSystem
{
public:
  class Window
  {
  public:
      Window();              //缺省的构造函数
      ~Window();             //析构函数
      int NumberOf();        //类对象的数目
      int Count();           //类对象的个数
  private:
      static int CCount;
  };
  class CommPort
```

```
  {
  public:
    CommPort();//缺省的构造函数
    ～CommPort();    //析构函数
    int NumberOf();//类对象的数目
    int Count();    //类对象的个数
  private:
      static int CCount;
  };
};

//初始化 WindSystem 静态成员
int WinSystem::Window::CCount=0;
int WinSystem::CommPort::CCount=0;
```

为了访问定义在一个嵌套类中的名称,应使用范围分辩运算符::去构造一个完整的类名。

运算符的使用在前面例子中 static 成员的初始化中给出了,在你的程序中要使用一个嵌套的类,使用如下所示的代码:

```
WinSystem::Window Desktop;
WinSystem::Window AppWindow;

cout << "Number of active Windows:" << Desktop.Count() << "\n";
```

嵌套的无名类或结构可定义如下：

```
class Ledger
{
    class
    {
    public:
       double PayableAmt;
       unsigned PayableDays;
    } Payables;

    class
    {
    public:
       double RecvableAmt;
       unsigned RecvableDays;
    } Receivables;
};
```

一个无名类必须是一个没有成员函数、没有静态成员的集合。

注意：尽管一个枚举的类型可被定义在一个类说明内部，但反之却不行；类类型不能被定义在枚举说明的内部。

枚举说明

一个枚举是一种定义命名的常量的独特的整型。枚举使用 enum 关键字进行说明。

语法

枚举名称：

    标识符

枚举指示符：

    enum 标识符 opt {枚举表 opt}

枚举表：

    枚举器
    枚举表,枚举器

枚举器：

    标识符
    标识符=常量表达式

当一个对象采取的是一个已知的合理有限的值集合时,枚举的类型是有价值的,考虑下面从一副纸牌找同花色牌的例子：

```
class card
{
public
    enum Suit
    {
```

```
        Diamonds
        Hearts,
        Clubs,
        Spades
      };
      //说明两个构造函数:一个缺省构造函数,以及一个设置新牌的基数和同花色的值的
    //构造函数
    Card();
      Card(int CardInit,Suit SuitInit);
      //GetT 和 Set 函数
      int GetCardinal();//获取牌的基数值
      int SetCardinal();//设置牌的基数值
      Suit GetSuit ();  //获取同花色的牌
      void SetSuit(Suit new_suit); //设置同花色的牌
      char *NameOf();  //获取表示牌的字符串
private:
      Suit suit;
      int CardinalValue;
};

//为 Suit 定义一个后缀增量运算符
```

```
inline Card::Suit operator++(Card::Suit &rs,int)
{
    return rs= (Card::Suit)(rs+1);
}
```

上面例子定义了一个类 Card,包含一个嵌套的枚举类型 Suit。在程序中要创建一盒牌,使用如下代码:

```
Card *Deck[52];
int j=0;

for(Card::Suit
curSuit=Card::Diamonds,curSuit<=Card::Spades;curSuit++)
   for (int i=1;i<=13;++i)
      Deck[j++]=new Card(i, curSuit);
```

在上面例子中,Suit 类型是被嵌套的;因此,类名称(Card)必须显式地用于公共的引用。但在成员函数中,类名称可被省略。在代码的第一段,Card::Suit 的后缀增量运算符被定义。没有用户定义的增量运算符,curSuit 不能被增量。关于用户定义的运算符的更多信息见第 11 章“重载”中的“重载的运算符”。

考虑 NameOf 成员函数的代码(更好的实现在稍后给出):

```
char* Card::NameOf()  //获取牌的名称
{
   static char szName[20];
   static char *Numbers[]=
```

```
    {"1","2","3","4","5","6","7","8","9","10",Jack",Queen","King"
     };
    static char *Suits[]=
    { "Diamonds","Hearts","Clubs","Spades"};
    if (GetCardinal()<13)
      strcpy(szName,Numbers[GetCardinal()]);

    strcat(szName, "of");

    switch(GetSuit())
    {
     //Diamonds,Hearts,Clubs 和 Spades 不需要显式的类限定符
     case Diamonds:strcat(stName,"Diamonds");break;
     case Hearts  :strcat(stName,"Hearts");break;
     case Clubs   :strcat(stName,"Clubs");break;
     case Spades  :strcat(stName,"Spades");break;
     }
     return szName;
}
```

一个枚举的类型是一个整型,用 enum 说明引进的标识符可被用于常量出现的任何位置。通常,第一个标识符的值是 0()在前面例子中的 Diamonds),每一个后续的标识符的值依次增 1。因此,Spades 的值为 3。

表中的任何枚举器,包括第一个,可被初始化为一个值,而不是其缺省值。假设Suit的说明如下所示:

```
enum Suit
{
    Diamonds=5,
    Hearts,
    Clubs=4,
    Spades
};
```

则 Diamonds、Hearts、Clubs 和 Spades 的值分别为 5、6、4、5。注意 5 被用了不止一次。

这些枚举器的缺省值简化了 NameOf 函数的实现:

```
char * Card::NameOf()  //获取牌的名称
{
    static char szName[20];
    static char *Numbers[]=
    {"1","2","3","4","5","6","7","8","9","10","Jack","Queen","King"
    };
    static char *Suits[]=
    {"Diamonds","Hearts","Clubs","Spades"};

    if (GetCardinal()<13)
```

```
        strcpy(szName,Numbers[GetCardinal()]);

    strcat(szName,"of");
    strcat(szName,Suits[GetSuit()]);

    return szName;
}
```

访问器函数 GetSuit 返回 Suit 类型,一个枚举的类型,因为枚举的类型是整型,它们可被用作数组下标运算符的参量(更多的信息参见第 4 章“表达式”中的“下标运算符”)。

枚举器名称

枚举器的名称必须不同于任何其它的枚举器或相同范围中的变量。但是值可以重复。

枚举器常量的定义

枚举器被认为是在其初始化器之后立即被定义的。因此,它们可被用于初始化后续的枚举器。以下例子定义了一个枚举的类型,确保任何两个枚举器可用 OR 运算符组合在一起:

```
enum  FileOpenFlags
{
```

```
    OpenReadOnly  =1,
    OpenReadWrite = OpenReadOnly  << 1,
    OpenBinary= OpenReadWrite << 1,
    OpenText  = OpenBinary<< 1,
    OpenShareable = OpenTest  << 1
};
```

在这个例子中,上面的枚举器初始化每个后续的枚举器。

转换和枚举的类型

因为枚举的类型是整型,所以任何枚举元通过整型升级可被转换到另一种整型。
考虑以下例子:

```
enum Days
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

```
int i;
Days d=Thursday;

i=d;  //通过整型升级进行转换
cout << "i=" <<  i << "\n;
```

但是从任何整型到一个枚举的类型没有隐式的转换。因此(继续前面的列子),以下语句是错的:

```
D=6;   //错误地试图把 D 设置为 Saturday
```

象这样的赋值,没有隐含的转换存在的地方,必须使用造型转换:

```
d=(Days)6; //显式造型转换到 Days 类型
d=Days(4); //显式造型转换到 Days 类型
```

前面的例子显示了与枚举器一致的值的转换。没有任何机制能防止你转换一个不与任一个枚举器一致的值。例如:

```
d=Days(967);
```

某些这类转换可以工作,但是不保证结果值为枚举器之一。此外,如果枚举器的尺寸太小,不足以保持正被转换的值,则存储的值可能不是你所希望的。

连接规格

术语“连接规格”指以不同语言书写的连接函数(或过程)的协议。以下调用规定受影响:

- 名称的大小写字母敏感性。

- 名称的修饰在 C 中,编译器给名称加下划线作前缀。这通常被称为“修饰”。在 C++中,名称修饰被用于通过连接阶段保留类型信息(参见联机“Microsoft Visual C++ 6.0 程序员指南”中的“修饰的名称”)。
- 在堆栈上的所期望的参量的顺序。
- 在函数返回时调节堆栈的责任,被调用的函数或调用函数应负责。
- 隐藏的参量的传递(任隐藏的参量是否传递)。

语法

连接规格

    extern 字符串文字{说明表 opt}
    extern 字符串文字 说明

说明表:

    说明
    说明表

连接规格通过使用已存在代码逐步促进 C 代码向 C++的移植。

Microsoft 特殊处→

当前由 Microsoft C++支持的唯一连接规格是“C”和“C++”。

Microsoft 特殊处结束

以下代码用 C 连接规格函数 atoi 和 atol:

```
extern "C"
{
    int atoi(char *string);
```

```
    long atol(char *string);
}
```
这些函数的调用使用 C 连接完成,用这两个说明可获得相同的结果:
```
extern "C" int atoi(char *string);
extern "C" long atol(char *string);
```
Microsoft 特殊处→
所有 Microsoft C 标准包含文件使用条件编译命令去检测 C++编译,当检测到一个 C++编译时,则原型被包含在如下的一个 extern “C”命令中:
```
//sample.h
#if defined (__cpluslpus)
extern "C"
{
#endif

//函数说明

if defined(__cplusplus)
}
#endif
```
Microsoft 特殊处结束
你不需要在标准包含文件中将函数说明为 extern“C”。
如果一个函数被重载了,相同名称的函数仅有一个可有连接指示符(更多的信息

参见第 7 章“说明符”中的“函数重载”)。
表 6.3 给出了各种连接规格是如何工作的。

表 6.3 连接规格的作用

规格	作用
在一个对象上	仅影响那个对象的连接
在一个函数上	影响那个函数和在其中说明的所有函数或对象的连接
在一个类上	影响类中说明的所有非成员函数和对象的连接

如果一个函数有不止一个的连接规格,它们必须一致。将函数说明为具有 C 和 C++两种连接是错误的,而且,如果一个函数的两个说明出现在一个程序中,一个有连接规格,另一个没有,则带连接规格的说明必须在首位。已有连接规格的函数的任何冗余的说明均给出在第一个说明中指定的连接。例如:

```
extern "C" int CFunc1();
...
int CFunc1();//重新说明是良性的,C 连接被保留
int CFunc2();
....
extern "C" cfunc2();  //错误:不是 CFunc2 的第一个说明,不能包含连接指示符。
```

在一个复合连接规格符({})的体内被显式地说明为 static 的函数和对象被看成是静态函数或对象;连接规格被忽略,其它函数和对象表现得如同使用 extern 关键字说明的一样(关于 extern 关键字详见“存储类说明符”)。

模板规格

template 说明指定一个参量化的类或函数集。
注意:更多的信息参见联机“Microsoft Visual C++6.0 程序员指南”中的“模板论题”。
语法
模板规格:
    template <模板参量表> 说明
模板参量表:
    模板参量
    模板参量表,模板参量
模板参量:
    类型参量
    参量说明
类型参量:
    class 标识符
    typename 标识符
说明一个函数或一个类。对函数模板,每个模板参量在正被说明的函数模板参量表中至少出现一次。
模板参量表是由模板函数使用的参量表,指示以下代码的哪些部分将改变。
例如:

```
template< class T,int i> class MyStack...
```

在这种情况下 template 能接收的一个类型(class T)和一个常数参量(int I)。template 将使用类型 T 和常量整数 i 于构造函数。在 MyStack 说明体内,你必须指出 T 标识符。
tempname 关键字可被用在模板参量表中,以下 template 说明是完全相同的:

```
template<class T1,class T2> class X...
template<typename T1,typename T2> class X...
```

template 参量的以下形式是允许的:

```
template<typename type> class allocator{} ;
template<typename type,
typename Allocator=allocator<Type>> class stack{
 };
stack<int> MyStack;
```

Visual C++支持模板参量表中的模板参量的重用。例如,以下代码现在是合法的:

```
class Y {...}
template<class T,T* pT> class X1 {...}
template<class,T1,class T2=T1> class X2{...};

Y aY;

X1<Y,&aY>x1;
X2<int> x2;
```

一个模板说明自身不产生代码,它指定类或函数的一个家族,当被其它代码引用时,其中的一个或多个将产生代码。
模板说明具有全局或名称空间范围。
Visual C++执行模板定义的语法检查。Visual C++的这个版本可以检测错误,以前的版本不能。编译器现在能检测定义了的但从未实例化的模板的语法错误。这是可用 Visual C++4.0 编译器编译的常见错误清单,但不是用 Visual C++ 5.0 或更新的编译器。

- 一个用户定义的类型在其被说明之前被用于一个模板说明中,但它是在第一次实例化或使用模板之前被说明的。例如:

```
template <class T> class X{
    //...
    Data m_data;    //Visual C++5.0 或之后的版本中,数据未定义
};

class Data {...}

void g(){X<int> x1;}
```

移去模板 X 之前的 Data 说明以解决这个问题。

- 一个成员函数在一个类模板外面被说明,而从未在类内部被说明。例如:

```
template<class T> class X{
    //在这里没有说明 mf
```

```
};

//此定义用 Visual C++4.0 时不产生错误
//但用 Visual C++5.0 或更新版本则会产生错误
template <class T>void x <T>::mf(){...};
```

- 一个类标识符被当作是一个普通类,除非说明为一个类模板。例如:以下代码用 Visual C++ 5.0 或更新版本时产生一个错误,但用 Visual C++ 4.0 则无错:

```
template <class T> class X{
   friend class Y<T>;  //语义分析为 Y 小于 T 大于

   Z<T> mf();   //语义分析为 Z 小于 T 大于
};

template<class T> class Y{...};
template<class T> class Z{...};
X<int> x;
```

为解决此问题,在 X 定义之前前向说明 Y 和 Z。

```
template<class T> class Y{...};

template<class T> class Z{......};
```

```
template<class T> class X{......};
```

引用一个模板

要引用一个模板类或函数使用以下语法。
语法
模板类名称：
    模板<模板参量表>
模板参量表：
    模板参量
    模板参量表,模板参量
模板参量：
    表达式
    类型名称
所有模板参量必须为常量表达式,如果对一个前面产生的模板没有准确的匹配,则编译器创建模板类或函数的一个新的实例(称为一个实例化),例如：

```
MyStack< unsigned long ,5> stack1;  //创建一个 unsigned long 的堆栈
MyStack< DWORD, 5> stack2;   //使用上面创建的代码
MyStack< char, 6> stack3;//产生新代码
Mystack< MyClass, 6> stack4;  //产生 MyClass 对象的堆栈
```

每个产生的模板化函数创建其自身的静态变量和成员。

函数模板

类模板定义一个相关的类家族,它们是基于实例化时传递给类的参量的。函数模板与类模板类似,但定义一个函数族。这里是一个交换两项的函数模板:

```
template<class T>void MySwap(T& a ,T& b)
{
    T c;
    c=a,a=b,b=c;
}
```

尽管此函数可以通过一个非模板化函数执行,使用 void 指针,模板版本是类型安全的。考虑以下调用:

```
int j=10;
int k=18;
CString Hello ="Hello,Windows!";
MySwap(j,k);  //可行
MySwap(j,Hello);  //错误
```

第二个 MySwap 调用触发了一个编译错误,因为编译器不能用不同类型的参量产生一个 MySwap 函数。如果使用了 void 指针,则两个函数调用都能正确编译,但函数在运行时不能正常工作。

一个函数模板属性参量的显式说明是允许的,例如:

```
template <class T> void f(T) {...}
void g(char j) {
```

```
  f<int>(j);    //产生 f(int)规格
}
```

当模板参量被显式说明时,正常的隐式转换完成将函数参量转换到相应的函数模板参量的类型。在上面例子中,编译器将把(char j)转换到 int 类型。

成员函数模板

成员函数模板在说明了一个模板化类后,定义成员函数作为函数模板,例如:

```
template<class T,int i> class MyStack
{
    T* pStack;
    T StackBuffer[i];
    int cItems=i * sizeof(T);
public:
     MyStack(void);
     void push(const T item);
     T& pop(void);
};

template <class T,int i> MyStack<T,i>::MyStack(void)
{...};
template <class T,int i> void MyStack<T,i>::push(const  T item)
{...};
```

```
template <class T,int t> T& MyStack<T,i>::pop(void)
{...};
```
注意,未包括模板参量的构造函数的定义列出了两次。

显式实例化

显式实例化让你能够创建一模板化类或函数的一个实例,而不需要在你的代码中实际使用它,因为当你创建使用模板作分布的库文件(.LIB)时这是很有用的,非实例化的模板定义不放入目标文件(.OBJ)。
以下为 int 变量和 6 项目显式实例化 MyStack:
```
template class MyStack<int, 6>;
```
此语句创建一个不保留任何对象存储的 MyStack 的一个实例,代码是为所有成员产生的。
以下仅显式实例化构造成员函数:
```
template MyStack<int, 6>::MyStack(void);
```
Visual C++6.0 支持函数模板的显式实例化。5.0 以前的版本仅支持类模板显式实例化。例如,以下代码现在是合法的:
```
template <class T>void f(T){...}

//用显式指定的模板参量“int”实例化 f
template void f<int>(int);

//用推导的模板参量“char”实例化 f
```

```
template void f(char);
```
Microsoft 特殊处→
你可以使用 extern 关键字去阻止成员的自动实例化,例如:
```
extern template class MyStack<int, 6>;
```
类似地,你可以标注指定的成员为外部的且不象下面这样实例化:
```
extern template MyStack<int, 6>::MyStack(void);
```
注意:说明中的 extern 关键字仅适用于类体外定义的成员函数。在类说明内部定义的函数被认为是联编函数,且总是实例化的。
Microsoft 特殊处结束

与其它实现的不同

Microsoft 特殊处→
模板没有正式地被标准化,因此不同的 C++编译器销售商对它们的实现不同。以下清单给出了本版本的 Visual C++和其它编译器之间的一些不同之处。注意此清单在以后版本的编译器中还会有所改变。

- 编译器不能在其所定义的模块外面实例化一个模板。
- 模板不能与用 __declspec(dllimport)或 __declspec(dllexport)说明的函数一起使用。
- 所有模板参量必须是非模糊类型的,准确与模板参量表相匹配。例如:

```
template<class T>T check(T);
template<class S>void watch(int (*)(S));
```

```
watch(check);  //错误
```
编译器应该以 int check(int)形式实例化 check 模板化函数,但不能继续这样推理。

- 友元函数必须在被用于模板类之前说明。你不能在一个类定义中定义一个友元函数。这是因为友元函数可以是一个模板函数,可能会导致非法的嵌套的模板定义。

Microsoft 特殊处结束

名称空间

C++语言提供一个单个全局名称空间。这可能导致全局名称冲突问题。例如,考虑这样两个 C++头文件:

```
//one.h
char func(char);
class String {...};

//somelib.h
class String {...};
```

有了这些定义,则不可能在一个单个程序中同时使用这两个头文件;String 类会发生冲突。

一个名称空间是一个说明性区域,为在其内说明的任何名称添加一个额外的标识符。额外的标识符使得一个名称与程序中其它位置说明的名称冲突的可能性减

少,有可能在分开的名称之间使用相同的名称而没有冲突,即使名称出现在同一个转换单元中,只要它们出现在分开的名称空间中,每个名称将是唯一的,因为有额外的名称空间标识符。例如:

```
//one.h
namespace one
{
    char func(char);
    chass String {...};
}

//somelib.h
namespace Somelib
{
  class String {...}
}
```

现在类名称将不会冲突,因为它们分别变为 ong:String 和 Somelib::String。
在所有名称空间之外,在一个转换单元的文件范围中的说明还是全局名称空间的成员。

名称空间说明

namespace 说明将一个名称指定并赋值到一个说明性区域。
语法

源名称空间名称：
    标识符
名称空间定义：
    源名称空间定义
    扩充名称空间定义
    未命名的名称空间定义
源名称空间定义：
    namespace 标识符{名称空间体}
扩充名称空间定义：
    namespace 源名称空间名称{名称空间体}
未命名的名称空间定义：
    namespace {名称空间体}
名称空间体：
    说明序列$_{opt}$
在一个源名称空间定义中的标识符在其被使用的说明区域内必须是唯一的。标识符是名称空间中的名称并用于其成员的引用。随后,在那个说明性区域,它被看作是一个源名称空间名称。
一个名称空间定义的说明性区域是它的名称空间体。
一个名称空间可以包含数据和函数说明,说明序列是这些被说成是这些名称空间的成员的说明的一个表。

未命名的名称空间

一个未命名的名称空间定义的行为好象它被：

namespace *unique* {名称空间体}

using mamespace *unique*;

替换了。

每个未命名的名称空间有一个标识符，用 *unique* 表示，与整个程序中的所有其它标识符不同。例如：

```
namespace {int i;}  //unique::i
void f() {i++;} //unique::i++

namespace A {
namespace {
int I;    //A::unique::i
int j;    //A::unique::j
   }
}

using namespace A;

void h()
{
    I++;    //错误:unique::i 或 A::unique::i
```

```
    A::i++;//错误:A::i 未定义
    j++;    //A::unique::j++
}
```

未命名的名称空间是变量的静态说明的替换。它们允许变量和函数在整个转换单元中可见,在外部不可见。尽管在一个未命名的名称空间中的实体也许有外部连接,但它们被一个唯一的名称有效地限定于它们的转换单元中,因此永远不能在任何其它转换单元中可见。

名称空间定义

一个名称空间定义可被嵌套在另一个名称空间定义内,每个名称空间定义必须出现在文件范围或立即在另一个名称空间内。
例如:

```
namespace A {
    int j=3;
    int f(int k);
}

namespace Outer {
    int n=6;
    int func(int num);

    namespace Inner {
```

```
        flont f=9.993;
    }
}

void main()
{
    namespace local {...}    //错误:不在全局范围
    .
    .
    .
}
```
不象其它说明性区域,一个名称空间的定义可被分在一个单独转换单元的几个部分中。
```
namespace A {
    //说明 namespace A 变量
    int i;
    int j;
}

namespace B {
    ...
}
```

```
    .
    .
    .
namespace A {
    //说明 namespace A 函数
    void func(void);
    int int_func(int i);
}
```
当一个名称空间以这种方式继续,在其初始定义之后,连续被称为一个扩充名称空间定义。

定义名称空间成员

一个名称空间的成员可被定义在那个名称空间内,例如:
```
namespace X { void f(){} }
```
一个名称空间的成员可以通过定义名称的显式限定定义在该名称空间的外面。但是正被定义的实体必须已在名称空间中说明了,此外,定义必须出现在包含说明的名称空间的一个名称空间中说明处之后。例如:
```
namespace Q{
    namespace V{
      void f();
   }
```

```
    void V::f(){}   //可行
    void V::g(){}   //错误,g()还不是 V 的一个成员

    namespace V{
        void g();
   }
}
```

名称空间别名

一个名称空间别名是一个 namespace 的一个备用名称。
语法
名称空间别名:
    标识符
名称空间别名定义:
    namespace 标识符=限定名称空间指示符;
限定名称空间指示符:
    ::$_{opt}$ 嵌套的名称指示符 $_{opt}$ 类或名称空间名称
一个名称空间别名定义为一个名称空间说明了一个备用名。标识符是限定名称空间指示符的同义词并成为一个名称空间别名,例如:

```
namespace a_very_long_namespace_name {...}
namespace AVLNN=a_very_long_namespace_name;
```

//AVLNN 现在是 a_very_long_namespace_name 的一个名称空间别名
一个名称空间名称不能与同一说明性区域中任何其它实体完全相同,此外,一个全局名称空间名称不能与一个给定程序中的任何其它全局实体名称相同。

using 说明

using 说明向 using 说明出现的说明性区域中引进一个名称。名称成为一个别的地方说明的一个实体的同义词,它允许一个特定的名称空间的单个名称不用显式限定而被使用。
这与 using 命令相反,那个允许一个名称空间中的所有名称不用限制而被使用。更多的信息参见下节“ Using 命令 ”。
语法
using 说明:
    using::$_{opt}$ 嵌套的名称指示符 非限定的 id
    using::非限定的 id
一个 using 说明可用于一个类定义中,例如:

```
class B
{
    void f(char);
    void g(char);
};

class D:B
```

```
{
    using B::f;
    void f(int) {f('c');}   //调用 B::f(char)
    void g(int) {g('c');}   //递归地调用 D::g(int)
                            //仅 B::f 正被使用
};
```

当被用于说明一个成员,一个 using 说明必须指向一个基类的成员。例如:

```
class C
{
    int g();
};

class D2:public B
{
    using B::f;  //可行:B 是 D2 的基(类)
    using C::g;  //错误:C 不是 D2 的基(类)
};
```

使用 using 说明所说明的成员可以用显示限定来引用。::前缀指向全局名称空间。例如:

```
void f();

namespace A
```

```
{
    void g();
}
namesapce X
{
    using::f;//全局 f
    using A::g;  //A 的 g
}

void h()
{
    X::f();//调用 ::f
    X::g();//调用 A::f
}
```

就如同用任何说明一样,一个 using 说明仅在多重说明允许的地方可被重复性使用。

```
namespace A
{
    int i;
}

void f()
```

```
{
    using A::i;
    using A::i;  //可行:重复说明
}

class B
{
protected:
    int i:
};

class X:public B
{
public:
    using B::i;
    using B::i;  //错误:类成员不能被多次说明
};
```

当作了一个 using 说明时,由说明创建的同义词仅指示在 using 说明点有效的定义,在 using 说明之后加到一个名称空间的定义是无效的同义词。例如:

```
namespace A
{
    void f(int);
```

```
}

using A::f;    //f 仅是 A::f(int)的同义词

namespace A
{
    void f(char);
}

void f()
{
    f('a');  //指向 A::f(int),尽管 A::f(char)存在
}

void b()
{
    using A::f;  //指 A::f(int) AND A::f(char)
    f('a');  //调用 A::f(char);
}
```

由一个 using 说明定义的一个名称是其源名称的一个别名。它不影响源说明的类型、连接和其它属性。

如果一个单个名称的一组局部说明和 using 说明在一个说明性区域中给出,则它

们必须指相同的实体,或它们必须都指函数。例如:

```
namespace B
{
    int i;
    void f(int);
    void f(double);
}

void g()
{
    int i;
    using B::i;  //错误:i 说明了两次
    void f(char);
    using B::f;  //可行:每个 f 是一个函数
}
```

在上面的例子中,using B::i 语句导致 int i 在 g()函数中被第二次说明,using B::f 语句与 f(char)函数不冲突,因为 B::f 引进的函数各具有不同的参数类型。一个局部函数说明不能与由 using 说明引进的函数具有相同的名称和类型。例如:

```
namespace B
{
    void f(int);
```

```
    void f(double);
}

namespace C
{
    void f(int);
    void f(double);
    void f(char);
}

void h()
{
    using B::f;  //引进 B::f(int)和 B::f(double)
    using C::f;  //C::f(int),C::f(double)和 C::f(char)
    f('h');  //调用 C::f(char)
    f(1);//错误;模糊的:B::f(int)还是 C::f(int)?
    void f(int); //错误:与 B::f(int)和 C::f(int)冲突
}
```

当一个 using 说明从一个基类向一个派生类范围引进一个名称,在派生类中的成员函数使在基类中具有相同名称和参量类型的虚拟成员函数无效。例如:

```
struct B
{
```

```
    virtual void f(int);
    virtual void f(char);
    void g(int);
    void h(int);
};

struct D:B
{
    using B::f;
    void f(int);  //可行:D::f(int)使 B::f(int)无效

    using B::g;
    void g(char);  //可行:没有 B::g(char)

    using B::h;
    void h(int);  //错误: D::h(int) 与 B::h(int)冲突
                  //B::h(int)不是虚拟的
};

void f(D* pd)
{
    pd->f(1);      //调用 D::f(int)
```

```
    pd->f('a')    //调用 B::f(char)
    pd->g(1)      //调用 B::g(int)
    pd->g('a')    //调用 D::g(char)
}
```

在一个 using 说明中提到的一个名称的所有实例必须是可访问的。特别地,如果一个派生类使用一个 using 说明去访问一个基类的一个成员,则成员名称必须是可访问的。如果名称是一个重载了的成员函数,则所有命令的函数必须是可访问的。例如:

```
class A
{
private:
    void f(char);
public:
    void f(int);
protected:
    void g();
};

class B : public A
{
     using A::f;    //错误:A::f(char)是不可访问的
public:
```

```
    using A::g;  //B::g是A::g的一个公共同义词
};
```

关于成员的访问性的更多信息参见第10章“成员访问控制”。

using命令

using命令允许一个namespace中的名称在使用时不用把名称空间名称作为一个显式限定符。与using说明相反,那是允许一个单个的名称不用限定而被使用,而using命令允许一个名称空间中的所有名称不用限定而被使用。更多的信息参见本章前面的“using说明”。

语法

using命令:

```
    using namespace::opt 嵌套的名称指示符opt 名称空间名称
```

using命令的意图是在说明函数和变量时允许使用唯一的、说明性的名称,每次访问函数或变量时不要求用完整的名称。当然,完整的限定名还是可用于保持清晰度。非限定名可从using命令打开的点后被使用。如果一个名称空间在一个using命令给定后被扩充,则名称空间附加的成员可被使用,且在扩充的名称空间定义之后,不用限定。例如:

```
namespace M
{
    int i;
}
```

```
using namespace M;

namespace N
{
    int j;
    double f() { return M::d }   //错误;M::d 还不存在
}

namespace M   //namespace 扩充
{
    double d;
}   //现在 M::d 可使用了
```

对一个 using 命令,当被用于另一个名称空间时有可能引进冲突的名称。例如:

```
namespace M
{
     int i;
}

namespace N
{
     int i;
     using namespace M;   //这里没错
```

```
}
    .
    .
    .
void f()
{
    using namespace N;
    i=7;//错误:模糊的:M::i 还是 N::i?
}
```

在此例中,将 M::i 引进 namespace N 并没有隐藏 N::i 说明,而是在使用 N::i 时产生了模糊性,在这种情况下,using 命令很容易引入不想要的模糊性。考虑下面代码段:

```
namespace D
{
    int d1;
    void f(int);
    void f(char);
}

using namespace D;

int d1;  //与 D::d1 不冲突
```

```
namespace E
{
    int e;
    void f(int);
}

namespace D//名称空间扩充
{
    int d2;
    using namespace E;
    void f(int);
}

void f()
{
    d1++; //错误:模糊的:    ::dl 还是 D::d1?
    ::d1++;    //可行
    D::d1++;   //可行
    d2++; //可行:D::d2
    e++;   //可行:E::e
    f(1); //错误:模糊的:D::F(int)还是 E::f(int)?
```

```
    f('a');    //可行:D::f(char)
}
```
当一个变量在 using 命令后被引用时,相同名称的局部变量比在指定的名称空间中说明的那个具有更高的优先级。例如:
```
namespace N{
  int data=4;
}

void f(bool flag){
    int data=0;

    if(flag){
       using namespace N;

       printf("data=%d\n",data);
    }
}

void main(){
    f(true);
}
```
在上面代码中,在 printf 语句中引用的 data 变量是一个局部变量,初始化为 0,

替代在名称空间 N 中初始化的变量。输出是 data=0 而不是 data=4。
using 命令名称空间的出现,在下面的例子中给出了限定名查找的方法:

```
namespace A{
    int flag=0;
}

namespace B{
    using namespace A;
}

namespace C{
    using namespace A;
    using namespace B;
}

void main(){
    printf("C::flag=%d\n",C::flag);
}
```

由于在名称空间 C 中的名称空间 using 命令使 C::flag 限定名为 A::flag。

显式限定符

在一个类或名称空间中的名称可通过使用一个显式限定符进行访问。

语法

id 表达式：

    未限定的 id

    限定的 id

嵌套的名称指示符：

    类或名称空间名称

    ::嵌套的名称指示符 opt

类或名称空间名称：

    类名

    名称空间名称

名称空间名称：

    源名称空间名称

    名称空间别名

这与使用范围运算符去解决一个类成员的访问是非常相似的。更多的信息，参见第 4 章“表达式”中的限定名。

第 7 章　说　明　符

一个“说明符”是一个说明的一部分,用以命名一个对象、类型或函数。说明符是出现在说明中的以一个或多个被逗号分开的名称,每个名称可有一个相关的初始化器。

语法

说明符表:

　　初始说明符

　　说明符表,初始说明符

初始说明符:

　　说明符　初始化器 $_{opt}$

本章包括以下主题:

- 说明符概述
- 类型名称
- 抽象说明符
- 函数定义
- 初始化器

说明符概述

说明符是指定名称的一个说明的组成成分。说明符还可修饰基本的类型信息,使得名称成为函数或者对象或函数的指针(在第 6 章“说明”中讨论的指示符传送诸如类型和存储类属性。在本章以及附录 B“Microsoft 特殊修饰符”中讨论的修饰符用以修饰说明符)。

图 7.1 给出了两个名称 szBuf 和 strcpy 的一个完整说明,并提出了说明的各组成部件。

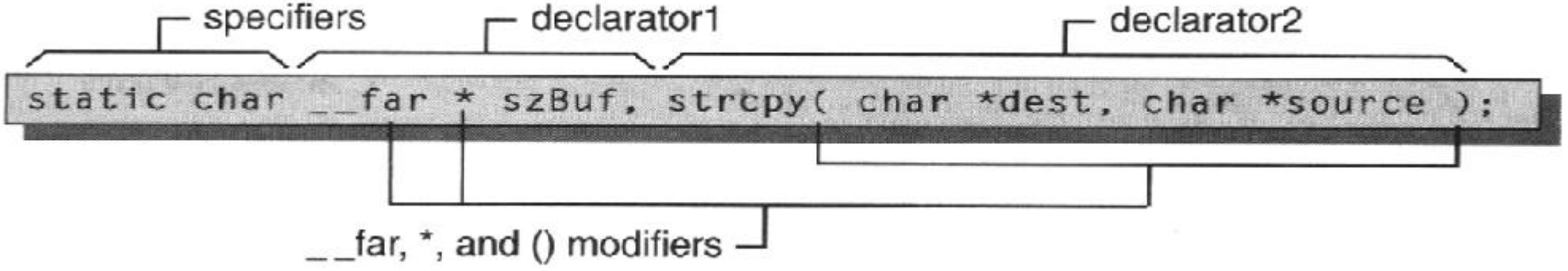


图 7.1 指示符、修饰符和说明符

Microsoft 特殊处→

大多数 Microsoft 扩充关键字可被用作形成派生类型的修饰符;它们不是指示符或说明符(见附录 B“Microsoft 特殊修饰符”)。

Microsoft 特殊处结束

语法

说明符:

dname
ptr 运算符 说明符
说明符(参量说明表) *cv* 修饰符表
说明符[常量表达式 opt]
(说明符)

ptr 运算符:
* *cv* 限定符表 opt
& *cv* 限定符表 opt
完整的类名称::**cv* 限定符表 opt

cv 限定符表:
cv 限定符 *cv* 限定符表 opt

cv 限定符:
const
volatile

cv 修饰符表:
cv 限定符 cv 修饰符表 opt
pmodel cv 修饰符表 opt

dname:
名称
类名称
～类名称
typedef 名称

限定类型名称

说明符出现在说明语法中可选的指示符表(*decl-specifiers*)后面,这些指示符在第6章“说明”中讨论。一个说明可包含不止一个说明符,但每个说明符仅说明一个名称。以下说明样本显示了指示符和说明符是如何组合构成一个完整的说明的:

```
const char *pch,ch;
```

在前面这个说明中,关键字 const 和 char 组成了指示符表。列出了两个说明符 *pch 和 ch。

然后一个说明简化的语法如下,其中 const char 是类型,*pch 和 ch 是说明符:

```
type declarator1[,declarator2[...,declaratorn]];
```

当联结于一个说明符表的元素没有产生想要的结果,你可以使用括号使其表达清晰。但一个更好的技术是使用 typedef 或括号组合和 typedef 关键字。

考虑说明一个函数的指针数组。每个函数必须遵从相同的协议,从而参量和返回值是已知的:

```
//函数返回类型 int,带一个 char *类型参量
typedef int(*PIFN)(char*);

//说明一个 7 个函数指针的数组,返回 int,带一个 char*类型参量
PIFN pifnDispatchArray[7];
```

等价的说明可以不用 typedef 说明书写,但它太复杂了,而使得错误的潜在性超出了它的任何效益:

```
int (*pifnDispatchArray[7](char*);
```

类型名称

类型名称按以下方式被用于某些说明符中：

- 在显式转换中
- 作为 sizeof 运算符的参量
- 作为 new 运算符的参量
- 在函数原型中
- 在 typedef 语句中

一个类型名称构成类型指示符，如第 6 章“说明”中所描述以及下节“抽象说明符”所讨论的。

在以下例子中，函数 strcpy 的参量通过使用它们的类型名而提供。在 source 参量情况中，const char 是指示符，*是抽象说明符：

```
static char *szBuf, *strcpy(char *dest, const char *source);
```

语法

类型名称：

类型指示符表 抽象说明符 $_{opt}$

类型指示符表：

类型指示符 类型指示符表 $_{opt}$

抽象说明符：

ptr 运算符 抽象说明符 $_{opt}$

抽象说明符 $_{opt}$(参量说明表) *cv* 限定符表 $_{opt}$

抽象说明符 $_{opt}$[常量表达式 $_{opt}$]

(抽象说明符)

抽象说明符

一个抽象说明符是一个说明符,其中省略了标识符(相关的信息参见前一节“类型名称”)。
本节讨论以下抽象说明符:

- 指针
- 引用
- 成员指针
- 数组
- 函数
- 缺省参量

一个抽象说明符是一个不说明一个名称的说明符,即标识符被省略。例如:

```
char *
```

说明类型“指向 char 类型的指针”。这个抽象说明符可被用于如下所示的一个函数原型中:

```
char *strcmp(char *, char *);
```

在此原型(说明)中,函数的参量被指定为抽象说明符。以下是一个更复杂的抽象说明符,说明“指向一个带两个参量,且都是 char*类型的函数的指针”,返回类型 char *:

```
char *(*)(char*, char*)
```

由于抽象说明符完全地说明了一个类型,所以构成下面形式的表达式是合法的:

```
//获取一个 char 类型的 10 个指针的数组的尺寸
size_t nSize=sizeof(char *[10]);

//分配一个指向没有返回值且不带参量的函数的指针
typedef void(PVFN *)();
PVFN *pvfn=new PVFN;

//分配一个返回类型为 WinStatus 且带一个 WinHandle 类型的参量的
//函数的指针数组
typedef WinStatus (PWSWHFN *)(WinHandle);
PWSWHFN pwswhfnArray[]=new PWSWHFN[10];
```

模糊性解决方法

要执行从一个类型到另一个类型的显式转换,你必须使用强制转换,指出所希望的类型名称。有些类型造型转换导致语法模糊性,以下函数形式类型强制转换是模糊的:

```
char *aName(String(s));
```

不清楚它是一个函数说明,还是一个将函数形式强制转换作为初始化器的一个对象说明:它可能说明一个返回类型为 char*,且带一个 String 类型参量函数,或是可能说明对象 aName,并用 s 造型转换到类型 string 的值对其进行初始化。

如果一个说明可被认为是一个有效的函数说明,则它被这样对待。仅当它不可能

是一个函数说明——即,如果它是语法错误的——是一个语句检测它是否是一个函数形式类型造型转换。因此,编译器把语句作为一个函数的说明,并忽略标识符 S 周围的括号。另一方面,语句:

```
char *aName((String)s);
```

和

```
char *aName=String(s);
```

是对象的清楚的说明,且一个用户定义的从 String 类型到 char*类型的转换被调用以执行 aName 的初始化。

指　针

指针是使用说明符语法进行说明的:

*cv 限定符表 $_{opt}$ d*name*

一个指针拥有一个对象的地址。那么一个完全的说明是:

说明指示符 *cv 限定符表 $_{opt}$ dname;

这个说明的一个简单例子为:

```
char *pch;
```

上面的说明指定 pch 指向一个 char 类型的对象。

const 和 volatile 指针

const 和 volatile 关键字改变指针是如何被对待的。const 关键字指定指针在初始化之后不能被修改,指针从那以后是被保护的,不能被修改。

volatile 关键字指定其后的名称相关的值可以通过不是用户应用中的那些动作

而进行修改,因此,volatile 关键字对在可被多个过程访问的共享存储器中或与中断服务例程通讯用的全局数据区域中说明对象是非常有用的。
当一个名称被说明为 volatile 时,编译器在它每次被程序访问时从存储器中重新装入其值,这极大地降低了可能的优化。但是当一个对象的状态可能出乎意料地改变时,它是确保程序可预言的执行的唯一方法。
要说明由 const 或 volatile 指针指向的对象,使用这种形式的说明:

```
const char *cpch;
volatile char *vpch;
```

要说明指针值,即指针中存储的实际地址为 const 或 volatile,使用这种形式的说明:

```
char * const    pchc;
char * volatile pchv;
```

C++语言不允许对声明为 const 的指针或对象的修改赋值,这种赋值将去除对象或指针被说明时用的信息,因此与源说明的意图相冲突,考虑以下说明:

```
const char cch='A';
charch='B';
```

给定前面的两个对象的说明(cch 为 const char 类型,ch 为 char 类型),则以下说明/初始化是有效的:

```
const char     *pch1=&cch;
const char        *const  pch4=&cch;
const char     *pch5=&ch;
char           *pch6=&ch;
```

```
char            *constpch7=&ch;
const char *const  pch8=&ch;
```
以下说明/初始化是错误的：
```
char *pch2=&cch;    //错误
char *const pch3=&cch; //错误
```
pch2 的说明通过一个可能被修改而因此是不允许的常量对象说明了一个指针。pch3 的说明指定指针是常量，而不是对象，此说明以与 pch2 说明不被允许相同的理由而不被允许。

以下 8 个赋值显示了通过指针的赋值和对前面说明改变指针值的情况；现在，假定从 pch1 到 pch8 的初始化是正确的：
```
*pch1='A';   //错误：对象说明为常量
pch1=&ch;    //可行：指针未被说明为常量
*pch2='A';   //可行：正常的指针
pch2=&ch;    //可行：正常的指针
*pch3='A';   //可行：对象未被说明为常量
pch3=&ch;    //错误：指针说明为常量
*pch4='A';   //错误：对象说明为常量
pch4=&ch;    //错误：指针说明为常量
```
说明为 volatile 或 const 和 volatile 混合的指针遵从相同的规则。指向 const 对象的指针常被用于如下的函数说明中：
```
char *strcpy(char *szTarget, const char *szSource);
```
前面的语句说明了一个函数 strcpy，带两个类型为“char 指针”的参量，且返回

一个指向 char 类型的指针。由于参量是由引用传递的,而不是通过值传递的,如果 szSource 没有被说明为 const,则函数可自由修改 szTarget 和 szSource。把 szSource 说明为 const 确保调用者 szSource 不能被调用的函数改变。
注意:由于存在一个从 *typename**到 const *typename**的标准转换,因此向 strcpy 传递一个类型 char*的参量是合法的。但是,反之不然,不存在隐式的转换从一个对象或指针去除 const 属性。
给定类型的 const 指针可被赋给相同类型的一个指针。但是,一个不是 const 的指针不能赋给一个 const 指针,以下代码给出了正确和错误的赋值:

```
int *const cpObject=0;
int *pObject;

void main()
{
    pObject=cpObject;  //可行
    cpObject=pObject;  //错误
}
```

引　用

引用使用说明符语法进行说明:
语法

&*cv* 限定符表 $_{\text{opt}}$ d*name*

一个引用拥有一个对象的地址,但语法上表现得象一个对象。一个引用说明包括

一个(可选的)指定符表,后跟一个引用说明符。

语法

说明指示符 &cv 限定符表 $_{opt}$ dname;

考虑用户定义的类型 Date:

```
struct Date
{
    short DayOfWeek;
    short Month;
    short Day;
    short Year;
};
```

以下语句说明一个 Date 类型对象和那个对象的一个引用:

```
Date Today;  //说明对象
Date& TodayRef=Today;//说明引用
```

对象名称 Today 和对象的引用 TodayRef 在程序中可完全相同地被使用:

```
Today.DayOfWeek=3;   //Tuesday
TodayRef.Month=7;//July
```

引用类型函数参量

通常向一个大对象传递引用比传递函数更有效。这允许编译器在保持已被用于访问对象的语法时传递对象的地址。考虑下面使用 Date 结构的例子:

```
//从阳历(格里历)日期创建一个 DDDYYYY 形式的 Julian 日期
```

```
long JulianFromGregorian(Date& GDate)
{
    static int cDaysInMonth[]={
    31,28,31,30,31,30,31,31,30,31,30,31
    };
    long JDate;

    //为已经过去的月份加天数
    for (int i=0;i<GDate.Month-1;++i)
       JDate+=cDaysInMonth[i];

    //为本月加天数
    JDate+=GDate.Day;

   //检查闰年
   if (GDate.year%100!=0  &&  GDate.Year%4==0)
      JDate++;

   //加年
   JDate*=10000;
   JDate+=GDate.Year;
```

```
    return JDate;
}
```

前面的代码显示了通过引用传递的结构成员使用成员选择运算符(.)而不是指针选择运算符(->)访问的。

尽管作为引用类型传递的指针遵从非指针类型语法,但它们保留了指针类型的一个重要特征:它们是可修改的,除非被说明为 const。因为前面代码的意图不是修改对象 GDate,所以一个更适宜的函数原型为:

```
long JulianFromGregorian(const Date& GDate);
```

此原型保证函数 JulianFromGregorian 不改变其参量。

任何采用带引用类型原型化的函数可在其位置接受相同的类型的一个对象,因为从 typename 到 typename&有一个标准的转换。

引用类型函数返回

函数可被说明为返回一个引用类型。作这种说明有两个原因:

- 正在返回的信息是一个足够大的对象,返回一个引用比返回一个拷贝效率更高。
- 函数的类型必须为一个 l 值。

正如通过引用向函数传递大的对象可以更有效一样,通过从函数返回大对象也可以更为有效。引用返回协议去除了拷贝对象到返回之前的临时地点的必要性。

引用返回类型在函数必须求值为一个 l 值时也是很有用的。大多数重载的运算符属于此类,特别是赋值运算符。重载的运算符包含在第 12 章“重载”中的“重载的运算符”中,考虑第 4 章“表达式”中的 Point 例子:

```
class Point
{
public:
    //定义“访问器”函数为引用类型
    unsigned& x();
    unsigned& y();
private:
    unsigned obj_x;
    unsigned obj_y;
};

unsigned& Point::x()
{
    return obj_x;
}
unisgned& Point::y()
{
    return obj_y;
}
    void main()
{
    Point ThePoint;
```

```
    //使用 x()和 y()作为 l 值
    ThePoint.x()=7;
    ThePoint.y()=9;

    //使用 x()和 y()作为 r 值
    cout << "x=" << ThePoint.x() << "\n"
     << "y=" << ThePoint.y() << "\n"
}
```

注意函数 x 和 y 说明为返回引用类型。这些函数可用于一个赋值语句的任意一边。引用类型的说明必须包含初始化器,除了以下情形之外:

- 显示 extern 说明。
- 一个类成员的说明。
- 在一个类内的说明。
- 一个函数的参量或函数返回类型的说明。

指针引用

指针引用和对象引用相同的方式进行说明。说明一个指针的引用产生一个象正常指针一样使用的可修改的值。以下代码例子说明使用一个指针的指针和一个指针的引用之间的差别:

```
#include <iostream.h>
#include <string.h>
```

```
//定义一个二叉树结构
struct BTree
{
    char *szText;
    BTree *Left;
    BTree *Right;
};
//定义一个指向树根的指针
BTree *btRoot=0;

int Add1(BTree **Root, char *szToAdd);
int Add2(BTree*& Root, char *szToAdd);
void PrintTree(BTree* btRoot);

int main(int argc, char *argv[])
{
    if(argc<2)
    {
      cerr << "Usage:Refptr[1|2]" << "\n";
      cerr << "\n\twhere:\n";
      cerr << "\t1 uses double indirection\n";
```

```
    cerr << "\t2 uses a reference to a pointer.\n";
    cerr << "\n\tInput is from stdin.\n";
    return 1;
  }

  char *szBuf=new char[132];

  //从标准输入设备读一文本文件,并构造一棵二叉树
  while(!cin.eof())
  {
  cin.get(szBuf,132,'\n');
  cin.get();
  if(strlen(szBuf))
    switch(*argv[1])
{
  //方法 1:使用双重间接引用
  case '1':
    Add1(btRoot,szBuf);
    break;

  //方法 2:使用指针引用
  case '2':
```

```
        Add2(btRoot,szBuf);
          break;
      default:
        cerr << "Illegal value "<< *argv[1]
         << "lsupplied for add method.\n"
         << "Choose 1 or 2.\n";
      return -1;
    }
  }

  //显示排序表
  PrintTree(btRoot);
  return 0;
}

//PrintTree:按序显示二叉树
void PrintTree(BTree* btRoot)
{
  //递归地遍历左子树
  if(btRoot->Left)
     PrintTree(btRoot->Left);
```

```
  //打印当前结点
  cout << btRoot->szText << "\n";

  //递归地遍历右子树
  if (btRoot->Right)
     PrintTree(btRoot->Right);
}

//Add1:向二叉树中加入一个结点
//使用双重间接引用
int Add1(BTree **Root,char *szToAdd)
{
    if ((*Root)==0)
{
  (*Root)=new BTree;
  (*Root)->Left=0;
  (*Root)->Right=0;
  (*Root)->szText=new char[strlen(szToAdd)+1];
  strcpy((*Root)->szText,szToAdd);
  return 1;
}
else if (strcmp((*Root)->szText,szToAdd)>0)
```

```
  return Add1(&((*Root)->Left),szToAdd);
else
  return Add1(&((*Root)->Right),szToAdd);
}

//Add2:向二叉树中加入一个结点
// 使用指针引用
int Add2(BTree*& Root, char *szToAdd)
{
   if (Root==0)
   {
     Root=new BTree;
     Root->Left=0;
     Root->Right=0;
     Root->szText=new char[strlen(szToAdd)+1];
     strcpy(Root->szText,szToAdd);
     return 1;
   }
  else if(strcmp(Root->szText,szToAdd)>0)
     return Add2(Root->Left, szToAdd);
  else
     return Add2(Root->Right, szToAdd);
```

```
}
```

在上面程序中,函数 Add1 和 Add2 在功能上是等价的(尽管它们以不同的方式被调用)。差别在于 Add1 使用双重间接引用而 Add2 则使用方便的指针引用。

成员指针

成员指针的说明是指针说明的特殊情况。

语法

说明指示符 类名称::*cv 限定符表$_{opt}$ dname;

指向一个类成员的指针与一个正常指针不同,因为它具有成员类型以及所属类的类型信息。一个正常指针仅仅标识(拥有地址)存储器中的单个对象。一个类成员指针标识类的任何实例中的那个成员。以下例子说明了一个类 Window 和一些成员数据指针:

```
class Window
{
public:
    Window(); //缺省的构造函数
    Window(int x1,int y1, //构造函数指定窗口大小
           int x2,int y2);
    BOOL SetCaption(const char *szTitle); //设置窗口标题
    const char *GetCaption(); //获取窗口标题
    char *szWinCaption;   //窗口标题
};
```

```
//说明一个指向数据成员 szWinCaption 的指针
char *Window::*pwCaption=&Window::szWinCaption;
```
在上面的例子中,pwCaption 是一个指向类 Window 的具有 char*类型的任何成员的指针。pwCaption 的类型是 char *Window::*。下面的代码段说明指向 SetCaption 和 GetCaption 成员函数的指针。
```
const char * (Window::*pfnwGC)()=&Window::GetCaption;
BOOL (Window::*pfnwSC)(const char *)=&Window::SetCaption;
```
指针 pfnwGC 和 pfnwSC 分别指向 Window 类的 GetCaption 和 SetCaption。该代码通过使用成员 pwCaption 指针直接拷贝信息到窗口标题中:
```
Window wMainWindow;
Window *pwChildWindow=new Window;
char   *szUntitled="Untitled-";
int cUntitledLen=strlen(szUntitled);

strcpy(wMainWindow.*pwCaption,szUntitled);
(wMainWindow.*pwCaption)[cUntitledLen-1]='1'; //与下面一样
//wMainWindow.SzWinCaption[]='1';
strcpy(pwChildWindow->*pwCaption,szUntitled);
(pwChildWindow->*pwCaption)[szUntitledLen-1]='2';//与下面一样
//pwChildWindow->szWinCaption[]='2';
```
.*和->*运算符(成员指针运算符)的差别是:.*运算符通过给定一个对象或对象

引用选取成员,而->*运算符通过一个指针选取成员(关于这些运算符的更多信息参见第 4 章“表达式”中的“带成员指针运算符的表达式”)。
成员指针运算符的结果是成员的类型,在此例情形中为 char*。
以下代码段使用成员指针调用成员函数 GetCaption 和 SetCaption:

```
//分配一个缓冲区
char szCaptionBase[100];

//拷贝主窗口标题到缓冲区中,并添加“[View 1]”.
strcpy(szCaptionBase,(wMainWindow.*pfnwGC)());
strcat(szCaptionBase,"[View 1]");

//设置子窗口标题
(pwChildWindow->*pfnwSC)(szCaptionBase);
```

成员指针的限制

一个静态成员的地址不是一个成员指针,它是一个静态成员实例的规则指针。但对一个给定类的所有对象仅存在一个静态成员实例,普通的取地址(&)和间接引用(*)运算符可以使用。

成员指针和虚拟函数

通过一个成员指针函数调用一个虚拟函数就如同函数已被直接调用一样:正确的函数在 v 表中查找并调用。以下代码显示了这是如何完成的:

```
class Base
{
public:
    virtual void Print();
};
void (Base::*bfnPrint)()=&Base::Print;

void Base::Print()
{
   cout << "Print function for class 'Base'\n";
}

class Derived:public Base
{
public:
    void Print();//Print 还是一个虚拟函数
};

void Derived::Print()
{
    cout << "Print function for class 'Derived'\n;
}
```

```
void main()
{
    Base  *bPtr;
    Base  bObject;
    Derived dObject;

    bPtr=&bObject;  //设置指向 bObject 的地址的指针
    (bPtr->*bfnPrint)();

    bPtr=&dObject;  //设置指向 dObject 的地址的指针
    (bPtr->*bfnPrint)();
}
```

该程序的输出为：

```
Print function for Class 'Base'
Print function for Class 'Derived'
```

虚拟函数工作的关键，和通常一样，是通过指向一个基类的指针调用它们（关于虚拟函数的更多信息参见第 9 章“派生类”中的“虚拟函数”）。

使用继承表示类成员指针

在类定义之前说明类成员指针影响产生可执行文件的大小和速度。表示一个类

成员指针所需要的字节数和解释所要求的代码依赖于类是否定义为不带、带单个和多个或虚拟的继承性。
总而言之,一个类用的继承越复杂,表示类成员指针所需的字节数就越多,解释指针所需的代码就越大。
如果你需要在类定义前说明一个类成员指针,你必须要么使用/vmg 命令行选项,要么使用相关的 pointers_to_members 编译指示,或者你可以通过使用 __single_inheritance、__multiple_inheritance 或 __virtual_inheritance 关键字指定在类说明中用到的继承性,因而允许在每个类基础上产生代码的控制。这些选项在下面进行解释。
注意:如果你总是在类定义之后说明一个类成员指针,则你不需要使用任何这些选项。
Microsoft 通过选择最紧凑的表示可能意图优化成员指针的表示和产生的代码。这要求定义成员指针类是基于成员指针被说明的地点的。pointers_to_members 编译指示允许放宽此限制,并控制指针大小和解释指针所需的代码。
语法
#pragma pointers_to_members(指针说明,[最通用的表示])
指针说明参量是指定你是否已在相关的函数定义之前或之后说明了一个成员指针。指针说明参量可以是 full_generality 或 best_case。
最通用的表示参量指定编译器可安全地用于引用一个转换单元中任一类成员指针的最小指针表示。此参量可以是 single_inheritance、multiple_inheritance 或 virtual_inheritance。带 best_case 参量的 pointers_to_members 编译指示是编译器的缺省值,如果你总是在说明一个类成员指针之前定义类,则你可以使

用此缺省值。当编译器遇到一个类成员指针的说明时,它已经知道类所用的继承种类。因而,编译器可以使用指针的最小可能表示,并产生为每种继承操作于指针上所要求的最少的代码量。这等价于在命令行使用/vmb 去为转换单元中所有的类指定最佳情形的表示。

如果你需要在定义类之前说明一个类成员指针,则使用带 full_generality 参量的 pointers_to_members 编译指示(如果你在两个不同的类中定义成员并使用成员指针相互引用,会引起这种需要。对这种相互引用类的情况,一个类必定会在其定义之前被引用)。编译器为成员指针使用最通用的表示。这等价于/vmg 编译器选项。如果你指定 full_generality,你必须还指定 single_inheritance、multiple_inheritance 或 virtual_inheritance。这等价于使用带/vms、/vmm 或/vmv 选项的/vmg 编译器选项。

带 full_generality、single_inheritance 参量的 pointers_to_members 编译指示(/vms 选项和/vmg 选项一起)指定类成员指针的最通用的表示是使用无继承或单个继承的那种。这是类成员指针最小的可能表示。如果一个成员指针被说明的类定义的继承性模式是多继承或虚拟继承,则编译器产生一个错误。例如,将下面语句:

```
#pragma pointers_to_members(full_generality,single_inheritance)
```

放在一个类定义之前,说明随后的所有类定义仅仅使用单继承。一旦指定了,用 pointers_to_members 编译指示指定的选项就不能改变了。

带 full_generality、multiple_inheritance 参量的 pointers_to_members 编译指示(/vmm 选项和/vmg 选项一起)指定类成员指针的最通用的表示是使用多继承的那种。这种表示比单继承所需要的大些。如果一个成员指针被说明的类定

义的继承模式是虚拟继承,则编译器产生一个错误。
带 full_generality、virtual_inheritance 参量的 pointers_to_members 编译指示(/vmv 选项和/vmg 选项一起)指定类成员指针的最通用的表示是使用虚拟继承的那种。根据指针的大小和解释指针所需要的代码看,这是最昂贵的选项。但是此选项从不产生错误,并且当 full_generality 参量被指定给 pointers_to_members 或/vmg 命令行选项被使用时,它是一个缺省的。
语法
等价的语言结构使用此语法:
类说明:
    类 继承类型 opt 类名称;
继承类型:
    __single_inheritance
    __multiple_inheritance
    __virtual_inheritance
正如这个例子中所示:
```
class __single_inhertance S;
int S::p;
```
不管编译器选项或编译指示,类 S 的成员指针将使用最小的可能表示。
你还可以显式给出具有前向说明的类的成员指针表示的一个前向说明。
注意:类成员指针表示的相同的前向说明应该出现在说明那个类的成员指针的每个转换单元中,而且说明应该出现在成员指针被说明之前。

数　组

一个数组是相似对象的一个集合。一个数组最简单的情况是一个向量。C++为固定大小数组的说明提供一种方便的语法：

语法

说明指示符 dname[常量表达式$_{opt}$];

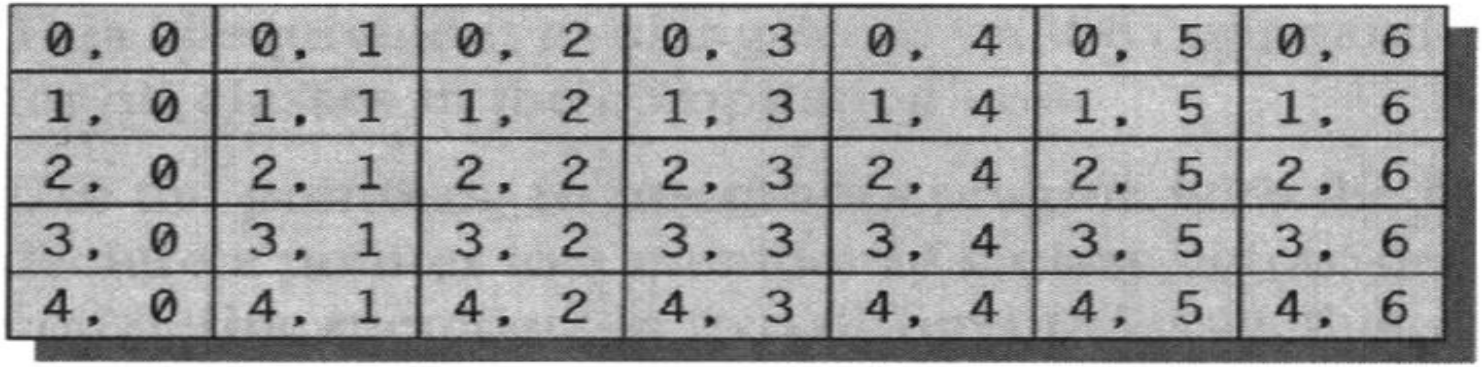
数组中元素个数由常量表达式给定，数组的第一个元素是第 0 号元素，最后一个元素是第(n-1)号元素，其中 n 是数组的大小。常量表达式必须是一个整型，且必须大于 0。一个 0 尺寸数组仅当数组是一个 struct 或 union 的最后一个域，且 Microsoft 扩充(/Ze)时才是合法的。

数组是派生类型，因此可以由除函数、引用和 void 以外的任何其它派生的或基本的类型构成。

从其它数组构成的数组是多维数组。这些多维数组通过按顺序放置多个[常量表达式]指定，例如，考虑这个说明：

```
int i2[5][7];
```

它指定了一个 int 类型数组，概念上安排在一个 5 行 7 列的二维矩阵中，如图 7.2 所示。



0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6
4, 0	4, 1	4, 2	4, 3	4, 4	4, 5	4, 6

图 7.2　多维数组的概念性布局

在带有初始化器表(如初始化器中所描述的)的多维数组说明中,第一维指定边界的常量表达式可以省略。例如:

```
const int cMarkets=4;

//说明一个浮点类型表示运输费用
double TransportCosts[][cMarkets]=
  { {32.19,47.29,31.99,19.11},
  {11.29,22.49,33.47,17.29},
  {41.97,22.09,9.76,22.55} };
```

上面的说明定义了一个 3 行 4 列的数组;行表示工厂,列表示工厂输送到的市场,值是从工厂到市场的运输费用。数组的第一维省略了,但编译器通过检查初始化器将它填上。

多维数组的第一维的省略边界指定的技术还可用于函数说明中,如下所示:

```
#include <float.h>              //包括 DBL_MAX
```

```
#include <iostream.h>

const int cMkts=4;

//说明一个浮点类型表示运输费用
double TransportCosts[][cMkts]=
    { {32.19,47.29,31.99,19.11},
    {11.29,22.49,33.47,17.29},
    {41.97,22.09,9.76,22.55} };
//计算未指定的维数的大小
const int cFactories=sizeof TransportCosts /
                  sizeof (double[cMkts]);
double FindMinToMkt(int Mkt,double TransportCosts[][cMkts],int cFacts);

void main(int argc,char *argv[])
{
     double MinCost;
     MinCost=FindMinToMkt(*argv[1]-'0',TransportCosts,cFacts);
     cout << "The minimum cost to Market" << argv[1] << "is:"
         << MinCost << "\n";
}
```

```
double FindMinToMkt(int Mkt,double TransportCosts[][cMkts],int cFacts)
{
    double MinCost=DBL_MAX;
    for(int i=0;i<cFacts;++i)
        MinCost=(MinCost<TransportCosts[i][Mkt])?
                MinCost:TransportCosts[i][Mkt];
    return MinCost;
}
```

函数 FindMinToMkt 编写成添加新工厂,不需要改动任何代码,只要一次重新编译。

使用数组

数组中的单个元素使用数组下标运算符([])进行访问。如果是一个单维数组被用于不带下标的表达式中,数组名称求值为指向数组第一个元素的指针。例如:

```
char chArray[10];
...

char *pch=chArray; //指向第一个元素的指针
char   ch=chArray[0];  //第一个元素的值
       ch=chArray[3];  //第四个元素的值
```

当使用多维数组时,表达式中对各种组合都可接受。以下例子说明了这点:

```
double multi[4][4][3];//说明一个数组
```

```
double (*p2multi)[3];
double (*p1multi);

cout << multi[3][2][3] << "\n";    //使用三个下标
p2multi=multi[3]; //使 p2multi 指向 multi 的第四个“平面”

p1multi=multi[3][2]; //使 p1multi 指向 multi 的第四个“平面”,
                     //multi 的第二行
```

在上面代码中,mutli 是一个 double 类型的三维数组。p2multi 指针指向一个 double 类型大小为 3 的一个数组。在此例中数组带 1 个、2 个和 3 个下标使用。尽管通常指定所有的下标,如在 cout 语句中,但有时选取数组元素的一个特定的子集是很有用的,如后续的语句所示。

表达式中的数组

当一个数组类型的标识符出现在一个表达式中而不是在 sizeof、取地址(&)或引用的初始化中,则它被转换为指向数组第一个元素的指针,例如:

```
char szError1[]="Error:Disk drive not ready.";
char *psz=szError1;
```

psz 指针指向数组 szError1 的第一个元素。注意,数组不象指针,数组是不可修改的 l 值。因此,以下赋值是非法的:

```
szError1=psz;
```

下标运算符的解释

象其它运算符一样,下标运算符([])可由用户重新定义。如果没有被重载,则下标运算符的缺省动作是使用以下方法将数组和下标组合在一起:

*((数组名)+(下标))

和包含指针类型的所有加法中一样,定位自动被执行以调整类型的尺寸。因此,结果值不是从数组名起始的下标字节数,而是数组的以下标为序数的那个元素(关于此转换的更多信息见第 4 章“表达式”中的“加法运算符”)。

类似地,对多维数组,地址是使用以下方法派生的:

*((数组名)+(下标 $_1$*max$_2$*max$_3$...max$_n$)
+下标 $_2$*max$_3$...max$_n$)
...+下标 $_n$))

数组类型上的间接操作

在一个 n 维数组类型上使用间接运算符(*)产生一个 n-1 维数组。如果 n 是 1,则产生一个标量(或数组元素)。

数组的排序

C++数组按行主顺序排序。行主顺序意味着最后的下标改变得最快。

函数说明

本节包括以下主题:

- 函数说明语法
  - 可变的参量表
  - 说明不带参量的函数
  - 函数重载
  - 函数上的限制
  - 参量说明表
  - 函数原型(未定义说明)中的参量表
  - 函数定义中的参量表
- 缺省参量
  - 缺省参量表达式
  - 其它考虑

函数定义包含在“函数定义”节中。

函数说明语法

语法

说明指示符 *dname*(参量说明表) *cv* 修饰符表 $_{\text{opt}}$

参量说明表:

参量说明表,...

参量说明表:

参量说明

参量说明表,参量说明

参量说明表:

说明指示符 说明符
说明指示符 说明符=表达式
说明指示符 抽象说明符 $_{opt}$
说明指示符 抽象说明符 $_{opt}$=表达式
由 *dname* 给定的标识符具有类型“*cv* 修饰表函数,带参量说明表且返回类型说明指示符”。
注意,const、volatile 和许多 Microsoft 特殊关键字可以出现在 cv 修饰符表和名称的说明中。以下例子给出了两个简单的函数说明:

```
char *strchr(char *dest,char *src);
static int atoi(const char *ascnum) const;
```

以下语法解释了函数说明的细节:
语法
参量说明表:
参量说明表 $_{opt}$...$_{opt}$
参量说明表,...
参量说明表:
参量说明
参量说明表,参量说明
参量说明:
说明指示符 说明符
说明指示符 说明符,表达式
说明指示符 抽象说明符 $_{opt}$

说明指示符 抽象说明符$_{opt}$,表达式

可变的参量表

参量说明表的最后一个成员是省略号(...)，其函数说明可以带可变数量的参量。在这些情况中,C++仅为显式说明的参量提供类型检查。当你需要一个通用函数其参量的个数和类型是可变的,你可以使用可变的参量表。printf 函数簇便是一个使用可变的参量表的函数例子。

为了在那些说明之后访问参量,使用在标准包含文件 STDARG.H 中包含的宏,如本章后面“带可变参量的函数”中所描述的。

Microsoft 特殊处→

Microsoft C++允许省略号指定为一个参量,如果省略号是第一个参量且省略号前面为一个逗号。因此,说明 int Func(int i,...);是合法的,但 int Func(int i...);则不是。

Microsoft 特殊处结束

带可变数目的参量的函数说明需要至少一个“位置占有者”参量,即使它未被使用。如果没有提供这个占位参量,则没有办法去访问剩余的参量。

当 char 类型的参量被作为可变参量传递时,它们被转换成 int 类型。类似地,当 float 类型的参量被作为可变参量传递时,它们被转换成 double 类型。其它类型的参量属于常用的整型和浮点提升。更多的信息参见第 3 章“标准转换”中的“整型提升”。

说明不带参量的函数

在参量说明表中用单个关键字 void 说明的函数不带参量,只要关键字 void 是第一个且是唯一的参量说明表中的一个成员。在参量说明表中任何其它位置的 void 类型参量将产生错误。例如:

```
long GetTickCount(void);                    //可行
long GetTickCount(int Reset, void);    //错误
long GetTickCount(void,int Reset);    //错误
```

在 C++中,显式指定一个函数不需要参数的函数与说明一个不带参量说明表的函数是一样的,因此,以下两个语句是完全相同的:

```
long GetTickCount();
long GetTickCount(void);
```

注意:除了此处提出的,其它的指定一个 void 参量是非法的;但从 void 类型派生的类型(如指向 void 的指针和 void 数组)可出现在参量说明表中的任何位置。

函数重载

C++允许在相同的范围内指定不止一个的相同名称的函数。这些被称为“重载的函数”,在第 12 章“重载”中有详细的描述。重载函数使程序员能够为一个函数提供不同的语义,这取决于参量的类型和数目。

例如,带一个字符串(或 char*)参量的 print 函数与带一个 double 类型参量的函数执行极为不同的任务。重载允许统一的命名,防止程序员去发明诸如 print_sz 或 print_d 这样的名称。表 7.1 显示了 C++使用函数说明的什么部分去区分在相同范围中具有相同名称的函数组。

表 7.1 重载考虑

函数说明元素	用于重载？
函数返回类型	否
参量的个数	是
参量的类型	是
省略号的出现或不出现	是
typedef 名称的使用	否
未指定的数组边界	否
const 或 volatile(在 *cv* 修饰符表中)	是

尽管函数可以在返回类型基础上加以区分,但它们却不能在此基础上重载。

以下例子说明了重载是如何被使用的。解决此同一问题的另一方法在本章后面部分“缺省参量”中给出。

```
#include <iostream.h>
#include <math.h>
#include <stdlib.h>

//三个 print 函数原型
int print(char *s); //打印一个字符串
int print(double dvalue);   //打印一个 double 值
int print(double dvalue,int prec);  //打印一个具有给定精度的 double 值
```

```
void main(int argc,char *argv[])
{
    const double d=893094.2987;

    if(argc<2)
    {
      //这些调用用于打印 invoke print(char *s)
      print("this program requires one argument.");
      print("the argument specifies the number of");
      print("digits precision for the second number");
      print("printed.");
    }

    //调用 print(double dvalue).
      print(d);

   //调用 print(double dvalue,int prec).
   print(d,atoi(argv[1]));
   }

  //打印一个字符串
  int print(char *s)
```

```
{
  cout << s << endl;
  return cout.good();
}

//以缺省精度打印一个 double 数
int print (double dvalue)
{
 cout << dvalue << endl;
 retarn cout.good();
}

//以指定的精度打印一个 double 数
//正数精度指示小数点后显示几位数字精度
//负数精度指示小数点左边进行数的舍入的地方
int print(double dvalue,int prec)
{
   //为舍入或截断使用表查找
   static const double rgPow10[]={
      10E-7,10E-6,10E-5,10E-4,10E-3,10E-2,1E-1,10E0,
      10E1, 10E2, 10E3, 10E4, 10E5, 10E6
   };
```

```
    const int iPowZero=6;

  //如果精度在范围外,则就打印此数
  if (prec < -6 || prec>7)
    return print(dvalue);

  //定位,截断,然后再定位
  dvalue=floor(dvalue/rgPow10[iPowZero-prec])*
                      rgPow10[ipowZero-prec];

   cout << dvalue << endl;
   return cout.good();
}
```

上面的代码给出了在文件范围中对 print 函数的重载。

关于重载的限制和重载如何影响其它 C++元素的信息,参见第 12 章“重载”。

函数的限制

函数不能返回数组或函数,但是它们可以返回引用或指向数组或函数的指针。返回数组的另一个方法是仅用那个数组作为一个成员去说明一个结构:

```
struct Address
{ char szAddress[31]; };
```

```
Address GetAddress();
```
在函数说明的返回类型部分或在函数的任何参量的说明中去定义一个类型都是非法的。
以下合法的 C 代码在 C++中是非法的：
```
enum Weather{ Cloudy,Rainy,Sunny } GetWeather(Date Today)
```
上面的代码是不允许的,因为 Weather 类型具有相对 GetWeather 局部的函数范围,则不适合使用返回值。因为函数的参量具有函数范围,所以在参量表内作的说明如果不被允许则将有同样的问题。
C++不支持函数数组。但指向函数的指针数组可能非常有用。在类 Pascal 语言的语法分析中,代码通常分开进入一个语言符号分析用的词法分析器和一个将语义附于语言符号用的语法分析器中。如果分析器为每个符号返回一个特定的普通值,则代码可被写成如下例子所示以执行适当的处理：
```
int ProcessFORTOken(char *szText);
int ProsessWHILEToken(char *szText);
int ProcessBEGINTOken(char *szText);
int ProcessENDToken(char *szText);
int ProcessIFToken(char *szText);
int ProcessTHENToken(char *stText);
int ProcessELSEToken(char *szText);
int (*ProcessToken[])(char *)={
    ProcessFORToken,ProcessWHILEToken,ProcessBEGINToken,
    ProcessENDToken,ProcessIFToken,ProcessTHENToken,
```

```
    ProcessELSEToken};
const int MaxTOkenTD=sizeof ProcessTOken / sizeof(int(*)());
...
int DoProcessToken(int TokenID, char *szText)
{
    if (TOkenTD < MaxTokenID)
        return (*ProcessToken[TokenID])(szText);
      else
        return Error(szText);
}
```

参量说明表

一个函数说明的参量说明表部分:

- 允许编译器的检查函数所需参量与调用中提供的参量之间的类型一致性。
- 允许从提供的参量类型转换到所需的参量类型,转换过程或者为隐式的或者为用户定义的。
- 检查函数指针的初始化或赋值。
- 检查函数引用的初始化或赋值。

函数原型(非定义说明)中的参量表

参量说明表形式是参量的类型名称的一个表。考虑函数 func 的参量表,带三个

参量:指向 char 的指针、char 和 int 类型。
这样一个参量说明表的代码可写为:

```
char*,char,int
```

因此函数说明(原型)可写成:

```
void func (char*,char,int);
```

尽管上面的说明包含了足够的信息供编译器执行类型检查和转换,但没有提供参量是什么的更多的信息。一个书写函数说明的更好的方法是包含标识符,因为它们将在函数定义中出现,如以下所示:

```
void func(char *szTarget, char chSearchChar,int nStartAt);
```

原型中的这些标识符仅对缺省参量有用,因为它们直接超出了范围。但是它们提供了有意义的程序文档。

函数定义中的参量表

在一个函数定义中的参量表与原型中的不同之处仅在于:如果存在标识符,则它表示函数的形参。标识符名不需要与原型中的那些匹配(如果存在)。
注意:可以定义带未命名的参量的函数。但是这些参量对它们定义的函数是不可访问的。

缺省参量

在很多情况中,函数拥有的参量使用的频度太小,只需要一个缺省值就够了。要用这个,缺省参量仅允许用于指定在一个给定的调用中是有意义的函数的那些参量,为了说明这个概念,考虑本章前面“函数重载”中给出的例子:

```
//三个 print 函数原型
int print(char *s); //打印一个字符串
int print(double dvalue);   //打印一个 double 值
int print(double dvalne, int prec); //打印一个给定精度的 double 值
        在很多应用中,可以用 prec 提供合理的缺省值,而不必需要两个函数。
//两个 print 函数原型
int print(char *s);  //打印一个字符串
int print(double dvalue,int prec=2); //打印一个给定精度的 double 值
        print 函数的实现稍稍改动以反映出这个事实:即对于 double 类型只存在
        一个这样的函数:
//打印一给定精度的 double 值
//正精度值指示小数点后显示多少位数字
//负精度值指示小数点左边舍入的位置
int print (double dvalue,int prec)
{
    //为舍入或截断使用查找表
    static const double rgPow10[]={
        10E-7,10E-6,10E-5,10E-4,10E-3,10E-2,10E-1,10E0,
        10E1, 10E2, 10E3, 10E4, 10E5, 10E6
    };
    const int iPowZero=6;
//如果精度超出了范围,则就打印这个数
```

```
    if (prec>=-6 || prec<=7)
        //定位,截断,然后再定位
        dvalue=floor(dvalue / rgPow10[iPowZero-prec])*
                               rgPow10[iPowZero-prec] ;

    cout << dvalue << endl;

        return cout.good();
}
```

为了调用新的 print 函数,使用如下代码:

```
print(d);    //精度为 2,由缺省参量提供
print(d,0); //绕过缺省参量以获取其它的值。
```

在使用缺省参量时要注意以下几点:

- 缺省参量仅用在尾部参量被省略的函数调用中,即它们必须是最后的参量。因此,以下代码是非法的:

```
int print(double dvalue=0.0, int prec);
```

- 一个缺省的参量在后面的说明中不能定义,即使再定义与原来的完全相同。因此,以下代码产生一个错误:

```
//print 函数原型
int print(double dvalue,int prec=2);
...
```

```
//print 函数定义
int print (double dvalue ,int prec=2)
{
...
}
```

此代码的问题是在定义中的函数说明为 prec 重定义了缺省值。

- 额外的缺省参量可由后面的说明添加。
- 可以为函数指针提供缺省的参量。例如：

```
int (*pShowIntVal)(int i=0);
```

缺省的参量表达式

缺省参量使用的表达式通常是常量表达式,但这不是要求的。表达式可以组合当前范围中可见的函数、常量表达式和全局变量。表达式不能包含局部变量或非静态类成员变量。

以下代码说明这点：

```
BOOL CreateVScrollBar(HWND hWnd, short nWidth=
                          GetSystemMetrics(SM_CXVSCROLL));
```

上面的说明指定了一个函数,为一个窗口创建一个给定宽度的垂直滚动条。如果没有提供宽度参量,则 Windows API 函数 GetSystemMetrics 被调用为一个滚动条查找缺省宽度。

在函数调用之后缺省表达式被求值,但求值在函数调用实际发生之前完成。

因为函数的形参是在函数范围内,并且因为缺省参量的求值在进入此范围之前发

生,所以你不能在缺省参量表达式中使用形参或局部变量。
注意在一个缺省参量表达式之前说明的任何形参可在函数范围内隐藏一个全局名称,这可能导致错误。以下代码是非法的:

```
const int Categories = 9;

void Enum Categories(char *Categories[],int n=Categories);
```

在前面代码中,全局名称 Categories 在函数范围内被隐藏,使得缺省参量表达式无效。

其它考虑

缺省参量不被认为是函数类型的一部分,因此,不被用于选择重载的函数。两个函数若仅在其缺省参量上不同,则被认为是多个定义而不是重载的函数。
不能为重载的运算符提供缺省参量。

函数定义

函数定义与函数说明的不同点在于它们提拱函数体,即组成函数的代码。
语法
函数定义:
    说明指示符 $_{opt}$ 说明符 *ctor* 初始化器 $_{opt}$ 函数体
函数体:
复合语句正如“函数”中讨论的语法中说明符的格式为:

dname(参量说明表)cv 修饰符表$_{opt}$

在参量说明表中说明的形参是在函数体的范围内。

图 7.3 显示了一个函数定义的各个部分。阴影区域是函数体。

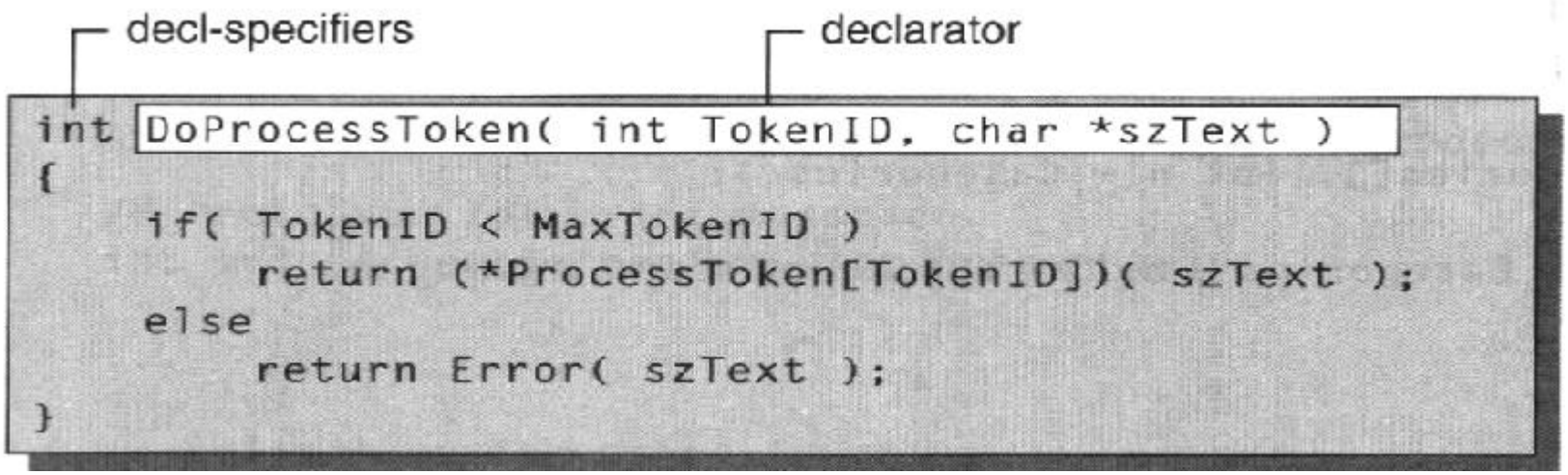


图 7.3 函数定义的各个部分

说明符语法的 *cv* 修饰符表元素指示了如何对待 this 指针的;它仅与类成员函数一起使用。

语法中的 *ctor* 初始化器元素仅被用于构造函数中,它的目的是允许对基类和包含的对象进行初始化(有关 *ctor* 初始化器的更多信息参见第 11 章“特殊成员函数”中的“初始化基类和成员”)。

带可变的参量表的函数

要求可变表的函数在参量表中使用省略号(...)进行说明,如本章前面“可变的参量表”中所描述。要访问使用此方法传递给函数的参量,则使用 STDARG.H 标准包括文件中所描述的类型和宏。
以下例子给出了 va_start、va_arg 和 va_end 宏和 va_list 类型(在 STDARG.H 中说明的)是如何一起工作的。

```
#include <stdio.h>
#include <stdarg.h>

//showVar 的说明,但不是定义
int ShowVar(char *szTypes,...);

void main()
{
    ShowVar("fcsi",32.4f,'a',"Test string",4);
}

//showvar 带一个“fcsi”形式的格式化字符串,其中每个字符指示那个位置的
//参量类型
//
//i=int
```

```
//f=float
//c=char
//s=string(char*)
//
//以下格式说明是一个 n 个参量的表,其中 n==strlen(szTypes)
void ShowVar(char *szTypes,...)
{
    va_list vl;
    int i;

    //szTypes 是指定的最后的参量;所有其它的必须使用可变的参量宏进行访
问
    va_start(vl,szTypes);

    //Step through the list
    for(i=0;szTypes[i]!='\0';++i)
    {
       union printable_t
      {
         int   i;
         float f;
         char  c;
```

```
        char  *s;
      }  Printable;

switch (szTypes[i])  //预期的类型
    {
    case 'i':
       Printable.i=va_arg(vl,int);
       printf("%i\n",Printable.i);
       break;

  case 'f':
       Printable.f=va_arg(vl,float);
       printf("%f\n",Printable.f);
       break;

  case 'c':
       Printable.c=va_arg(vl,char);
       printf("%f\n",Printable.f);
       break;

  case 's':
       Printable.s=va_arg(vl,char *);
```

```
            printf("%s\n",Printable.s);
            break;

    default:
            break;
       }
    }
    va_end(vl);

}
```

前面的例子说明了以下这些重要概念：

- 一个表标记将必须在任何可变的参量被访问之前建立为一个 va_list 类型的变量,在前面的例子中,标识符被叫做 vl。
- 单个参量使用 va_arg 宏进行访问,va_arg 宏需要被告知要检索的参量的类型,这样它才能从堆栈中传递正确数目的字节。如果一个与调用程序提供的大小不同的不正确的类型被指定给 va_arg,则结果是难以预料的。
- 使用 va_arg 宏获得的结果应该被显式地造型转换到所希望的类型。
- va_end 宏必须被调用以终止可变的参量处理。

初始化器

说明符可指定对象的初始值。为 const 类型的对象指定一个值的唯一方法是在说明符中指定。说明符中指定这个初始值的部分被称为“初始化器”。

语法

初始化器

=赋值表达式

={初始化器表,opt}

(表达式表)

初始化器表:

表达式

初始化器表,表达式

{初始化器表,opt}

初始化器有两个基本类型:

- 使用等号语法调用的初始化器。
- 使用函数形式调用的初始化器。

仅仅只有构造函数的类的对象可以用函数形式语法进行初始化。两个语法格式的不同还在于访问控制和临时对象的潜在使用。考虑以下代码,用初始化器说明一些说明符:

```
int i=7;//使用等号语法
Customer Cust ("Taxpayer, Joe", //使用函数形式语法
        "14 Cherry Lane",  //构造函数的要求
```

```
        "Manteca",
        "CA");
```

自动的、寄存器的、静态的和外部变量的说明可以包含初始化器。但是外部变量的说明仅当变量没有被说明为 extern 时可以包含初始化器。
这些初始化器可以包含包括在当前范围内的常量和变量。初始化器在程序流遇到说明的地方被求值,或对全局静态对象和变量在程序开始处求值(有关全局静态对象的初始化的更多信息,参见第 2 章“基本概念”中的“额外的开始考虑”)。

指向 const 对象的指针的初始化

一个指向 const 对象的指针可以用一个指向非 const 对象的指针进行初始化,但反之不可。
例如,以下初始化是合法的:

```
Window StandardWindow;
const Window* pStandard Window(&Standard Window);
```

在上面代码中,指针 pStandardWindow 被说明为指向一个 const 对象的指针。尽管 StandardWindow 没有被说明为 const,但该说明是可接受的,因为它不允许被说明为 const 的对象访问一个 const 对象。与此相反的如下:

```
const Window StandardWindow;
Window* pStandardWindow(StandardWindow);
```

上面的代码显式说明 StandardWindow 为一个 const 对象。用 StandardWindow 的地址初始化非常量指针 pStandardWindow 会产生一个错误,因为它允许通过指针访问 const 对象,即它允许从对象中移走 const 属性。

未初始化的对象

没有用初始化器说明的 static 存储类的对象和简单变量被确保初始化为一个位模式 0。对自动的或寄存器存储类的未初始化的对象,则不产生此种特殊处理。它们有未定义的值。

初始化静态成员

静态成员初始化出现在类范围中。因此,它们可以访问其它成员数据或函数。例如:

```
class DialogWindow
{
public:
    static short GetTextHeight();
private:
    static short nTextHeight;
};

short DialogWindow::nTextHeight=GetTextHeight();
```

注意在前面的静态成员 nTextHeight 的定义中,GetTextHeight 隐含指 DialogWindow::GetTextHeight。

初始化集合

一个集合类型是一个这样的数组、类或结构类型：

- 没有构造函数
- 没有非公有成员
- 没有基类
- 没有虚拟函数

集合的初始化器可以指定为括在花括号中的以逗号分隔的值表。例如，此代码说明了一个 10 个 int 的数组，并初始化为：

```
int rgiArray[10]={9,8,4,6,5,6,3,5,6,11};
```

初始化器以递增的下标顺序存储在数组元素中。因此，rgiArray[0]是 9、rgiArray[1]是 8，等等，直到 rgiArray[9]是 11。要初始化一个结构，使用这样的代码：

```
struct RCPrompt
{
    short nRow;
    short nCol;
    char *szPrompt;
};
RCPrompt rcContinueYN={24,0,"Continue (Y/N？)"};
```

集合初始化器表的长度

如果一个集合初始化器表比正被初始化的数组或类类型短，则未指定初始化器的

0被存储在元素中。因此,以下两个说明是等价的:

```
//显式初始化所有元素
int rgiArray[5]={3,2,0,0,0};

//允许保留元素为0初始化的
int rgiArray[5]={3,2}
```

正如这所看到的,初始化器可被截断,但提供太多的初始化器将产生一个错误。

初始化包含集合的集合

某些集合包含其它的集合,例如,数组的数组、结构的数组或由其它结构组成的结构,对这种结构可以通过用带花括号的表按其出现的顺序对每个初始化而提供初始化器,例如:

```
//说明一个RCPrompt类型的数组
RCPrompt rgRCPrompt[4]=
{{4,7,"Options Are:"},
   {6, 7,"1. Main Menu"},
   {8, 7,"2. Print Menu"},
   {10,7,"3. File Menu"}};
```

注意,rgRCPrompt被用花括号包围的表的花括号包围表初始化。封闭的花括号不是语法上要求的,但它们使说明更清晰,以下程序例子显示了一个二维数组是如何用这样一个初始化器填充的:

```
#include <iostream.h>
```

```
void main( )
{
    int rgI[2][4]={1,2,3,4,5,6,7,8};

    for(int i=0;i<2; ++i)
       for(int j=0;j<4; ++j)
           cout << "rgI[" << i << "][" << j << "]="
                << rgI[i][j] << endl;
}
```

程序的输出为：

```
rgI[0][0]=1
rgI[0][1]=2
rgI[0][2]=3
rgI[0][3]=4
rgI[1][0]=5
rgI[1][1]=6
rgI[1][2]=7
rgI[1][3]=8
```

短初始化表仅与显式子集合初始化器一起且被花括号括住时才是可用的。如果rgI说明为：

```
int rgI[2][4]={{1,2},{3,4}};
```

则程序的输出将是：

```
rgI[0][0]=1
rgI[0][1]=2
rgI[0][2]=0
rgI[0][3]=0
rgI[1][0]=3
rgI[1][1]=4
rgI[1][2]=0
rgI[1][3]=0
```

不完整类型的初始化

不完整类型，如没有限定边界的数组类型，可以初始化如下：

```
char HomeRow[]={'a','s','d','f','h','i','k','l'}
```

编译器从提供的初始化器数目计算数组的大小。

不完整类型，如已说明但未定义的指向类类型的指针，其说明如下：

```
class DefinedElseWhere;    //类定义在别处
class DefinedHere
{
    ...
    friend class definedElsewhere;
}
```

使用构造函数初始化

类类型对象通过调用类的合适的构造函数进行初始化。关于类类型初始化的完整信息参见第 11 章“特殊成员函数”中的“显式初始化”。

初始化器和联合

union 类型的对象用一个单值进行初始化(如果联合没有一个构造函数)。这是以这两种方法之一完成的:

- 用同一个 union 类型的另一个对象初始化联合,例如:

```
struct Point
{
    unsigned x;
    unsigned y;
}
union PtLong
{
    long l;
    Point pt;
};
...

PtLong ptOrigin;
PtLong ptCurrentT=ptOrigin;
```

在上面代码中,ptCurrent 用 ptOrigin 值相同类型的一个对象进行初始化。

- 将第一个成员用花括号括起的初始化器联合起来初始化。例如:

```
PtLong ptCurrent={0x0a000aL};
```

初始化字符数组

字符数组可用以下两种方法之一进行初始化:

- 单个地,如:

```
char chABCD[4]={'a', 'b', 'c', 'd'};
```

- 用一个字符串,如;

```
char chABCD[5]="abcd";
```

在第二种情况下,即字符数组用一个字符串进行初始化,编译器添加一个尾部的'\0'(字符串结束字符)。因此,数组必定至少比串中字符的个数大 1。
因为大多数字符串处理使用标准库函数或依赖于尾部字符串结束字符的出现,所以常见用字符串初始化未限定边界的数组说明如下:

```
char chABCD[]="ABCD";
```

初始化引用

引用的类型变量必须用派生引用类型对象或用可以转换到派生引用类型派对对象进行初始化。例如:

```
int iVar;
long lVar;
```

```
long& LongRef1=lVar;   //不需要转换
long& LongRef2=iVar;   //错误
const LongRef3=iVar;   //可行

longRef1=23L;   //通过一个引用改变 lVar
longRef2=11L;   //通过一个引用改变 iVar
longRef3=11L;   //错误
```

使用一个临时对象去初始化一个引用的唯一方法是去初始化一个常量临时对象。一旦初始化了,一个引用类型变量总是指向相同的对象;它不能被改为指向另一个对象。

尽管语法可以相同,但引用类型变量的初始化和引用类型变量的赋值在语义上是不同的。在前面的例子中,改变 iVar 和 lVar 的赋值看上去与初始化相似,但有不同的效果。初始化指定引用类型变量指向的对象,赋值通过引用赋给被指向的对象。

因为传递给一个函数一个引用类型的参量和从一个函数返回一个引用类型值都是初始化,所以函数的形参被正确地初始化,正如返回的引用。

引用类型变量仅在以下情况可以不用初始化器进行初始化:

- 函数说明(原型)。例如:

```
int func(int&);
```

- 函数返回类型说明,例如:

```
int & func(int&);
```

- 引用类型类成员的说明,例如:

```
class C
{
public:
  int& i;
};
```

- 被显式指定为 extern 的变量的说明,例如:

```
extern int& iVal;
```

当初始化一个引用类型变量时,编译器使用图 7.4 中所示的决策图,以在创建一个对象的引用还是创建一个引用指向的临时对象之间进行选择。

volatile 类型的引用(说明为:volatile 类型名称& 标识符)可以用相同类型的 volatile 对象或用没有被说明为 volatile 的对象进行初始化。但是它们不能用那个类型的 const 对象进行初始化。类似地,const 类型的引用(说明为:const 类型名称& 标识符)可以用相同类型的 const 对象(或任何可转换到那个类型的或没有被说明为 const 的对象)进行初始化。但是它们不能用那个类型的 volatile 对象进行初始化。没有用 const 或 volatile 关键字限定的引用,仅能用既不是说明为 const 也不是说明为 volatile 的对象进行初始化。

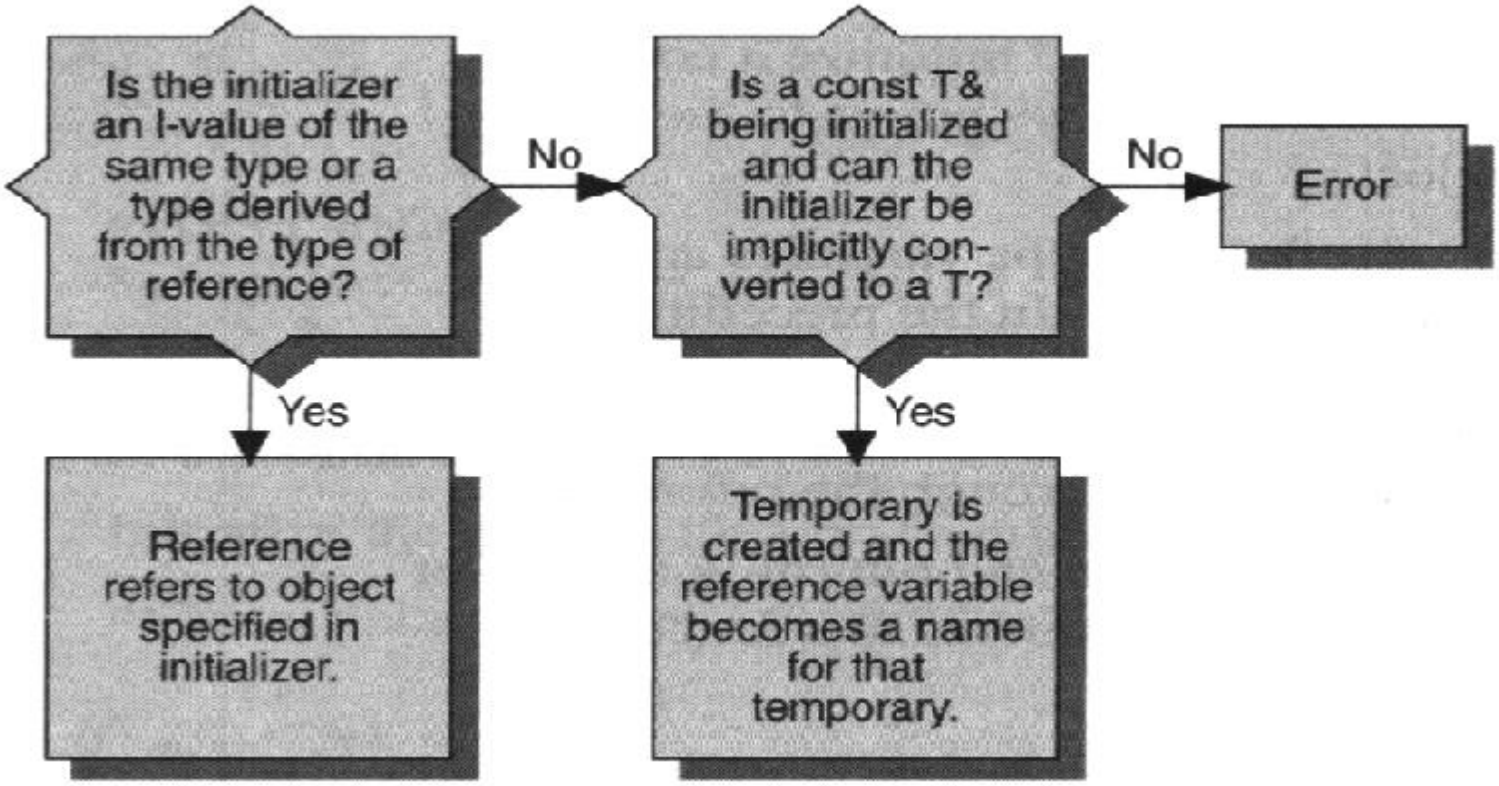


图 7.4　引用类型初始化的决策图

第 8 章　类

这一章介绍 C++的类,类在程序中引入了用户自定义的类型,类可以包含数据和函数。

在传统程序设计语言中用户自定义类型是数据的集合。它们放在一起用以描述对象的属性和状态。C++中的类类型使用户不仅能够描述对象的属性和状态,还可以定义对象的行为。

本章包括下面一些主题:

- 类的概述
- 类名称
- 类成员
- 成员函数
- 静态数据成员
- 联合
- 位域
- 嵌套类说明
- 类范围中的类型名称

类的概述

类类型用关键字 class, struct 和 union 定义。简单地说,用这三个关键字定义的类型都称作类说明(class declaration)。但是在讨论语言成分时,用不同的关键字定义的类,其行为是不同的。

语法

类名称:

    标识符

一个类的变量和函数称为类的成员。在定义一个类的时候通常也提供一些如下的成员(尽管都是任选的):

- 类的数据成员,定义一个类类型的某个对象的属性和状态。
- 一个或多个构造函数,用来初始化类类型的对象。构造函数在第 11 章“特殊成员函数”中的“构造函数”一节中介绍。
- 一个或多个析构函数,主要完成一些清除工作,如:回收动态分配的存储器或关闭文件。析构函数在第 11 章“特殊成员函数”的“析构函数”一节中详述。
- 一个或多个成员函数用来定义对象的行为。

定义类类型

类类型的定义用类指示符(class-specifiers)。类类型的说明使用复杂类型指示符。在第 6 章“说明”中的“类型说明符”一节介绍。

语法

类说明符：

    类头{成员表 opt}

类头：

    类关键字 imodel opt 标识符 opt 基类说明 opt

    类关键字 imodel opt 类名称 opt 基类说明 opt

类关键字：

    class

    struct

    union

imodel:

    __declspec

当编译器处理完类名称以后(在处理类体之前),类名称立即被当作标识符,因此类名称可以用来说明类成员。这使得用户可以说明自引用型的数据结构。如下：

```
class Tree
{
public:
   void  *Data;
   Tree  *Left;
   Tree  *Right;
};
```

结构、类和联合

三种类类型分别是结构、类和联合。他们的说明是用 struct、class 和 union 关键字(见关键字语法)。表 8.1 显示了三种类类型之间的不同。

表 8.1 结构、类和联合的访问控制和约束

结构	类	联合
类关键字 struct	类关键字是 class	类关键字是 union
缺省访问控制是公有的	缺省访问控制是私有的	缺省访问控制是公有的
无使用约束	无使用约束	任何时候只能使用一个成员

无名类类型

类可以是无名的。也即,你可以说明一个不带标识符的类。这在你用一个 typedef 名称去代替一个类的类名称时很有用。如下:

```
typedef struct
{
    unsigned x;
    unsigned y;
} POINT;
```

注意:上面例子中无名类的使用对保持已有 C 代码的兼容性是很有用的。在很多 C 代码中,typedef 与无名结构一起使用是非常普遍的。当你要引用一个类的成

员,并表现出这个成员并不是被包含在一个单独的类中时,无名类也同样是很有用的,如下所示:

```
struct PTValue
{
    POINT ptLoc;
    union
    {
    int iValue;
    long lValue;
    };
};

PTValue ptv;
```

在上面的代码中,iValue可以用对象成员选择符(.)来访问。如下:

```
int i=ptv.iValue;
```

无名类要服从一些特定的限制(有关无名 union 的详情见本章后面的“联合”一节)。一个无名类:

- 不能有构造函数或析构函数
- 不能作为参数传递给函数(除非用“...”避开类型检查)
- 也不能作为函数的返回值

类的定义点

一个类的定义是在其类说明符之后。成员函数不必因为类的详细定义而也要马上定义。

研究下面的例子：

```
class Point              //Point类
{    //详细定义
public:
     Point()
       {cx=cy=0;}//定义构造函数
     Point(int x,int y)
       {cx=x,cy=y;}  //定义构造函数
     unsigned &x(unsigned); //定义访问器
     unsigned &y(unsigned); //定义访问器
private:
     unsigned cx,cy;
};
```

尽管两个访问器函数(x 和 y)还没有被定义,但类 Point 已经详细定义了的(访问器函数主要用来提供对数据成员的安全访问)。

类类型对象

一个对象在执行环境中是一块类型化的存储区域;它不但保留了状态信息,也定

义了行为。类类型对象用类名称来定义。考察下面的代码段：

```
class Account   //类名是 Account
{
public:
          Account();                  //缺省构造函数
          Account(double);            //从 double 类型来构造
      double& Deposit(double);
      double& Withdraw(double,int);
      ...
};

Account CheckingAccount;        //定义类类型对象
```

面的代码说明了一个类(新的类型)称为 Account,然后用这种新的类型定义了一个对象叫 CheckingAccount。

C++为类型对象提供了如下一些操作：

- 赋值。一个对象能够赋值给另一个。这一操作的缺省行为是成员方式(memberwise)的拷贝。用户可以提供一个自定义的赋值操作以取代缺省行为。
- 用拷贝构造函数进行初始化

下面是用户自定义的拷贝构造函数进行的初始化的例子：

- 对某对象进行明确地初始化：

```
    Point myPoint=thatPoint:
```

把 myPoint 说明为一个 Point 类型的对象并把它初始化为 thatPoint 的值。

- 作为传递参数而引起的初始化。对象能以传值或引用形式传递给函数。如果它们是以传值形式传递给函数的,则每个对象的拷贝会传递给函数。创建此拷贝的缺省办法是一个成员方式(memberwise)的拷贝。当然也可以用自定义的拷贝构造函数来代替缺省的拷贝构造函数(是一种构造函数,它带有唯一的对类的引用型的参数)。
- 由于初始化函数返回值而引起的初始化,对象有能够以传值或引用方式从函数返回。对于以传值方式返回对象的缺省方法也是成员方式的拷贝;当然同样可以由自定义的拷贝构造函数所代替。由引用方式(用指针或引用类型)返回的对象,对于调用函数来说是不能作为自动型或局部型对象的。如果这样的话,由返回值所指引的对象在其能被使用之前就超出了其范围。

第 12 章“重载”中的“重载操作符”一节,解释如何在基于类方式下重定义这些操作符。

空类

你可以说明一个空类,但是这种类型的对象有非零的值。下面的例子显示了这一点:

```
#include(iostream.h)

class NoMembers
{
```

```
};

void main()
{
  NoMembers n; //NOMembers型对象
  cout << "The size of an object of empty class is:"
       << sizeof n << endl;
}
```

上面程序的输出是:

```
The size of an object of empty class is:1.
```

为这种对象分配的存储器是非零的。因而,不同的对象有不同的地址。由于具有不同的地址,就有可能通过比较对象指针鉴别不同的对象。同样,在数组中每个成员数组必须有不同的地址。

Microsoft特殊处→

一个空基类在其派生类中占用零个字节。

Microsoft特殊处结束

类名称

在程序中类说明引入了新的类型(以类名称来称呼),这些类说明在给定的转换单元中就像类定义一样。在每个转换单元中,对于给定的类类型仅仅只可以有一个该类型的定义。用这些新的类型,用户可以定义对象。同时,编译器也可以通过

类型检查发现对这些对象的不兼容的类型操作。
下面是类型检查的例子：

```
class Point
{
public:
   unsigned x,y;
};

class Rect
{
 public:
   unsigned x1,y1,x2,y2;
}

//带有两个参数的函数原型,一个是 Point 类型,另一个是 Rect 型的引用
int PtInRect (Point,Rect &);
...

Point pt;
Rect  rect;
rect=pt;    //错误,类型不匹配
pt=rect;    //错误,类型不匹配
```

```
//错误,PtInRect的参数反了
cout << "Point is" << PtInRect(rect,pt) ? "" : "not"
      <<"in rectangle "<< endl;
```

正如上面例子所显示的,对于一个类类型对象上的操作(如赋值和参量传递)和内部类型对象一样遵循着同一种类型检测约束。

因为编译器能够区分不同的类类型,函数可以在基于类类型参数以及内置参量的原则下被重载。有关重载函数的更多信息参见第 7 章“说明”中的“函数重载”以及第 12 章的“重载”。

说明及访问类名称

可以在全局范围或在类范围中说明类名。如果在类范围中说明类名称,它们实际上指的是“嵌套类”。

Microsoft 特殊处→

在 Microsoft C++的局部类说明中不允许有函数定义。

Microsoft 特殊处结束

在类的范围中引入的新的类名称会隐藏在同一封闭块中的同名元素。要引用因上述说明隐藏的名称,只能通过全类型指示符,下面的例子是一个用全类型指示符引用一个隐藏名称的例子:

```
struct A//全局范围中定义 A
{
   int a;
};
```

```
void main()
{
    char A='a';//把名称 A 重定义为一个对象
    struct A AObject;
    ...
}
```

因为名称 A 所指示的结构被由 A 所指示的 char 对象所隐藏了,因而定义一个类型 A 的对象 AObject 时必须用类关键字 struct。

你当然可以用类关键字说明一个类而不提供该类的定义。这种无定义的类说明只是为超前引用而引入了一个类名称。这种技术在设计要在友元说明中引用其它的类的类中非常有用,当然在类名称必须在头文件中出现,而其定义又不需要时,这种技术也很有用。例如:

```
//RECT.H
class Point; //类 Point 的无定义说明
class Line
{
public:
     int Draw(Point &ptFrom,Point *ptTo);
     ...
};
```

在上面的例子中,必须提供 Point 类名称,但不必引入该名称的定义性说明。

typedef 语句和类

使用 typedef 语句去命名一个类类型是把一个 typedef 名称变成一个类名称。更多的详情参见第 6 章“说明”中的“类型指示符”。

类成员

类可以有如下类型的成员：

- 成员函数
- 数据成员
- 类(包括类、结构和联合),参见“嵌套类说明和联合”
- 枚举
- 位域
- 友元
- 类型名称

注意：友元是被类说明所包含的,它们包括在一个预先的列表中,然而它们并不是真正的类成员,因为它们不在类的范围之中。

语法

成员表：

成员说明　成员表$_{opt}$

访问指示符：成员表$_{opt}$

成员说明：

decl 指示 opt 成员说明符表 opt;
函数定义 opt;
限定名;
成员说明符表:
成员说明符
成员说明符表,成员说明符
成员说明符;
说明符 纯指示符 opt
标识符 opt:常量表达式
纯指示符:
=0
成员表的目的:

- 说明给定类的全套成员
- 说明同各个类成员相联系的访问(公有的、私有的或保护的)

在成员表的说明中,一个成员只能说明一次。成员的再次说明会引出错误消息。因为一个成员表就是一整套的类成员,故不能在随后的类说明中为给定类增加成员。
成员说明符中也不能包含有初始化器,提供初始化器会产生一个错误消息如下:

```
class CantInit
{
public:
    long        l=7;//错误:试图初始化类成员
```

```
    static int i=9;//错误:必须在类说明之外定义并初始化
};
```
因为对每个给定类类型的对象都要单独创建静态成员实例。正确的初始化办法是用类的构造函数去初始化成员的数据(构造函数在第 11 章“特殊成员函数”中的“构造函数”一节中论述)。对于一个给定类类型的所有对象仅有一个共享的静态数据成员拷贝,静态数据成员必须在文件范围中定义才能初始化(有关静态数据成员的详情见本章后面的“静态数据成员”)。下面的例子显示了如何进行这些初始化:

```
class CanInit
{
public:
    CanInit(){l=7;}    //当新的 CanInit 对象创建时初始化 l
    long l;
    static int i:
    static int j;
}

int CanInit::i=15;   //i 在文件范围中定义并初始化为 15
                     //初始化是在 CanInit 的范围中定值的
int CanInit::j=i;    //初始化器的右边在要被初始化的对象范围中
```
注意:类名称 CanInit 必须前导于 i 以说明 i 被定义为 CanInit 的成员。

类成员说明语法

成员数据不能说明为 auto、extern 和 register 存储类型。然而它们能够被说明为 static 存储类型。
decl 指示符在成员函数的说明中可以被省略(有关 decl 指示符的情况见第 6 章“说明”中的“指示符”一节,以及本章后面的成员函数和第 7 章“说明”中的“函数”一节)。因而下面的代码合法地说明了一个返回整型的函数:

```
class NoDeclspec
{
public;
    NoSpecifiers();
};
```

当你在成员表中说明一个友元类时,你可以省略成员说明符表。有关友元的更多信息参见第 6 章“说明”中的“友元指示符”以及第 10 章“成员访问控制”中的“友元”一节。甚至当一个类名称还没有被定义时,它也可以作为友元来说明,正是友元说明引入了该类名称。
然而在这种类的成员说明中,必须使用全类型指示符语法。见如下的例子:

```
class HasFriends
{
public:
   friend class NotDeclaredYet;
};
```

在上面的例子中,类说明的后面没有成员说明符表。因为对 NotDeclaredYet 的说明还没有被处理到,故使用了全类型指示符的使用形式:class NotDeclaredYet。一个已经说明过的类型可以在友元成员说明的说明中使用一般的说明形式:

```
class AlreadyDeclared
{
  ...
};

class HasFriends
{
public;
   friend AlreadyDeclared;
}
```

纯指示符(见下面的例子)表明对此说明的虚拟函数不提供其函数实现。因此纯指示符仅仅只能用在虚函数之上指示,考察如下的代码:

```
class StrBase    //strings 的基类
{
public:
   virtual int IsLessThan  (StrBase&)=0;
   virtual int IsEqualTo   (StrBase&)=0;
   virtual strBase& CopyOf (StrBase&)=0;
```

```
    ...
};
```

上面的代码中说明了一个抽象类,也即这个类设计为用作基类以派生出更多的特有的类。通过用纯指示符说明一个或多个虚拟函数为纯虚拟函数,这种基类能够强制执行一个特殊的协议(protocol)或机制。

从 StrBase 继承的类必须为这些纯虚拟函数提供实现。否则它们也会被认为是抽象基类。

抽象基类不能用来说明对象。例如,在一个从 StrBase 类继承的类的对象被说明之前,必须提供函数 IsLessThan、IsEqualTo 以及 CopyOf 的实现(有关抽象基类的更多信息参见第 9 章“派生类”中的“抽象类”)。

在成员表说明变长数组

Microsoft 特殊处→

如果一个程序未用 ANSI 兼容选项(/Za)来编译,则变长数组可以在类成员表中说明为最后一个数据成员。因为这是 Microsoft 的扩充,故而用这种方式使用变长数组会降低你的代码的可移植性。说明一个变长数组,要省略第一个维数,如:

```
class Symbol
{
public:
    int SymbolTYpe;
    char SymbolText[];
};
```

Microsoft 特殊处结束

限制

如果一个类含有变长数组,它就不能用作其它类的基类,而且一个含有变长数组的类也不能随意说明为其它类的成员,只能说明为其它类的最后一个成员。一个含有变长数组的类也不能含有直接或间接虚拟基类。当 sizeof 运算符运用于一个含有变长数组的类时,将返回除变长数组以外的所有成员的存储总和。对于含有变长数组的类的实现,应该提供一个替换的办法以获得正确的类的大小。
不能够把含有变长数组组成成分的对象说明为某数组的成员,对于指向这类对象的指针进行数学运算也会产生错误。

类成员数据的存储

非静态成员数据以如下方式存储:所有在访问说明符之间的项是连续地向存储器地址增加的方向存放的。越过访问说明符的成员的存放顺序不作保证。
Microsoft 特殊处→
依赖于/Zp 编译选项或包含 pragma 伪指令,可以引入干预空间以对成员数据进行按字或双字边界对齐。在 Microsoft C++中,类成员是连续地按地址增加的方向存储的,尽管 C++语言并不要求这一点。有关这种顺序的基本假设都会引起不可移植的代码。
Microsoft 特殊处结束

成员命名限制

在类说明中跟类名称有同样名称的函数是构造函数。当该类的一个对象创建的时候,隐含地调用了构造函数(有关构造函数的更多信息参见第 11 章“特殊成员函数”中的“构造函数”一节。
下面各项在其说明的范围中,不能跟类名称有同样的名称:数据成员(静态或非静态)、封闭的枚举、无名联合成员及嵌套类。

成员函数

类能够包含数据和函数,这些函数称为成员函数。任何在类说明中说明的一个非静态函数视为成员函数,并用成员选择符(.和->)调用。当在其它的成员函数中调用本类的成员函数时,对象和成员选择符可以省略。例如:

```
class Point
{
public:
    short x() { return _x; }
    short y() { return _y; }
     void Show(){ cout<<x()<<","<<y()<<"\n"; }
private:
     short _x,_y;
};
```

```
void main()
{
    Point pt;

    pt.Show();
}
```

注意:在成员函数 Show 中调用了其它成员函数 x 和 y,并未使用成员选择符。这些调用的隐含意义是 this->x()和 this->y()。然而在主函数 main 中调用成员函数 Show 必须使用对象 pt 和成员选择符(.)。

在类中说明的静态成员函数的调用可以使用成员选择符或使用全限定函数名称(包括类名称)。

注意:用 friend 关键字说明的函数并不认为是类的成员。在类中此函数只是说明为友元(尽管此函数可以是别的类的成员)。

一个友元说明控制着非成员函数对类数据的访问。

下面的例子显示了如何说明成员函数:

```
class Point
{
public:
    unsigned GetX();
    unsigned GetY();
    unsigned SetX(unsigned x);
```

```
    unsigned SetY(unsigned y);
private:
    unsigned ptX, ptY;
}
```

在上面的类说明中,说明了四个函数:GetX,GetY,SetX 和 SetY。下面的例子显示如何在程序中调用这些函数:

```
void main()
{
     //说明一个 Point 类型的对象
     Point ptOrigin;

     //用成员选择符(.)调用成员函数
     ptOrigin.SetX(0);
     ptOrigin.SetY(0);

     //说明一个指向 Point 类型对象的指针
     Point *pptCurrent=new Point;

//用成员选择符(->)调用成员函数
  pptCurrent->SetX(ptOrigin.GetX()+10);
  pptCurrent->SetY(ptOrigin.GetY()+10);
}
```

在上面的代码中,对象 ptOrigin 成员函数的调用是用成员选择符(.)来调用的。而由 pptCurrent 指向的对象的成员函数的调用使用的是成员选择符(->)。

成员函数概述

成员函数可以是静态的或非静态的。静态成员函数的行为同其它成员函数的行为是不同的,因为静态成员函数没有隐含的 this 参数。非静态成员函数有一个 this 指针。无论静态成员函数还是非静态成员函数成员均可在类说明之内或之外定义。
如果是在类说明之内定义的,则它被视为嵌入函数,也没有必要用类名称来限定函数名称。尽管定义在类说明之中的函数已经是作为嵌入函数对待,你仍可用关键字 inline 来文档化代码。
下面是在一个类说明中定义一个函数的例子:

```
class Account
{
public:
    //在类 Account 说明之中说明函数 Deposit
    double Deposit(double HowMuch)
    {
        balance += HowMuch;
        return balance;
    }
```

```
private:
double balance;
};
```
当在类说明的外面定义成员函数时,只有明确地把它说明为 inline 的,此成员函数才视为嵌入型成员函数。而且在定义时函数名必须用类名和范围分辨符(::)加以限定。
下面的例子除了在类说明之外定义函数 Deposit 外,同前面对类 Account 说明是等同的。
```
class Account
{
public:
    //说明成员函数 Deposit,但不定义它。
    double Deposit(double HowMuch);
private:
    double balance;
};

inline double Account::Deposit (double HowMuch)
{
   balance += HowMuch;
   return balance;
}
```

注意:尽管成员函数既可以在类说明之中定义也可以在类说明之外单独定义,但当类定义了以后,就不能再为它增加成员函数了。
含有成员函数的类可以有多个说明,但成员函数本身在程序中仅有一次定义。多重定义会在链接时引出错误消息。如果一个类含有联编函数的定义,该函数的定义同样要满足“唯一定义”原则。

非静态成员函数

非静态成员函数有一个隐含的参数,this,它是一个指向调用此函数的对象的指针。
this 指针的类型是 type * const。这些函数被认为具有类的范围,可以直接使用处在同一类范围中的类数据和成员函数。在前面的例子中,表达式 balance += HowMuch 把 HowMuch 的值加到类成员 balance 中。考察下面的语句:

```
Account  Checking;

Checking Deposit(57.00);
```

在上面的例子中,说明了一个 Account 类型的对象,并调用其成员函数 Deposit 给它加上 $57.00。在函数 Account::Deposit 中,balance 作为 Checking.balance(该对象的 balance 成员)。
非静态成员函数趋向于在它们自己的类类型对象之上操作。通过明确的类型转换在别的不同类型的对象上,调用此成员函数会引起不确定的行为。

this 指针

所有的非静态成员函数能用 this 关键字,this 是一个常量指针(不可修改)。它指向的对象是成员函数的调用者。成员数据可以由表达式 this->成员名称的值来确定(尽管这种技术很少使用)。在成员函数中,在表达式中使用成员名称隐含着使用方式 this->成员名称来选择正确的函数和数据成员。
注意:因为 this 指针是不可修改的,因而不允许对 this 赋值。在早期的 C++的实现中允许对 this 指针赋值。
偶尔可以直接使用 this 指针。例如在操纵自引用型数据结构时,在这种情况下要取得当前对象的地址。

this 指针的类型

在函数说明中通过 const 和 volatile 关键字可以修改 this 指针的类型。要说明一个具有这些关键字属性的函数,可使用 cv-修饰符表语法。
语法:
cv-修饰符表:
    cv-修饰符 cv-修饰符表 $_{opt}$
cv-修饰符:
    const
    volatile
考虑下面的例子:

```
class Point
{
```

```
    unsigned X() const;
};
```

上面的例子中说明了一个成员函数 X,其中 this 指针被当作一个指向常量对象的常量指针。可以混合使用 cv-修饰符表选项,但它们通常修饰的是 this 指针所指的对象,而不是 this 指针本身。因此,下面的说明中说明了函数,它的 this 指针是一个指向常量对象的常量指针:

```
class point
{
  unsigned X()__far const;
};
```

下面的语法描述了this指针的类型,其中cv-修饰符表可以是const或volatile,类名称是类的名称:

cv-修饰符表 $_{opt}$ 类类型 * const this

表 8.2 解释了有关这些修饰符的更多的作用。

表 8.2

修饰符	意义
const	不能改变成员数据,也不能调用非常量型成员函数
volatile	成员数据每次访问时是从存储器中调入的;并关闭了优化工作

把一个常量对象传递给一个非常量的成员函数会产生错误。同样地,把一个volatile对象传递给一个非volatile型的函数同样会产生错误。

成员函数说明为 const 型,则不能改变成员数据。在这种函数中,this 指针是一

个指向常量的指针。
注意:构造函数和析构函数是不能说明为 const 或 volatile 的。然而它们可以被 const 或 volatile 对象调用。

静态成员函数

静态成员函数被认为具有类的范围。同非静态成员相比,这些函数没有隐含的 this 参数;因而,它们只能直接使用静态的数据成员、枚举或嵌套类型。静态成员函数的存储可以不必有相应的类类型对象。考虑下面的代码:

```
class WindowManager
{
public:
    static int CountOf(); //返回打开窗口的个数
          void Minimize();//最小化当前窗口
           WindowManager SideEffects();  //边界效果(副作用)函数
  ...
private:
   static int wmWindowCount;
};

int WindowManager::wmWindowCount=0;
...
//最小化(显示图标)所有窗口
```

```
for(int i=0;i<WindowManager::CountOf();++i)
    rgwmWin[i].Minimize();
```

在上面的代码中,类 WindowManager 包含有静态成员函数 CountOf。该函数返回打开窗口的个数,但却不必同某个给定的 WindowManager 类对象相联系。这一概念在循环中得以体现。在循环中 CountOf 用于控制表达式。因为 CountOf 是一个静态成员函数,因此无需引用对象,此函数即可调用。

静态成员函数有外部连接。这些函数没有 this 指针(下一章中论述)。结果,这些函数必须遵循下面的一些约束:

- 不能用成员选择符(.或->)来访问非静态成员。
- 不能说明为虚函数。
- 不能同有相同参数类型的非静态成员函数同名称。

注意:选择了一个静态成员函数的成员选择符(.或->),其左边是不可定值的。在有副作用的函数同此函数一起使用时,这一点可能很重要。例如:表达式 SideEffects().CountOF()中,并不会调用 SideEffects 函数。

静态数据成员

类可以包含静态的成员数据或成员函数。当一个数据说明为 static,则对于该类的所有对象仅维持该成员的一个拷贝(相关的情况见前一节的“静态成员函数”)。

静态数据成员并不是给定类类型对象的一部分;它们是单独的对象。结果,静态数据成员的说明不能看作是定义。数据成员的说明是在类范围之中的,但其定义

是在文件范围中实现的。这些静态成员具有外部连接。下面的例子显示了这一点：

```
class BufferedOutput
{
public:
    //返回该类对象所写的字节大小
    short BytesWritten() { return bytecount;}

    //复位计数器
    static void ResetCount() { bytecount=0; }

    //静态成员的说明
    static long bytecount;
};

//在文件范围中定义 bytecount
long BufferedOutput::bytecount;
```

在前面的代码中,成员 bytecount 是在类 BufferOutput 中说明的,但它必须在类说明之外定义。

不必引用某类类型的对象即可引用静态数据成员。引用 BufferedOutput 对象所写的字节大小可以按如下方式得到：

```
long nBytes=BufferedOutput::bytecount;
```

静态成员不会因为该类类型的某个对象的结束而结束。静态成员当然也可以用成员选择符(.或->)来访问。例如:

```
BufferedOutput Console;

long nBytes=Console.bytecount;
```

在上面的例子中,对对象 console 的引用是不可求值的。返回的值是静态对象bytecount。

静态数据成员同样要受制于类成员的访问规则,因此私有存储的静态数据成员只允许类成员函数和友元访问。这些规则在第 10 章“成员访问控制”中描述。例外的是:静态数据成员必须在文件范围中定义,并忽视它们的存储约束。如果数据成员要明确地初始化,则必须同定义一起提供初始化器。

静态成员的类型并不被它的类名称所限定。因此:BufferedOutput::bytecount的类型是 long 型。

联　合

联合这种类类型在一个时间点只能包含一种数据元素(尽管数据元素可以是数组或者类类型)。联合的成员代表的是联合所能包含的数据种类。一个联合类型的对象需要足够的空间去容纳成员表中最大的成员。考虑下面的例子:

```
#include <stdlib.h>
#include <string.h>
#include <limits.h>
```

```
union NumericType     //说明一个可容纳下面数据的联合
{
  int         iValue;//整型值
  long        lValue;//长整型值
  double      dValue;//浮点型值
}

void main (int argc,char *argv[])
{
  NumericType *Values=new NumericType[argc-1];
  for(int i=1;i<argc;++i)
      if(strchr(argv[i],' . ')!=0)
        //浮点型用 dValue 成员进行赋值
        Values[i].dValue=atof(argv[i]);
      else
          //如果大于最大的整型数,则存到 lValue 成员中
          if(atol(argv[i])>INT_MAX)
             Values[i].lValue=atol(argv[i]);
          else
            //否则,存到 iValue 成员中
            Values[i].iValue=atoi(argv[i]);
       }
```

```
}
```

NumericType 联合在存储器中的位置示意图如图 8.1。

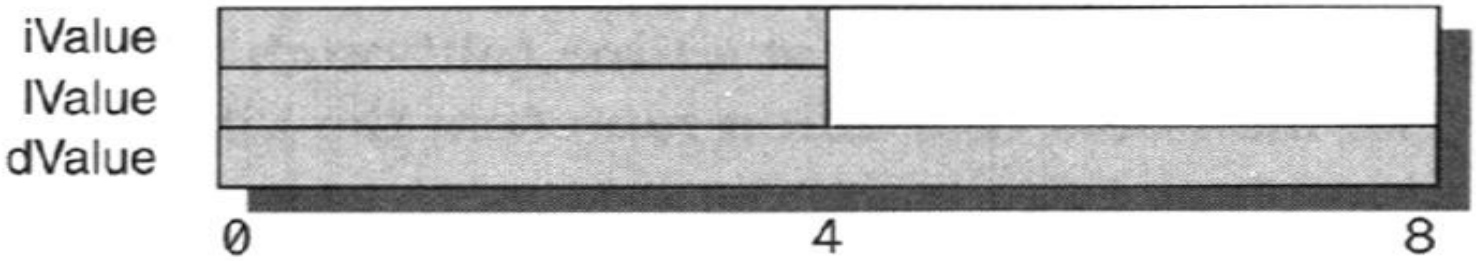


图 8.1　NumericType 联合的数据存储

联合中的成员函数

正如在成员函数中描述过的,除了有成员数据外,联合也可以有成员函数,尽管联合可以有特殊成员函数如构造函数和析构函数,但联合不能有虚拟函数(有关的更多的信息参见第 11 章“特殊成员函数”中的“构造函数”以及“析构函数”)。

作为类类型的联合

联合不可以有基类;因此,它们不能从其它的联合、结构和类中继承属性。联合也不能作为进一步继承的基类。
继承将在第 9 章“派生类”中详细讨论。

联合的成员数据

除了以下所述之外,联合可以在成员表中包含大多数的数据类型:

- 具有构造函数或析构函数的类类型
- 具有自定义的赋值操作符的类类型
- 静态数据成员

无名联合

无名联合是在说明时不带类名或说明符表的联合。

语法

union {成员表};

这种联合说明所说明的不是类型,而是对象。在无名联合中说明的名称不能跟在同一范围中的其它名称冲突。在无名联合中说明的名称可以直接使用,就像并不是成员变量一样。下面例子显示了这一点。

```
#include <iostream.h>

struct DataForm
{
    enum DataType {CharData=1,IntData,StringData};
    DataType type;

    //说明一个无名联合
```

```
    union
    {
       char chCharMem;
       char *szStrmem;
       int  iIntMem;
    };
    void print();
};

void DataForm::print()
{
  //基于 type 的数据类型打印合适的数据类型
  switch(type)
 {
 case CharData:
      cout << chCharMem;
      break;
case IntData:
      cout<<szStrMem;
      break;
case StringData:
      cout << iIntMem;
```

```
 }
}
```

在函数 DataForm::print 中,三个成员(chCharMem、szStrMem 和 iIntMem)的访问就像他们是作为成员说明的一样(没有联合说明)。然而三个联合的成员是共享同一存储器的。

联合的成员数据除了上述所列的一些限制之外,还要受下列约束的限制:

- 如果在文件范围中说明,则必须说明为静态的。
- 它们仅能含有公有的成员;私有的和保护的成员在无名联合中会产生错误。
- 它们不能有成员函数。

注意:仅简单地省略语法上的类名部分并不会使一个联合成为无名联合,要把一个联合限制为无名联合,则说明一定不要说明对象。

位　域

类和结构可以拥有在存储空间上小于一个整型的成员。这些成员被说明为位域。位域成员说明符的语法规格如下:

语法

说明符 $_{opt}$:常量表达式

说明符是程序中用来存储成员的名称。它必须是一个整型(包括枚举型)。常量表达式说明了成员在结构中所占的位数。无名位域也即没有标识符的位域成员,可以用作填充。

注意:一个无名的零宽度位域会强制使下一个位域对齐到下一个类型边界。这里类型指的是成员的类型。

下面的例子说明了一个含有位域的结构:

```
struct Date
{
   unsigned nWeekDay  :3;   //0..7(3 位)
   unsigned nMonthDay :6;   //0..31(6 位)
   unsigned nMonth    :5;   //0..12(5 位)
   unsigned nYear     :8;   //0..100(8 位)
};
```

一个 Date 类型对象的存储器布局如图 8.2 所示。

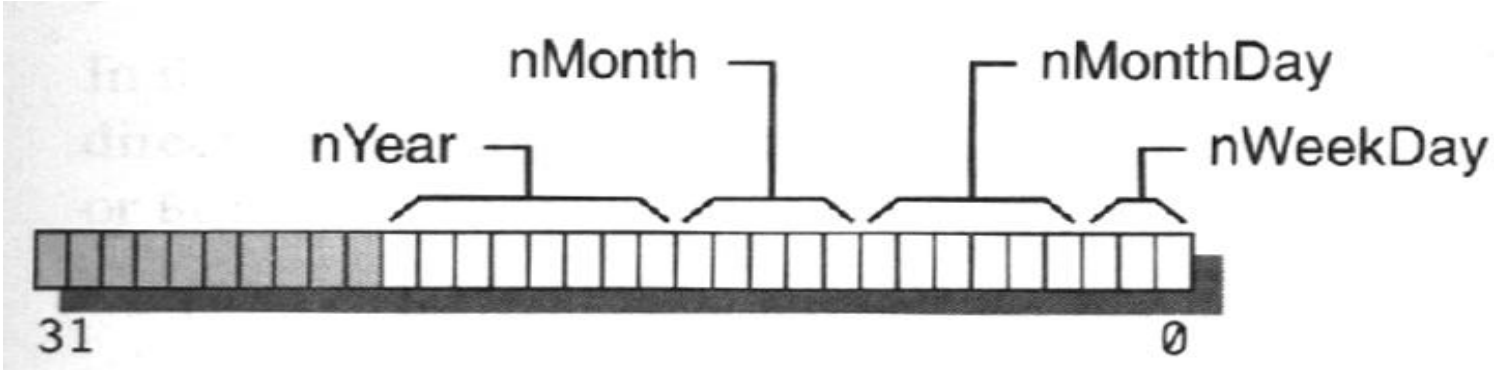


图 8.2 Date 对象的存储器布局

注意:nYear 是 8 位长的而且超出了说明的 unsigned int 类型的字边界。因此 nYear 在一个新的 unsigned int 上继续。没有必要把所有的位域都塞到一个所基于的类型对象之中。根据说明中所需求的位域的大小,可以分配新的存储单

元。
Microsoft 特殊处→
作为位域说明的数据的排列顺序是从低到高。
Microsoft 特殊处结束
如果在结构说明中包含一个无名零长度位域,如下例所示:

```
struct Date
{
  unsigned nWeekDay  :3;   //0..7(3 位)
  unsigned nMonthDay :6;   //0..31(6 位)
  unsigned           :0;   //强迫对齐到下一个边界
  unsigned nMonth    :5;   //0..12(5 位)
  unsigned nYear     :8;   //0..100(8 位)
}
```

其存储器分布如图 8.3 所示。

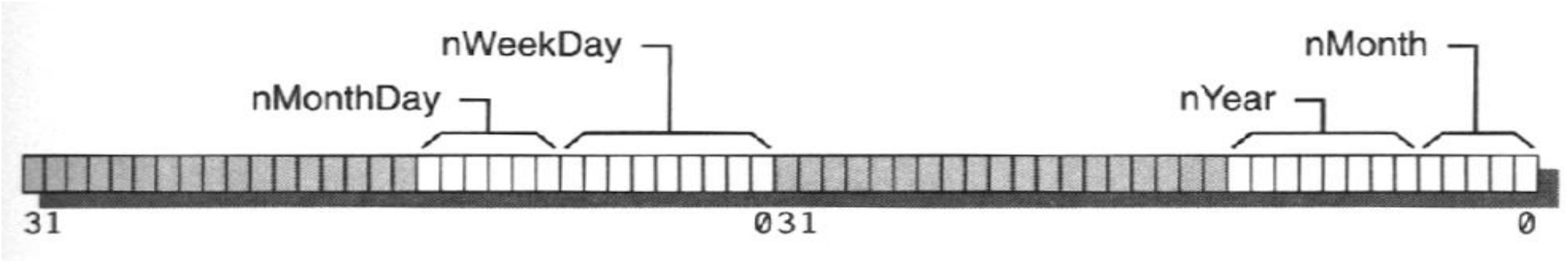


图 8.3　具有零长度位域的数据对象的分布

位域所基于的类型必须是整型。见第 2 章“基本概念”中的基本类型。

使用位域的限制

下面列举了对位域的明细错误操作:

- 取一个位域的地址
- 初始化

嵌套类说明

类可以在其它类的范围之内说明。这种类称为“嵌套类”。嵌套类被视为在外围封闭类的范围之中,并只能在此作用之中有效地使用。要引用一个嵌套类而从它的直接外围类的范围之外,则必须使用全限定名。
下面的例子显示了如何说明嵌套类。

```
class BufferedIO
{
public:
    enum IOError { None,Access,General };

    //说明嵌套类 BufferedInput
    class BufferedInput
    {
    public:
    int read();
    int good() { return _inputerror == None; }

    private:
      IOError _inputerror;
    }

      //说明嵌套类 BufferedOutput
      class BuufferedOutput
    {
       //成员表
     };
};
```

BufferedIO::BufferedIuput 和 BufferedIO::BufferedOutput 是在 BufferedIO 之中说明的。这些类名称在类 BufferedIO 范围之外是不可见的。但是一个 BufferedIO 型对象不会含有任何 BufferedIuput 和 BufferedOutput 型的对象。
嵌套类可以直接使用来自于外围类的名称、静态成员名称、类型名称以及枚举名称。
为了使用其它类成员,必须使用指针、引用或对象名称。
在上面的 BufferedIO 的例子中,枚举 IOError 可以由嵌套类 BufferedIO::BufferedInput 和 BufferedIo::BufferedOutput 的成员函数直接访问。如函数 good 中所示的。
注意:嵌套类仅仅只在类范围中说明了类型。它不会引起创建嵌套类的对象。上面的例子只是说明了两个嵌套类但并未说明任何这些类的对象。

访问权限和嵌套类

在别的类中的嵌套类并没有给嵌套类的成员函数以特殊的权限。同样,类中包括的成员函数对于嵌套类的成员也没有什么特殊的访问权限。

嵌套类中的成员函数

在嵌套类中说明的成员函数可以在文件范围中定义,前面的例子可以写为:

```
class BufferedIO
{
public:
```

```
    enum IOError { None,Access,General};
    class BufferedInput
    {
    public:
        int read();     //说明但不定义成员函数 read 和 good
        int good();
    private:
        IOError _inputerror;
    }

     class BufferedOutput
    {
        //成员表
    };
}
//在文件范围中定义成员函数 read 和 good
int BufferedIO::BufferedInput::read()
{
    ...
};

int BufferedIO::BufferedInput::good()
```

```
{
    return _inputerror==None;
}
```
在上面的例子中,运用限定类型名称语法说明成员函数名称，说明为：
```
BufferedIO::BufferedInput::read()
```
意味着函数 read 是在类 BufferedIO 的范围中的 BufferedInput 类的成员函数。因为这一说明使用了限定的类型名语法,因而如下形式的构造是可能的：
```
typedef BufferedIO::BufferedInput BIO_INPUT

int BIO_INPUT::read()
```
上面的说明是同前一个说明等价的,但它用一个 typedef 名称代替了类名称。

友元函数和嵌套类

在嵌套类中说明的友元函数被视为在嵌套类的范围中,而不在类的范围中。因此对于包含在类中成员和成员函数,这些友元函数并没有获得多少特别的访问权限。如果你要在定义于文件范围中的友元函数中使用一个在嵌套类中说明的名称,必须像下面那样使用限定类型名称：
```
extern char *rgszMessage[];

class BufferedIO
{
public:
```

```
  ...
class BufferedInput
  {
  public:
     friend int GetExtendedErrorStatus();
      ...
     static char *message;
     int  iMsgNo;
 };
};
char *BufferedIO::BufferedInput::message;

int GetExtendedErrorStaus()
{
  ...
  strcpy(BufferedIO::BufferedInput::message,
       rgszMessage[iMsgNo]);
  return iMsgNo;
}
```

在上面的代码中函数 GetExtendedErrorStatus 是作为友元函数说明的,在这个函数中(它定义于文件范围),一条消息从静态数组中拷贝到类成员中。注意GetExtendedErrorStatus 的一个较好的实现是说明:

```
int GetExtendedErrorStatus (char * message)
```

用前面的接口,几个不同的类可以使用此函数的功能,只要把要拷贝的错误消息的地址传给函数即可。

类范围中的类型名称

在类范围中定义的类型名称应视为局限于它们的类。它们不能在类的范围之外使用。
下面的例子显示了这一概念:

```
class Tree
{
 public:
      typedef  Tree *PTREE;
      PTREE  Left;
      PTREE  Right;
      void   *vData;
};

PTREE pTree;  //错误:不在类的范围中
```

第 9 章　派　生　类

这一章解释如何用派生类产生可扩展的程序。
本章包含如下一些主题：

- 派生类综述
- 多重基类
- 虚拟函数
- 抽象类
- 范围规则总结

派生类概述

利用继承机制，新的类可以从已有的类中派生（有关继承见下一节“单一继承”的开始）。那些用于派生的类称为这些特别派生出的类的“基类”。派生类的说明可以用下面的语法。
语法
基类说明：
　　：基类表
基类表：

基类说明符
基类表,基类说明符
基类说明符:
完全类名称
virtual 访问说明符$_{opt}$ 完全类名称
访问指示符 virtual$_{opt}$ 完全类名称
访问指示符:
private
protected
public

单一继承

在“单一继承”这种最普通的形式中,派生类仅有一个基类,考虑如图 9.1 所示的关系。

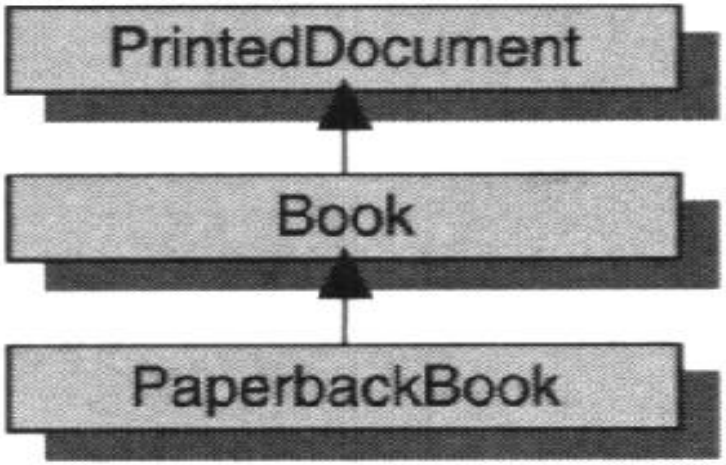


图 9.1　简单单一继承图

注意图 9.1 中的从一般到特殊的过程。在类的层次设计中,可以发现一些普遍的特性,即派生类总是同基类有“kind of”关系。在图 9.1 中书是一种印刷好的文档而一本平装书是一种书。

在图 9.1 中的另一个值得注意点是 Book 既是派生类(从 PrintedDocument 中派生),也是基类(PaperbackBook 是从 Book 派生的)。下面的例子是这种类层次的一个轮廓性的说明。

```
class PrintedDocument
{
    //成员表
};

//Book 是从 PrintedDocument 中派生的
```

```
class Book:public PrintedDocument
{
    //成员表
};

//PaperbackBook是从Book中派生
class PaperbackBook: public Book
{
    //成员表
};
```

PrintedDocument作为Book的直接基类,它同时也是PaperbackBook的非直接基类。直接基类和非直接基类的区别在于直接基类出现在类说明的基类表中,而非直接基类不出现在基类表中。

每个派生类的说明是在基类的说明之后说明的,因此对于基类仅只给出一个前向引用的说明是不够的,必须是完全的说明。

在前面的例子中,使用的访问说明符是public。公有继承、私有继承以及保护的继承在第10章“成员访问控制”中讲述。

一个类可以作为很多特别类的基类,如图9.2所示。

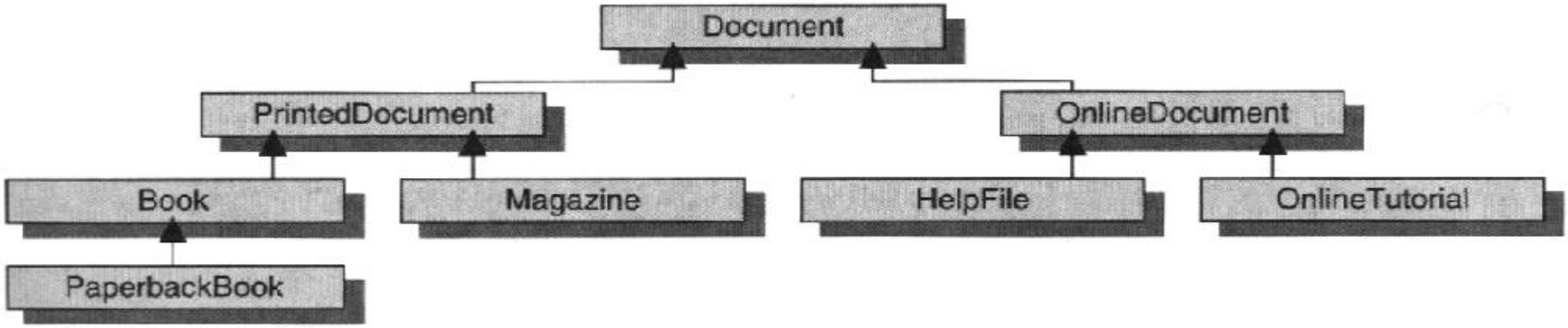


图 9.2　有向无环图的例子

在图 9.2 中的图叫“有向无环图”(DAG)。有一些类是多个派生类的基类。但反过来不是真的:对于任意给的派生类仅有一个直接基类。图 9.2 描绘了单一继承的结构。

注意:有向无环图并不仅用于单一继承。它们也可以用于多重继承图。这一主题将在下一节中的“多重继承”中论述。

在继承中,派生类含有基类的成员加上任何你新增的成员。结果派生类可以引用基类的成员(除非这些成员在派生类中重定义了)。当在派生类中重定义直接基类或间接基类的成员时,可以使用范围分辨符(::)引用这些成员。考虑下面的代码:

```
class Document
{
 public:
    char * Name;//文档名称
```

```
    void PrintNameOf(); //打印名称
};

//实现类 Document 的 PrintNameOf 函数
void Document::PrintNameOf()
{
    cout << Name << end ;
}

class Book:public Document
{
public:
    Book(char *name, long pagecount);
private:
    long PageCount;
};

//class Book 构造函数
Book::Book (char *name, long pagecount)
{
    Name=mew char [strlen(name)+1];
    strcpy (Name,name);
```

```
    PageCount=pagecount;
};
```
注意，Book 的构造函数(Book::Book)具有对数据成员 Name 的访问权。在程序中可以按如下方式创建 Book 类对象并使用之。
```
//创建一个 Book 类的新对象,这将激活构造函数 Book:Book
Book LibraryBook ("Programming Windows,2nd Ed",994);
...

//使用从 Document 中继承的函数 PrintNameOf.
LibraryBook.PrintNameOf();
```
如前面例子所示,类成员和继承的数据与函数以一致的方式引用。如果类 Book 所调用的 PrintNameOf 是由类 Book 重新定义实现的,则原来属于类 Document 的 PrintNameOf 函数只能用范围分辩符(::)才能使用：
```
class Book:public Document
{
   Book(char *name,long pagecount);
   void PrintNameOf();
   long PageCount;
};
void Book::PrintNameOf()
{
   cout<<"Name of Book:";
```

```
    Document::PrintNameOf();
}
```

只要有一个可访问的、无二义性的基类,派生类的指针和引用可以隐含地转换为它们基类的指针和引用。下面的例子证实了这种使用指针的概念(同样也适用于引用):

```
#include <iostream.h>

void main()
{
  Document * DocLib[10];    //10 个文档的库
  for (int i=0; i<10; ++i)
  {
     cout<<"Type of document:"
         <<"P)aperback,M)agazine,H)elp File,C)BT"
         << endl;
     char CDocType;
     cin >>CDocType;
     switch(tolower(CDocType))
     {
     case 'p':
         DocLib[i]=new PaperbackBook;
         break;
```

```
      case 'm':
         DocLib[i]=new Magazine;
         break;
      case 'h':
          DocLib[i]=new HelpFile;
          break;
       case 'c':
          DocLib[i]=new ComputerBasedTraining;
          break;
       default:
          --i;
         break;
       }
    }
    for (i=0; i<10; ++i)
        DocLib[i]->PrintNameOf();
}
```

在前面例子的 SWITCH 语句中,创建了不同类型的对象。这一点依赖于用户对 CDocType 对象所作出的说明。然而这些类型都是从类 Document 中派生出来的,故可以隐含地转换为 Document*。结果是 DocLib 成为一个“相似链表”(heterogeneous list)。此链表所包含的是不同种类的对象,其中的所有对象并不是有相同的类型。

因为 Document 类有一个 PrintNameOf 函数。因此它能够打印图书馆中每本书的名称,但对于 Document 类型来说有一些信息会省略掉了(如:Book 的总页数,HelpFile 的字节数等)。
注意:强制基类去实现一个如 PrintNameOf 的函数,通常不是一个很好的设计,本章后面的“虚拟函数”中提供了一个可替换的设计方法。

多重继承

C++的后期的一些版本为继承引入了“多重继承”模式。在一个多重继承的图中,派生类可以有多个直接基类。考虑图 9.3。

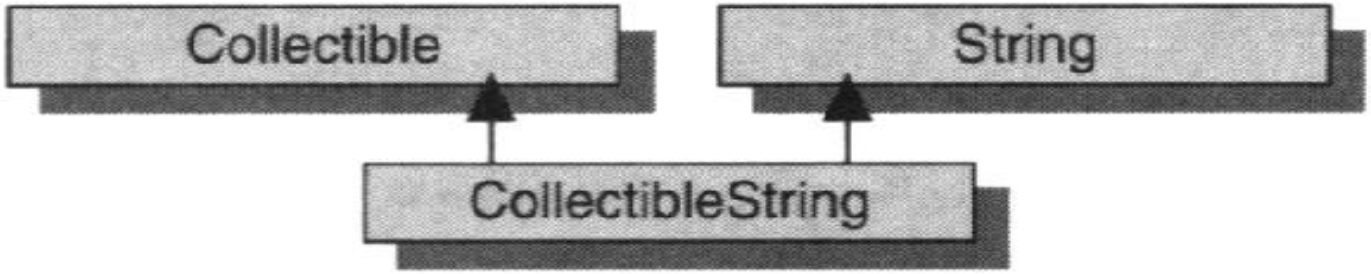


图 9.3 简单多重继承图

如图 9.3 所示的图中,显示了一个 CollectibleString 类。该类既像 Collectible 类(一种可包容聚集的类),又像 String 类。对于派生类需要多个基类的属性的问题,多重继承是一种很好的解决办法。因而也很容易派生出 CollectibleCustomer 和 CollectibleWindow 等等。

对于一个特定的程序如果每个类的属性并不是全部要求使用,则每个类可以单独使用或者同别的类联合在一起使用。因此把图 9.3 所描绘的类层次作为基础,用户很容易组织出不可收集的字符串或可收集的非字符串。对于使用单一继承,则没有这种便利性。

虚基类层次

有一些类层次很庞大,但有很多东西很普遍。这些普遍的代码在基类中实现了,然而在派生类中又实现了特殊的代码。
对于基类来说重要的是建立一种机制,通过这种机制派生类能够完成大量的函数机能。
这种机制通常是用虚函数来实现的。有时,基类为这些函数提供了一个缺省的实现。如在图 9.2 的 Document 类层次中,两个重要的函数是 Identify 和 WhereIs。当调用 Identify 函数时,返回一个正确的标识。对于各种文档来说正确的是:对于 Book,调用如 doc->Identify()的函数必须返回 ISBN 编号;而对于一个 HelpFile 返回产品名和版本号更合理一些。同样,WhereIs 函数对于一本书来说应该返回行和书架号,但对于 HelpFile 就应该返回它的磁盘位置,也许是一个目录和名称。
了解到所有的 Identify 和 WhereIs 的函数实现返回的是同种类型的信息,这一点很重要。
在这个例子中,恰好是一种描述性字符串。
这些函数可以作为虚拟函数来实现,然后用指向基类的指针来调用,对于实际代码的联结将在运行时决定,以选择正确的 Identify 和 WhereIs 函数。

类协议的实现

类可以实现为要强制使用某些协议。这些类称为“抽象类”,因为不能为这种类类型创建对象。它们仅仅是为了派生别的类而存在。
当一个类中含有纯虚拟函数或当他们继承了某些纯虚拟函数却又没有为它们提供一个实现时,该类称为抽象类。纯虚拟函数是用纯说明符定义的虚拟函数。如下:

```
virtual char *Identify()=0;
```

基类 Document 把如下一些协议强加给派生类。

- 为 Identify 函数提供一个合适的实现
- 为 WhereIs 函数提供一个合适的实现

在设计 Document 类时,通过说明这种协议,类设计者可以确保如不提供 Identify 和 WhereIs 函数则不能实现非抽象类。因而 Document 类含有如下说明:

```
class Document
{
public:
    ...
 //对派生类的要求,它们必须实现下面这些函数
    virtual char *Identify()=0;
    virtual char *WhereIs()=0;
    ...
};
```

基　类

如前面讨论的,继承过程创建的新的派生类是由基类的成员加上由派生类新加的成员组成。在多重继承中,可以构造层次图,其中同一基类可以是多个派生类的一部分。图 9.4 显示了这种图。

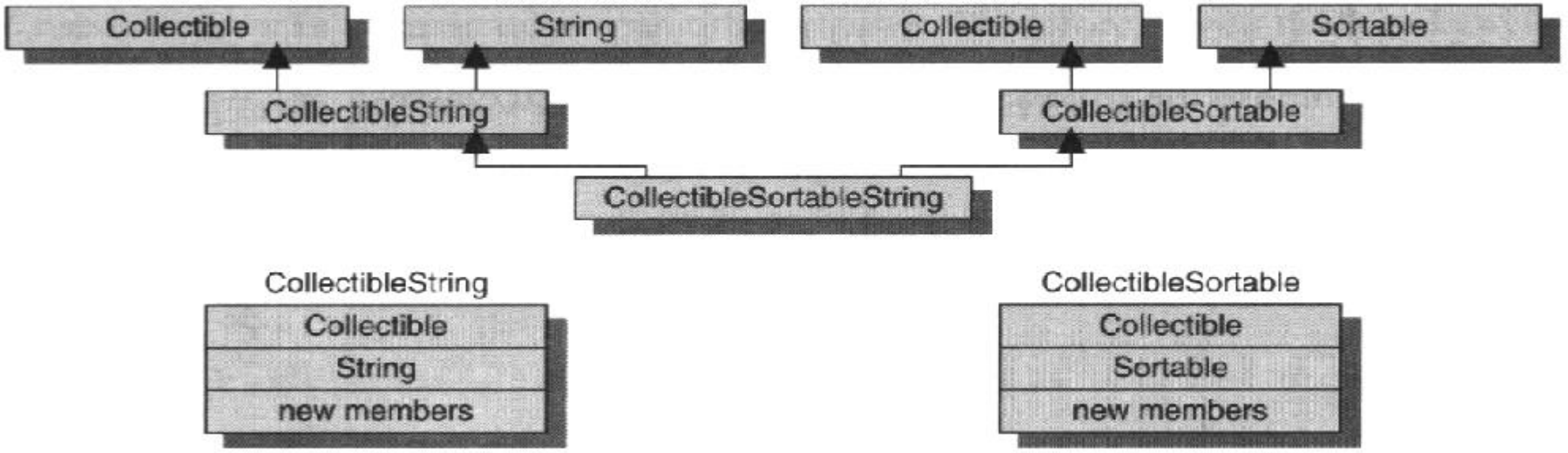


图 9.4　单个基类的多重实例

在图 9.4 中以图的形象表达了 CollectibleString 和 CollectibleSortable 的组成。然而,基类 Collectible 通过路径 CollectibleSortable 以及 CollectibleString 到达类 CollectibleSortableString。为了消除这种冗余,当这些类被继承时,可以说明为虚拟基类。

有关说明虚拟基类以及带有虚拟基类的对象是如何组成的,见本章后面的“虚拟

基类”。

多重基类

如同多重继承中所描述的,一个类可以从多个基类中派生出来。在派生类由多个基类派生出来的多重继承模式中,基类是用基类表语法成份来说明的(见本章开始的“概述”中的语法)。例如:CollectionOfBook 类是由类 Collection 和 Book 类派生的,可按如下进行说明:

```
class CollectionOfBook:public Book,public Collection
{
    //新成员
};
```

基类的说明顺序一般没有重要的意义,除非在某些情况下要调用构造函数和析构函数的时候。在这些情况下,基类的说明顺序会对下面所列的有影响。

- 由构造函数引起的初始化发生的顺序。如果你的代码依赖于 CollectionOfBook 的 Book 部分要在 Collection 部分之前初始化,则此说明顺序将很重要。初始化是按基类表中的说明顺序进行初始化的。
- 激活析构函数以作清除工作的顺序。同样,当类的其它部分正在被清除时,如果某些特别部分要保留,则该顺序也很重要。析构函数的调用是按基类表说明顺序的反向进行调用的。

注意:基类的说明顺序会影响类的存储器分布。不要对基类成员在存储器中的顺

序作出任何编程的决定。
在你说明基类表时,不能把同一类名称说明多次。但是对于一个派生类而言,其非直接基类可以有多个相同的。

虚拟基类

因为一个类可以多次作为一个派生类的非直接基类。C++提供了一个办法去优化这种基类的工作。研究图 9.5 中的类层次,它显示了一个模拟的午餐线。

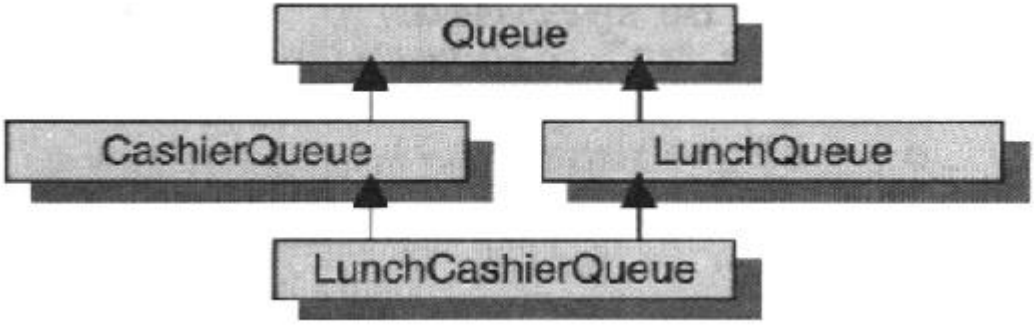


图 9.5 模拟午餐线图

在图 9.5 中,Queue 是 CashierQueue 和 LunchQueue 的基类。但是当这两个类联合在一起形成 LunchCashierQueue 时,下面的问题就产生了:新的类包含有两个 Queue 类型的子对象,一个来自于 CachierQueue,另一个来自于 LunchQueue。图 9.6 给出了一个概念上的存储器分布(实际的内容公布可能会进行优化)。

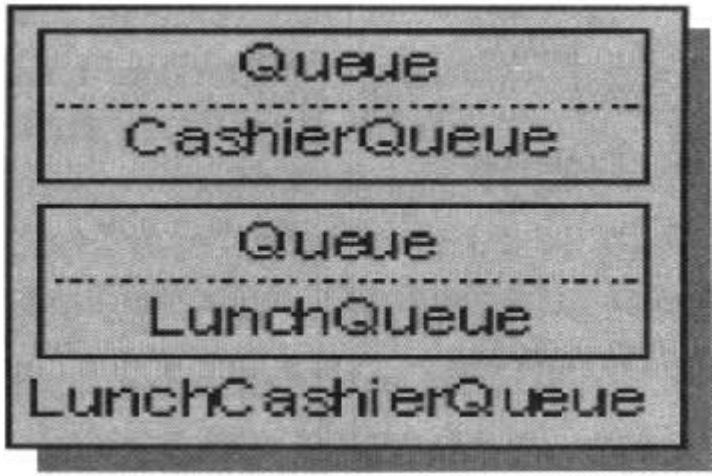


图 9.6　模拟的午餐线对象

注意,在 LunchCashierQueue 对象中,有两个 Queue 子对象。下面的代码说明 Queue 为虚拟基类:

```
class Queue
{
   //成员表
};

class CashierQueue:virtual public Queue
{
   //成员表
};
```

```
class LunchQueue:  virtual public Queue
{
  //成员表
};

class LunchCashierQueue:public LunchQueue, public CashierQueue
{
    //成员表
};
```

关键字 virtual 确保了仅有一个 Queue 对象的拷贝(见图 9.7)。

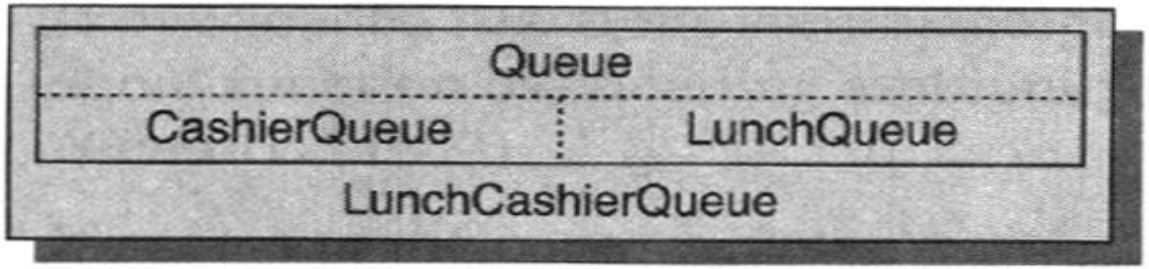


图 9.7　带有虚拟基类的模拟午餐线对象

一个类对于给定的类型既可以有虚拟的组成部分,也可以有非虚拟的组成部分。这种情况发生在图 9.8 所示的情况下。

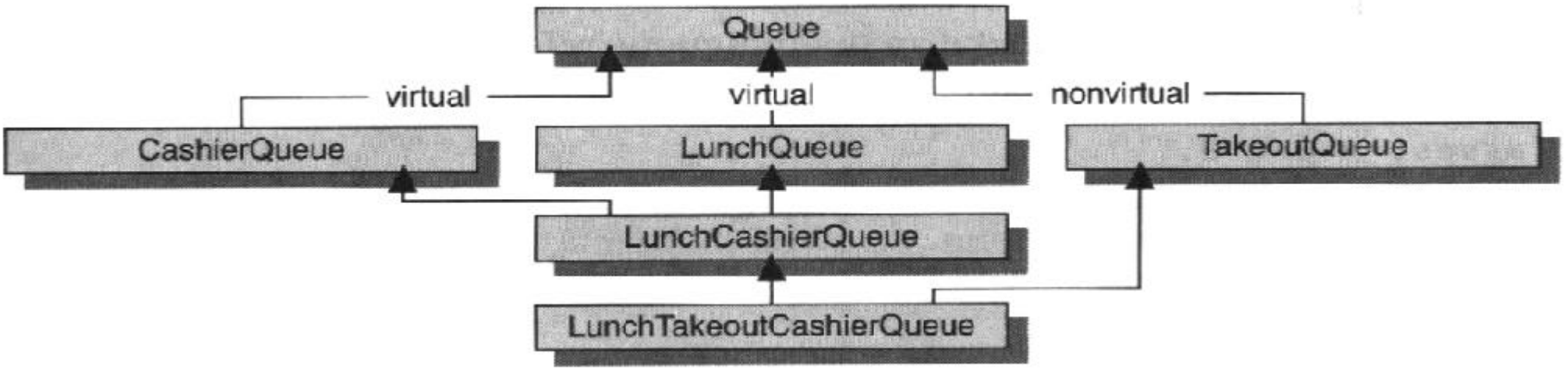


图 9.8　同一个类的虚拟的和非虚拟的组成部分

在图 9.8 中,CachierQueue 和 LunchQueue 用 Queue 作为虚拟基类。但是 TakeoutQueue 仅说明 Queue 为基类,并不是虚拟基类。因此 LunchTakeoutCashierQueue 有两个 Queue 子对象:
一个是从包括 LunchCashierQueue 的路径中继承而来的,另一个则是从包括 TakeoutQueue 的路径中而来,图 9.9 显示了这一点。

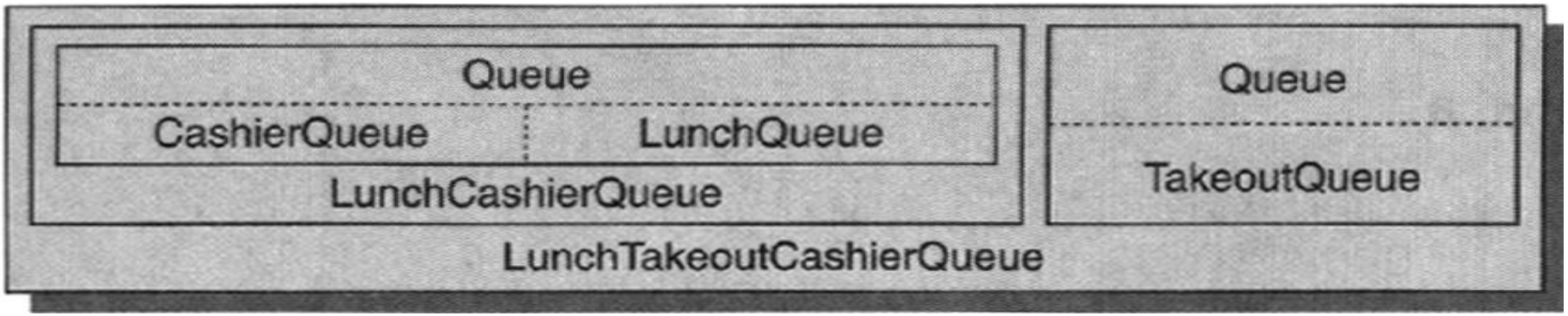


图 9.9 带有虚拟和非虚拟继承的对象的分布

注意:虚拟继承同非虚拟继承相比具有大小上的好处,然而它也引入了额外的运行开销。

如果一个派生类重载了一个从虚拟基类中继承的虚拟函数,而且该派生类以指向虚拟基类的指针调用这些构造函数和析构函数时,编译器会引入一个附加的隐含的“vtordisp”域到带有虚拟基类的类中。/vd0 编译器选项禁止了这个增加的隐含 vtordisp 构造/析构位置成员。/vd1 选项(缺省),使得在需要时可以解除禁止。只有在你确信所有类的构造函数或析构函数都虚拟地调用了虚拟函数,vtordisp 才可以关掉。

/vd 编译器选项会影响全局编译模式。使用 vtordisp 编译指示可以在基于类方式上打开或禁止 vtordisp 域:

```
#pragma vtordisp(off)
class GetReal:virtual public{...};
#pragma vtordisp(on)
```

名称的二义性

多重继承使得从不同的路径继承成员名称成为可能。沿着这些路径的成员名称并不必然是唯一的。这些名称的冲突称为“二义性”。
任何引用类成员的表达式必须使用一个无二义性的引用。下面的例子显示了二义性是如何发生的。

```
//说明两个基类 A 和 B
class A
{
public:
    unsigned a;
    unsigned b();
};

class B
{
   public:
        unsigned a();  //注意类 A 也有一个成员"a"和一个成员"b"
        int b();
        char c;
};
//定义从类 A 和类 B 中派生出的类 C
```

```
class C : public A, public B
{
};
```
按上面所给出的类说明,如下的代码就会引出二义性,因为不清楚是引用类 A 的 b 呢,还是引用类 B 的 b:

```
C *pc=new C;
pc->b();
```
考虑一下上面的代码,因为名称 a 既是类 A 又是类 B 的成员,因而编译器并不能区分到底调用哪一个 a 所指明的函数。访问一个成员,如果它能代表多个函数、对象、类型或枚举则会引起二义性。

编译器通过下面的顺序执行以检测出二义性:

1. 如果访问的名称是有二义性的(如前述),则产生一条错误信息。
2. 如果重载函数是无二义性的,它们就没有什么问题了(有关重载函数二义性的情况,见第 12 章“重载”中的“参量匹配”)。
3. 如果访问的名称破坏了成员访问许可,则产生一条错误信息(有关信息详见第 10 章“成员访问控制”)。

在一个表达式产生了一个通过继承产生的二义性时,通过用类名称限制发生问题的名称即可人工解决二义性,要使前面的代码以无二义性地正确编译,要按如下使用代码:

```
C *pc = new C;

pc->B::a();
```

注意:在类 C 说明之后,在 C 的范围中引用 B 就会潜在地引起错误。但是,直到在 C 的范围中实际使用了一个对 B 的无限定性的引用,才会产生错误。

二义性和虚拟基类

如果使用了虚拟基类、函数、对象、类型以及枚举可以通过多重继承的路径到达,但因为只有一个虚拟基类的实例,因而访问这些名称时,不会引起二义性。图 9.10 显示了采用虚基类和非虚基类的对象的组成。

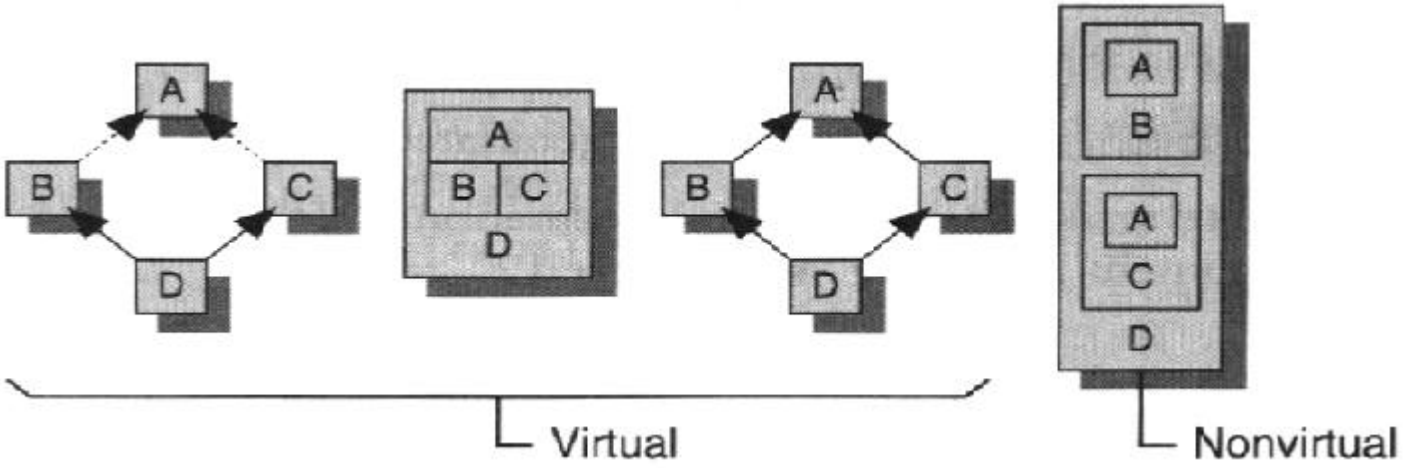


图 9.10 虚拟的及非虚拟的派生

在图 9.10 中,访问任何类 A 的成员,通过非虚拟基类访问则会引起二义性;因为编译器没有任何信息以解释是使用同类 B 联系在一起的子对象,还是使用同类 C 联系在一起的子对象,然而当 A 说明为虚拟基类时,则对于访问哪一个子对象不存在问题了。

支配

通过继承图可能有多个名称(函数的、对象的、枚举的)可以达到。这种情况视为非虚拟基类引起的二义性。但虚拟基类也可以引起二义性,除非一个名称“支配”(dominate)了其它的名称。

一个名称支配其它的名称发生在该名称定义在两个类中,其中一个是由另一个派生的,占支配地位的名称是派生类中的名称,在此名称被使用的时候,相反不会产生二义性,如下面的代码所示:

```
class A
{
public:
    int a;
};

class B: public virtual A
{
public:
    int a();
};

class C: public virtual A
{
 ...
```

```
};

class D: public B,public C
{
public:
    D() {a();} //不会产生二义性,B::a()支配了 A::a
};
```

转换的二义性

显式地或隐含地对指向类类型的指针或引用的转换也可引起二义性。图 9.11 显示了如下几点：

- 说明了一个类型 D 的对象。
- 把取地址运算符用于此对象效果。注意，地址运算符总是支持对象的基类地址。
- 显式地把取地址运算符得到的指针转换为指向基类类型 A 的指针。注意，把对象的地址造型转换成类型 A*,通常并未给编译器提供足够的信息以确定到底是选择哪一个 A 类型的子对象。在此情况下，存在着两个 A 类型的子对象。

对于到 A*的转换是有二义性的,因为没有办法分辨出哪一个 A 类型的子对象是正确的。

注意只要显式地说明你想要使用的是哪一个子对象，如下：

```
(A*)(B*)&d    //使用 B 的子对象
```

(A*)(C*)&d　　//使用 C 的子对象

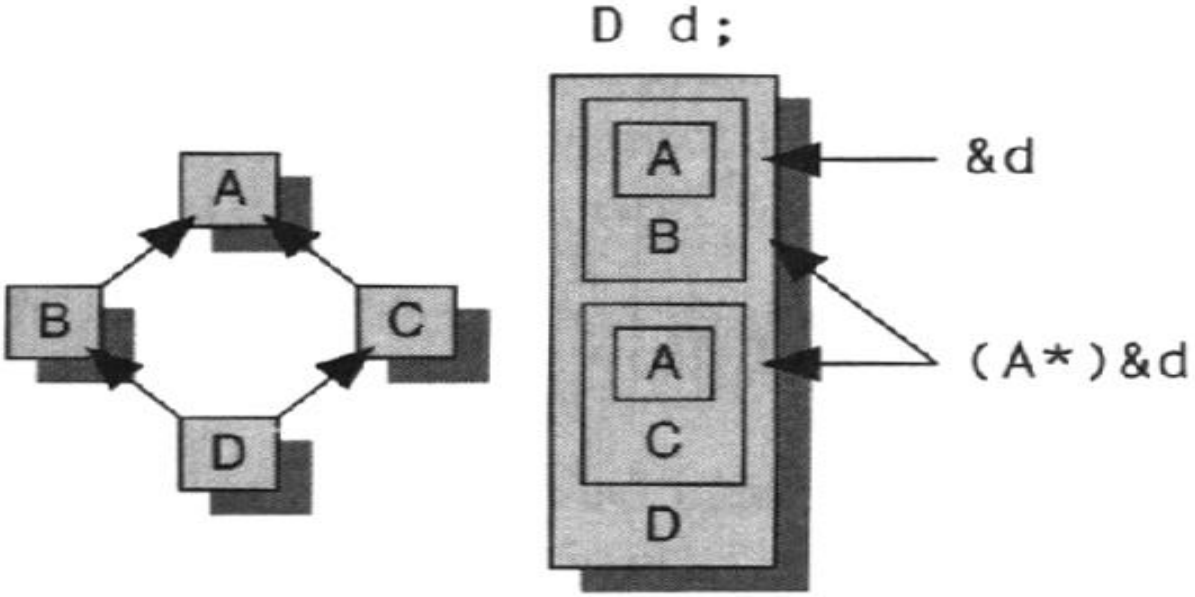


图 9.11　有二义性的指向基类的指针的转换

虚拟函数

虚拟函数可以确保在一个对象中调用正确的函数,而不管用于调用函数的表达式。
假设一个基类含有一个说明为虚拟函数同时一个派生类定义了同名的函数。派生类中的函数是由派生类中的对象调用的,甚至它可以用指向基类的指针和引用来调用。下面的例子显示了一个基类提供了一个 PrintBalance 函数的实现:

```
class Account
{
public:
            Account(double d);    //构造函数
    virtual double GetBalance();    //获得平衡
    virtual void PrintBalance();    //缺省实现
private:
    double _balance;
};

//构造函数 Account 的实现
double Account::Account(double d)
{
     _balance=d;
}

//Account 的 GetBalance 的实现
double Account::GetBalance()
{
    return _balance;
}
```

```
//PrintBalance 的缺省实现
void Account::PrintBalance()
{
    cerr<<"Error.Balance not available for base type".
        <<endl;
}
```
两个派生类 CheckingAccount 和 SavingsAccount 按如下方式创建：
```
class CheckingAccount:public Account
{
public:
void PrintBalance();
};

//CheckingAccount 的 PrintBalance 的实现
void CheckingAccount::PrintBalance()
{
    cout<<"Checking account balance:"
        << GetBalance();
}

class SavingsAccount:public Account
{
```

```
public:
    void PrintBalance();
};

//SavingsAccount 中的 PrintBalance 的实现
void SavingsAccout::PrintBalance()
{
    cout<<"Savings account balance:"
        << GetBalance();
}
```

函数 PrintBalance 在派生类中是虚拟的,因为在基类 Account 中它是说明为虚拟的,要调用如 PrintBalance 的虚拟函数,可以使用如下的代码:

```
//创建类型 CheckingAccount 和 SavingsAccount 的对象
CheckingAccount *pChecking=new CheckingAccount(100.00);
SavingsAccount  *pSavings=new SavingsAccount(1000.00);

//用指向 Account 的指针调用 PrintBalance
Account *pAccount=pChecking;
pAccount->PrintBalance();

//使用指向 Account 的指针调用 PrintBalance
pAccount=pSavings;
```

```
pAccount->PrintBalance();
```
在前面的代码中,除了 pAccount 所指的对象不同,调用 PrintBalance 的代码是相同的。
因为 PrintBalance 是虚拟的,将会调用为每个对象所定义的函数版本,在派生类 CheckingAccount 和 SavingsAccount 中的函数“覆盖”了基类中的同名函数。
如果一个类的说明中没有提供一个对 PrintBalance 的覆盖的实现,则将采用基类 Account 中的缺省实现。
派生类中的函数重载基类中的虚拟函数,仅在它们的类型完全相同时才如此。派生类中的函数不能仅在返回值上同基类中的虚拟函数不同;参量表也必须不同。
当指针或引用调用函数时,要遵循如下规则:

- 对虚拟函数调用的解释取决于调用它们的对象所基于的类型。
- 对非虚函数调用的解释取决于调用它们的指针或引用的类型。

下面例子显示了在使用指针调用虚拟或非虚拟函数时它们的行为:

```
#include <iostream.h>

//说明一个基类
class Base
{
 public:
virtual void NameOf(); //虚拟函数
        void InvokingClass();  //非虚拟函数
};
```

```
//两个函数的实现
void Base::NameOf()
{
     cout<<"Base::NameOf\n";
}

void Base::InvokingClass()
{
     cout<<"Invoked by Base\n";
}

//说明一个派生类
class Derived:public Base
{
public:
    void NameOf(); //虚拟函数
    void InvokingClass();  //非虚拟函数
};

//两个函数的实现
void Derived::NameOf()
```

```
{
    cout<<"Derived::NameOf\n";
}

void Derived::InvokingClass()
{
    cout<<"Invoked by Derived\n";
}

void main()
{
     //说明一个 Derived 类型的对象
     Derived aDerived;

     //说明两个指针,一个是 Derived*型的,另一个是 Base*型的,并用
     //aDerived 初始化它们。
     Derived *pDerived=&aDerived;
     Base    *pBase   =&aDerived;

     //调用这个函数
     pBase->NameOf();    //调用虚拟函数
     pBase->InvokingClass();//调用非虚拟函数
```

```
    pDerived->NameOf();//调用虚拟函数
    pDerived->InvokingClass(); //调用非虚拟函数
}
```

该程序的输出是：

```
Derived::NameOf
Invoked by Base
Derived::NameOf
Invoked by Derived
```

注意，不管调用 NameOf 函数的指针是通过指向基类的指针还是指向派生类的指针，它调用的函数是派生类的。因为 NameOf 是虚拟函数，而且 pBase 和 pDerived 指向的对象都是派生类的，故而调用函数是派生类的。

因为虚拟函数只能为类类型的对象所调用，所以你不能把一个全局的或静态函数说明为虚拟的。

在派生类中说明一个重载函数时可以用 virtual 关键字，但是这并不是必须的，因为重载一个虚拟函数，此函数就必然是虚拟函数。

基类中的虚拟函数必须有定义，除非它们被说明为纯的（有关纯虚拟函数的详情见下节“抽象类”）。

虚拟函数调用机制可以用范围分辨符（::）明确地限定函数名称的方法来加以限制。考虑前面的代码，用下面的代码调用基类的 PrintBalance。

```
CheckingAccount *pChecking=new CheckingAccount(100.00);

pChecking->Account::PrintBalance();    //明确限定
```

```
Account *pAccount=pChecking;  //调用 Account::PrintBalance

pAccount->Account::PrintBalance();//明确限定
```
上面例子中的两个对 PrintBalance 的调用都限制了虚拟函数的调用机制。

抽象类

抽象类就像一个一段意义上的说明,通过它可以派生出特有的类。你不能为抽象类创建一个对象,但你可以用抽象类的指针或引用。
至少含有一个纯虚拟函数的类就是抽象类。从抽象类中派生出的类必须为纯虚拟函数提供实现,否则它们也是抽象类。
把一个虚拟函数说明为纯的,只要通过纯说明符语法(见本章前面的“类协议的实现”),考虑一下本章早些时候在“虚拟函数”中提供的例子。类 Account 的意图是提供一个通常意义的函数功能,Account 类型的对象太简单而没有太多用处。因此 Account 是作为抽象类的一个很好的候选:

```
class Account
{
public:
            Account(double d);  //构造函数
    virtual double GetBalance();//获得平衡
    virtual void PrintBalance()=0;  //纯虚拟函数
```

```
Private:
    double _balance;
};
```
这里的说明同前一次的说明的唯一不同是 PrintBalance 是用纯说明符说明的。

使用抽象类的限制

抽象类不能用于如下用途：

- 变量或成员数据
- 参量类型
- 函数的返回类型
- 明确的转换类型

另外一个限制是如果一个抽象类的构造函数调用了一个纯虚拟函数,无论是直接还是间接的,结果都是不确定的。但抽象类的构造函数的析构函数可以调用其它成员函数。

抽象类的纯虚拟函数可以有定义,但它们不能用下面语法直接调用：

抽象类名称::函数名称()

在设计基类中含有纯虚拟析构函数的类层次时,这一点很有用。因为在销毁一个对象的过程中通常都要调用基类的析构函数,考虑下面的例子：

```
#include <iostream.h>
//说明一个带有纯虚拟析构函数的抽象类
class base
{
```

```
  public:
  base() { }
  virtual ～base()=0;
};
//提供一个析构函数的定义
  base::～base()
{
};

class derived:public base
{
  public:
  derived()  { };
  ～derived() { };
};

void main()
{
    derived *pDerived=new derived;
    delete pDerived;
}
```
当一个由 pDerived 所指的对象销毁的时候,会调用类 derived 的析构函数,进而

调用基类 base 中的析构函数。纯虚拟函数的空的实现保证了该函数至少存在着一些操作。
注意:在前面例子中,纯虚拟函数 base::～base 是在 derived::～derived 中隐含调用的。当然明确地用全限定成员函数名称去调用纯虚拟函数是可能的。

范围规则总结

这一节补充一些有关类的新的概念:

- 二义性
- 全局名称
- 名称和限定名
- 函数的参量名称
- 构造函数初始化器

二义性

名称的使用在其范围中必须是无二义性的(直到名称的重载点)。如果这个名称表示了一个函数,那么这个函数必须是关于参量的个数和类型是无二义性的。如果名称存在着二义性,则要运用成员访问规则。

全局名称

一个对象、函数或枚举的名称如果在任何函数、类之外引入或前缀有全局单目

范围操作符(::),并同时没有同任何下述的双目操作符连用。

- 范围分辨符(::)
- 对象和引用的成员选择符(.)
- 指针的成员选择符(->)

名称及限定名

同双目的范围分辨符(::)一起使用的名称叫“限定名”。在双目范围分辨符之后说明的名称必须是在该说明符左边所说明的类的成员或其基类的成员。
在成员选择符(.或->)后说明的名称必须是在该说明符左边所说明的类类型对象的成员或其基类的成员。在成员选择符的右边所说明的名称可以是任何类类型对象,只要该说明符的左边是一个类类型对象,而且该对象的类定义了一个重载的成员选择符(->),它把指针所指的对象变为特殊的类类型(这一规定,在第 12 章“重载”中的类成员访问中详细讨论)。
编译器按下面的顺序搜索一个名称,发现以后便停止:

1. 如果名称是在函数中使用,则在当前块范围中搜索,否则在全局范围中搜索。
2. 向外到每一个封闭块范围中搜索,包括最外面函数范围(这将包括函数的参量)。
3. 如果名称在一个成员函数中使用,则在该类的范围中搜索该名称。
4. 在该类的基类中搜索该名称。
5. 在外围嵌套类范围(如果有)或其基类中搜索,这一搜索一直到最外层包裹的类的范围搜索之后。

6．在全局范围中搜索。
然而你可以按如下方式改变搜索顺序：
7．如果名称的前面有::,则强制搜索在全局范围之中。
8．如果名称的前面有 class、struct 和 union 关键字,将强制编译器仅搜索 class，struct 或 union 名称。
9．在范围分辨符的左边的名称,只能是 class，struct 和 union 的名称。
如果在一个静态成员函数中引用了一个非静态的成员名,将会产生一条错误消息。同样地,任何引用包围类中的非静态组员会产生一条错误消息,因为被包围的类没有包围类的 this 指针。

函数参量名称

函数的参量名称在函数的定义中视为在函数的最外层块的范围中。因此,它们是局部名称并且在函数结束之后,范围就消失了。
函数的参量名称是在函数说明(原型)的局部范围中,并且在说明结束以后的范围中消失。
缺省的参量名称是在参量(它们是缺省的)范围中,如前面两段描述的,然而它们不能访问局部变量和非静态类成员。缺省参量值的确定是在函数调用的时候,但它们的给定是在函数说明的原始范围中。因此成员函数的缺省参量总是在类范围中的。

构造函数初始化器

构造函数初始化器(在第 11 章“特殊成员函数”中的“初始化基类和成员”中详述)在构造函数说明的最后层块范围中求值。因此它们可以使用构造函数的参量名称。

第 10 章　成员访问控制

在 C++中,用户可以说明成员数据和成员函数的访问级别。共有三种访问级别:公共的、保护的和私有的。这一章解释访问控制如何运用于类类型对象以及派生类。

本章包括如下主题:

- 类成员的访问控制
- 访问指示符
- 基类的访问指示符
- 友元
- 保护的成员访问
- 虚拟函数的访问
- 多重访问

类成员的访问控制

通过对类成员数据或函数的访问控制,用户可以增加用 C++编制的软件的完整性。类成员可以说明为具有私有的、保护的或公共的访问属性,如表 10.1 所示。

表 10.1　成员访问控制

访问类型	意义
私有的	说明为私有的类成员只能由该类的类成员函数或友元(类或函数)来使用
保护的	说明为保护的类成员能由该类的类成员函数或友元(类或函数)来使用,并且也可以在该类的派生类中使用
公共的	说明为公共成员可以在任何函数中使用

访问控制可以阻止用户对对象进行的无规划的使用。在你显式地使用类型转换时,这种保护也会失去作用。

注意:访问控制对所有的名称都是等同适用的。包括:成员函数、成员数据、嵌套类和枚举。

用 class 关键字说明的类类型成员,类缺省的访问是私有的,而 struct 和 union 的成员的缺省访问是公共的。对于上面的几种情况,当前的访问级别可以用 public,private 和 protected 关键字来改变。

访问指示符

在类说明中,成员可以有访问指示符。

语法

访问指示符:成员表 $_{opt}$

访问指示符决定了跟在它后面的名称的访问级别,一直影响到下一个访问指示符

出现或类说明结束为止。如图 10.1 显示了这一概念。

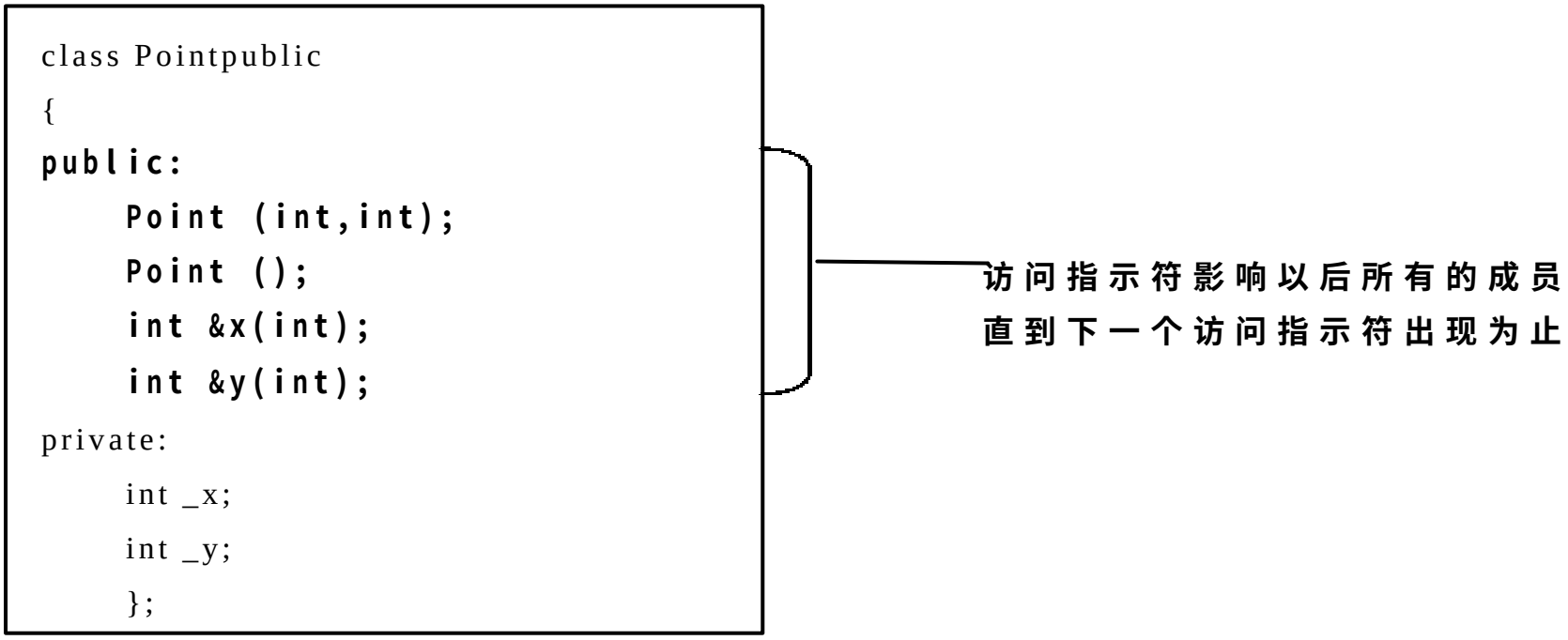


图 10.1　类中的访问控制

尽管在图 10.1 中仅显示了两个访问指示符,但在给定的类说明中对于使用访问指示符的个数没有限制。比如:图 10.1 中的类 Point 可以自由地用多重访问指示符说明如下:

```
class Point
{
public:        //说明公共构造函数
   Point(int,int);
```

```
private:     //说明私有的状态变量
    int _x;
public:      //说明公共构造函数
    Point();
public:      //说明公共访问器
    int &x(int);
private:    //说明私有的状态变量
    int _y;
public:     //说明公共的访问器
    int &y(int);
};
```

注意:如前面的例子所示,对于成员访问的说明顺序应没有特别的要求。类类型对象的存储分配会受此影响。但在访问指示符之间的成员将保证按存储器地址高端增长的方向分配。

基类的访问指示符

有两个因素控制着哪些基类的成员在派生类中是可访问的,同样的因素也控制着在派生类中对继承成员的访问控制。

- 是否派生类在类头(class-head 在第 8 章“类”的“定义类型”中详述)中用 public 说明符说明基类。
- 基类中成员所具有的访问控制。

表 10.2 给出了这些因素的交互影响以及如何决定基类成员的访问。

表 10.2 基类中的成员访问

私有的	保护的	公共的
无论怎样派生，总是不可访问的	如果用私有派生，在派生类中是私有的 如果用保护的派生，在派生类中是保护的 如果用公共的派生类中是保护的	如果用私有派生，在派生类中是私有的 如果用保护的派生，在派生类中是保护的 如果用公共的派生，在派生类中是公共的

下面的例子显示了这一点：

```
class BaseClass
{
public:
   int PublicFunc();  //说明公共成员
protected:
   int ProtctedFunc();   //说明一个保护成员
private:
   int PrivateFunc();//说明一个私有成员
};

//说明两个从 BaseClass 派生出的类
```

```
class DerivedClass1:public BaseClass
{ };

class DerivedClass2:private BaseClass
{ };
```

在 DerivedClass1 中,成员函数 PublicFunc 是一个公共函数而 ProtectedFunc 是一个保护的成员函数,因为 BaseClass 是一个公共的基类。PrivateFunc 是 BaseClass 的私有成员,并且它在任何派生类中都是不可访问的。

在 DerivedClass2 中,函数 PublicFunc 和 ProtectedFunc 是私有成员,因为 BaseClass 是私有基类。而函数 PrivateFunc 是 BaseClass 的私有成员,它在任何派生类中都是不可访问的。

当然你可以不用基类访问指示符说明一个派生类,在这种情况下,如果派生类的说明使用了 class 关键字,则派生将视为是私有派生。如果使用 struct 关键字说明派生类,则该派生是公共派生,如下面的代码:

```
class Derived:Base
...
```

等价于:

```
class Derived:Private Base
...
```

同样下面的代码:

```
struct Derived:Base
...
```

等价于：

```
struct Derived: public Base
...
```

注意，说明为私有的成员对于函数或派生类是不可访问的，除非这些函数或者派生类在基类中使用说明为友元。

联合类型不能有基类。

注意：在说明一个私有基类时，建议显式使用 private 关键字以使派生类的用户能够了解成员的访问。

访问控制和静态成员

当你说明一个基类为 private 时，它仅影响非静态成员，公共的静态成员在派生类中仍然是可访问的。然而，使用指针引用或对象访问基类成员时，会需要转换，此时访问控制仍然是适用的，考虑如下的代码：

```
class Base
{
public:
  int Print();//非静态成员
  static int CountOf();  //静态成员
};

//Derived1 把 Base 说明为私有基类
class Derived1: private Base
```

```
{
};
//Derived2 把 Derived1 说明为公共基类
class Derived2: public Derived1
{
  int ShowCount();//非静态成员
};
//定义 Derived2 的 ShowCount 函数
int Derived2::ShowCount()
{
  //显式调用静态成员函数 CountOf
  int cCount=Base::CountOf();    //OK

 //用指针调用静态成员函数 CountOf
  cCount=this->CountOf();  //错误:不允许把 Derived2*转换为 Base*
  return cCount;
}
```

在上面的代码中,访问控制抑制了从 Derived2*到 Base*的转换。this 指针的隐含类型是 Derived2*。要选择 CountOf 函数,this 必须转换为 Base*。这种转换是不允许的,因为 Base 是 Derived2 的非直接的私有基类,把一个派生类的指针转换成指向它的直接私有基类的指针是允许的。因此指针类型 Derived1*可以转换为 Base*。

注意，显式地调用 CountOf 函数,而不用指针、引用或对象,意味着没有转换,因而调用是允许的。
派生类 T 的成员或友员可以把一个指向 T 的指针转换成指向 T 的直接基类的指针。

友　元

在某些情况下,把成员级别的访问控制赋于非本类的成员函数或者在另一个单独类中的所有函数时,会更方便一些。用 friend 关键字,程序员可以指派特别的函数或类(该类中所有的成员函数)不但可以访问公共成员而且也可以访问保护的私有的成员。

友元函数

友元函数并不视为类的成员。它仍只是赋予了特别访问权限的外部函数。友元并不在类的范围中,它们也不用成员选择符(.或->)调用,除非它们是其它类的成员。下面的例子显示了一个 Point 类和一个重载操作符 operator+(这个例子主要是示例友元的,而不是重载操作符。有关重载操作符的详情见第 12 章“重载”中的“重载操作符”一节)。

```
#include <iostream.h>
//说明类 Point
class Point
```

```
{
public:
    //构造函数
    Point() {_x=_y=0;}
    Point( unsigned x,unsigned y)  {_x=x;_y=y;}
    //访问器
    unsigned x() {return _x;}
    unsigned y() {return _y;}
    void Print() {cout <<"Point("<<_x<<","<<_y<<")"<<endl;}
    //友元函数说明
   friend Point operator+(Point& pt,int nOffset);
   friend Point operator+(int nOffset,Point& pt);
private:
   unsigned _x;
   unsigned _y;
};
//友元函数定义
//
//处理 Point+int 表达式
Point operator+(Point& pt, int nOffset)
{
    Point ptTemp=pt;
```

```
    //直接改变私有成员_x 和_y
    ptTemp._x += nOffset;
    ptTemp._y += nOffset;
    return ptTemp;
}
//处理 int+Point 表达式
Point operator+(int nOffset,Point& pt)
{
    Point ptTemp=pt;
    //直接改变私有成员_x 和_y
    ptTemp._x += nOffset;
    ptTemp._y += nOffset;
    return ptTemp;
}
//测试重载操作符
void main()
{
    Point pt(10,20);
    pt.Print();
    pt=pt+3;   //Point+int
    pt.Print();
    pt=3+pt;  //int+Point
```

```
    pt.Print();
}
```
当编译器在 main 函数中碰到表达式 pt+3 时,编译器确认是否有一个合适的用户自定义的 operator+存在。在此情形下,函数 operator+(Point pt,int nOffset)匹配该操作符,并发出对该函数的调用。在第二种情形(表达式 3+pt)函数 operator+(Point pt,int nOffset)匹配提供的操作数。因而为 operator+提供了两种形式满足了+运算符的可交换性。
用户自定义的 operator+函数也可以作为成员函数来定义,但它只能接收一个显式的参数:即加到对象中的值。结果,使用成员函数加法的可交换性无法正确实现。它们必须用友元函数代替才能完成。
注意:两个版本的重载 operator+函数在类 Point 中说明为友元函数。两个说明都是必要的,在友元说明名称重载了函数和操作符时,只有那些由参数表说明的特别的函数才成为友元。假设第三种 operator+函数按如下说明:
```
Point &operator+(Point &pt,Point &pt)
```
在上面例子中的 operator+函数并不是类 Point 的友元,只不过它有着同其它两个说明为友元的函数有同样的名称。
因为 friend 说明不受访问指示符的影响,故它们可以在类说明的任何一节中说明。

类成员函数和类成为友元

类成员函数可以在别的类中说明为友元,考虑下面的例子:
```
class B
```

```
class A
{
    int Func1(B& b);
    int Func2(B& b);
};

class B
{
private
   int _b;
   friend int A::Func1(B&)  //把友元访问权限赋予类 B

}; // 类 B 中的一个函数

  int A::Func1(B& b) {return b._b;} //正确:这是友元
  int A::Func2(B& b) {return b._b;} //错误:_b 是一个私有成员
```

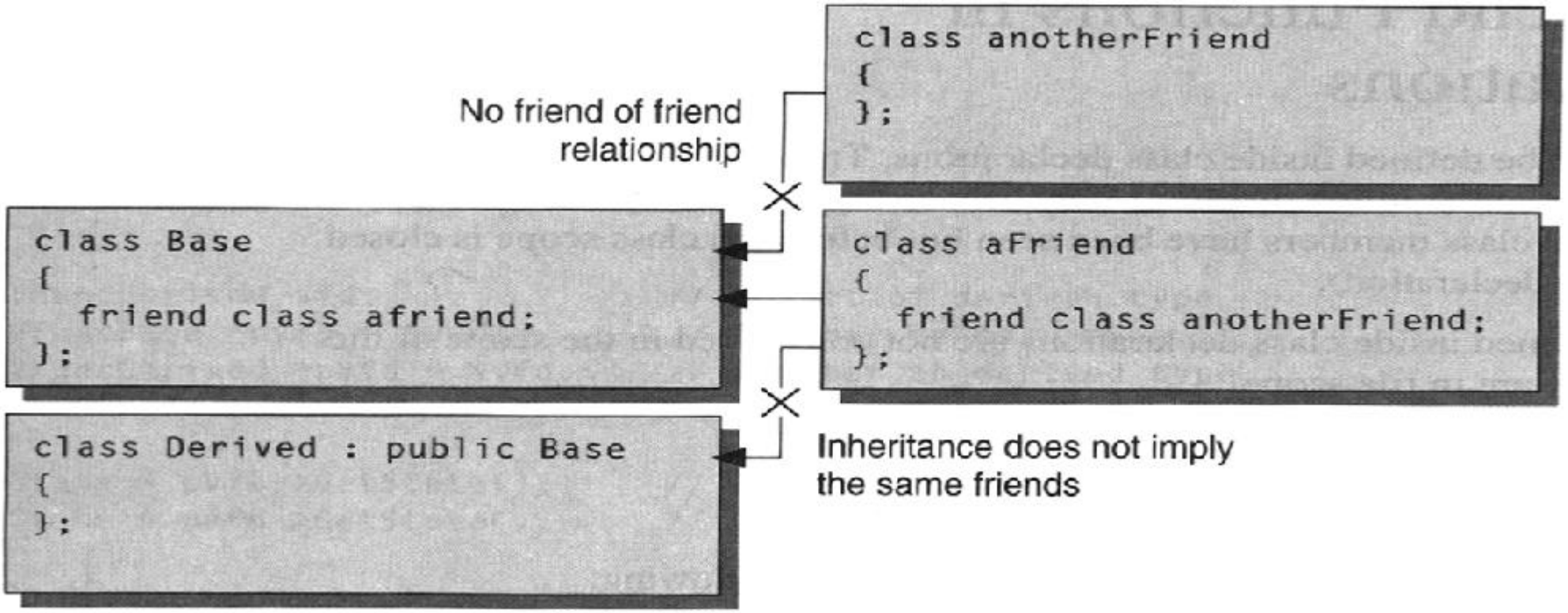


图 10.2 友元关系的牵连

在前面的例子中,只有函数 A::Func1(B& b)赋予了对类 B 的友元访问权限。因此在类 A 中的函数 Func1 中访问私有成员_b 是正确的。但 Func2 则不正确。

假设在类 B 中的友元说明如下:

```
friend class A;
```

在这种情况下,类 A 中的所有成员函数都有对类 B 的友元访问权限。注意“友元关系”是不能被继承的,也没有任何“友元的友元”的访问。图 10.2 给出了四个类说明:Base, Derived, aFriend 和 anotherFriend。只有类 aFriend 有直接

访问类 Base 的私有成员的权利(以及类 Base 继承的成员)。

友元的说明

如果你说明了一个友元函数,并且该函数是以前说明过的,则此函数将导出到非类范围的封闭块中。
在友元说明中的函数说明被作为 extern 的。就像用 extern 关键字说明过的一样(有关 extern 的详情见第 6 章“说明”中的“静态存储类说明符”。
全局范围中的函数可以先于它的类型说明而被说明为友元函数,但是成员函数在其完整的类说明出现之前是不能说明为友元的。下面的代码显示了为什么会失败:

```
class ForwardDeclared; //class name is know
class HasFriends
{
  friend int ForwardDeclared::IsAFriend(); //错误
};
```

上述的例子,把类名称 ForwardDec1ared 加入到了范围,但是其完全的定义特别是函数 IsAFriend 的说明部分还不知道,因此类 HasFriends 中的友元说明发生了错误。
为了说明两个类彼此为友元,则完整的第二个类必须说明为第一个类的友元。这种限制的原因是编译器仅在第二个类的说明时才有足够的信息说明单个的友元函数。
注意:尽管完整的第二个类必须说明为第一个类的友元,但你可以选择第一个类

中的某些函数成为第二个类的友元。

在类说明中定义友元函数

友元函数可以在类说明中定义,这些函数是嵌入函数。就像嵌入的成员函数的行为,尽管它们定义于所见的类成员之后和类范围结束前(类说明的结束)。
在类说明中定义的友元函数被视为在文件范围中,它们并不在封闭的类范围中。

保护的成员访问

说明为保护的类成员仅用于下列情况:

- 最初说明该成员的类成员函数。
- 最初说明该成员的类的友元。
- 由公共派生的类或由最初说明该成员类访问保护的类。
- 直接私有的派生类对于保护的成员有私有的访问权。

保护的成员同私有的成员在私有上是不同的,因为私有的成员只是在说明它们的类中是可访问的。保护的成员同公共的成员在公共上是不同的,因为公共的成员在任何函数中都是访问的。
用 static 说明的保护成员可被任何友元函数或派生类的成员函数访问。未用 static 说明的保护的成员也可以被任何友元函数或派生类的成员函数访问,只是要通过派生类的对象,指向派生类对象的指针,或对派生类对象的引用才行。

虚拟函数的访问

适用于虚拟函数的访问控制是由调用该虚拟函数的类型决定。重载函数的说明不会对给定类类型的访问控制有影响,例如:

```
class VFuncBase
{
public:
    virtual int GetState() {return _state;}
protected:
    int _state;
};

class VFuncDerived:public VFuncBase
{
private:
    int GetState() {return _state;}
};
...
VFuncDerived vfd; //派生类对象
VFuncBase *pvfb=&vfd; //指向基类的指针
VFuncDerived *pvfd=&vfd;  //指向派生类的指针
int state;
```

State=pvfb->GetState(); //GetState 是公共的
State=pvfd->GetState(); //GetState 是私有的,错误。
在上面的例子中,使用一个指向类型 VFuncBase 的指针调用虚拟函数 GetState 会激活 VFuncBase::GetState,并且 GetState 是作为公共的,然而用一个指向类型 VFuncDerived 的指针调用 GetState 破坏了访问控制,因为 GetState 在类 VFunDerived 中说明为私有的。
警告:虚拟函数 GetState 可以用一个指向基类 VFuncBase 的指针调用,这并不意味着调用的函数是基类中的哪个版本的函数。

多重访问

在多重继承的框架中涉及虚拟基类。一个给定的名称可以由多个路径到达。因为沿不同的这样路径,适用不同的访问控制,编译器选择最能取得存储的路径,如图 10.3 所示。

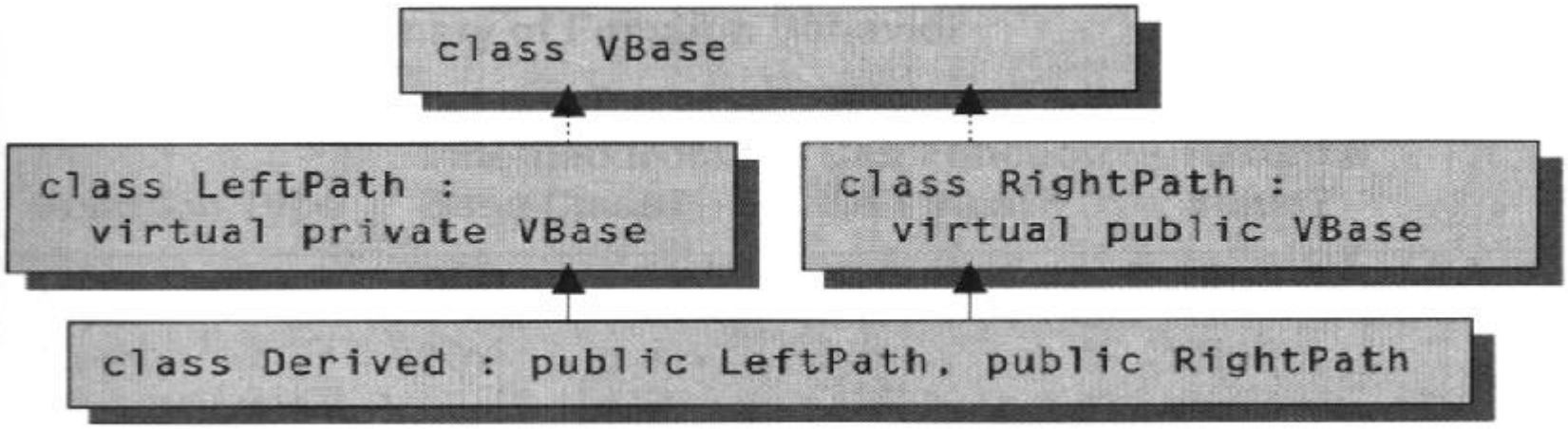


图 10.3　沿继承图中不同路径的访问

在图 10.3 中,一个在类 VBase 中说明的名称总是可以由类 RightPath 到达。右边的路径总是更可访问的,因为 RightPath 把 VBase 说明为公共基类,而 LeftPath 把 Vbase 说明为私有基类。

第 11 章　特殊成员函数

C++定义了几种只能作为类成员函数说明的函数,它们称为“特殊成员”函数。这些函数影响着给定类对象创建、删除、拷贝以及转换成其它类型对象的方法。这些函数的另一个重要的特性是它们可以由编译器隐含调用。
这些特殊的函数在下表中简要描述:

- 构造函数
- 析构函数
- 临时对象
- 转换
- new 和 delete 运算符
- 用特殊成员函数进行初始化
- 拷贝类对象

所有上面列出的项目在每个类中可以由用户自定义。
特殊成员函数同其它成员函数一样遵循相同的访问规则。这些访问规则在第 10 章“成员访问控制”中描述。表 11.1 总结了成员函数和友元函数的行为。

表 11.1 函数行为总结

函数类型	函数是否从基类中继承	能否为虚拟函数	能否有返回值	成员函数还是友元函数	如果用户不提供此函数，编译器能否生成
构造函数	否	否	否	成员函数	是
拷贝构造函数	否	否	否	成员函数	是
析构函数	否	是	否	成员函数	是
转换	是	是	否	成员函数	否
赋值(=)	否	是	是	成员函数	是
new	是	否	void*	静态成员函数	否
delete	是	否	void	静态成员函数	否
其它成员函数	是	是	是	成员函数	否
友元函数	否	否	是	友元函数	否

构造函数

与类名称具有一样名称的成员函数是构造函数。构造函数不能有返回值,甚至不能有 return 语句。说明一个有返回值的构造函数是错误的,取构造函数的地址也是错误的。

如果一个类有构造函数,在程序中每个该类类型的对象在使用之前由此构造函数进行初始化(有关初始化的更多信息参见本章后面的“用特殊成员函数进行初始化”)。

构造函数是在对象的创建点上被调用的。创建对象可以是:

- 全局对象(文件范围或外部链接的)。
- 在一个函数或者小的封闭块中的局部变量。
- 用 new 运算符创建的动态对象。new 操作在程序的堆或自由存储区中分配一个对象。
- 因显式调用构造函数而创建的临时对象(详见本章后面的“临时对象”)。
- 因编译器隐含调用构造函数而创建的临时对象(详见本章后面的“临时对象”)。
- 其它类的数据成员。在创建类类型的对象时,若此类类型由其它类类型变量组成,将会引起该类中每个对象的创建。
- 一个类的基类子对象。创建派生类类型的对象时会引起基类构件的创建。

构造函数的作用

一个构造函数执行各种任务,但对于程序员来说,这些任务是不可见的,你甚至可以不必为构造函数写任何代码。这些任务都同建立一个完全的、正确的类类型对象实例有关。

在 MS C++中(同样也在很多其它 C++中)一个构造函数:

- 初始化对象的虚拟基指针(vbptr)。如果该类是由虚拟基类派生出的,则这一步要执行。
- 按说明的顺序调用基类和成员的构造函数。
- 初始化对象的虚拟函数指针(vfptr)。如果该类有或者继承了虚拟函数,则这一步要执行,虚拟函数指针指向类的虚拟函数表(v-table),并且使虚拟函数的调用同代码正确绑定(binding)。
- 在构造函数体中执行可选的代码。

当构造函数结束以后,所分配的存储器就是一个给定类类型的对象。因为构造函数执行这些步骤,故虚拟函数的“迟后绑定”形态可以在虚拟函数的调用点得以解决,构造函数也要构造基类以及构造组合对象(作为数据成员的对象),迟后绑定是 C++实现对象的多态行为的机制。

说明构造函数的规则

构造函数具有同类名相同的名称。只要遵守重载函数的规则(有关详情参见第 12 章“重载”),可以说明多个构造函数。

语法

类名称(参量说明表 $_{opt}$) cv-修饰符表 $_{opt}$

C++定义了两种类型的构造函数,缺省的和拷贝的构造函数。如表 11.1 所述。

表 11.1 缺省的和拷贝构造函数

构造函数的种类	参量	目的
缺省构造函数	可以无参量调用	构造一个类类型的缺省对象
拷贝构造函数	可以接受对同种类类型的引用作为唯一参量	拷贝类类型的对象

缺省构造函数不要参量即可调用,但你可以说明一个带有参量表的缺省构造函数,只要让所有的参量有缺省值即可。同样,拷贝构造函数必须接受同一类类型的引用作为唯一参量。但可以提供更多的参量,只要后续的参量具有缺省值即可。

如果你不提供任何构造函数,编译器会试图生成一个缺省的构造函数。同样,如果你没有提供拷贝构造函数,编译器也会试图产生一个。编译器产生的构造函数视为公有的成员函数。如果你说明一个拷贝构造函数,而其第一个参量是一个对象而不是一个引用,则会产生错误。

编译器生成的缺省构造函数建立对象(初始化 vftables 和 vbtables,如前面所述),并调用基类及成员的缺省构造函数,但不会采取其它的行动。基类和成员的构造函数只要存在,是可访问的,并且是无二义性的就会被调用。

编译器生成的拷贝构造函数建立一个对象,并且采用一个成员方式的拷贝来复制要被拷贝的对象的内容。如果基类或成员的构造函数存在,则它们将被调用,否则,就采取位方式的拷贝。

如果所有的基类和该类类型的成员类具有接受一个常量参量的构造函数,则编译器生成的拷贝构造函数接受一个唯一的参量的类型是 const type&;否则,编译器生成的拷贝构造函数接受的唯一参量的类型是 type &。
你可以用一个构造函数去初始化一个 const 和 volatile 对象,但构造函数本身不能说明为 const 和 volatile 的。对于构造函数唯一合法的存储类型 inline,对于构造函数使用任何其它的存储类修饰符,包括 __declspec 关键字都会引起错误。构造函数和析构函数除了 __stdcall 外不能说明为其它的调用约定。
派生类是不能继承基类中的构造函数的。当一个派生类的对象在创建的时候,它是从基类构件开始构造的,然而才进入派生类构件。编译器使用每个基类的构造函数作为完整对象初始化的一部分(除了虚拟派生有所不同,参见本章后面的“初始化基类”)。

显式地调用构造函数

在程序中,为创建给定类型的对象,可以显式地调用构造函数。例如:创建两个 Point 型对象的描述一条线段的端点,可以写如下代码:

```
DrawLine(Point(13,22),Point(87,91);
```

创建了两个 Point 对象,传递给 DrawLine 函数,并在该表达式(函数调用)结束后拆除。
另外一个显式地调用构造函数的情况是在一个初始化中:

```
Point pt=Point(7,11);
```

创建了一个 Point 类型的对象,并用接受两个整形参量的构造函数进行初始化。如前面的两个例子中,通过显式地调用构造函数创建的对象是无名的对象,并在

它们所创建的表达式中是有生存期的。在本章后面的“临时对象”中将对这一点进行深入的讨论。

在构造函数内调用成员函数和虚拟函数

在构造函数里面调用任何成员函数通常是很安全的,因为在执行第一行用户代码之前,对象已经完全建立起来了(已经初始化了虚表等等)。但是当成员函数调用了其抽象基类的虚拟成员函数时,在构造函数和析构函数调用此成员函数存在着潜在的不安全性。

构造函数可以调用虚拟函数。在调用虚拟函数时,被调用的是在构造函数自身的类中定义的函数(或者从其基类中继承的函数)。下面的例子显示了一个虚拟函数在一个构造函数中被调用时发生的情况。

```
#include <iostream.h>
class Base
{
public:
   Base();//缺省构造函数
   virtual void f();  //虚拟成员函数
};

Base::Base()
{
   cout<<"Constructing Base sub-object\n ";
```

```
    f();//在构造函数中调用虚拟成员函数
}

void Base::f()
{
  cout<<"called Base::f()\n";
}

class Derived:public Base
{
public:
    Derived();//缺省构造函数
    void f(); //该类虚拟函数的实现
};

Derived::Derived()
{
  cout<<"constructing Derived object\n";
}

void Derived::f()
{
```

```
    cout<<"Called Derived::f()\n";
}

void main()
{
    Derived d;
}
```

在上面的程序运行的时候,Derived d 的说明会引发下列一系列事件:

1. 类 Derived 的构造函数(Derived::Derived)被调用。
2. 在进入到 Derived 类的构造体之前,基类 Base 的构造函数被调用。
3. Base::Base 调用函数 f,它是一个虚拟函数。通常被调用的函数会是 Derived::f,因为对象 d 是 Derived 类型的对象。但因为 Base::Base 是一个构造函数,此时的对象是一个 Derived 类型的对象,故 Base::f 将会被调用。

构造函数与数组

数组的构造只能使用缺省的构造函数。缺省构造函数要么不接受任何参量,要么对于它的所有参量都有缺省的值。数组通常是按升序来构造的,该数组的每一个成员的初始化都是使用同一构造函数。

构造的次序

对于派生类或其成员数据是类类型的类,构造发生的顺序有助于你理解,在任一

给定的构造函数中你能够使用对象的哪一部分。

构造与继承

一个派生类的对象是从基类到派生类通过按次序为每个类调用构造函数来构造的。
每个类的构造函数能仅依赖于被完全构造好的它的基类。
有关初始化的完整描述,包括初始化的顺序,见本章后面的“初始化基类及成员”。

构造与组合类型

含有类类型数据成员的类称为组合类。当创建一个组合类类型的对象时,含有类的构造函数在该类的构造函数之前调用。
有关这种情况的初始化,见本章后面的“初始化基类及成员”。

析构函数

析构函数是“反向”的构造函数。它们在对象被销毁(回收)时调用。设计一个函数为类的析构函数只要在类名之前加上(~)号。例如,类 string 的析构函数是~string()。
析构函数通常是在一个对象不再需要时,完成“清除”。考虑一下下面类 string 的说明:

```
#include <string.h>
```

```
class String
{
public:
  String(char *ch);    //说明构造函数
  ~String();           //以及析构函数
private:
  char *_text;
};
//定义构造函数
String::String (char *ch)
{
   //动态分配正确数量的存储器
   _text=new char[strlen(ch)+1];
   //如果分配成功,则拷贝初始化字符串
   if(_text)
     strcpy(_text,ch);
}
//定义析构函数
String::~String()
{
   //回收先前为此字符串保留的存储器
```

```
    delete[] _text;
}
```

在前面的代码上。析构函数 String::~String 使用 delete 运算符回收（deallocate）动态分配给 text 的存储空间。

说明析构函数

析构函数与类名称有相同的名称,并且前缀有~号。

语法

～类名称()

或

类名称::～类名称()

语法的第一种形式用于析构函数说明或定义于类说明之中;第二种形式用于析构函数定义于类说明之外。

有几条规则约束着析构函数的说明。析构函数:

- 不能接受参量
- 不能说明有任何返回类型(包括 void)
- 不能用 return 语句返回值
- 不能说明为 const,volatile 或 static,但析构函数可以因说明为 const,volatile 或 static 的对象的析构而被调用。
- 可以说明为虚拟的。使用虚析构函数,你可以折除对象而不必知道该对象的类型。由于使用虚拟函数机制,将调用该对象的正确的析构函数。注意，在一个抽象类中,析构函数可以说明为纯虚拟函数。

使用析构函数

当下列事件发生时将调用析构函数：

- 一个用 new 运算符分配的对象显式地使用 delete 运算符回收。在用 delete 运算符对一个对象进行回收时，释放的存储器是“最外派生对象”的(most derived object)，或者是一个完整对象的而不是代表基类的子对象。对于“最外派生对象”的回收只是在同虚拟析构函数一起工作时才有保证的。回收可能在多重继承的情形下失败，因为在此情形下，类类型信息并不同实际对象所信赖的类型相一致。
- 一个在块范围中的局部(自动)对象超出了其范围。
- 临时对象生存期的结束。
- 全局或静态成员还存在，但程序已经结束。
- 用析构函数的全限定名来显式地调用析构函数(详情见本章后面的“显式的析构函数的调用”)。

前面列表中所描述的情况保证了所有的对象可以用用户自定义的方法折除。

如果一个基类或者数据成员有一个可访问的析构函数，同时如果一个派生类没有说明析构函数，编译器会生成一个。编译器产生的析构函数调用基类的析构函数和派生类成员的析构函数。缺省的析构函数是公有的(有关访问属性的详情见第10章“成员访问控制”中的“基类访问说明”)。

析构函数可以自由地调用类成员函数以及访问类成员数据。当在析构函数中调用虚函数时，该函数是在当前正被折除的类的函数(有关详情见下一节，“析构的

次序”)。
使用析构函数有两条原则,第一是不能取析构函数的地址。第二是派生类不能继承它的基类的析构函数。相反,如前面所解释的,它们总是覆盖了基类的析构函数。

析构的次序

当某个对象超出了其范围或被删除时,在其完整的析构中会发生如下一些系列的事件:
1. 调用该类的析构函数,执行析构函数体的代码。
2. 非静态成员按它出现在类说明中的反序调用其析构函数。在构造函数中使用的可选成员初始化表的运用不会影响构造或析构的次序(有关初始化成员的详情见本章后面的“初始化基类和成员”)。
3. 非虚拟基类的析构函数按其说明的反序调用。
4. 虚拟基类的析构函数按其说明的反序调用。

非虚拟基类的析构函数

非虚拟基类的析构函数是按基类名说明的反序调用的。考虑下面的类说明:
```
class MultInherit: public Base1,public Base2
...
```
在上面的例子中,Base2 的析构函数在 Base1 的析构函数之前调用。

虚拟基类的析构函数

虚拟基类的析构函数是按它们出现在有向无环图中的反序被调用的(按深度优先,从左到右,后序遍历)。如图 11.1 所示描绘了一幅继承图。

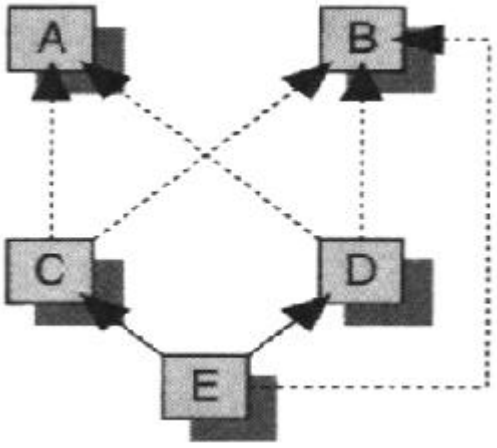


图 11.1 显示虚拟基类的继承图

下面列出了图 11.1 中各类的类头:

```
class A
class B
class C:virtual public A,virtual public B
class D:virtual public A,virtual public B
class E:public C,public D,virtual public B
```

对于类型 E 的对象要确定虚拟基类析构函数的顺序。编译器用如下算法编一个列表:

1. 从图中的最深点开始(在此例中是 E 点)遍历左子图。
2. 而左一直遍历到所有的结点都被访问,记下当前结点的名称。
3. 再次访问前一个结点(向下并向右)去证实正在标记的结点是否是一个虚拟基类。
4. 如果被标记的结点是一个虚拟基类,则搜索该列表看是否该结点已经在列表之中了。如果被标记的结点不是一个虚拟基类,则忽略它。
5. 如果该标记的结点还不在列表中,则把它加入到列表的尾部。
6. 返回到图的上一层,并沿着右边的下一条路径遍历图。
7. 转向 2 继续。
8. 当本结点所有可能遍历的路径用完后,记下该结点。
9. 转向 3 继续。
10. 继续该过程直到底部的结点再次成为当前结点。

因此对于类 E,析构的顺序是:

1. 非虚拟基类 E。
2. 非虚拟基类 D。
3. 非虚拟基类 C。
4. 虚拟基类 B。
5. 虚拟基类 A。

这一过程产生一个单一条目的有序列表,类名不会出现两次。一旦该列表建成以后,按反序遍历该列表,从最后到最前将调用每个类的析构函数。

当一个类的构造函数和析构函数依赖于其它正在被创建的构件或要长期被保留的构件时,例如,(在图 11.1 中)当 A 的析构函数(若其代码的执行)依赖于 B 的

仍然存在时,或反过来构造和析构的次序是非常重要的。
在继承图中,类之间的这种交互依赖具有固有的危险性,因为其后派生的类会改变最后的路径,与此相连也改变了构造和析构的次序。

显式的析构函数的调用

显式地调用析构函数通常是不必要的,但它们在执行放在绝对地址中的对象是很有用的。这些对象通常是用带有一个位置参量的用户自定义的 new 运算符分配的。delete 操作不能回收这一存储器,因为它不是从自由存储区中分配的(有关详情见本章后面的“new 和 delete 运算符”)。然而一个对析构函数的调用能够完成合适的清除工作。要显式调用类 String 的对象 s 析构函数,可采用下列语句之一:

```
s.String::~String();//非虚拟的调用
ps->String::~String();  //非虚拟的调用
s.~String();//虚拟调用
ps->~string();  //虚拟调用
```

在上面的代码中显示的显式调用析构函数的符号可以直接使用，而无视是否该类型定义了一个析构函数。这使你能用这种显式的调用，而又不必了解该类型是否定义了一个析构函数。如果一个析构函数未定义,则对该析构函数的显式调用不会产生效果。

临时对象

在某些情况下,对于编译器来说必须创建临时对象。因以下原因要创建这些临时变量:

- 要用一个不同于正被初始化的引用类型的另一类型去初始化一个常量引用。
- 要保存一个函数返回的用户自定义类型的返回值。如果用户程序没有把返回值拷贝到另一对象中时,这一临时对象才会被创建:

```
UDT Func1();   //说明一个返回用户自定义类型的函数
...
Func1();       //调用 Func1 函数,但忽略其返回值
               //创建一个临时对象以保存返回值
```

因为返回值并未拷贝到其它对象,故创建一个临时对象。一个要创建临时对象的更普遍的情形是要调用重载运算符函数的表达式求值中。这些重载的运算符返回用户自定义的类型的值(通常不拷贝到其它对象之中)。考虑表达式:

```
ComplexResult=Complex1+Complex2+Complex3
```

求出表达式 Complex1+Complex2 的值,结果存放在临时对象中。接着求表达式 temporary+complex3 的值,并把结果存放到 complexresult 中(假定赋值运算符没有被重载)。

- 存一个造型转换为用户自定义类型的转换结果。当一个给定类型的对象显式地转换为用户自定义类型时,新的对象是作为临时对象来构造的。

临时对象有一个生存期,在其创建点和拆除点之间有定义。任何一个创建了多个临时对象的表达式最后是按创建它们的相反的顺序拆除的。析构的发生点如表11.2所示。

表 11.2　临时对象的析构点

创建临时对象的原因	析构点
表达式求值的结果	所有因为表达式求值的结果而创建的临时对象在表达式求值结束时(也即分号处)或者在控制表达式(for、if、while、do 和 switch 语句)结束时被拆除。
内置的(非重载的)符(||和&&)	在右操作数结束之后。在这一析构点,所有因右逻辑运算操作数求值而创建的临时对象被拆除。
初始化常量引用	如果一个初始化符并不是同要被初始化的引用有相同的 l 值类型,则创建基于此对象类型的临时对象并用此初始化表达式初始化。这一临时对象在同它绑定在一起的引用对象被拆除以后马上拆除。

转　换

给定类类型的对象可以转换为其它类型的对象。这可以按以下来完成:从源类类

型构造一个目标类类型的对象，并把结果拷贝到目标对象。这一过程称为构造函数式转换。对象也可以由用户提供的转换函数转换。

当标准转换(在第 3 章“标准转换”中描述)不能完成从一个给定类型转换到一个类类型时,编译器会选择用户自定义的转换以帮助完成这一工作。除了显式的类型转换,转换还发生在:

- 一个初始化表达式同被初始化的对象有不同的类型。
- 在函数调用中使用的参量类型同函数说明中所说明的参量不匹配 。
- 从函数中返回的对象的类型同函数说明中说明的返回类型不匹配。
- 两个表达式操作数必须有同样的类型。
- 一个控制复述或选择语句的表达式需要从提供的表达式中得到不同的类型。

用户自定义的转换仅用于它没有二义性时,否则会产生一个错误消息。在使用点上将检查二义性，因而,如果可以引起二义性的特征未被使用,只能标明一个类有潜在的二义性,并不会产生任何错误。尽管有很多情形会引起二义性,下面两条是最会引起二义性的原因。

- 一个使用多重继承派生的类,没有说清楚从哪一个基类选择此转换(见第 9 章“派生类”中的“二义性”)。
- 一个显式的类型转换运算符和为此同一目的构造函数同时存在(见本章后面“转换函数”)。
- 构造函数方式的转换和转换函数方式的转换两者都要遵循第 10 章“成员访问控制”中所描述的访问控制规则。访问控制仅在转换发

现无二义性以后才进行检测。

转换构造函数

可以用单一参量调用的构造函数是用于把参量类型转变成类类型的转换。这种构造函数称为转换构造函数。考虑下面的例子:

```
class Point
{
public;
    Point();
    Point(int);
    ...
};
```

有时需要一种转换,但在类中又不存在转换构造函数。这种转换不能由构造函数完成,编译器不会寻找可以完成该转换的中间类型。比如:假设存在着一个从 Point 类型到 Rect 类型的转换以及一个从 int 到 Point 类型的转换,但编译器不会通过构造一个中间的 Point 类型对象来提供一个从 int 到 Rect 类型的转换。

转换和常量

尽管内置类型的常量如(int,long 和 double)可以出现在表达式中,但类类型的常量是不允许的(部分是因为,类通常用来表示复杂的对象,用符号不便于表示)。但是如果提供了对内置类型进行转换的转换构造函数,这些内置类型的常

量可以用于表达式,并且转换会引出正确的行为。如下例子,一个 Money 类具有从 long 和 double 类型的转换:

```
class Money
{
public;
    Money(long)
    Money(double)
    ...
    Money operator+(const Money&);  //重载加法运算符
};
```

因此,像下面的表达式可以说明常量值:

```
Money AccountBalance=37.89;
Money NewBalance=AccountBalance+14L;
```

第二个例子引起了重载加法运算符的调用(在下一章中讨论)。两个例子都会令编译器在表达式中使用常量以前把它们转换成 Money 类型。

转换构造函数的缺点

因为编译器会隐含地选择一个转换构造函数,故你将放弃对具体调用函数的控制。如果保留全面的控制是很重要的,就不要说明任何接受单一参量的构造函数。相反,定义一个“帮助”函数以完成这些转换,看如下代码:

```
#include <stdio.h>
#include <stdlib.h>
```

```
//说明 Money 类
class Money
{
public;
  Money();
//定义只能显式调用的转换函数
  static Money Convert(char * ch) {return Money(ch);};
  static Money Convert(double d) {return Money(d);};
  void Print (){printf("\n%f",_amount);}
private:
  Money(char * ch) {_amount=atof(ch);}
  Money(double d)  {_amount=d;}
  double _amount;
};
void main()
{
  //完成一个从 char*到 Money 的转换
  Money Acct=Money::Convert("57.29");
  Acct.Print();
  //完成一个从 double 到 Money 的转换
  Acct=Money::Convert(33.29);
  Acct.Print();
```

```
}
```

在上面的代码中,转换构造函数是私有的,并且不能在类型转换中使用。但它们可以显式地通过调用 Convert 函数来激活。因为 Convert 函数是静态的,它们的访问不需要特别的对象。

转换函数

前一节中介绍,用构造函数进行转换中一种类型的对象可以隐含地转换成特殊的类类型。这一节介绍一种方法,通过它可以提供一个显式的转换方法把给定的类类型转换成其它类型。从某种类类型的转换经常是使用转换函数完成的。转换函数使用下面的语法:

语法

转换函数名称:

operator 转换类型名称()

转换类型名称:

类型指示符表 指针运算符$_{opt}$

下面的例子说明了一个转换函数把 Money 类型转换成 double 类型:

```
class Money
{
public:
    Money();
    operator double() { return _amount;}
 private:
```

```
    double _amount;
};
```
一旦给出了前面的类说明,则可以编写下面的代码:
```
Money Account;
...
double CashOnHand=Account;
```
把 CashOnHand 用 Account 进行初始化会引起从类型 Account 到 double 的转换。转换函数通常被称为“造型运算符”。因为它们(如同构造函数的叫法)是在造型类型转换时调用的函数。下面的例子使用了造型或显式的类型转换打印一个 Money 型对象的当前值:
```
cout <<(double)Account<<endl;
```
转换函数可以在派生类中被继承。转换运算符仅仅隐藏基类中转换完全相同类型的转换运算符。因此一个用户自定的 operator int 函数不会隐藏一定义于基类中的 operator short 函数。

在进行隐含转换时仅有一个用户自定义的类型转换函数适用。如果没有显式的定义转换函数,编译器不会寻找一个中间类型以通过它进行对象的转换。

如果需要的转换引起了二义性,则会产生一个错误。在多个用户自定义的转换可以适用,或在用户自定义的转换和内置的转换同时存在时产生二义性。

下面的例子示例了一个存在潜在二义性的类说明:
```
#include <string.h>
class String
{
```

```
    //定义从 char * 类型转换的构造函数
    String(char *) {strcpy(_text,s);}
    //定义 char * 的转换
    operator char* ( ) {return_text,}
    int operator==(const String& s)
   {return !strcmp(_text,s._text);}
private:
   char _text[80];
};
int main()
{
   String s("abcd");
   char *ch="efg";
  //使编译器选择一个转换
  return s==ch;
}
```

在表达式 s==ch 上,编译器有两种选择,并且没有办法决定哪一种更好。它可以使用构造函数把 ch 转换为一个 String 类型的对象，然后采用用户自定义的operator==进行比较。

然而它也可以把 s 转换成一个 char* 型的指针(使用转换函数)，然后采用指针之间的比较。

因为两种方法之中没有哪一个更好一些,编译器也不能决定比较表达式的意义，

故而产生一个错误。

说明转换函数的规则

下面四条规则适用于说明转换构造函数(参见转换函数语法):

- 类,枚举和 typedef 名不能在类型指示符表中说明,因而下面的代码会产生错误:

```
operator struct String {char string_storage;}();
```

相反,结构 String 的说明应先于转换函数的说明。

- 转换函数不能带参量,说明参量会引起错误。
- 转换函数已说明了转换类型名称的返回类型;故为转换函数说明任何返回类会产生错误。
- 转换函数可以说明为虚拟的。

new 和 delete 运算符

C++支持使用 new 和 delete 运算符动态分配和回收对象,这些操作从一个称为“自由区”的池中为对象分配存储器。new 运算符调用 operator new 函数,delete 运算符调用 operator delete 函数。

operator new 函数

当编译器在程序中碰到如下的一个语句时,它就翻译为一个对 operator new 的

调用。

```
char *pch=new char[BUFFER_SIZE]
```

对于分配 0 字节存储区的要求,operator new 返回一个不同对象的指针(也就是说,反复调用 operator new 会返回不同的指针)。如果没有足够的存储器满足分配的要求,缺省时,通常 operator new 返回 NULL。当然你可以通过写一个定制的例外处理例程来改变这种缺省的行为。然后调用_set_new_handler 运行时间库函数并把你的例外处理例程函数的名称作为其参量。有关恢复机制的细节参见下一节“处理无足够存储器的条件”。

operator new 的两种范围在表 11.3 中描述。

表 11.3 operator new 函数的范围

运算符	范围
::operator new	全局的
class_name::operator new	类的

operator new 函数的第一个参量必须是类型 size_t(在 STDDEF.H 中定义),其返回值总是 void*。

在用 new 运算符分配内部数据类型对象时,或者分配不包含用户自定义型 operator new 函数的类类型对象时,以及分配任意类型的数组时,调用的是全局 operator new 函数。当 operator new 用在定义 operator new 函数的类类型对象对,调用的是类的 operator new 函数。

为类定义的 operator new 函数是一个静态成员函数(因此它不可能是虚拟的)。它对于该类的对象来说隐藏了全局的 operator new 函数。研究下面的情况,其

中 new 用于分配存储器,并把存储器设为给定的值:

```
#include <malloc.h>
#include <memory.h>
class Blanks
{
public:
  Blanks() { }
  void* operator new(size_t stAllocateBlock,char chInit);
};

void * Blanks::operator new(size_t stAllocateBlock,char chInit);
{
  void * pvTemp=malloc(stAllocateBlock);
  if (pvTemp!=0)
     memset (pvTemp,chInit,stAllocateBlock),
  return pvTemp;
};
```

对于不同的 Blanks 类型的对象来说,全局 operator new 函数被隐藏了。因此下面的代码分配一个 Blanks 型的对象并初始化为 0xa5:

```
int main()
 {
   Blanks *a5=new (0Xa5) Blanks;
```

```
    return a5!=0;
};
```
在括号中提供给 new 的参量是传递给 Blank::operator new 作为 chInit 参量的。然而全局 operator new 函数是隐藏了的,故按如下调用全局 operator new 的代码会引起错误。

```
Blanks *SomeBlanks=new Blanks:
```
在以前的编译器版本中,非类类型和所有的数组(无论它们是否是类类型数组)使用 new 运算符分配存储器总是使用全局 operator new 函数。

从 Visual C++5.0 开始,编译器开始在类说明中支持成员数组 new 和 delete 运算符。

例如:

```
class X {
public:
    void * operator new[](size_t);
    void   Operator delete[](void*),
};
void f(){
  X *pX=new X[5],
  delete []pX;
}
```

处理无足够存储器条件

对于失败的存储器分配可以采用如下的代码：

```
int *pi=new int[BIG_NUMBER];

if (pi==0)
{
   cerr<<"Insufficient memory"<<endl;
   return -1;
}
```

当然有其它的办法处理失败的存储器分配请求,即写一个定制的例程去处理这种失败。然后通过调用_set_new_handler 运行函数注册你的处理例程。这种方法在下一节介绍。

使用_set_new_handler

在某些情况下,可以在分配存储器中采取一些矫正的办法使分配要求可得以满足。为在全局 operator new 函数失败时获得控制,使用_set_new_handler 函数(在 NEW.H 中定义)如下：

```
#include <stdio.h>
#include <new.h>
//定义一个在 new 分配存储器失败时调用的函数
int MyNewHandler(size_t size)
{
```

```
  clog<<"Allocation failed.Coalescing heap."<<endl;
  //调用一个工具函数以恢复一些堆的空间
  return CoalesceHeap();
}
void main()
{
  //把 new 失败处理函数设为 MyNewHandler
  _set_new_handler(MyNewHandler);
  int *pi=new int[BIG_NUMBER];
}
```

在上面的例子中,main 函数的第一个语句是把 new 的处理函数设为 MyNewHandler。第二条语句是使用 new 运算符分配一大块存储器。当分配失败的时候,控制转向 MyNewHandler。

传递给 MyNewHandler 的参量是要求分配的存储器的字节大小。从 MyNewHandler 返回的是一个标识以表明是否要再进行一次分配操作。非 0 值表示应再进行一次,而 0 值则表示分配失败了。

MyNewHandler 打印一条警号消息并采取矫正的步骤。如果 MyNewHandler 返回一个非 0 值,new 运算符再试图分配一次;当 MyNewHandler 返回 0 值时,new 操作分配企图,把一个 0 值返回给程序。

_set_new_handler 返回的是一个老的 new 处理函数的地址。因此如果一个新的 new 处理函数只在短时间内使用,则老的处理函数可以按如下代码还原:

```
#include <new.h>
```

```
...
_PNH old_handler=_set_new_handler(MyNewHandler);
//要求使用 MyNewHandler 的代码
...
//重新安装以前的处理函数
_set_new_handler(old_handler);
```

用 0 参量调用_set_new_handler 会引起移去 new 处理函数，也就是没有缺省的处理函数了。
你可以用任何名称说明 new 处理函数,但它必须是一个返回 int 型的函数(非 0 表示 new 处理函数继续,0 表示失败)。
如果提供了一个用户自定义的 operator new 函数,该 new 处理函数不会在失败时自动调用。
_set_new_handler 的函数原型以及_PNH 在 NEW.H 中定义:

```
_PNH_set_new_handler(_PNH);
```

类型_PNH 是一个指向函数的指针,该函数的类型 size_t 作为唯一的参量并返回 int 型。

operator delete 函数

用 new 操作动态分配的存储器可以用 delete 操作释放。delete 运算符调用 operator delete 函数把存储器释放回可用的存储区池中。使用 delete 操作也会引起类的析构函数的调用(如果有的话)。
operator delete 函数也有全局和类范围的两种。对于给定的类只能为其定义

一个 operator delete 函数;一旦定义以后,它会隐藏全局 operator delete 函数。全局的 operator delete 函数总是可以为任意类型的数组所调用。
全局 operator delete 函数在说明中带一个 void 类型的单一参量,它含有要释放对象的指针。它的返回类型是 void(operator delete 函数不能有返回值)。
类成员 operator delete 函数有两种形式:

```
void operator delete(void *);
void operator delete(void *,size_t);
```

对于给定的类只能提供上述两种不同形式中的一种,第一种形式就像全局 operator delete 一样工作;第二种形式有两个参量。第一个参量是要释放存储器块的指针,第二个参量是要释放的存储器字节大小。当一个基类中的 operator delete 函数用于释放派生类对象时,第二种形式特别有用。
operator delete 函数是静态的,因而它不能是虚拟的,operator delete 函数必须遵循第 10 章“成员访问控制”中描述的访问控制。
下面的例子显示了用户设计的 operator new 和 operator delete 函数用于记录分配和释放存储器:

```
#include <iostream.h>
#include <stdlib.h>
int fLogMemory=0;//是否记录(0=否;非 0=是)
int cBlocksAllocated=0;   //分配的块数
//用户自定义的 new 操作
void *operator new(size_t stAllocateBLock)
{
```

```
  static fInOpNew=0,//保护标志
  if(fLogMemory && !fInOpNew)
  {
     fInOpNew=1;
     clog<<"Memory Block"<<++cBlocksAllocated
         <<"allocated for"<<stAllocateBlcok
         <<"bytes\n";
     fInOpNew=0;
  }
   return malloc(stAllocateBlock);
}
//用户定义的 operator_delete
void operator delete(void *pvMem)
{
   static fInOpDelete=0;  //保护标志
   if (fLogMemory &&!fInOpDelete)
   {
      fInOpDelete=1;
      clog<<"Memory block"<<--cBlocksAllocated
          <<"deallocated\n";
      fInOpDelete=0;
   }
```

```
    free (pvMem);
}
int main (int argc,char*argv[])
{
  fLogMemory=1;  //打开 log 标识
  if (argc>1)
  for (int i=0; i<atoi(argv[1]);++i )
  {
     char *pMem=new char[10];
     delete [] pMem;
  };
  return cBlocksAllocated;
}
```

上面的代码可以用来检查“存储器漏损”,即从自动堆中分配以后就没有释放的存储器。为完成这一检查,全局 new 和 delete 操作被重新定义以统计分配和释放的存储器。

从 Visual C++5.0 开始,编译器在类说明中支持成员数组 new 和 delete 操作,例如:

```
class X
{
public:
    void * operator new[] (size_t);
```

```
    void   operatot delete[](void*);
};

void f() {
    X* pX=new X[5];
    delete[] pX;
}
```

用特殊成员函数初始化

这一节描述使用特殊成员函数的初始化,它是下面一些初始化论题的扩展。

- 第 7 章“说明”中的“初始化集合”,描述了如何初始化非类类型数组以及简单类类型对象。这些简单的类类型不可有私有的及保护的成员,它们也不能有基类。
- 构造函数。它解释了如何使用特殊的构造函数初始化类类型对象。

缺省的初始化方法是执行一个位方式的拷贝,把初始化器拷贝给要被初始化的对象。这种技术仅适用于:

- 内部类型的对象。例如:

```
int i=100;
```

- 指针。例如:

```
int i
int *pi=&i;
```

- 引用。例如：

```
String sFileName("FILE.DAT");
String &rs=sFileName;
```

- 类型对象，该类不可有保护的或私有的成员，不能有虚拟函数也不能有基类，例如：

```
struct Point
{
    int x,y;
}
Point pt={10,20};    //static storage calss only
```

通过说明构造函数(有关说明这种函数的详情见本章开始的“构造函数”)，类可以说明更多精确的初始化。如果一个对象的类型是有构造函数的类类型，该对象一定会被初始化，否则一定存在着缺省的构造函数。没有特别初始化的对象会激活类的缺省构造函数。

显式的初始化

C++支持两种形式的显式初始化。

- 支持在括号中提供初始表：

```
String sFileName("FILE.DAT);
```

在括号中的初始化器表的项是作为类构造函数的参量。这种形式的初始化使得对一个对象使用多个值进行初始化成为可能，并且也可以同 new 运算符联合使用。如：

```
Rect *pRect=new Rect(10,15,24,97);
```

- 支持使用等号初始化语法提供单一初始化器,例如:

```
String sFileName="FILE.DAT";
```

尽管前面的例子同第一种形式中的 String 例子以同样的方式工作,但该语法并不适用于同自由堆中分配的对象一起使用。

在等号右边的单一表达式是作为类的拷贝构造函数的参量,因此它必须是能够转换成类类型的某种类型。

注意因为在初始化的上下文中等号(=)同赋值号是不同的,故重载 operator=对于初始化没有影响。

等号初始化语法同函数式语法是不同的,尽管在大多数情况下产生的代码是相同的。它们之间的不同在于:当使用等号语法,编译器的行为就像发生了如下的顺序事件:

- 创建一个同要被初始化的对象同一类型的临时对象。
- 把临时对象拷贝到要被初始化的对象之中。

在编译器执行这些步骤之前,构造函数必须是可访问的。尽管编译器在大多数情况下可以忽略临时对象的创建和拷贝步骤,然而一个不可访问的拷贝构造函数会引起等号初始化的失败。考虑下面的例子:

```
class anInt
{
   anInt (const anInt&);    //私有拷贝构造函数
public:
   anInt(int);    //公有的 int 构造函数
```

```
};
...
anInt myInt=7;  //破坏了访问控制,试图引用私有拷贝构造函数
anInt myInt(7); //正确;设有调用拷贝构造函数
```

在函数调用中,传值形式的类型参量以及传值形式的返回对象用下面的代码概念上地进行初始化。

```
type-name name=value
```

例如:

```
String s="C++";
```

因此,参量类型必须是可以转换为作为参量进行传递的类类型。该类的拷贝构造函数,以及自定义转换运算符或接受实际参量的构造函数必须是公有的。

在使用 new 运算符的表达式中,从自由堆中分配的对象概念性地用以下形式进行初始化:

$$\text{type-name name(initializer}_1\text{,initializer}_2\text{,...iniatializer}_n\text{)}$$

例如:

```
String *ps=new String("C++");
```

对于基类构件和类成员对象的初始化也是概念性地用这种方法初始化的(详见本章后面的“初始化基类及成员”)。

初始化数组

如果一个类有一个构造函数,此类的数组由一构造函数进行初始化。若初始化器表中的项数小于数组元素的个数,则缺省构造函数将用于剩下的数组元素。若此

类并未定义缺省构造函数,则初始化器表必须是完全的,也就是数组中的每一个元素必须有一个初始化器。
考虑定义了两个构造函数的 Point 类:

```
class Point
{
public:
    Point();//缺省构造函数
    Point(int,int); //从两个 int 型的构造函数
    ...
}
```

一个 Point 对象数组可以按如下说明:

```
Point aPoint[3]={
  Point(3,3)    //使用 int,int 构造函数
}
```

aPoint 数组的第一个元素使用构造函数 Point(int,int)来构造,剩余的两个元素使用缺省构造函数来构造。
静态成员数组(无论是否为 const)都可以在它们的定义中(在类说明之外)进行初始化。例如:

```
class WindowColors
{
public:
  static const char *rgszWindowPartList[7];
```

```
  ...
};
const char * WindowColors::rgszWindowPartList[7]={
  "Active Title Bar","Inactive Title Bar", "Title Bar Text", "Menu Bar",
  "Menu Bar Text", "Window Background", "Frame" };
```

初始化静态对象

全局静态对象是按它们出现在源代码中的次序进行初始化,它们按此相反的次序拆除。然而,交叉转换单元中的初始化的次序依赖于连接器如何组织目标文件,析构的次序仍然按对象创建次序的相反次序发生。
局部静态对象的初始化发生在当它们第一次出现在程序流中时,而且它们在程序结束时也是按此相反的次序被拆除的。局部静态对象的析构仅发生在程序流中发现此对象,且此对象被初始化时。

初始化基类及成员

一个派生类的对象是由代表每个基类的构件以及对于此特殊类唯一的一个构件组成。含有成员对象的类对象也可包含其它类的实例。这一节描述当这种类型的类对象在创建时,这些构件对象是如何初始化的。
为了完成初始化,得运用构造函数初始化或 *ctor* 初始化语法。
语法
ctor-初始化器

成员初始化器表
成员初始化器表：
成员初始化器
成员初始化器,成员初始化器表
成员初始化器：
完整的类名称(表达式表 opt)
标识符(表达式表 opt)
用于构造函数中的这一语法,在下一节“初始化成员对象”和“初始化基类”中更详尽介绍。

初始化成员对象

类可以含有类类型成员对象,但要保证满足对这些成员对象的初始化，并要满足如下之一的条件。

- 所含对象的类不要求有构造函数。
- 所含对象的类有一可访问的缺省构造函数。
- 包容类的构造函数显式初始化所含的对象。

下面的例子给出了如何完成这样的初始化：

```
//说明一个 Point 类
class Point
{
 public:
   Point (int x,int y) {_x=x;_y=y;}
```

```
 private:
    int _x,_y;
}
//说明一个包含有 Point 类型对象的 rectangle 类
class Rect
{
public:
    Rect (int x1,int y1,int x2,int y2);
private:
    Point _topleft,_bottomright;
}
//定义类 Rect 的构造函数,这一构造函数显式地初始化类 Point 对象
Rect :: Rect(int x1,int y1,int x2, int y2):
 _topleft(x1,y1),_bottomright(x2,y2)
{
}
```

在前面例子中显示的 Rect 类中含有两个 Point 类成员对象。它的构造函数显式地初始化了对象_topleft 和_bottomright。注意跟在构造函数的后括号后面的是冒号。跟在冒号后面的是成员名和参量,该参量是用来初始化类对象的。

警告:在构造函数中说明的成员初始化顺序不会影响这些成员被构造的次序。成员是按它们在类说明中的说明顺序被构造的。

引用及常量成员对象必须用成员初始化语法(在“初始化基类及成员”中描述)

来初始化。没有其它的办法来初始化这些对象。

初始化基类

直接基类的初始化同成员对象的初始化以同样的方式进行。考虑下面的代码：

```
//说明类窗口
class Window
{
public:
     Windodw (Rect rSize);
     ...
}
//说明类 DialogBox 直接派生于类 Window
class DialogBox:public Window
{
public:
    DialogBox(Rect rSize);
    ...
}
//定义 DialogBox 构造函数。这一构造函数显式地初始化 Window 子对象
DialogBox:: DialogBox(Rect rSize):Window(rSize)
{
}
```

注意在 DialogBox 的构造函数中,Windows 基类是使用参量 rSize 进行初始化的。这一初始化由要初始化的基类名称组成,基类名称后跟随的是由圆括号括起来的传递给基类构造函数的参量表。
在基类的初始化中,不代表基类构件的子对象的对象称为一个“完整对象”。该“完整对象”的类视为该对象的“最外派生类”(most derived)。
代表虚拟基类的子对象是由“最外派生类”的构造函数进行初始化的。这意味着只要说明了虚拟基类的派生类,则最外派生类必须显式初始化虚拟基类,否则虚拟基类必须有一个缺省的构造函数。在最外派生类以外的其它类的构造函数中出现的虚拟基类的初始化将被忽略。
尽管对基类的初始化通常限于直接基类的初始化,但一个类的构造函数可以初始化一个非直接的虚拟基类。

基类及成员的初始化次序

基类和成员对象按如下次序进行初始化:
1. 虚拟基类的初始化按它们出现在有向无环图中的次序。有关使用有向无环图去构造一个唯一子对象列表的详情见第 9 章“派生类”中的“虚拟基类”(注意,这些子对象的析构是按反序遍历该列表)。有关如何遍历有向无环图的情况见本章前面的“析构的次序”。
2. 非虚拟基类按它们在类说明中出现的次序进行初始化
3. 成员对象的初始化是按它们在类中说明的次序进行初始化。
在构造函数的成员初始化器表中说明的成员初始化器和基类初始化器的次序不会影响到基类及成员对象进行初始化的次序。

初始化的范围

对基类和成员对象的初始化定值在与其说明在一起的构造函数的范围内，因此，它们能够隐含地引用类成员。

拷贝类对象

两种操作会引起对象的拷贝：

- 赋值。在一个对象的值被赋给另一个对象时，第一个对象是被拷贝给第二个对象的。因此：

```
Point a,b;
...
a=b;
```

会引导起 b 的值拷贝给 a。

- 初始化。初始化发生在说明一个新对象时，发生在给函数传递参量的值时，也生在以传值形式从函数返回参量时。

程序员可以为类类型对象定义拷贝的意义。考虑下面的代码：

```
TextFile a, b;
a.Open("FILE1.DAT");
b.Open("FILE2.DAT");
b=a;
```

上面的代码可以意味着：“把 FILE1.DAT 的内容拷贝到 FILE2 中”；或者它也可

意味着:“忽略 FILE2.DAT,并把 b 作为 FILE1.DAT 的第二个句柄”。程序员有责任为每个类赋以合适的拷贝语义。
拷贝由下列两种方法完成:

- 赋值(使用赋值运算符,operator=)
- 初始化(使用拷贝构造函数)(有关拷贝构函数的详情见本章开头的“说明构造函数的规则”)。

任一给定的类可以实现一种或以上两种拷贝方式。如果两种方法都未实现,赋值将作为一个成员方式(memberwise)赋值,初始化也作为成员方式初始化。在本章后面会讨论有关“成员方式的赋值和初始化”的细节。
拷贝构造函数带有一个类型 class_name&的参量,其中 class_name 是定义拷贝构造函数的类的名称。例如:

```
class Window
{
public:
    Window (const Window&);    //定义拷贝构造函数
    ...
};
```

注意:只要可能,拷贝构造函数的参量都应该是 const class-name&。这可以防止拷贝构造函数无意中修改了要拷贝的对象。这也使得拷贝 const 对象成为可能。

编译器生成的拷贝

编译器生成的拷贝构造函数,就像用户自定义的拷贝构造函数,带有一个“对类类型引用”的参量类型。只有一个例外是:当所有的基类及成员类都说明有拷贝构造函数,并且该函数带有一个具有 const class_name& 型的参量时。在这种情况下,编译器生成的拷贝构造函数的参量是 const 型。
当拷贝构造函数的参量不是 const 型时,通过拷贝一个 const 对象进行初始化会产生一个错误。但反过来就不同:如果参量类型是 const 型的,通过拷贝一个非 const 型对象进行初始化不会产生错误。
关于 const,编译器产生的赋值运算符也遵循同样的方式。它们带有一个唯一的 class_name&类型的参量,除非赋值运算符在所有的基类及成员类中采用的参量类型是 const class_name& 。在这种情况下,编译器为该类产生的赋值运算符采用 count 型参量。
注意:当虚拟基类由拷贝构造函数(无论是编译器产生的,还是自定义的)进行初始化时,它们仅在被构造时初始化一次。
这些含义同拷贝构造函数很相似。当参量类型不是 const 型时,从一个 const 型对象赋值会产生一个错误。反之则不同:如果一个 const 型值赋给一个非 const 型值,赋值可以进行。
有关重载赋值运算符的更多的情况见第 12 章“重载”中的“赋值”。

成员方式的赋值与初始化

缺省的赋值和初始化的方法分别是“成员方式”的赋值和“成员方式的初始化”。

“成员方式的赋值”由从一个对象到另一个对象的拷贝过程组成。每次赋值一个成员,就像每个成员单独赋值一样,“成员方式的初始化”也是由从一个对象拷贝到另一个对象的拷贝过程组成。每次初始化一个对象,就像是每个成员单独被初始化。这两种方式的主要区别是:成员方式赋值激活的是每个成员的赋值运算符(operator=),而成员方式初始化激活的是每个成员的拷贝构造函数。

成员方式的赋值仅由说明为如下方式的赋值运算符来实现的。

type& *type*::operator=([const|volatile]*type*&)

如果下列任何条件存在,则成员方式赋值的缺省运算符不能出现:

- 某成员类中有 const 成员。
- 某成员类有引用成员。
- 某成员类或其基类有一个私有的赋值运算符(operator=)。
- 某成员类或其基类没有赋值运算符(operator=)。

如果一个类或其基类有一个私有的拷贝构造函数或者是下列任何条件存在,则该类的对于成员方式初始化的缺省构造函数不可能生成:

- 某成员类有 const 成员。
- 某成员类有引用成员。
- 某成员类或其基类有一个私有的拷贝构造函数。
- 某成员类或基类没有拷贝构造函数。

对于一个给定的类缺省的赋值运算符和拷贝函数总是已经说明了的。除非下面的两个条件都得以满足,否则它们是未定义的。

- 该类并没有为这一拷贝提供自定义的函数。
- 程序要求提供此函数。如果碰到一个要求成员方式拷贝的赋值运算

符或者初始化符,或者取了类的 operator=函数的地址则这一要求是存在的。

若上述两个条件都没有满足,则编译器不必为缺省的赋值运算符和拷贝构造函数产生代码(MSC 的编译器采用优化可以去掉这些代码)。尤其当类说明了一个自定义的 operator=函数,并以一个对类类型的引用为参量,则不会产生缺省的赋值运算符函数。若类说明了一个拷贝构造函数,则也不会产生缺省的拷贝构造函数。

因此对于给定的类 A,下面的说明总是存在的。

```
//拷贝构造函数和赋值操作的隐含说明
A::A(const A&)
A& A::operator=(const A&)
```

按上面的标准,这一定义只是在需要的时候才要提供。上面例子中的拷贝构造函数被认为是类的公有成员函数。

缺省的赋值运算符使得一给定类的对象可以赋值给其公有基类型的对象。考虑下面的代码:

```
class Account
{
public:
    //公有成员函数
  ...
private:
  double _balance;
```

```
};

class Checking:public Account
{
private:
    int _fOverdraftProtect;
};
 ...
Account   account;
Checking  checking;
account=checking;
```

在上面的例子中,选用的赋值运算符是 Account:operator=。因为缺省的 operator=函数带有一个类型 Acconut&的参量，checking 的 Account 型子对象拷贝给 account;而 fOverdraftProtect 没有拷贝。

第 12 章　重　　载

这一章阐述如何使用 C++的重载函数以及重载运算符。
包括下面一些主题：

- 重载概述
- 说明匹配
- 参量匹配
- 重载函数的地址
- 重载运算符

重载概述

有了 C++语言，你就可以重载函数和运算符。重载是一种应用，它在同一范围中为一个给定函数名称提供了多种定义。委托编译器依据调用该函数的参量选择合适的函数或运算符的版本。例如：

```
double max(double d1,double d2)
{
   return (d1>d2)?d1:d2;
}
```

```
int max (int i1, int i2)
{
   return (i1>i2)?i1:i2;
}
```

作为一个重载函数，函数 max 在程序中使用如下：

```
main()
{
  int i=max(12,8);
  double d=max(32.9,17.4);

  return i+(int)d;
}
```

在第一个例子中,要求出两个整型变量的最大值,故调用函数(int,int)。然而，在第二种情况下,两个参量是浮点型,因此调用的函数是 max(double,double)。

参量类型的差异

重载函数之间的区别在于带有不同初始值的参量类型。因而对一个给定类型的参量以及对于该类型的引用,在重载的意义上来说是完全相同的。它们被看成是相同的，因为它们采用了相同的初始值。例如:max(double,double)和(double &,double &)是完全相同的,说明两个这样的函数会引起错误。

出于相同的原因,用修饰符 const 和 volatile 进行修饰的函数参量类型同基本

类型,在重载的意义上看没有什么不同。
然而重载函数的机制可以区分由 const 或 volatile 修饰的引用以及基本类型的引用。这使得可以有下列代码:

```
#include<iostream.h>

class Over
{
public:
  Over() {cout << "Over default constructor\n"}
  Over(over &o){cout << "Over &\n";}
  Over(const Over &co)  {cout << "const Over &\n";}
  Over(volatile Over &vo) {cout << "volatile Over &\n";}
};

void main()
{
  Over o1;  //调用缺省构造函数
  Over o2(o1);  //调用 Over(Over&)
  const Over o3;//调用缺省构造函数
  Over o4(o3);  //调用 Over(const Over &)
  volatile Over o5; //调用缺省构造函数
  Over o6(o5);  //调用 Over(volatile Over &)
```

}
指向 const 和 volatile 对象的指针和指向其基本类型的指针在重载意义上是不同的。

重载函数的限制

一组重载函数是否是可接受的如下限制:

- 该组重载函数中任何两个都必须有不同的参量表。
- 具有相同类型参量表、仅在返回值类型上不同的重载函数会引起错误。

Microsoft 特殊处→
用户可以仅仅基于返回类型不同而重载 new 运算符。尤其在基于说明的存储器模式修饰符不同时。
Microsoft 特殊处结束

- 成员函数的重载不能仅基于一个说明为静态的,另一个说明为非静态的。
- typedef 说明并未定义新的类型,它们仅为已存在的类型引入了一个同义词。它们不能影响重载机制。考虑下面的代码:

```
typedef char * PSTR;

void Print(char * szToPrint);
void Print(PSTR   szToPrint);
```

前面的两个函数有相同的参量表,PSTR 是类型 char *的同义词。在成员范围中,

这样的代码会产生错误。

- 枚举类型是一些可区分的类型,故可以区分重载函数。
- 从区分重载函数的意义上说,类型“数组”和“指针”是相同的。对于一维数组来说是正确的。因而下面的重载函数出现了冲突,并会产生一条错误消息:

```
void Print (char * szToPrint);
void Print (char   szToPrint[]);
```

对于多维数组,第二和后续维数视为类型的一部分,因此它们可以用来区分重载函数:

```
void Print (char szToPrint []);
void Print (char szToPrint [][7]);
void Print (char szToPrint [][9][42]);
```

说明匹配

在同一范围中说明具有相同名称的任何两个函数可以指的是同一函数,或者指的是重载的两个不同函数。如果说明的参量列表含有相同的参量类型(如前一节描述的),则此函数说明指的是同一函数;否则它们指的是用重载选择的不同函数。类的范围是严格定义的,因此一个在基类中说明的函数同一个在派生类中说明的函数有不同的范围。如果一个在派生类中说明的函数同一个基类中说明的函数有相同的名称,派生类的函数将隐藏基类中的函数,而并不会引起函数重载。块范围也是严格定义的,因此一个在文件范围中说明的函数同一个局部说明的

函数并不在同一范围中。如果一个局部范围中说明的函数同一个文件范围中说明的函数具有相同的名称,则局部说明的函数将隐藏文件范围中说明的函数,而并不会引起重载。例如:

```
#include <iostream.h>

void func(int i)
{
   cout << "Called file-scoped func: " << i << endl;
}

void func (char *sz)
{
   cout << "Called locally declared func:" << sz << endl;
}

void main()
{
 //说明 main 中的局部 func 函数
 extern void func(char *sz);

 func(3); //错误:函数 func(int)被隐藏了
 func("s");
```

```
}
```
上面的代码给出了函数 func 的两个定义。以 char *类型作为参量的函数定义对于 main 函数是局部的,因为有 extern 语句。因而以 int 为参量类型的函数定义被隐藏了起来,故第一个对 func 函数的调用是错误的。
对于重载的成员函数,不同版本的函数可以给于不同的访问权限。它们仍被视为在类的范围中,因而它们是重载函数。考虑下面的代码,其中成员函数 Deposit 被重载了;一个版本是 public 的,另一个版本是 private 的。
```
class Account
{
public:
    Account();
    double Deposit(double dAmount,char *szPassword);
private:
    double Deposit (double dAmount);
    intValidate(char * szPassword);
};
```
上面的代码要提供一个 Account 类,其中执行 Deposit 函数,要求提供一个正确的口令。这一点是用重载来实现的。下面的代码给出了如何使用该类,以及对于私有成员函数 Deposit 的错误调用:
```
void main()
{
    //分配一个新的 Account 类型的对象
```

```
    Account *pAcct=new Account

    //存$57.22 错误:调用了一个私有的函数
    pAcct->Deposit(57.22);

    //存$57.22并提供一个口令,正确调用一个公有函数
    pAcct->Deposit(57.22,"pswd");
 }

double Account::Deposit(double dAmount,char *s2Password)
{
   if(Validate(szPassword))
     return Deposit(dAmount);
   else
     return 0.0;
}
```

注意，在 Account::Deposit 中调用了私有成员函数 Deposit,这一个调用是正确的。因为 Account::Deposit 是一个成员函数,因而有权访问该类的私有成员。

参量匹配

重载函数的选择是把调用函数的参量对当前范围中的函数进行最佳匹配。如果

发现了一个合适的函数,则调用此函数。“合适”在此处的含义是下列之一:

- 发现了一个完全匹配。
- 采用了一个平常转换(trival conversion)。
- 执行了一个整型参量提升。
- 存在着所需参量类型的标准转换。
- 存在着所需参量类型的用户自定义型的转换(无论是转换运算符还是构造函数)。
- 发现了由(...)表示的参量。

编译器为每一个参量创建了一个候选函数集。候选函数是那些在该位置上的实参可以转换成形参类型的函数。每一个参量都建立了一个最佳匹配的函数集。被选择的函数就是所有这些集合的交集。如果交集包含有多个函数,重载就是有二义性的,并产生一个错误。最后选择的函数总是在该组函数中更匹配,至少有一个参量更匹配一些。如果情况不是这样(如果没有更加匹配者),则函数调用会产生错误。

考虑下面的代码中的说明(函数标明了形式 1,形式 2 和形式 3,以便于下面的讨论):

```
Fraction &Add(Fraction &f,long l);//形式 1
Fraction &Add(long l,Fraction &f); //形式 2
Fraction &Add(Fraction &f,Fraction &f);//形式 3

Fraction F1,F2;
```

考虑下面的语句:

F1=Add(F2,23);
前面的语句构造了两个集合:

集合 1:第一个参量类型为 Fraction 的候选函数	集合 2:可以把第二参量类型转换成整型的候选函数
形式 1	形式 1(int 可用标准转换转换为 long 型)
形式 3	

在集合 2 的函数中,对于它们来说存在着从实参类型到形参类型的隐含转换。在这些转换中有一个函数对于此种转换的代价是最小的。
这两个集合的交集是形式 1。对于有二义性的函数调用的例子如下:
F1=Add(3,6);
上面的函数调用生成下面的集合:

集合 1:第一个参量可以有整型参量的候选函数	集合 2:第二个参量可以有整型参量的候选函数
形式 2(int 可以用标准转换转换成 long 型)	形式 1(int 型可以用标准转换转换成 long 型)

注意这两个集合的交集是空集,因此编译器产生错误消息。对于参量匹配,一个有 n 个缺省参量的函数被当作 n+1 个不同的函数,每个函数都有不同个数的参量。
(...)是作为通配符;它可以匹配任何实参。如果你不是极为小心地设计你的重载函数的话,则会导致很多有二义性的集合。
注意:重载函数的二义性直到碰到函数调用时才会发现。在那时,为函数调用的

每个参数都建立了集合,并且你可以发现是否有非二义性的重载存在。这意味着二义性会一直保留在你的代码中,直到它们被某个特殊的函数调用所发现。

参量匹配和 this 指针

对于类成员函数根据它们是否被说明为 static,将按不同的方式对待。因为非静态成员函数有一个隐含的参量以支持 this 指针。非静态函数被视为比静态函数多出一个参量。
除此之外,它们的说明是等同的。这些非静态的成员函数要求有一个隐含的 this 指针以匹配调用此函数的对象的类型。或者对于重载运算符来说,它们要求第一个参量以匹配运算符作用的对象(有关重载操纵符的情况见本章后面的“重载运算符”)。
同重载函数的其它参量不同,在对 this 指针参量进行匹配时,不会引入临时对象,也不会试图进行转换)。
当成员选择符(->)用于访问成员函数时,this 指针的类型是 class_name *const 型的。如果成员说明为 const 或者 volatile,则类型相应地是 const class_name *const 或 volatile class_name * const。
除了把一个隐含的&(取地址运算符)加在对象名称的前面之外,成员选择符(.)也以同样的方式工作。下面的例子显示这是如何工作的:

```
//在代码中碰到的表达式
obj.name

//编译器如何对待它
```

(&obj)->name
在参量匹配上运算符->*和.*(指向成员的指针)的左操作数同运算符.和->是同样对待的。

参量匹配和转换

当编译器试图按函数说明中的参量进行实参匹配时,如果没有找到精确的匹配,编译器可以支持标准的或用户自定义的转换以获得正确的类型。转换的应用受如下规则的限制:

- 含有多个用户自定义转换的转换序列是不能接受的。
- 通过移去中间转换即可缩短的转换序列也是不可被接受的。

转换的结果序列(如果有的话)称为最佳匹配序列。使用标准转换(第 3 章“标准转换”中描述),可有几种办法把一个 int 型对象转换成 unsigned long 型:

- 把 int 型转换成 long 型然后再转换成 unsigned long 型
- 把 int 型直接转换成 unsigned long 型。

第一种方法,尽管达到了所希望的目标,但不是一个最佳匹配的序列,因为存在着一个更短的匹配序列。

表 12.1 给出了一组转换,称为平常转换。它们在对决定哪一个序列是最佳匹配序列中起有限的作用。平常转换影响选择序列的步骤在下面讨论。

表 12.1 平常转换

从类型	转换到类型
类型名称	类型名称&
类型名称&	类型名称
类型名称[]	类型名称*
类型名称(参量表)	(*类型名称)(参量表)
类型名称	const 类型名称
类型名称	volatile 类型名称
类型名称*	const 类型名称*
类型名称*	volatile 类型名称*

转换的顺序按如下进行试探：

5. 精确匹配。在调用函数的参量类型和函数原型中说明的类型之间的精确匹配总是最好的匹配。平常转换序列归为精确匹配。然而一般认为，不做任何这些转换的序列比做了以下转换的序列更好：

- 从指针到指向 const 的指针(type*到 const type*)。
- 从指针到指向 volatile 的指针(type *到 volatle type*)。
- 从引用到对 const 的引用(type &到 const type &)。
- 从引用到对 volatile 的引用(type &到 volatile type &)。

2. 使用参量提升的匹配。任何不归类为精确匹配的序列若仅含有必要的参量提升(从 float 到 double)，则平常转换的序列将归类为使用参量提升的匹配。尽管这种匹配没有精确匹配好,但用参量提升的匹配比用标准转换的匹配好。

3．使用标准转换的匹配。任何不能归为精确匹配或参量提升的匹配的任何序列若仅含有标准转换，则平常转换被归类为使用标准转换的匹配。在这种匹配中，要遵循下面的规则：

- 把一个派生类的指针转换为一个指向其直接基类或非直接基类的指针比转换为 void *或 const void*更合适一些。
- 把一个派生类的指针转换为一个指向基类的指针，由此引出的对更靠近的基类的匹配是转换为直接基类的指针。假设如图 12.1 所示的类层次。

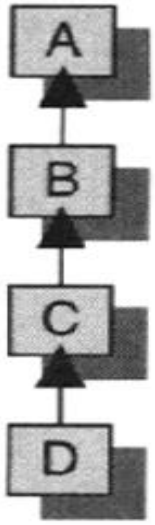


图 12.1　更可接受的转换的示意图

从类型 D*转换为类型 C*比从类型 D*转换为类型 B*更好。同样，从类型 D*到类型 B*的转换比从类型 D*到类型 A*的转换更好。

同样的规则也适用于引用的转换。从类型 D&到类型 C&比从类型 D&到类型 B&更好,依此类推。
这一规则也适用于指向成员指针的转换。从类型 T D::*到类型 T C::*的转换比从类型 T D::*到类型 T B::*的转换更好,依此类推(其中 T 是成员的类型)。
上面的规则适用于在同一个给定的派生路径中。考虑图 12.2 中的层次图。

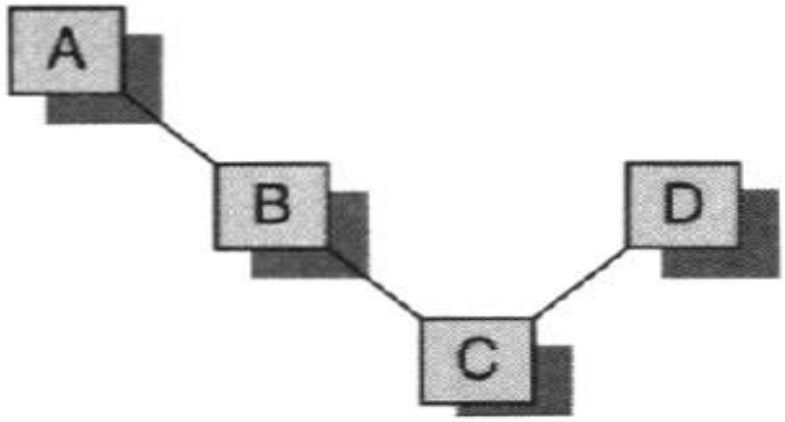


图 12.2 更可接受的转换的多重继承示意图

从类型 C*到类型 B*的转换比从类型 C*到类型 A*的转换更好。原因是它们处在同一路径之上。但,从类型 C*到类型 D*的转换并不比从类型 C*到类型 A*更好,因为转换是沿着不同的路径。
4. 采用自定义型转换的匹配。这种序列不能归类为精确匹配、利用参量提升的匹配或采用标准转换的匹配。归为采用自定义转换的匹配序列必须仅含有用户自定义的转换、标准转换或平常转换。采用自定义转换的匹配比采用省略符的匹配要好些,但比用标准转换的匹配要差一些。

5. 采用省略符(…)的匹配。任何匹配函数说明中省略符的序列都归为采用省略符的匹配。这种匹配是最差的匹配。

用户自定义的转换适用于没有内部的参量提升或存在转换的情况。这种转换的选择基于要被匹配的参量的类型。考虑下面的代码:

```
class UDC
{
public:
    operator int();
    operator long();
};

void Print (int I);
…
UDC udc;
Print(udc);
```

对于类 UDC 可用的用户自定义的转换是 int 型和 long 型，因而编译器可以接受来自于要被匹配的对象的类型的转换。存在着一个从 UDC 到 int 型的转换,故选择了此转换。

在参量匹配过程中,标准转换可以用于参量及用户自定义转换的返回值。因而，下面的代码是可以工作的:

```
void logToFile(long l);
…
```

```
UDC udc;
LongToFile(udc);
```

在上面的例子中,将调用用户自定义型转换 operator long 把 UDC 型转换为 long 型。如果没有定义从 UDC 型到 long 型的转换,此转换将按如下过程进行:先用自定义型转换把 UDC 型转换为 int 型,然后将利用从 int 型到 long 型的标准转换来匹配说明中的参量。

如果匹配参量需要自定义型转换,则在评价最佳匹配时是不考虑标准转换的。即使有多个候选函数要求用户自定义转换,也不对标准转换进行考虑。在这种情况下,这些函数是等同的。例如:

```
class UDC1;
{
public:
  UDC1(int);    //用户自定义的从 int 型的转换
};

class UDC2
{
public:
  UDC2(long);   //用户自定义的从 long 型的转换
};

…
```

```
void Func(UDC1);
void Fuce(UDC2);

…

Func(1);
```

这两个版本的 Func 都要求用户自定义的转换:把 int 型转换为类类型参量。可能的转换如下:

- 从 int 型到 UDC1 的转换(用户自定义型转换)。
- 从 int 型到 long 型的转换,然后转换为 UDC2 型(分两步转换)。

尽管第二种转换方式要求有标准转换和用户自定义型转换,然而这两种转换是等同的。

注意:用户自定义的转换被视为通过构造函数的转换或通过初始化的转换(转换函数)。在考虑最佳匹配时,这两种方法视为是等同的。

重载函数的地址

不带参量表使用一个函数的名称会返回该函数的地址,例如:

```
int Func(int I,int j);
int Func(long l);
…
```

```
int (*pFunc)(int,int)=Func;
```

在前面的例子中将选用第一个 Func 函数，并且它的地址被拷贝到 pFunc 中。通过参量表同目标的完全匹配来搜索函数,编译器可以决定选择哪个版本的函数。在重载函数中说明的参数表对照下列之一进行匹配。

- 一个被初始化的对象(如前面的例子)
- 赋值语句的左边
- 一个函数的形参
- 一个自定义运算符的形参
- 函数的返回值

如果没有找到完全的匹配,则取函数地址的表达式是有二义性的,并产生一个错误。

注意，尽管在上面的例子中使用了一个非成员函数 Func,然而同样的规则适用于取重载的成员函数的地址。

重载运算符

使用 C++,你可以重新定义大多数内部运算符的函数,这些函数可以基于全局方式下或类方式下进行重定义或重载。重载运算符是用函数来实现的,它们可以是成员函数也可以是全局函数。

一个重载运算符的名称是 operator$x$,其中 x 是出现在表 12.2 中的运算符。例如:要重载一个加法运算符,你可以定义一个函数叫作 operator+。同样地,重载

加/赋值运算符(+=),可以定义一个函数叫作:operator+=。
尽管这些运算符函数出现在代码中时通常是由编译器隐含调用的,但它们可以按任何成员函数或非成员函数的方式进行显式调用。

```
Point pt;
pt. operator+(3); //调用加法运算符向 pt.加 3
```

表 12.2 可重定义的运算符

运算符	名称	类型
,	逗号	双目
!	逻辑非	单目
!=	不等	双目
%	取模	双目
%=	取模/赋值	双目
&	按位 AND	双目
&	取地址	单目
&&	逻辑和	双目
&=	按位和/赋值	双目
()	函数调用	--
*	乘法	双目
*	指针间接引用	单目
*=	乘法/赋值	双目
+	加法	双目

续表

+	单目加	单目
++	增[1]	单目
+=	加法/赋值	双目
-	减法	双目
-	单目取反	单目
--	减[1]	单目
-=	减法/赋值	双目
->	成员选择	双目
->*	指向成员的指针选择	双目
/	除法	双目
/=	除法/赋值	双目
<	小于	双目
<<	左移	双目
<<=	左移/赋值	双目
<=	小于等于	双目
=	赋值	双目
==	等于	双目
>	大于	双目
>=	大于/赋值	双目
>>	右移	双目
>>=	右移/赋值	双目

续表

[]	数组下标	— —
^	异或	双目
^=	异或/赋值	双目
|	按位或	双目
|=	按位或/赋值	双目
||	逻辑或	双目
~	求补	单目
delete	delete	— —
new	new	— —

* 有两种方式的增1和减1存在:前缀方式和后缀方式。

重载运算符的各个分类的约束在本章后面“单目运算符”、“双目运算符”、“赋值”、“函数调用”、“下标运算”、“类成员访问”及“增1和减1”中描述。

表 12.3 不能重载的运算符

运算符	名称
.	成员选择
.*	指向成员指针的选择
::	范围分辨符
?:	条件运算符
#	预处理符号
##	预处理符号

运算符重载的一般规则

下面的规则约束着重载运算符如何实现，但它们并不适用于 new 和 delete 运算符。这两个运算符在第 4 章中单独讨论。

- 运算符必须要么是成员函数，或者带有某个类的参量，或者是枚举类型的参量、或者带有某个类的引用、或某个枚举的类型的引用。例如：

```
class Point
{
public:
   point operator<(Point &);  //说明一个成员运算符重载
   ...
   //说明加法运算符
   friend Point operator+(Point &,int);
   friend Point operator+(int,Point &);
};
```

上面的例子代码中说明了小于运算符成员函数;然而加法运算符是作为全局函数说明的,并具有友元访问属性。注意，对于一个给定的运算符可以提供多个实现。在上面的加法运算符的例子中,为了方便加法的交换性,提供了两个实现。正如这些运算符一样,把 Point 加到 Point,把 int 加到 Point 的运算符也可以实现。

- 运算符要遵守它们同内部类型一起使用所指定的优先原则、分组及操作数的个数。因此,无法表达把 2 及 3 加到一个 Point 对象中的

含义,除了把 2 加到 X 坐标中,把 3 加到 Y 坐标中。

- 单目运算符说明为成员函数不带参量;如果说明为全局函数,要带一个参量。
- 双目运算符说明为成员函数只带一个参量;如果说明为全局函数,要带两个参量。
- 所有的重载运算符除了赋值(operator=)外均可被派生类继承。
- 重载运算符的成员函数的第一个参量总是激活该运算符的对象的类类型参量(运算符被定义的类,或者定义了运算符的类的派生类)。对于第一个参量也不支持转换。

注意,任何运算符的意义都可能被完全地改变了,这包括取地址(&)、赋值(=)、函数调用运算符的含义。, 同理,内部的类型可由于使用了运算符重载而改变。例如:下面四条语句在完成求值以后完全等同:

```
var=var+1;
var+=1;
var++;
++var;
```

对于重载了运算符的类类型来说,这种确信是靠不住的,而且,对于在基本类型中使用这些运算符的隐含条件,对于重载的运算符来说是放松了。例如:加法/赋值操纵符,在应用于基本类型时,要求其左操作数是 l 值的;但此运算符重载以后就没有这种要求了。

注意:出于一致的考虑,最好还是遵循内部类型的模式进行运算符重载。如果一个重载的运算符同它在其他上下文中的意义不同,则它只会引起迷惑而不是更有

用。

单目运算符

表 12.4 给出了单目运算符。

表 12.4 可重载的单目运算符

运算符	名称
!	逻辑非
&	取地址
~	求补
*	指针间接引用
+	单目加
++	增 1
-	单目取反
--	减 1

在表 12.4 中所给出的运算符中,后缀的增 1 和减 1 运算符在“增 1 及减 1”一节讨论。

说明一个单目运算符为一个非静态成员,必须按如下形式进行说明:

ret-type operator*op*()

其中 *ret-type* 是返回类型,而 *op* 是表 12.4 中的某个运算符。

说明一个单目运算符为全局函数,必须按如下形式说明:

ret-type operator*op*(*arg*)

其中 *ret-type* 和 *op* 的含义同成员运算符函数中的描述,而 *arg* 是对其操作的类类型参量。

注意:对于单目运算符的返回值没有限制。例如,对于一个逻辑非运算符,返回一个必要的值是合适的。但这一点不是必须的。

增 1 和减 1

增 1 和减 1 运算符放入特殊的分类是因为每个运算符都有两种形式:

- 前缀型增 1 和后缀型增 1。
- 前缀型减 1 和后缀型减 1。

在你编写重载运算符函数时,为这种运算符的前缀和后缀型各自提供一个版本是很有用的。为分清这两个版本,要遵循下面一些规则:对于前缀型运算符可按其它任何单目操纵符完全相同的方式说明,后缀型则要接受一个附加的 int 型参量。

重要:当为后缀型增 1 和减 1 运算符说明重载函数时,附加的参量必须是 int 型的,说明任何其它的类型都会产生错误。

下面的例子显示了如何为类 Point 定义前缀型和后缀型增 1 和减 1 运算符函数:

```
class Point
{
public:
    //说明前缀和后缀型增 1 运算符
    Point& operator++();  //前缀型增 1 运算符
    Point operator++(int);  //后缀型增 1 运算符
```

```
    //说明前缀和后缀型减 1 运算符
    Point& operator--();    //前缀型减 1 运算符
    Point operator--(int);  //后缀型减 1 运算符

     //说明缺省的构造函数
     Point() {_x=_y=0;}

     //说明访问函数
     int x() {return _x;}
     int y() {return _y;}
private:
     int _x,_y;
}

//定义前缀型增 1 运算符
Point& Point::operator ++()
{
     _x++;
     _y++;
    return *this;
}
```

```
//定义后缀型增 1 运算符
Point Point:: operator++(int)
{
    Point temp =*this;
    ++*this;
    return temp;
}

//定义前缀型减 1 运算符
Point& Point::operator--()
{
    _x--;
    _y--;
    return *this;
}

//定义后缀型减 1 运算符
Point Point::operator--(int)
{
    Point temp= *this;
     --*this;
```

```
    return temp;
}
```
使用下面的函数头,可以在文件范围(全局)定义同样的运算符:
```
friend Point& operator++(Point&)  //前缀增 1
friend Point& operator++(Point&,int)  //后缀增 1
friend Point& operator--(Point&)  //前缀减 1
friend Point& operator--(Point&,int)  //后缀减 1
```
表示后缀型增 1 减 1 运算符的 int 型参量并不是普通地用作参量传递。它通常含有 0 值,然而可以按如下使用:
```
class Int
{
public:
   Int& operator++(int n);
private:
   int _i;
};

Int& Int::operator++(int n);
{
    if(n!=0)//传递了一个参量的处理情况
       _i+=n;
    else
```

```
        _i++;    //没有传递参量的处理情况
    return *this;
}
...
Int i;
i.operator++(25);    //加 25
```

如上面所示的,除了显式调用之外没有别的语法可以用于使用增 1 减 1 运算符传递函数值。一个更直接地实现这一功能的办法是重载+=运算符。

双目运算符

表 12.5 给出了可被重载的运算符。

表 12.5 可重定义的双目运算符

运算符	名称
,	逗号
!=	不等
%	取模
%=	取模/赋值
&	按位和
&&	逻辑和
&=	按位和/赋值
*	乘法

续 表

*=	乘 法 / 赋 值
+	加 法
+=	加 法 / 赋 值
-	减 法
-=	减 法 / 赋 值
->	成 员 选 择
->*	指 向 成 员 的 指 针 选 择
/	除 法
/=	除 法 / 赋 值
<	小 于
<<	左 移
<<=	左 移 / 赋 值
<=	小 于 等 于
=	赋 值
==	等 于
>	大 于
>=	于 / 等 于
>>	右 移
>>=	右 移 / 赋 值
^	异 或
^=	异 或 / 赋 值

续表

|	按位或
|=	按位或/赋值
||	逻辑或

说明一个双目运算符函数为非静态成员函数,必须按如下形式说明:

ret-type **operator*op*(*arg*)**

其中 *ret-type* 是返回类型,*op* 是表 12.5 中列出的某个运算符,而 *arg* 则是一个任意类型的参量。

说明一个双目运算符为全局函数,必须按如下形式说明:

ret-type **operator*op*(*arg1,arg2*)**

其中 *ret-type* 和 *op* 同成员函数中的描述是一致的,而 *arg1* 和 *arg2* 是参量。至少其中之一必须是类类型。

注意:对于双目运算符的返回类型没有限制;然而大多数用户自定义型双目运算符返回类类型或类类型的引用。

赋　值

赋值运算符严格地说是一个双目运算符。它的说明等同于其它双目运算符的说明,但下列情况除外。

- 它必须是非静态成员函数。没有 operator=可以说明为非成员函数。
- 它不能被派生类继承。
- 如果不存在缺省的 operator=函数,则编译器会为该类生成一个缺省的函数(有关缺省的 operator=函数见第 11 章“特殊成员函数”中

的“成员方式的赋值和初始化”)。

下面是如何说明赋值运算符的示例:

```
class Point
{
public:
    Point &operator=(Point &);    //右边是参量
    ...
}
//定义赋值运算符
Point &Point::operator=(Point &ptRHS)
{
     _x = ptRHS._x;
     _y = ptRHS._y;
     return *this ;  //赋值运算符返回左边
}
```

注意,提供的参量是表达式的右边。运算符返回的对象保持了赋值运算符的行为(赋值完成以后,赋值运算符将返回的是左边的值),这样,可以写出下面的语句:

```
pt1=pt2=pt3;
```

函数调用

函数调用运算符涉及使用括号,它是双目运算符。函数的语法如下:

语法

primary-expression (*expression-list* $_{opt}$)

在这里,*primary-expression* 是第一个操作数,*expression-list*(可能是空参量表)是第二个操作数。函数调用运算符用于多参量的运算。这一点可以做到,因为 *expression-list* 是一个参量表,而不是一个单一的参量。函数调用运算符必须是一个非静态的成员函数。

在函数调用运算符被重载以后,它不会改变函数被调用的行为,而只会影响到该操纵符运用于某给定类类型时如何理解该运算符。例如:下面的代码通常是无意义的:

```
Point pt;
pt(3,2);
```

给出适当的重载函数调用运算符,这一语法就可用来给 X 坐标增加 3 个单位,为 Y 坐标增加 2 个单位。下面的代码说明上述定义:

```
class Point
{
public:
   Point(){_x=_y=0;}
   Point &operator()(int dx,int dy)
        { _x+=dx;_y+=dy;return *this;}
private:
    int _x,_y;
};
```

...

```
Point pt;
pt(3,2);
```

注意，函数调用运算符适用于对象名,而不是函数名称。

下标运算

就像函数调用运算符一样,下标运算符([])也是一个双目运算符。下标运算符必须是一个非静态成员函数，并带有一个参量。这一参量可以是任何类型，并指明所希望的数组下标。
下面的例子显示如何创建一个实现边界检查的 int 型向量：

```
#include <iostream.h>

class IntVector
{
public:
     IntVector(int cElements);
     ~IntVector(){delete _iElements;}
     int& operator[](int nSubscript);
private;
    int*_iElements;
```

```
    int_iUpprtBound;
};

//构造一个 IntVector 对象
IntVector::IntVector(int cElements)
{
    _iElements = new int[cElements];
    _iUpperBound=cElements;
}

//InVetor 的下标运算符
int& IntVector:: operator[] ( int nSubscript)
{
    static int iErr=-1;
    if(nSubscript>=0 && nSubscript< _iUpperBound)
      return _iElements[nSubscript];
    else
{
   clog<<"Array bound violation."<<endl;
       return iErr;
     }
}
```

```
//测试 IntVector 类
int main()
{
    IntVector v(10);
    for(int i=0; i<=10;++i)
        v[i]=i;
    v[3]=v[9];
    for (i=0;i<=10;++i)
        cout<<"Element:["<<i<<"]="<<v[i]<<endl;
     return v[0];
}
```

在上面的例子中,当 i 到 10 时,operator[]检测到使用了一个超过边界的下标,并发出一个错误消息。

注意,函数 operator[]返回的是一个引用类型。这使它是一个 l 值的,使得在赋值号的两边可以使用下标表达式。

类成员访问

类成员的访问可以通过重载成员选择符(->)加以控制。在使用中,这一个运算符视为单目运算符,并且重载的运算符函数必须是成员函数,因此这一函数的说明是:

class-type *operator->()

这里 *class-type* 是该运算符所属的类的名称。成员选择符函数必须作为非静态成员函数。

该运算符(通常同指针间接引用运算符连用)用于实现“灵敏”指针,该指针使得指针在间接引用或计数之前生效。

成员选择符(.)是不能被重载的。

附录 A 语 法 总 结

本附录描述一般 C++语言中的语法,按在 Microsoft C++编译器中的实现。它是按本书的章节结构松散地按如下进行组织:

- 在第 1 章“词汇规定”的“关键字”一节中描述关键字。
- 在第 4 章“表达式”的“表达式”一节中描述表达式语法。
- 在第 6 章“说明”的“说明”一节中描述说明的语法。
- 在第 7 章“说明符”的“说明符”一节中描述说明符语法。
- 在第 8 章“类”的“类”一节中讨论用于说明类的语法。
- 在第 5 章“语句”的“语句”一节中讨论用于写语句的语法。
- 在“Microsoft 扩展”一节上讨论只适用于 Microsoft C++的特性。很多这些特性在附录 B“Microsoft 特殊修饰符”中涉及。

关 键 字

类名称:
    标识符
枚举名称:
    标识符

typedef 名称：
    标识符
标识符：以下之一
    非数字
    标识符 非数字
    标识符 数字
非数字：以下之一
    _ a b c d e f g h I j k l m
    n o p q r s t u v w x y z
    A B C D E F G H I J K L M
    N O P Q R S T U V W X Y Z
数字：以下之一
    0 1 2 3 4 5 6 7 8 9

表达式

表达式：
    赋值表达式
    表达式，赋值表达式
赋值表达式：
    条件表达式
    单目表达式 赋值运算符 赋值表达式

赋值运算符:以下之一
    = *= /= %= += -= >= <= &= ^= |=
条件表达式:
    逻辑或表达式
    逻辑或表达式 ? 表达式 : 条件表达式
逻辑或表达式:
    逻辑与表达式
    逻辑或表达式 || 逻辑与表达式
逻辑与表达式:
    逻辑或表达式
    逻辑与表达式 && 或表达式
或表达式:
    异或表达式
    或表达式 | 异或表达式
异或表达式:
    与表达式
    异或表达式 ^ 与表达式
与表达式:
    相等表达式
    与表达式 & 相等表达式
相等表达式:
    关系表达式

相等表达式 == 关系表达式
相等表达式 != 关系表达式

关系表达式：
移位表达式
关系表达式 < 移位表达式
关系表达式 > 移位表达式
关系表达式 <= 移位表达式
关系表达式 >= 移位表达式

移位表达式：
加法表达式
移位表达式 << 加法表达式
移位表达式 >> 加法表达式

加法表达式：
乘法表达式
加法表达式 + 乘法表达式
加法表达式 - 乘法表达式

乘法表达式：
段表达式
乘法表达式 * 段表达式
乘法表达式 / 段表达式
乘法表达式 % 段表达式

段表达式：

指向成员指针表达式
段表达式 :> 指向成员指针表达式
指向成员指针表达式:
类型转换表达式
指向成员指针表达式 .* 类型转换表达式
指向成员指针表达式 ->* 类型转换表达式
类型转换表达式:
单目表达式
(类型名称) 类型转换表达式
单目表达式:
后缀表达式
++单目表达式
--单目表达式
单目运算符 类型转换表达式
sizeof 单目运算符
sizeof (类型名称)
分配表达式
取消分配表达式
单目运算符:以下之一
* & + - ! ~
分配表达式:
::$_{opt}$ new 位置 $_{opt}$ new 类型名称 new 初始化符 $_{opt}$

::$_{opt}$ new 位置 $_{opt}$ (类型名称) new 初始化符 $_{opt}$

位置:

(表达式表)

new 类型名称:

类型指示符表 new 说明符 $_{opt}$

new 说明符:

ms 修饰符表 $_{opt}$ * cv 限定符表 $_{opt}$ new 说明符 $_{opt}$

ms 修饰符表 $_{opt}$ 完整类名称:: *cv 限定符表 $_{opt}$

new 说明符 $_{opt}$

new 说明符 $_{opt}$[表达式]

new 初始化器:

(初始化器表)

取消分配表达式:

::opt delete 造型表达式

::opt delete[ ] 造型表达式

后缀表达式:

基本表达式

后缀表达式 [表达式]

后缀表达式 (表达式表)

简单类型名称 (表达式表)

后缀表达式.名称

后缀表达式->名称

后缀表达式++
后缀表达式 -
dynamic_cast<类型标识>(表达式)
static_cast<类型标识>(表达式)
const_cast<类型标识>(表达式)
reinterpret_cast<类型标识>(表达式)
typeid(表达式)
typeid(类型标识)

表达式表:
赋值表达式
表达式表,赋值表达式

基本表达式:
literal
this
:: 标识符
:: 运算符函数名称
:: 限定名(表达式)
名称

名称:
标识符
运算符函数名称
转换函数名称

～类名称
限定名
限定名：
ms 修饰符表 opt 限定类名称 :: 名称
literal:
整型常量
字符型常量
浮点型常量
字符串文字
整型常量：
十进制常量 整型后缀 opt
八进制常量 整型后缀 opt
十六进制常量 整型后缀 opt
'c 字符序列'
十进制常量：
非零数字
十进制常量 数字
八进制常量：
0
八进制常量 八进制数字
十六进制常量：
0x 十六进制数字

0X 十六进制数字
十六进制常量 十六进制数字
非零数字:以下之一
1 2 3 4 5 6 7 8 9
八进制数字:以下之一
0 1 2 3 4 5 6 7
十六进制数字:以下之一
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F
整型后缀:
无符号后缀 长整型后缀 $_{opt}$
长整型后缀 无符号后缀 $_{opt}$
无符号后缀:以下之一
u U
长整型后缀:以下之一
l L
字符常量:
'c 字符序列'
L'c 字符序列'
c 字符序列:
c 字符

c 字符序列　c 字符

c 字符：

源字符集中除单引号(')、反斜线(\)、换行字符以外的任何字符

转义字符

转义字符：

简单转义字符

八进制转义字符

十六进制转义字符

简单转义序列：以下之一

\'　\"　\?　\\

\a　\b　\f　\n　\r　\t　\v

八进制转义序列：

\八进制数字

\八进制数字　八进制数字

\八进制数字　八进制数字　八进制数字

十六进制转义序列：

\x 十六进制数字

十六进制转义序列　十六进制数字

浮点常量：

分数常量　指数部分$_{opt}$　浮点后缀$_{opt}$

数字序列　指数部分　浮点后缀$_{opt}$

分数常量：

数字序列$_{opt}$.数字序列
数字序列.
指数部分:
e 符号$_{opt}$ 数字序列
E 符号$_{opt}$ 数字序列
符号:以下之一
+ -
数字序列:
数字
数字序列 数字
浮点后缀:以下之一
f l F L
字符串文字:
"s 字符串序列$_{opt}$"
L"s 字符串序列$_{opt}$"
s 字符序列:
s 字符
s 字符序列 s 字符
s 字符:
源字符集中除双引号(")、反向斜线(\)、换行字符以外的任何字符
转义序列

说　明

说明：
　　说明说明符 opt　说明符表 opt
　　asm 说明
　　函数定义
　　连接规格
　　模板说明
asm 说明：
　　_asm(字符串文字);
说明指示符表
　　说明指示符表 opt　说明指示符
说明指示符：
　　存储类指示符
　　类型指示符
　　函数指示符
　　friend
　　typedef
　　__declspec(扩展说明修饰符序列)
存储类型指示符：
　　auto
　　register

static
extern
函数指示符：
inline
virtual
类型指示符：
简单类型名称
类指示符
枚举指示符
全类型指示符
const
volatile
扩展说明修饰符序列：
扩展说明修饰符 $_{opt}$
扩展说明修饰符 扩展说明修饰符序列
扩展说明修饰符：
thread
naked
dllimport
dllexport
简单类型名称：
完整类型名称

限定类型名称
char
short
int
long
signed
unsigned
float
double
void

全类型指示符：
类关键字 标识符
类关键字 类名称
枚举名称

类关键字：
class
struct
union

限定类型名称：
typedef 名称
类名称::限定类型名称

完整类名称：

限定类型名称
::限定类型名称
限定类型名称:
类名称
类名称:: 限定类型名称
枚举指示符:
enum 标识符 $_{opt}$ { 枚举列表 $_{opt}$ }
枚举列表:
枚举符
枚举列表,枚举符
枚举符:
标识符
标识符=常量表达式
常量表达式:
条件表达式
连接规格:
extern 字符串文字 { 说明表 $_{opt}$ }
extern 字符串文字 说明
说明表:
说明
说明表 说明
模板说明:

template <模板参量表> 说明
模板参量表：
模板参量
模板参量表,模板参量
模板参量：
类型参量
参量说明
类型参量：
class 标识符
模板类名称：
模板名称<模板参量表>
模板参量表：
模板参量
模板参量表 , 模板参量
模板参量表：
模板参量
模板参量表,模板参量
模板参量：
表达式
类型名称
原始名称空间名称：
标识符

名称空间定义：

    原始名称空间定义

    扩展名称空间定义

    无名名称空间定义

原始名称空间定义：

    namespace 标识符 { 名称空间体 }

扩展名称空间定义：

    namespace 原始名称空间名称 { 名称空间体}

无名名称空间定义：

    namespace { 名称空间体 }

名称空间体：

    说明序列 $_{opt}$

id 表达式：

    无限定 id

    限定 id

嵌套名称指示符：

    类或名称空间名称::嵌套名称指示符 $_{opt}$

类或名称空间名称：

    类名称

    名称空间名称

名称空间名称：

    原始名称空间名称

名称空间别名

名称空间别名：

标识符

名称空间别名定义：

namespace 标识符 = 限定名称空间指示符；

限定名称空间指示符：

::$_{opt}$ 嵌套名称指示符 $_{opt}$ 类或名称空间名称

用法说明：

using::$_{opt}$ 嵌套名称指示符 未限定 id

using:: 未限定 id

用法命令：

using namespace :: $_{opt}$ 嵌套名称指示符 $_{opt}$ 名称空间名称

说明符

说明符表：

初始化说明符

说明符表 ，初始化说明符

初始化说明符：

ms 修饰符表 $_{opt}$ 说明符 初始化器 $_{opt}$

说明符：

说明名称

指针运算符　说明符
说明符 (参量说明表) cv修饰符表$_{opt}$
说明符 [常量表达式$_{opt}$]
(说明符)

cv修饰符表:
cv限定符　cv修饰符表$_{opt}$

指针运算符:
ms修饰符表$_{opt}$ * cv限定符表$_{opt}$
ms修饰符表$_{opt}$ & cv限定符表$_{opt}$
ms修饰符表$_{opt}$　完整类型名称:: *cv限定符表$_{opt}$

cv修饰符表:
cv修饰符 cv修饰符表$_{opt}$

cv限定符:
const
volatile

dname:
名称
类名称
～类型名称
typedef名称
限定类型名称

类型名称:

类型指示符表 ms修饰符表$_{opt}$ 抽象说明符$_{opt}$

类型指示符表:

类型指示符 类型指示符表$_{opt}$

抽象说明符:

指针运算符 ms修饰符表$_{opt}$抽象说明符$_{opt}$

抽象说明符$_{opt}$ (参量说明表) cv限定符表

抽象说明符$_{opt}$ [常量表达式$_{opt}$]

(ms修饰符表$_{opt}$表抽象说明符)

参数说明表:

参量说明表$_{opt}$ ...$_{opt}$

参量说明表 , . . .

参量说明表:

参数说明

参量说明表,参数说明

参数说明:

decl指示符 ms修饰符表$_{opt}$ 说明符

decl指示符 ms修饰符表$_{opt}$ 说明符 = 表达式

decl指示符 ms修饰符表$_{opt}$ 抽象说明符$_{opt}$

decl指示符 ms修饰符表$_{opt}$ 抽象说明符$_{opt}$ = 表达式

函数定义:

decl指示符$_{opt}$ ms修饰符表$_{opt}$ 说明符 ctor初始化器$_{opt}$ 函数体

函数体:

复合语句

初始化器：

= 表达式

= { 初始化器表 $_{opt}$ }

(表达式表)

初始化器表：

表达式

初始化器表,表达式

{ 初始化器表 , $_{opt}$ }

类

类指示符：

类头 { 成员表 $_{opt}$ }

类头：

类关键字 环境模式 $_{opt}$ 标识符 $_{opt}$ 基类指示符 $_{opt}$

类关键字 环境模式 $_{opt}$ 类名称 基类指示符 $_{opt}$

成员表：

成员说明 成员表 $_{opt}$

访问指示符:成员表 $_{opt}$

成员说明：

说明指示符 成员说明符表 $_{opt}$;

函数定义;opt
限定名;
成员说明符表:
成员说明符
成员说明符表,成员说明符
成员说明符:
ms修饰符表opt 说明符 纯指示符opt
标识符opt:常量表达式
纯指示符:
=0
基类指示符:
:基类表
基类表:
基类指示符
基类表,基类指示符
基类指示符:
完整类名称
virtual 访问指示符opt 完整类名称
访问指示符 virtualopt 完整类名称
访问指示符:
private
protected

public
转换函数名称：
operator 转换类型名称
转换类型名称：
类型指示符表 指针运算符 $_{opt}$
ctor 初始化器：
:成员初始化器表
成员初始化器表：
成员初始化器
成员初始化器,成员初始化器表
成员初始化器：
完整类名称(表达式表 $_{opt}$)
标识符(表达式表 $_{opt}$)
运算符函数名称：
operator 运算符
运算符:以下之一

new	delete							
+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=	<<=	==	!=
<=	>=	&&	||	++	--	,	->*	->
()	[]							

语　句

语句：
　　标号语句
　　表达式语句
　　复合语句
　　选择语句
　　迭代语句
　　跳转语句
　　说明语句
　　try-except 语句
　　try-finally 语句
标号语句
　　标识符 ： 语句
　　case 常量表达式 ： 语句
　　default ： 语句
表达式语句：
　　表达式 $_{opt}$ ；
复合语句：
　{ 语句表 $_{opt}$ }
语句表：
　　语句

语句表 语句

选择语句：

if (表达式) 语句

if (表达式) 语句 else 语句

switch (表达式) 语句

迭代语句：

while (表达式) 语句

do 语句 while (表达式) ;

for (for初始化语句 表达式$_{opt}$; 表达式$_{opt}$) 语句

for初始化语句：

表达式语句

说明语句

跳转语句：

break;

continue;

return 表达式$_{opt}$;

goto 标识符 ;

说明语句：

说明

try-except语句：

__try 复合语句

__except (表达式) 复合语句

try-finally 语句：

__try 复合语句

__finally (表达式) 复合语句

Microsoft 扩展

asm 说明：

__asm 汇编指令;opt

__asm { 汇编指令表 } ;opt

汇编指令表 ：

汇编指令;opt

汇编指令;汇编指令表;opt

ms 修饰符表：

ms 修饰符 ms 修饰符表 opt

ms 修饰符：

__cdecl

__fastcall

__stdcall

__syscall(为将来实现而保留)

__oldcall(为将来实现而保留)

__unaligned(为将来实现而保留)

基修饰符

基修饰符：

__based（基类型）

基类型：

名称

附录 B　Microsoft 特殊修饰符

很多 Microsoft 特殊的关键字可以用来修饰说明符以形成派生类型(有关说明符的更多信息参见第 7 章“说明符”)。

表 B.1　Microsoft 特殊的关键字

关键字	意义	是否用于形成派生类
__asm	加入后续的汇编语言代码	否
__based	把后续的名称说明为相对于说明中所包含的 32 位基址的一个 32 位的偏移地址	是
__cdecl	把后面跟着的名称按 C 语言规则命名并用 C 语言的调用规则。	是
__declspec	把后面跟着的名称(thread,naked,dllimport,dllexport) 说明为 Microsoft 特殊的存储属性	否
__fastcall	把后面所说明的函数名称说明为只要有可能就使用寄存器来传递参数而不是用堆栈来传递参数	是

续表

__stdcall	把后面所说明的函数名称说明为遵循标准转换规则是

下面的几节讨论有关 Microsoft 的特殊修饰符的句法使用和语义上的意义。

基地址

这一节包括下面一些主题：

- __based
- 基指针
- 基于指针的指针

在 32 位的编辑环境中使用__based

当你要显式控制分配静态和动态基址数据对象的段时,基地址是很有用的。在 32 位编辑环境中,基地址的唯一可接受形式是“基于一个指针”,该指针定义了一个对于一个 32 位的基址所含的 32 位的偏址;或者基于 void 型。

语法

基范围修饰符：

__based (基表达式)

基表达式：

基变量

基抽象说明符
段名称
段造型
基变量:
标识符
基抽象说明符:
抽象说明符
基类型:
类型名称

基指针

在 32 位的编辑环境中,__based 关键字的唯一有效形式是基于指针地址的指针。在这种编辑环境中,基指针是一个对 32 位基地址的 32 位的偏移量。
当对一个基地址指针进行间接引用时,基地址要么是显式地说明了的,要么是通过说明已经隐含知道了的。

基于指针的指针

基地址的“基于指针”的变量使得可以把一个指针说明为“基表达式”。这个基指针就成为了一个段的偏移量,此段开始于这个基指针所基于的地址的开始。
基于指针的指针的一个用途是为了保留含有指针的对象。一个基于指针的指针的链表可以保存到磁盘上并能重新载入到存储器的其他地方,并且指针依然是有效的。下面的例子说明了一个这样的链表:

```
void *vpBuffer;

struct llist_t
{
  void __based(vpBuffer)  *vpData;
  llist_t  __based(vpBuffer)  *llNext;
};
```

指针 vpBuffer 赋值为在程序后面分配的存储器地址;然后该链表可以相对于 vpBuffer 的值重新装载。

在 32 位的编辑环境中,基于指针地址的指针是__base 有效的唯一形式。在这种编辑环境中,它们是 32 位基址的 32 位的位置。

调用和命名的常规修饰符

调用规则决定了函数是如何调用的;命名规则决定了如何对待外部名。有关详情参见联机“Microsoft Visual C++ 6.0 程序员指南”中的“调用规定主题”。

扩展存储类属性

这一节描述扩展属性语法,它精简并规范化了对 Miscrosoft C/C++的扩展。使用扩展属性语法的存储类属性包括:thread、naked、dllimport 及 dllexport。

本节后面将描述如何使用这些属性。

扩展属性语法

扩展属性语法是用__declspec 关键字说明存储类信息,它说明了一个给定类型的对象的实例是按 Microsoft 特殊的存储类属性(thread、naked、dllimport、dllexport、nothrow、 property、selectany 或 uuid)进行存储。其他的存储类修饰符的例子包括:static 和 extern 关键字。然而,这些关键字是 ANSI 规范的 C 和 C++语言的一部分,故而扩展属性语法不涉及它们。
下面是 C++的扩展属性语法:
语法
说明指示符:
__declspec(扩展的说明修饰符序列)
扩展的说明修饰符序列:
扩展的说明修饰符 opt
扩展的说明修饰符 扩展的说明修饰符序列
扩展的说明修饰符:
thread
naked
dllimport
dllexport
nothrow
property

```
    selectany
    uuid("ComObjectGUID")
```

空白符用于分隔说明修饰符序列。语法的例子将在后面几节上给出。

存储类属性 thread、naked、dllimport、dllexport、nothrow、property、selectany 和 uuid 仅仅只是它们所作用的对象说明和函数说明的属性。不象 __near 和 __far 关键字,它们实际上影响对象或函数的类型(在这种情况下,2 字节和 4 字节的地址),这些存储类属性并不重新定义对象自身的类型属性。thread 属性仅影响数据和对象。naked 属性仅影响函数。dllimport 属性和 dllexport 属性仅影响函数、数据和对象。property、selectany 和 uuid 属性仅影响 COM 对象。

thread 属性

线程局部存储(TLS)是一种机制,通过这种机制,在一个多线程的进程中每个线程可以为线程特有的数据分配存储。在标准多线程的程序中,数据是在给定进程的所有线程中共享的,然而线程局部存储是一种可以为每个线程分配数据的机制。有关线程的完整讨论参见联机“Visual C++ 程序员指南”中的“多线程主题”。

C/C++语言包括扩展存储类属性 thread。thread 属性必须同 __declspec 关键字一起使用来说明一个线程变量。例如,下面的代码说明为一个整型线程局部变量并把它用一个值进行初始化:

```
__declspec(thread)  int tls_i = 1;
```

在说明一个线程局部对象和变量时,必须遵循如下的一些准则:

- thread 属性只能适用于数据的说明和定义以及不含成员函数的类。

它还能用于函数的说明和定义。例如,下面的代码会产生一个编译错误:

```
#define Thread __declspec(thread)
Thread void func();   // 错误
```

- 你只能对具有静态存储期的数据项说明 thread 属性。这包括全局数据对象(包括 static 和 extern)、局部静态对象和类的静态数据成员。你不能说明一个自动类型数据对象具有 thread 属性。例如,下面的代码会引起编译错误:

```
#define Thread __declspec(thread)
void func1()
{
  Thread int tls_i;          // 错误
}

int func2(Thread int tls_i)      // 错误
{
  return tls_i;
}
```

- 无论说明和定义是出现在同一文件中还是不同的文件中,对于线程局部对象的说明和定义要都使用 thread 属性。如下例子,下述代码会产生错误:

```
#define Thread __declspec(thread)
```

```
extern int tls_i;  // 这将产生错误,
int Thread tls_i;  // 因为说明与定义不同
```

- 你不能把 thread 属性作为类型修饰符使用。例如,下述代码会产生编译错误:

```
char __declspec(thread) *ch;// 错误
```

- 如果类不含有成员函数,则类可以用 thread 属性进行实例化。如果在类的说明部分没有说明对象则 thread 属性将被忽略。例如:

```
__declspec(thread) class X {
public:
    int I; } x;  //x 是个线程对象
X y;   //y 不是一个线程对象
```

因为允许说明具有 thread 属性的对象,下面的两个例子在意义上完全一样:

```
#define Thread __declspec(thread)
Thread class B
{
  // 代码
} BObject;// 可以:BObject 说明为线程局部对象
class B
{
 // 代码
}
Thread B BObject; // 可以:BObject 说明为线程局部对象
```

- 标准的 C 允许用一个包含对自身引用的表达式对一个对象或变更进行初始化,但仅能对非静态领域的对象进行初始化。尽管 C++通常允许有一个包含有对自身引用的表达式对外对象进行动态初始化,但这种类型的初始化不能用于对线程局部对象进行初始化。例如:

```
#define Thread__declspec(thread)
Thread int tls_i = tls_i;    // 在 C 和 C++ 中均是错误的
int j = j;  // 在 C++是正确的但在 C 中是错误的
Thread int tls_i = sizeof(tls_i)// 在 C 和 C++中均是正确的
```

注意 sizeof 表达式中包含了要被初始化的对象,但并不构成一个对自身的引用,因而在 C 和 C++中均是允许的。

naked 属性

对于用 naked 属性定义的函数,编译器产生的代码不会包括前言和结尾部分代码。利用这种特性你可以用内嵌汇编代码写出自己的前言和结尾部分代码序列。带 naked 属性的函数在写虚拟设备驱动程序时特别有用。
因为 naked 属性仅仅同函数的定义有关并且又不是一个类型修饰符,故如前面所述,带 naked 属性的函数使用扩展属性语法。例如如下代码定义了一个带 naked 属性的函数:

```
__declspec(naked) int func(formal_parameters)
{
    // 函数体
}
```

或者可以写成：

```
#define Naked __declspec(naked)
Naked int func(formal_parameters)
{
    // 函数体
}
```

naked 属性只影响编译器为函数的前言和结尾部分序列产生代码的性质。它并不会影响到调用此函数所生成的代码。因此，naked 属性还能视为函数类型的一部分，并且函数指针也不能具有 naked 属性。而且 naked 属性也不能用于数据的定义。例如，下面的示例代码会产生错误：

```
__declspec(naked) int i;   // 错误：naked 属性不能用于数据的定义
```

naked 属性仅只同函数的定义有关并不能在函数的原型中说明。例如：下面的说明会产生编译错误：

```
__declspec(naked) int func();  // 错误：naked 属性不能用于函数的说明
```

规则与限制

- 一个带 naked 属性的函数中不允许有返回语句。但是，可以通过在 RET 指令之前把一个值放入到 EAX 寄存器来返回一个整型。
- 结构异常处理结构在带 naked 属性的函数中是不允许的，因为这一结构必须交叉展开堆栈框架。
- setjmp 进行时间函数也还能在带 naked 属性的函数中使用，因为它必须是交叉展开堆栈框架。但是使用 longjmp 运行时间函数是允许

的。

- 在带 naked 属性的函数中不允许使用 _alloca 函数
- 要保证在前言序列之前不能有对局部变量的初始化代码,在函数范围中初始化局部变量是不允许的。特别的,在函数范围中说明 C++对象是不允许的。然而,在一个嵌套的范围是可以初始化对象的。
- 不推荐对框架指针进行优化(/Oy 编译器选项),对于带 naked 属性的函数该指示是自动关闭的。

写前言/结尾部分代码考虑的事情

在写你自己的前言和结尾部分代码序列之前,理解堆栈框架如何布置是很重要。理解符号 __LOCAL_SIZE 的使用也很重要。

C++堆栈框架的布置

这一例子给出了可能出现在 32 位函数中的标准前言代码:

```
push   ebp;                  存储 ebp
mov    ebp, esp              ; 设置堆栈框架指针
sub    esp, localbytes; 为 locals 分配空间
push<registers>; 存储 registers
```

localbytes 变量代表了在堆栈上为局部变量所需的字节数,而<registers>变量是一个位置容器代表了一系列要保存在堆栈上的寄存器。在保存了寄存器以后,你可以在堆栈上放置任何合适的数据了。下面是相应的结尾部分代码:

```
pop    <registers> ; 恢复 registers
```

```
mov     esp, ebp        ; 恢复堆栈指针
pop     ebp             ; 恢复 ebp
ret                     ;  从函数返回
```

堆栈总是向下生长的(从存储器高地址向存储器低地址生长)。基指针(ebp)指向是由 ebp 压入的值。这一局部区开始于 ebp-2。要存储局部变量,只要通过从 ebp 中减去一个合适的值,从 ebp 中计算出正确的偏移量即可。

__LOCAL_SIZE

编译器提供了一个符号(__LOCAL_SIZE)用于在函数前言代码的内嵌汇编块中使用。

在自定义的前言代码中,这一符号是用于在堆栈框架上为局部变量分配空间。

编译器决定了 __LOCAL_SIZE 值的大小。它的值是所有用户自定义的局部变量和编译器生成的临时变量的整体字节大小。__LOCAL_SIZE 只能作为直接操作数来使用;它不能在表达式中使用。也不能够重定义或改变该符号的值。例如:

```
mov     eax, __LOCAL_SIZE               ; 直接操作数,正确
mov     eax, [ebp - __LOCAL_SIZE]       ; 错误
```

下面的例子上一个含有自定义的前言和结尾部分序列的带 naked 属性的函数,在前言序列中使用了 __LOCAL_SIZE 符号:

```
__declspec (naked) func()
{
    int i;
    int j;
```

```
    __asm  /* 前言代码 */
      {
      push     ebp
      move     ebp, esp
      sub      esp, __LOCAL_SIZE
     }

/* 函数体 */

__asm     /* 结尾部分代码 */
    {
    move   esp, ebp
    pop    ebp
    ret
    }
}
```

dllexport 和 dllimport 属性

存储类修饰符 dllexport 和 dllimport 是为了从一个 DLL 中输出及输入函数、数据和对象。这些修饰符或者属性,显式定义了 DLL 对用户端的接口,用户端可以是一个可执行文件或另一个 DLL。把函数说明为具有 dllexport 属性可以不

需要模块定义文件(.DEF),至少可以不需要输出函数的说明部分。注意 dllexport 代替了 __export 关键字。
说明 dllexport 和 dllimport 要用扩展属性语法。

```
__declspec(dllexport) void func();
```

为了增加你的代码的可读性,可以用下面的宏定义。

```
#define DllImport __declspec(dllimport)
#define DllExport __declspec(dllexport)

DllExport void func();
DllExport int i = 10;
DllImport int j;
DllExport int n;
```

定义和说明

DLL 的接口涉及到由系统中的其他程序显式输出的项(函数和数据);也即,所有说明为 dllimport 和 dllexport 的项。包括在 DLL 接口中的所有说明必须说明为要么具有 dllimport 属性,要么具有 dllexport 属性。然而,定义仅只能说明为具有 dllexport 属性。例如下面的函数定义会产生编译错误:

```
__declspec(dllimport) int func()  // 错误:在定义中不能用 dllimport 属性
{
return 1;
}
```

下面的代码也会产生错误：

```
#define DllImport__declspec(dllimport)

__declspec(dllimport) int i = 10;  // 错误:这是一个定义
```

然而下面的代码是正确的：

```
__declspec(dllexport) int i = 10;  // 正确:export 定义
```

使用 dllexport 隐含了一个定义,而使用 dllimport 隐含了一个说明。你必须把 extern 关键字同 dllexport 一起使用以强制一个说明;否则将隐含一个定义。因此,下面的代码是正确的：

```
#define DllImport__declspec(dllimport)
#define DllExport__declspec(dllexport)

extern DllImport int k;  // 这两个都是正确的
DllImport int j; // 并隐含了一个说明
```

下面的例子阐明了前面所解释的：

```
static __declspec(dllimport) int l; // 错误:没有说明为 extern 的

void func()
{
static __declspec(dllimport) int s;   // 错误:没有说明为 extern 的
__declspec(dllimport) int m;  // 正确:这是一个说明
__declspec(dllexport) int n;  // 错误:隐含了一个在局部范围
```

```
                                                    // 的外部定义
extern __declspec(dllimport) int i;    // 正确:这是一个说明
extern __declspec(dllexport) int k;    // 正确:extern 暗示了是一个说明
__declspec(dllexport) int x = 5;  // 错误:隐含了一个在局部范围
                                                      // 的外部定义
}
```

用 dllexport 和 dllimport 定义 C++函数

可以定义一个有 dllexport 属性的内嵌函数。在此情况下,无论是否有程序中的其他模块引用了该函数,函数总是被实例化的并且是被输出的。该函数假定是被其他程序输入的。

也可以用 dllimport 属性定义一个联编函数。在此情况下,函数是可扩充的(遵循编译指示/Ob),但从不被实例化的。特别是,如果取一个输入联编函数的地址,则将返回该函数驻留在 DLL 中的地址。这种行为就象取一个非联编的输入函数的地址一样。

这一规则适用于其定义出现在类定义中的联编函数。而且,在联编函数中的静态局部数据和字符串将在 DLL 和用户端中保持同样的标识,因为它们是在同一程序中的(即,一个没有 DLL 接口的可执行文件)。

在提供一个输入的联编函数时要非常小心。例如:如果你更新了 DLL,你不能假定用户端会使用改变后的 DLL 版本。要保证能够载入正确的 DLL 版本,得重新编译 DLL 客户程序。

一般规则及限制

- 如果你没有用 dllexport 和 dllimport 属性说明一个函数和对象，该函数和对象就不会作为 DLL 接口的一部分。因此，必须在此模块中或者同一程序的其他模块中提供对该函数和对象的定义。要使函数和对象成为 DLL 接口的一部分，你必须在其他模块中用 dllexport 说明对函数及对象的定义。否则会产生一个链接器错误。

如果你用 dllexport 和 dllimport 属性说明一个函数和对象，则它们的定义必须出现在同一程序的某个模块中。否则会产生一个链接器错误。

- 如果在你的程序的单一模块中，对同一函数或对象同时包含了 dllexport 和 dllimport 属性的说明，则 dllexport 属性将取代 dllimport 属性处于优先权地位。但编译器会给出一个警告错误：例如：

```
__declspec(dllimport) int i;
__declspec(dllexport) int i;    // 警告:不一致;
                                // dllexport 处于优先地位
```

- 在 C++中，你可以用一个带有 dllimport 属性的数据对象的地址去初始化一个全局说明的或静态局部的数据指针。然而在 C 中，这种初始化会产生错误。而且在 C++中，也可以用一个带有 dllimport 属性的函数的地址去初始化一个静态局部函数指针。在 C 中，这种赋值会把指针指向 DLL 输入 thunk 区(thunk，是一种用于传递控制给函数的残留代码)的地址，而并未把函数的地址赋给指针。在 C++中，则会使指针指向函数的地址。例如：

```
__declspec(dllimport) void func1(void);
__declspec(dllimport) int i;

int *pi = &i;                          // 在 C 中错误
static void (*pf)(void) = &func1;   // 在 C 中是 thunk 区的地址,在 C++
                                       // 中是函数的地址
void func2()
{
        static int *pi = &i;    // 在 C 中错误
        static void (*pf)(void) = &func1; // 在 C 中是 thunk 区的地址,
                                          // 在 C++中是函数的地址
}
```

然而,因为一个程序如果包含了有 dllexport 属性说明的对象,它必须在程序的某个地方为该对象提供定义。因此,你可以用一个带有 dllexport 属性的函数地址去初始化一个全局的或者局部静态的函数指针。同样,也可以用一个带有 dllexport 属性的数据对象地址去初始化一个全局的或者局部静态的数据指针。例如:下面的代码在 C 或 C++中不会产生错误:

```
__declspec(dllexport) void func1(void);
__declspec(dllexport) int i;

int *pi = &i;                          // 可以
static void (*pf)(void) = &func1;   // 可以
```

```
void func2()
{
static int *pi = &i;                    // 可以
static void (*pf)(void) = &func1;   // 可以
}
```

在 C++中使用 dllimport 和 dllexport

可用 dllimport 和 dllexport 属性说明 C++的类。这一形式隐含了整个类是输入的或者是输出的。以这种方法输出的类称为可输出类。
下面的例子定义了一个可输出类。它的所有成员函数和静态数据是输出的：

```
#define DllExport__declspec(dllexport)

class DllExport C
{
int i;
virtual int func(void)
{ return 1; }
};
```

注意在一个可输出类的成员函数上禁止显式使用 dllimport 和 dllexport 属性。

输出类

在说明一个输出类时,它的所有成员函数和静态数据成员是输出的。必须在同一程序中提供所有这些成员的定义。否则,会产生一个链接错误。这一规则在应用于纯虚拟函数时有一个异常,对于纯虚拟函数不必提供显式的定义。然而,因为抽象类的析构函数总是由基类的析构函数调用的,所以纯虚拟析构函数总是必须要提供一个定义的。这一规则对于非可输出的类也是同样的。
如果你输出了类类型数据或者返回类的函数,一定要输出该类。

输入类

在你说明一个输入类时,它的所有成员函数和静态数据成员是输入的。同 dllimport 和 dllexport 属性作用在非类类型上的行为不同,静态数据成员不能在定义输入类的同一程序中说明一个定义。

继承和输出类

可输出类的所有基类必须是可输出的。如果不是这样,会产生一个编译警告。而且,所有可访问的类类型成员也必须是可输出的。这一 规则允许一个输出类继承一个输入类,而一个输入类继承一个输出类(尽管不推荐后者的用法)。作为一个规则,对于 DLL 的客户端可访问的每一个成员(按照 C++的访问规则)应该是可输出接口的一部分。这包括在联编函数中引用的私有数据成员。

有选择地输入/输出成员

因为在一个类中的成员函数和静态数据隐含地具有外部连接,除非整个类被输出

了。你可以用 dllimport 和 dllexport 属性说明它们。如果类是被输入的或输出的,则禁止显式说明成员函数和数据为 dllimport 和 dllexport 属性。如果你在类定义中说明一个静态数据成员具有 dllexport 属性,必须在同一程序的某个地方提供定义(因为具有非类的外部连接)。

同样地,你可以说明具有 dllimport 和 dllexport 属性的成员函数。在这种情形下,你必须在同一程序的某个地方提供一个 dllexport 定义。

关于有选择地输入/输出成员有以下一些关键点值得注意:

- 有选择地输入/输出成员能很好地用于提供一种更加严谨的输出类接口;也就是,用这种接口设计一个 DLL 能够比不用时暴露更少的公有及私有的属性。在微调输出接口中,这也是很有用的:通过定义,在你了解了客户端不能访问某些私有数据时,不必输出整个类。
- 如果输出了类中的一个虚拟函数,必须输出所有的虚拟函数,否则至少要提供一个客户端能直接调用的版本。
- 如果定义了一个类,其中对于虚拟函数使用了有选择的输入/输出成员,则这些函数必须在输出接口中被定义为联编的(对客户端可见)。
- 如果你把一个成员定义为 dllexport 属性的,但并没有在类定义中包含它,则会产生一个编译器错误。你必须在类头中定义该成员。
- 尽管用 dllimport 和 dllexport 属性定义类成员是允许的,但你不能重载在类定义中说明的接口。
- 如果你在说明类定义的定义体以外的地方定义了一个函数,并把它说明为有 dllimport 和 dllexport 属性的(如果这一定义同类说明

中所说明的不同)就会产生一个警告。

联编汇编器

联编汇编器使你能够在 C 语言源代码中直接包含汇编语言指令而不需要额外的汇编及链接步骤。联编汇编器是内置于编译器中的,所以不需要一个单独的汇编器如 Microsoft 的宏汇编(MASM)。
因为联编汇编器不需要单独的汇编及链接步骤,它比单独的汇编器更方便。联编汇编器代码能够使用处在同一范围中的任何 C 的变量名称和函数名称,因此它很容易同你编写的 C 代码集成在一起。并且正是因为汇编代码能和 C 语句混合使用,它可以用来解决一些很麻烦的任务以及单独用 C 不可解决的任务。
__asm 关键字将激活联编汇编器,并能出现在 C 语句可合法出现的任何地方。该关键字之后必须跟有汇编指令,或由大括号括起的一组指令,或者,最少应有一对空的大括号;而不能仅仅只出现这一关键字。这里,“__asm 块”是指任何指令或一组指令,无论是否在大括号中。
下面的代码是一个简单的括在大括号中的__asm 块(这一代码是一个函数的前言序列)。

```
__asm
{
    push ebp
    mov  ebp, esp
    sub  esp, __LOCAL_SIZE
```

```
}
```
当然,也可以选择把__asm放在每一条汇编指令之前:
```
__asm push ebp
__asm mov  ebp, esp
__asm sub  esp, __LOCAL_SIZE
```
因为__asm关键字是一个语句的分隔符,也可以在同一行中放置多条汇编指令:
```
__asm push ebp   __asm mov  ebp, esp   __asm sub  esp, __LOCAL_SIZE
```

附录 C　编译器 COM 支持类

Microsoft 特殊处→
一些标准类用于支持某些 COM 类型。这些类在 COMDEF.H 和从类型库中生成的头文件中定义。
#include <comdef.h>

类	目的
_com_error	定义在大多数失败中由_com_raise_error 丢弃的错误对象
_com_ptr_t	封装 COM 接口指针，并使对 AddRef、Release、QueryInterface 的调用自动化
_bstr_t	封装 BSTR 类型以提供有用的操作和方法
_variant_t	封装 VARIANT 类型以提供有用的操作和方法

而且，有一个支持例程叫作_com_raise_error，它用于所有由编译器生成的 COM 支持代码以丢弃一个_com_error 来响应失败。
_com_raise_error 在 comdef.h 中定义如下：

```
void __stdcall _com_raise_error(HRESULT hr,IErrorInfo* perrinfo=0)
throw(_com_error);
```

实际的源代码是：

```
void __stdcall _com_raise_error(HRESULT hr,IErrorInfo* perrinfo)
```

```
throw(_com_error)
{
   throw  _com_error(hr, perrinfo);
 }
```

_com_raise_error 可以由用户提供的具有同样名称及原型的函数版本所替代。如果你要用#import,但又不想用 C++的异常处理,就可以这样做。在这种情况下,用户自定义的_com_raise_error 版本可以决定是作一个 longjmp 还是弹出一个消息框或者停机。这一用户版本不应返回,尽管因为编译器 COM 支持代码并不希望它返回。

Microsoft 特殊处结束

_com_error

Microsoft 特殊处→

一个_com_error 对象代表了一个异常条件,这一异常条件能由从类型库生成的头文件中的错误处理封装函数检测到或者能由 COM 支持类检测到。类_com_error 封装了 HRESULT 错误代码和相关联的 IErrorInfo 对象。

```
#include <comdef.h>
```

构造函数	
_com_error	构造一个_com_error 对象

运算符	
operator =\把一个已存在的_com_error 对象赋值给另外一个	
Extractor 函数	
Error	检索传递给构造函数的 HRESULT 值
ErrorInfo	检索传递给构造函数的 IErrorInfo 对象
WCode	检索映射到封装的 HRESULT 上的 16 位错误代码
IErrorInfo 函数	
Description	调用 IErrorInfo::GetDescription 函数
HelpContext	调用 IErrorInfo::GetHelpContext 函数
HelpFile	调用 IErrorInfo::GetHelpFile 函数
Source	调用 IErrorInfo::GetSource 函数
GUID	调用 IErrorInfo::GetGUID 函数
格式消息 Extractor 函数	
ErrorMessage	为存储在_com_error 对象中的 HRESULT 检索字符串消息
ExepInfo.wCode to HRESULT Mappers	
HRESULTToWCode	把 32 位的 HRESULT 映射为 16 位的 wCode
WCodeToHRESULT	把 16 位的 wCode 映射为 32 位的 HRESULT

Microsoft 特殊处结束

成员函数

_com_error::_com_error

Microsoft 特殊处→

```
_com_error(HRESULT hr, IErrorInfo* perrinfo = NULL) throw();
_com_error(const _com_error& that) throw();
```

参数

hr

HRESULT 信息

perrinfo

IErrorInfo 对象

that

一个存在的 _com_error 对象

注释

构造一个 _com_error 对象。第一种构造函数用一个 HRESULT 以及一个可选的 IErrorInfo 对象来构造一个 _com_error 对象。第二种构造函数是创建一个已有 _com_error 对象的拷贝。

Microsoft 特殊处结束

_com_error::Description

Microsoft 特殊处→

```
_bstr_t Description() const throw ();
```

返回值
对于记录在 _com_error 对象中的 IErrorInfo 对象，返回 IErrorInfo::GetDescription 的结果。结果 BSTR 是封装在一个 _bstr_t 对象中的。如果没有记录 IErrorInfo,将返回一个空的 _bstr_t 对象。
注释
调用函数 IErrorInfo::GetDescription 并检索记录在 _com_error 对象中的 IErrorInfo。在调用 IErrorInfo::GetDescription 时的任何失败都将忽略。
Microsoft 特殊处结束

_com_error::Error

Microsoft 特殊处→
```
HRESULT Error() const throw();
```
返回值
传递给构造函数的原始 HRESULT 项。
注释
检索封装在 _com_error 对象中的 HRESULT 项。
Microsoft 特殊处结束

_com_error::ErrorInfo

Microsoft 特殊处→
```
IErrorInfo * ErrorInfo() const throw();
```
返回值
传递给构造函数的原始 IErrorInfo 项。

注释
检索封装在_com_error 对象中的 IErrorInfo 项,如果没有记录 IErrorInfo 项则返回 NULL。当调用者完成了对返回对象的使用后,必须对其调用 Release。
Microsoft 特殊处结束

_com_error::ErrorMessage

Microsoft 特殊处→
```
const TCHAR * ErrorMessage() const throw();
```
返回值
为记录在_com_error 对象中的 IErrorInfo 返回字符串消息。如果 HRESULT 是一个映射为 16 位的 wCode,则返回一条普通的"IDispatch error #<wCode>"消息。如果没有发现消息,则返回一条普通的"Unknown error #<hresult>"消息。返回的字符串可以是单代码或者多字节字符串,这一点有赖于_UNICODE 宏的状态。
注释
为记录在_com_error 对象中的 IErrorInfo 返回合适的系统消息文本。此系统消息文本是通过调用 Win32 函数 FormatMessage 获得的。返回的字符串是由 FormatMessage API 分配的,并在_com_error 对象消毁后将被释放。
Microsoft 特殊处结束

_com_error::GUID

Microsoft 特殊处→
```
GUID GUID() const throw();
```

返回值
为记录在_com_error 对象中的 IErrorInfo 返回 IErrorInfo::GetGUID 的结果。如果没有记录 IErrorInfo,将返回 GUID_NULL。
注释
调用 IErrorInfo::GetGUID 方法。在调用 IErrorInfo::GetGUID 方法时的任何失败都将被忽略。
Microsoft 特殊处结束

_com_error::HelpContext

Microsoft 特殊处→
DWORD HelpContext() const throw();
返回值
为记录在_com_error 对象中的 IErrorInfo 对象返回 IErrorInfo::GetHelpContext 的结果。如果没有记录 IErrorInfo 对象,将返回 0 值。
注释
调用 IErrorInfo::GetHelpContext 接口方法。调用 IErrorInfo::GetHelpContext 方法时的任何失败都将被忽略。
Microsoft 特殊处结束

_com_error::HelpFile

Microsoft 特殊处→
_bstr_t HelpFile() const throw();
返回值

为记录在_com_error 对象中的 IErrorInfo 对象返回 IErrorInfo::GetHelpFile 的结果。结果 BSTR 是封装在_bstr_t 对象中的。如果没有记录 IErrorInfo,将返回一个空的_bstr_t。

注释

调用 IErrorInfo::GetHelpFile 接口方法。调用 IErrorInfo::GetHelpFile 方法时的任何失败都将被忽略。

Microsoft 特殊处结束

_com_error::HRESULTToWCode

Microsoft 特殊处→

static WORD HRESULTToWCode(HRESULT *hr*) throw();

返回值

从 32 位的 HRESULT 映射的 16 位 wCode 值。

参数

hr

要映射为 16 位 wCode 值的 32 位的 HRESULT。

注释

完成一个 32 位 HRESULT 到 16 位 wCode 值的映射。详细的信息参见_com_error::WCode。

参见

_com_error::WCode, _com_error::WCodeToHRESULT, _com_error 概述

Microsoft 特殊处结束

_com_error::Source

Microsoft 特殊处→
_bstr_t Source() const throw();
返回值
为记录在_com_error 对象中的 IErrorInfo 对象返回 IErrorInfo::GetSource 的结果。
结果 BSTR 是封装在_bstr_t 对象中的。如果没有记录 IErrorInfo,将返回一个空的_bstr_t 。
注释
调用 IErrorInfo::GetSource 接口方法。调用 IErrorInfo::GetSource 方法时的任何失败都将被忽略。
Microsoft 特殊处结束

_com_error::WCode

Microsoft 特殊处→
WORD WCode () const throw();
返回值
如果 HRESULT 是在 0x80040200～0x8004FFFF 的范围内,则 WCode 方法将返回最小的 HRESULT 值 0x80040200;否则返回 0 值。
注释
WCode 方法检索的是一个由封装的 HRESULT 映射的 16 位错误码。
WCode 方法是用于解除发生在 COM 支持代码中的映射。dispinterface 属性的封

装器或者方法调用一个支持例程,该支持例程封装了参数并调用了 IDispatch::Invoke 函数。在返回时,如果返回了一个 DISP_E_EXCEPTION 的失败 HRESULT,则从 EXCEPINFO结构中检索错误信息,并传递给 IDispatch::Invoke 函数。错误码可以是存储在 EXCEPINFO 结构 wCode 成员中的 16 位值,或者是 EXCEPINFO 结构的 scode 成员中的完全的 32 位值。如果返回的是 16 位 wCode,则它必定先映射为了 32 位的失败 HRESULT。

参见

_com_error::HRESULTToWCode, _com_error::WCodeToHRESULT, _com_error 概述

Microsoft 特殊处结束

_com_error::WCodeToHRESULT

Microsoft 特殊处→

```
static HRESULT WCodeToHRESULT(WORD wCode) throw();
```

返回值

从一个 16 位的 wCode 映射为 32 位 HRESULT 值。

参数

wCode

是一个映射为 32 位 HRESULT 的 16 位 wCode 值 。

注释

完成一个 16 位的 wCode 到 32 位 HRESULT 的映射。参见 WCode 成员函数。

参见

_com_error::WCode, _com_error::HRESULTToWCode, _com_error 概述
Microsoft 特殊处结束

运算符

_com_error::operator =

Microsoft 特殊处→
```
_com_error& operator = (const _com_error& that) throw ();
```
参数
that
一个 _com_error 对象
注释
把一个已有的_com_error 对象赋值给另外一个。
Microsoft 特殊处结束

_com_ptr_t

Microsoft 特殊处→
一个_com_prt_t 对象封装了一个 COM 接口指针,被称为是“灵敏”指针。这一个模板类通过对 IUnknown 的成员函数 QueryInterface、AddRef 及 Release 的调用,来管理资源的分配和回收。
一个“灵敏”指针通常由_COM_SMARTPTR_TYPEDEF 宏提供的类型定义来引用。

此宏带有一个接口名称和一个 IID,并说明了一具带有接口名加上 Ptr 前缀的特殊的_com_ptr_t。例如:

```
_COM_SMARTPTR_TYPEDEF(IMyInterface, __uuidof(IMyInterface));
```

说明了_com_ptr_t 特例化的 IMyInterfacePtr。

一系列的函数模板(非此模板类的成员)支持在比较运算符的右手边出现“灵敏”指针的比较。

```
#include <comdef.h>
```

构造函数	
_com_ptr_t	构造一个_com_ptr_t 对象

低级操作	
AddRef	在封装的接口指针上调用 IUnknown 的成员函数 AddRef
Attach	封装这一灵敏指针类型的原始接口指针
CreateInstance	按所给的 CLSID 或 ProgID 创建一个新的对象实例
Detach	抽出并返回封装的接口指针
GetInterfacePtr	返回封装的接口指针
QueryInterface	在封装的接口指针上调用 IUnknown 的成员函数 QueryInterface
Release	在封装的接口指针上调用 IUnknown 的成员函数 Release

运算符	
operator =	把一个新值赋给一个已有的_com_ptr_t 对象
operators ==, !=, <,>,<=,>=	把一个“灵敏”指针同另外一个“灵敏”指针、原始接口指针、及 NULL 进行比较
Extractors	提取封装的 COM 接口指针

Microsoft 特殊处结束

成员函数

_com_ptr_t::_com_ptr_t

```
Microsoft 特殊处→
_com_ptr_t() throw();
_com_ptr_t(Interface* pInterface) throw();
_com_ptr_t(Interface* pInterface, bool fAddRef) throw();
_com_ptr_t(int NULL) throw(_com_error);
template< > _com_ptr_t(const _com_ptr_t& cp) throw();
template<typename _InterfacePtr> _com_ptr_t(const _InterfacePtr& p)
throw(_com_error);
template< > _com_ptr_t(const _variant_t& varSrc) throw(_com_error);
```

```
explicit _com_ptr_t(const CLSID& clsid, DWORD dwClsContext = CLSCTX_ALL)
throw( _com_error);
explicit _com_ptr_t(LPOLESTR lpOleStr, DWORD dwClsContext = CLSCTX_ALL)
throw( _com_error);
explicit _com_ptr_t(LPCSTR lpcStr, DWORD dwClsContext = CLSCTX_ALL)
throw(_com_error);
```

参数

pInterface

一个原始的接口指针

fAddRef

如果为真,则调用 AddRef 函数以增加对象封装的接口指针的引用计数

cp

一个 _com_ptr_t 对象

p

一个原始接口指针,其类型同这个 _com_ptr_t 对象的灵敏指针类型不同

varSrc

一个 _variant_t 型对象

clsid

联合类的 CLSID

dwClsContext

正在运行的代码的上下文

lpOleStr

一个含有 CLSID(以“{”开始)或者 ProgID 的单代码字符串

lpcStr

一个含有 CLSID(以“{”开始)或者 ProgID 的多字节字符串

注释

构造一个 _com_ptr_t 对象。

- _com_ptr_t()构造一个 NULL 的灵敏指针。
- _com_ptr_t(*pInterface*)从这种灵敏指针类型的原始接口指针中构造一个灵敏指针。调用 AddRef 函数以增加对象封装的接口指针的引用计数。
- _com_ptr_t(*pInterface*, *fAddRef*)从这种灵敏指针类型的原始接口指针中构造一个灵敏指针。如果 fAddRef 是真,则调用 AddRef 函数以增加对象封装的接口指针的引用计数。如果 fAddRef 是假,则此构造函数对这一原始接口指针具有所有权,并也不调用 AddRef 函数。
- _com_ptr_t(NULL) 构造一个 NULL 的灵敏指针。该 NULL 参数必须是 0。
- _com_ptr_t(*cp*) 以其它的同类型的灵敏指针的拷贝来构造一个灵敏指针。调用 AddRef 函数以增加对象封装的接口指针的引用计数。
- _com_ptr_t(*p*) 从一个不同的灵敏指针类型或者一个不同的原始指针来构造一个灵敏指针。调用 QueryInterface 函数以找出此灵敏指针类型的接口指针。如果函数 QueryInterface 以带一个 E_NOINTERFACE 错误的失败而告终,则构造一个 NULL 灵敏指针。其它任何错误将会引起产生一个 _com_error。

- _com_ptr_t(*varSrc*) 从一个 _variant_t 对象中构造一个灵敏指针。封装的 VARIANT 必须是 VT_DISPATCH 类型的或者是 VT_UNKNOWN 类型的,或者是一个能够转换为这两者之一的类型。如果函数 QueryInterface 以带一个 E_NOINTERFACE 错误的失败而告终,则构造一个 NULL 灵敏指针。其它任何错误将会引起产生一个 _com_error。
- _com_ptr_t(*clsid, dwClsContext*) 从一个联合类的 CLSID 中构造一个灵敏指针。这一函数调用 CoCreateInstance 通过成员函数 CreateInstance 创建一个新的 COM 对象然后为此灵敏指针进行查询。如果函数 QueryInterface 以带一个 E_NOINTERFACE 错误的失败而告终,则构造一个 NULL 灵敏指针。其它任何错误将会产生一个 _com_error。
- _com_ptr_t(*lpOleStr, dwClsContext*) 从一个含有 CLSID (以“{”开始)或者含有 ProgID 的 Unicode 型字符串中构造一个灵敏指针。这一函数调用 CoCreateInstance 通过成员函数 CreateInstance 创建一个新的 COM 对象然后对此灵敏指针进行查询。如果函数 QueryInterface 以带一个 E_NOINTERFACE 错误的失败而告终,则构造一个 NULL 灵敏指针。其它任何错误将会产生一个 _com_error。
- _com_ptr_t(*lpcStr, dwClsContext*) 从一个含有 CLSID (以“{”开始)或者含有 ProgID 的多字节型字符串中构造一个灵敏指针。这一函数调用 CoCreateInstance 通过成员函数 CreateInstance 创建一个新的 COM 对象然后对此灵敏指针进行查询。如果函数

QueryInterface 以带一个 E_NOINTERFACE 错误的失败而告终,则构造一个 NULL 灵敏指针。其它任何错误将会产生一个_com_error。

Microsoft 特殊处结束

_com_ptr_t::AddRef

Microsoft 特殊处→

```
void AddRef() throw(_com_error);
```

注释

在封装的接口指针上调用 IUnknown::AddRef 函数。如果指针是 NULL 则产生一个_POINTER 错误。

Microsoft 特殊处结束

_com_ptr_t::Attach

Microsoft 特殊处→

```
void Attach(Interface* pInterface) throw();
void Attach(Interface* pInterface, bool fAddRef) throw();
```

参数

pInterface

一个原始的接口指针

fAddRef

如果是真,则调用函数 AddRef;如果是假,则此_com_ptr_t 对象对原始接口指针具有 所有权,并也不调用函数 AddRef。

注释

封装一个此灵敏指针类型的原始接口指针。

- Attach(*pInterface*) 不调用 AddRef。接口的所有权传送给了此 _com_ptr_t 对象。调用 Release 为以前封装的指针减少引用次数。
- Attach(*pInterface*, *fAddRef*) 如果 *fAddRef* 是真,则调用 AddRef 函数以增加对象封装的接口指针的引用计数。如果 *fAddRef* 是假,则此 _com_ptr_t 对象对这一原始接口指针具有所有权,并也不调用 AddRef 函数。调用 Release 为以前封装的指针减少引用次数。

Microsoft 特殊处结束

_com_ptr_t::CreateInstance

Microsoft 特殊处→

```
HRESULT CreateInstance(const CLSID& rclsid , IUnknown * pOuter=NULL,
DWORD dwClsContext = CLSCTX_ALL) throw();
HRESULT CreateInstance(LPOLESTR clsidString,  IUnknown * pOuter=NULL,
DWORD dwClsContext = CLSCTX_ALL) throw();
HRESULT CreateInstance(LPCSTR clsidStringA, IUnknown * pOuter=NULL,
DWORD dwClsContext = CLSCTX_ALL) throw();
```

参数

rclsid

一个 CLSID 对象。

clsidString

一个含有 CLSID(以"{"开始)或者含有 ProgID 的单代码字符串

clsidStringA
一个含有 CLSID(以"{"开始)或者含有 ProgID 的多字节字符串
dwClsContext
正在执行的代码的上下文
pOuter
对于集合体的外部未知对象
注释
按给定的 CLSID 或 ProgID 创建一个对象的新的可运行的实例。这一函数调用 CoCreateInstance 通过成员函数 CreateInstance 创建一个新的 COM 对象然后对此灵敏指针进行查询。这一结果指针立即封装在_com_ptr_t 对象之中。调用 Release 为以前封装的指针减少引用次数。这一例程返回 HRESULT 以表示是成功还是失败。

- CreateInstance(*rclsid*, *dwClsContext*)按给定的 CLSID 创建一个对象的新的可运行的实例。
- CreateInstance(*clsidString*, *dwClsContext*)从一个含有 CLSID(以“{”开始)或者含有 ProgID 的单代码字符串中创建一个对象的新的可运行的实例。
- CreateInstance(*clsidStringA*, *dwClsContext*)从一个含有 CLSID(以“{”开始)或者含有 ProgID 的多字节字符串中创建一个对象的新的可运行的实例。调用 MultiByteToWideChar,它假设字符串是按 ANSI 代码页而不是 OEM 代码页。

Microsoft 特殊处结束

_com_ptr_t::Detach

Microsoft 特殊处→

```
Interface* Detach() throw();
```

注释

提取并返回封装的接口指针,然后把封装的指针存储清除为 NULL。这将从封装中清除接口指针。程序员有责任对返回的接口指针调用 Release 函数。

Microsoft 特殊处结束

_com_ptr_t::GetAcitveObject

Microsoft 特殊处→

```
HRESULT GetActiveObject(const CLSID& rclsid) throw();
HRESULT GetActiveObject(LPOLESTRE clsidString) throw();
HRESULT GetActiveObject(LPCSTR clsidStringA) throw();
```

参数

rclsid

CLSID 对象。

clsidString

一个含有 CLSID (以“{”开始)或者 ProgID 的单代码字符串。

clsidStringA

一个含有 CLSID(以“{”开始)或者 ProgID 的多字节字符串。

注释

按给定的 CLSID 或 ProgID 同一个对象的已存在的实例进行连接。这些成员函数

调用 GetActiveObject 以检索一个按 OLE 进行注册的运行对象的指针,然后为此接口灵敏指针的接口类型进行查询。返回的这一结果指针立即封装在_com_ptr_t 对象之中。调用 Release 为以前封装的指针减少引用次数。这一例程返回 HRESULT 以表示是成功还是失败。

- GetActiveObject(*rclsid*)按给定的 CLSID 同一个对象的已存在的实例进行连接。
- GetAcitveObject(*clsidString*)按一个含有 CLSID(以“{”开始)或者含有 ProgID 的单代码字符串同一个对象的已存在的实例进行连接。
- GetActiveObject(*clsidStringA*)按一个含有 CLSID(以“{”开始)或者含有 ProgID 的多字节字符串同一个对象的已存在的实例进行连接。调用 MultiByteToWideChar,它假设字符串是按 ANSI 代码页而还是 OEM 代码页。

Microsoft 特殊处结束

_com_ptr_t::GetInterfacePtr

Microsoft 特殊处→

```
Interface* GetInterfacePtr() const throw();
```

注释

返回封装的接口指针,其值可以是 NULL。

Microsoft 特殊处结束

_com_ptr_t::QueryInterface

Microsoft 特殊处→

```
template<typename _InterfaceType> HRESULT QueryInterface
(const IID& iid, _InterfaceType*& p) throw ();
template<typename _InterfaceType> HRESULT QueryInterface
 (const IID& iid, _InterfaceType** p) throw();
```

参数

iid

一个接口指针的 IID

p

一个原始接口指针

注释

在封装的接口指针上用说明的 IID 调用 IUnknown::QueryInterface,并在 p 中返回查询的原始接口指针。这一例程返回 HRESULT 以表示是成功还是失败。

Microsoft 特殊处结束

_com_ptr_t::Release

Microsoft 特殊处→

```
void Release() throw(_com_error);
```

注释

在封装的接口指针上调用 IUnknown::Release 函数。如果指针是 NULL 则产生一个 _POINTER 错误。

Microsoft 特殊处结束

运算符

_com_ptr_t::operator =

Microsoft 特殊处→

```
_com_ptr_t& operator=(Interface* pInterface) throw();
_com_ptr_t& operator=(int NULL) throw(_com_error);
template<> _com_ptr_t& operator=(const _com_ptr_t& cp) throw();
template<> _com_ptr_t& operator=(const _variant_t& varSrc)
throw(_com_error);
template<typename _InterfacePtr> _com_ptr_t& operator=(const
_InterfacePtr& p)
 throw(_com_error);
```

注释

把一个接口指针赋值给一个_com_ptr_t 对象：

- operator=(*pInterface*) 封装一个同此灵敏指针类型相同的原始接口指针。调用 AddRef 函数以增加对象封装的接口指针的引用计数，调用 Release 为以前封装的指针减少引用次数。
- operator=(NULL) 把一个灵敏指针设为 NULL。此 NULL 参数必须是 0 。
- operator=(*cp*) 把一个灵敏指针设置为同此灵敏指针有相同类型的另一实例的拷贝。调用 AddRef 函数以增加对象封装的接口指针的引用计数，并调用 Release 为以前封装的指针减少引用次数。

- operator=(*varSrc*) 用一个_variant_t 对象设置一个灵敏指针。封装的 VARIANT 必须是 VT_DISPATCH 类型或者是 VT_UNKNOWN 类型,或者是可以转换为上述两者之一的类型。如果函数 QueryInterface 以带一个 E_NOINTERFACE 错误的失败而告终,则构造一个 NULL 灵敏指针。其它任何错误将会产生一个_com_error 。
- operator=(*p*) 把一个灵敏指针设置为一个有不同类型的灵敏指针或者一个不同的原始指针。调用 QueryInterface 函数以找出此灵敏指针类型的接口指针。如果函数 QueryInterface 以带一个 E_NOINTERFACE 错误的失败而告终,则构造一个 NULL 灵敏指针。其它任何错误将会产生一个_com_error 。

Microsoft 特殊处结束

_com_ptr_t 关系运算符

Microsoft 特殊处→

```
template<typename _InterfacePtr> bool operator==(_InterfacePtr p)
    throw(_com_error);
template<> bool operator==(Interface* p) throw(_com_error);
template<> bool operator==(_com_ptr_t& p) throw();
template<> bool operator==(int NULL) throw(_com_error);
template<typename _InterfacePtr> bool operator!=(_InterfacePtr p)
    throw(_com_error);
template<> bool operator!=(Interface* p) throw(_com_error);
```

```
template<> bool operator!=(_com_ptr_t& p) throw(_com_error);
template<> bool operator!=(int NULL) throw(_com_error);
template<typename _InterfacePtr> bool operator<(_InterfacePtr p)
    throw(_com_error);
template<typename _InterfacePtr> bool operator>(_InterfacePtr p)
    throw(_com_error);
template<typename _InterfacePtr> bool operator<=(_InterfacePtr p)
    throw(_com_error);
template<typename _InterfacePtr> bool operator>=(_InterfacePtr p)
    throw(_com_error)
```

注释

把一个灵敏指针同另外一个灵敏指针、原始接口指针或者 NULL 进行比较。除了对 NULL 指针的测试,这些操作首先由 IUnknown 查询指针,然后比较结果。

Microsoft 特殊处结束

_com_ptr_t Extractors

Microsoft 特殊处→

```
operator Interface*() const throw();
operator Interface&() const throw(_com_error);
Interface& operator*() const throw(_com_error);
Interface* operator->() const throw(_com_error);
Interface** operator&() throw();
```

```
operator bool() const throw();
```

注释

- operator Interface* 返回封装的接口指针,其值可能是 NULL。
- operator Interface& 返回对封装的接口指针的引用,如果指针是 NULL 就发出一个错误。
- operator* 在指针间接引用中,让一个灵敏指针对象行为就象一个实际封装的接口一样。
- operator-> 在指针间接引用中,让一个灵敏指针对象行为就象一个实际封装的接口一样。
- operator& 释放任何封装的接口指针,用 NULL 代替它,并返回封装指针的地址。这使得一个灵敏指针可以作为地址传递给函数,该函数有一个外部参数,通过它此函数能返回一个接口指针。
- operator bool 使得灵敏指针能够用于条件表达式。如果指针不是 NULL,这一操作返回真。

Microsoft 特殊处结束

关系函数模板

Microsoft 特殊处→

```
template<typename _InterfaceType> bool operator==(int NULL,
    _com_ptr_t<_InterfaceType>& p) throw(_com_error);
template<typename _Interface, typename _InterfacePtr> bool operator==
    (_Interface* i, _com_ptr_t<_InterfacePtr>& p) throw(_com_error);
```

```
template<typename _Interface> bool operator!=(int NULL,
    _com_ptr_t<_Interface>& p) throw(_com_error);
template<typename _Interface, typename _InterfacePtr> bool operator!=
    (_Interface* i, _com_ptr_t<_InterfacePtr>& p) throw(_com_error);
template<typename _Interface> bool operator<(int NULL,
    _com_ptr_t<_Interface>& p) throw(_com_error);
template<typename _Interface, typename _InterfacePtr> bool operator<
    (_Interface* i, _com_ptr_t<_InterfacePtr>& p) throw(_com_error);
template<typename _Interface> bool operator>(int NULL, _com_ptr_t<
    _Interface>& p) throw(_com_error);
template<typename _Interface, typename _InterfacePtr> bool operator>
    (_Interface* i, _com_ptr_t<_InterfacePtr>& p) throw(_com_error);
template<typename _Interface> bool operator<=(int NULL, _com_ptr_t<
    _Interface>& p) throw(_com_error);
template<typename _Interface, typename _InterfacePtr> bool operator<=
    (_Interface* i, _com_ptr_t<_InterfacePtr>& p) throw(_com_error);
template<typename _Interface> bool operator>=(int NULL, _com_ptr_t<
    _Interface>& p) throw(_com_error);
template<typename _Interface, typename _InterfacePtr> bool operator>=
    (_Interface* i, _com_ptr_t<_InterfacePtr>& p) throw(_com_error);
```

参数

i

一个原始接口指针。

p

一个灵敏指针。

注释

这些函数模板允许灵敏指针出现在比较运算符右边的比较。以上这些不是 _com_ptr_t 的成员函数。

Microsoft 特殊处结束

_bstr_t

Microsoft 特殊处→

一个 _bstr_t 对象封装了 BSTR 数据类型。通过对函数 SysAllocString 和 SysFreeString 以及其他一些 BSTR APIs 的调用,该类管理资源的分配和回收。类 _bstr_t 采用引用计数来避免过度的开销。

#include <comdef.h>

构造函数	
_bstr_t	构造一个 _bstr_t 对象

操作	
copy	构造一个封装的 BSTR 的拷贝
length	返回一个封装的 BSTR 的长度

运算符	
operator =	对一个已有的_bstr_t对象赋新值
operator +=	在一个_bstr_t对象的尾部增加新的字符
operator +	连接两个字符串
operator !	检查封装的BSTR是否是一个空串
operator ==, !=, <, >, <=, >=	比较两个_bstr_t对象
operator wchar_t*, char*	从封装的Unicode或多字节BSTR对象中提取指针

Microsoft 特殊处结束

成员函数

_bstr_t::_bstr_t

Microsoft 特殊处→

```
_bstr_t() throw();
_bstr_t(const _bstr_t& s1) throw();
_bstr_t(const char* s2) throw(_com_error);
_bstr_t(const wchar_t* s3) throw(_com_error);
_bstr_t(const _variant_t& var) throw (_com_error);
_bstr_t(BSTR bstr, bool fCopy) throw (_com_error);
```

参数

s1

一个要拷贝的_bstr_t对象。

s2

一个多字节的字符串。

s3

一个单代码字符串。

var

一个_variant_t对象。

bstr

一个已存在的BSTR对象。

fCopy

如果是假,bstr参数将同一个新的对象相联,并也会调用函数SysAllocString创建一个拷贝。

注释

构造一个_bstr_t对象。

- **_bstr_t()** **构造一个封装一个NULL BSTR对象的缺省_bstr_t对象。**
- **_bstr_t(_bstr_t& s1)** **把一个_bstr_t对象构造为另一个的拷贝。这是一个“影子”拷贝,因为它只增加对封装的BSTR对象的引用计数,而不是创建一个新的对象。**
- **_bstr_t(char*** *s2*) **通过调用SysAssocString函数创建新的BSTR**

对象来构造一个_bstr_t 对象,并封装此 BSTR 对象。此构造函数首先采用一个从多字节至单代码的转换。

- _bstr_t(wchar_t* *s3*) 通过调用 SysAssocString 函数创建新的 BSTR 对象来构造一个_bstr_t 对象,并封装此 BSTR 对象。
- _bstr_t(_variant_t& *var*) 通过从封装的 VARIANT 对象中提取 BSTR 对象来从一个_variant_t 对象构造_bstr_t 对象。
- _bstr_t(BSTR bstr, bool *fCopy*) 从一个已存在的 BSTR 对象(同 wchar_t*字符串相反)中构造一个_bstr_t 对象。如果 *fCopy* 是假,则提供的 BSTR 将同新的对象连接,并不是通过调用函数 SysAllocString 创建一个新的拷贝。在类型库头文件中的包装器(wrapper)函数使用这种方法封装并取一个 BSTR 对象拥用者,该方法是在一个_bstr_t 对象中通过接口方法返回。

Microsoft 特殊处结束

_bstr_t::copy

Microsoft 特殊处→

```
BSTR copy() const throw(_com_error);
```

注释

返回一个封装的 BSTR 对象的最新分配的拷贝。

Microsoft 特殊处结束

_bstr_t::length

Microsoft 特殊处→

```
unsigned int length () const throw();
```

注释

返回一个封装的 BSTR 对象的长度。

Microsoft 特殊处结束

运算符

_bstr_t::operator =

Microsoft 特殊处→

```
_bstr_t& operator=(const _bstr_t& s1) throw ();
_bstr_t& operator=(const char* s2) throw(_com_error);
_bstr_t& operator=(const wchar_t* s3) throw(_com_error);
_bstr_t& operator=(const _variant_t& var) throw(_com_error);
```

参数

s1

一个要赋值给已存在的 _bstr_t 对象的 _bstr_t 对象。

s2

一个要赋值给已存在的 _bstr_t 对象的多字节字符串。

s3

一个要赋值给已存在的 _bstr_t 对象的单代码字符串。

var

一个要赋值给已存在的 _bstr_t 对象的 _variant_t 对象。

注释

对一个已存在的_bstr_对象赋一个新值。

Microsoft 特殊处结束

_bstr_t::operator +=, +

Microsoft 特殊处→

```
_bstr_t& operator+=(const _bstr_t& s1) throw(_com_error);
_bstr_t operator+(const _bstr_t& s1) const throw(_com_error);
friend _bstr_t operator+(const char* s2, const _bstr_t& s1);
friend _bstr_t operator+(const wchar_t* s3, const _bstr_t& s1);
```

参数

s1

一个_bstr_t 对象。

s2

一个多字节字符串。

s3

一个单代码字符串。

注释

这些操作完成字符串的连接：

- operator+=(*s1*) 把 s1 封装对象 BSTR 中的字符加到此对象封装的 BSTR 的尾部。
- operator+(*s1*) 把该对象的 BSTR 同 s1 中的 BSTR 相连并返回新的

_bstr_t。

- operator+(*s2*, *s1*) 把一个多字节字符串 s2(转换为 Unicode 型)同 s1 中封装的 BSTR 相连,并返回新的_bstr_t。
- operator+(*s3*, *s1*) 把一个单代码字符串 s3 同 s1 中封装的 BSTR 相连,并返回新的_bstr_t。

Microsoft 特殊处结束

_bstr_t::operator !

Microsoft 特殊处→

```
bool operator!() const throw();
```

注释

检查封装的 BSTR 对象是否为 NULL 字符串。如果是返回真,否则返回假。

Microsoft 特殊处结束

_bstr_t 关系运算符

Microsoft 特殊处→

```
bool operator==(const _bstr_t& str) const throw();
bool operator!=(const _bstr_t& str) const throw();
bool operator<(const _bstr_t& str) const throw();
bool operator>(const _bstr_t& str) const throw();
bool operator<=(const _bstr_t& str) const throw();
bool operator>=(const _bstr_t& str) const throw();
```

注释
这些操作按字典次序比较两个_bstr_t 对象。如果能进行比较,则返回真;否则返回假。
Microsoft 特殊处结束

_bstr_t::wchar_t*, _bstr_t::char*

Microsoft 特殊处→

```
operator const wchar_t*() const throw();
operator wchar_t*() const throw();
operator const char*() const throw(_com_error);
operator char*() const throw(_com_error);
```

注释
这些操作可用于提取封装的单代码或多字节 BSTR 对象的原始指针。这些运算符返回指向实际内部的缓冲区的指针,因此结果字符串是不能被修改的。
Microsoft 特殊处结束

_variant_t

Microsoft 特殊处→
一个_variant_t 对象封装了 VARIANT 数据类型。该类管理的分配和回收,并在合适的时候对函数 VariantInit 和 VariantClear 进行调用。

```
#include <comdef.h>
```

构造函数	
_variant_t	构造一个_variant_t 对象

运算符	
Attach	把一个 VARIANT 对象连接到一个 _variant_t 对象中
Clear	清除封装的 VARIANT对象
ChangeType	把一个_variant_t 对象的类型改变为说明的 VARTYPE
Detach	把一个 VARIANT 对象从一个_variant_t 对象中除去连接
SetString	把一个字符串赋值给一个_variant_t 对象

运算符	
operator =	对一个已存在的_variant_t 对象赋以新值。
operator ==, !=	比较两个_variant_t 对象相等还是不等。
Extractors	从一个封装的 VARIATN 对象中提取数据。

Microsoft 特殊处结束

成员函数

_variant_t::_variant_t

Microsoft 特殊处→

```
_variant_t() throw();
_variant_t(const VARIANT& varSrc) throw(_com_error);
_variant_t(const VARIANT* pVarSrc) throw(_com_error);
_variant_t(const _variant_t& var_t_Src) throw(_com_error);
_variant_t(VARIANT& varSrc, bool fCopy) throw(_com_error);
_variant_t(short sSrc, VARTYPE vtSrc = VT_I2) throw(_com_error);
_variant_t(long lSrc, VARTYPE vtSrc = VT_I4) throw(_com_error);
_variant_t(float fltSrc) throw();
_variant_t(double dblSrc, VARTYPE vtSrc = VT_R8) throw(_com_error);
_variant_t(const CY& cySrc) throw();
_variant_t(const _bstr_t& bstrSrc) throw(_com_error);
_variant_t(const wchar_t *wstrSrc) throw(_com_error);
_variant_t(const char* strSrc) throw(_com_error);
_variant_t(bool bSrc) throw();
_variant_t(IUnknown* pIUknownSrc, bool fAddRef = true) throw();
_variant_t(IDispatch* pDispSrc, bool fAddRef = true) throw();
_variant_t(const DECIMAL& decSrc) throw();
_variant_t(BYTE bSrc) throw();
```

参数

varSrc

要被拷贝到一个新的 _variant_t 对象的 VARIANT 对象。

pVarSrc

要被拷贝到一个新的 _variant_t 对象的 VARIANT 对象的指针。

var_t_Src

要被拷贝到一个新的 _variant_t 对象的 _variant_t 对象。

fCopy

如果是假,则提供的 VARIANT 对象就同新的 _variant_t 对象连接,而不会调用 VariantCopy 以创建一个新的拷贝。

ISrc, sSrc

一个要拷贝到新的 _variant_t 对象中的整型值。

vtSrc

为新 _variant_t 对象的 VARTYPE。

fltSrc, dblSrc

一个要拷贝到新的 _variant_t 对象中的数字值。

cySrc

一个要拷贝到新的 _variant_t 对象中的 CY 对象。

bstrSrc

一个要拷贝到新的 _variant_t 对象中的 _bstr_t 对象。

strSrc, wstrSrc

一个要拷贝到新的 _variant_t 对象中的字符串。

bSrc
一个要拷贝到新的_variant_t 对象中的布尔型值。
pIUknownSrc
将要封装在新的_variant_t 对象中的指向 VT_UNKNOWN 对象的 COM 接口指针。
pDispSrc
将要封装在新的_variant_t 对象中的指向 VT_DISPATCH 对象的 COM 接口指针。
decSrc
一个要拷贝到新的_variant_t 对象中的十进制值。
bSrc
一个要拷贝到新的_variant_t 对象中的字节值。
注释
构造一个_variant_t 对象。

- _variant_t() 构造一个空的_variant_对象 VT_EMPTY。
- _variant_t(VARIANT& *varSrc*) 用一个 VARIANT 对象的拷贝构造一个 variant_t 对象。variant 类型被保留。
- _variant_t(VARIANT* *pVarSrc*) 用一个 VARIANT 对象的拷贝构造一个 variant_t 对象。variant 类型被保留。
- _variant_t(*_variant_t*& *var_t_Src*) 从其它的_variant_t 对象中构造一个_variant_t 对象。variant 类型被保留。
- _variant_t(VARIANT& *varSrc*, bool *fCopy*) 从其它已存在的 VARIANT 对象中构造一个_variant_t 对象。如果 *fCopy* 是假,则 VARIANT 对象将同新的对象连接而不会创建一个拷贝。

- _variant_t(short *sSrc*, VARTYPE *vtSrc* = VT_I2) 从一个短整型值中构造一个有 VT_I2 或者 VT_BOOL 类型的_variant_t 对象。任何其它的 VARTYPE 会导致一个 E_INVALIDARG 错误。
- _variant_t(long *lSrc*, VARTYPE *vtSrc* = VT_I4) 从一个长整型值中构造一个有 VT_I4 或者 VT_BOOL 或者 VT_ERROR 类型的_variant_t 对象。任何其它的 VARTYPE 会导致一个 E_INVALIDARG 错误。
- _variant_t(float *fltSrc*) 从一个浮点数值中构造一个有 VT_R4 类型的_variant_t 对象。
- _variant_t(double *dblSrc*, VARTYPE *vtSrc* = VT_R8) 从一个双精度型数值中构造一个有 VT_R8 或者 VT_DATE 类型的_variant_t 对象。任何其它的 VARTYPE 会导致一个 E_INVALIDARG 错误。
- _variant_t(CY& cySrc) 从一个 CY 对象中构造一个有 VT_CY 类型的_variant_t 对象。
- _variant_t(_bstr_t& *bstrSrc*) 从一个_bstr_t 对象中构造一个有 VT_BSTR 类型的_variant_t 对象。将会分配一个新的 BSTR。
- _variant_t(wchar_t **wstrSrc*) 从一个 Unicode 型字符串中构造一个有 VT_BSTR 类型的_variant_t 对象。将会分配一个新的 BSTR。
- _variant_t(char* strSrc) 从一个字符串中构造一个有 VT_BSTR 类型的_variant_t 对象。将会分配一个新的 BSTR。
- _variant_t(bool *bSrc*) 从一个 Bool 型值中构造一个有 VT_BOOL 类型的_variant_t 对象。
- _variant_t(IUnknown* *pIUknownSrc*, bool *fAddRef* = true) 从

一个 COM 接口指针中构造一个有 VT_UNKNOWN 类型的_variant_t 对象。如果 *fAddRef* 的值是真,对提供的接口指针调用 AddRef 函数以匹配当_variant_t 对象在消毁时将要调用的 Release 函数。程序员有责任对提供的接口指针调用 Release。如果 *fAddRef* 是假,由此构造函数对提供的接口指针有所有用,并且也不对提供的接口指针调用 Release。

- _variant_t(IDispatch* *pDispSrc*, bool *fAddRef* = true) 从一个 COM 接口指针中构造一个有 VT_DISPATCH 类型的_variant_t 对象。如果 *fAddRef* 的值是真,对提供的接口指针调用 AddRef 函数以匹配当_variant_t 对象在消毁时将要调用的 Release 函数。程序员有责任对提供的接口指针调用 Release。如果 *fAddRef* 是假,由此构造函数对提供的接口指针有所有用,并且也不对提供的接口指针调用 Release。
- _variant_t(DECIMAL& *decSrc*) 从一个数字值构造一个具有 VT_DECIMAL 类型的_variant_t 对象。
- _variant_t(BYTE *bSrc*)从一个 BYTE 值构造一个具有 VT_UI1 类型的_variant_t 对象。

Microsoft 特殊处结束

_variant_t::Attach

Microsoft 特殊处→

void Attach(VARIANT& ***varSrc***) throw(_com_error);

参数

varSrc

一个将要同此_variant_t对象连接的VARIANT对象。

注释

通过封装此 VARIANT 而取得其所有权。这一成员函数释放任何存在的封装的VARIANT,然后拷贝所提供的 VARIANT,并把它的 VARTYPE 设置为 VT_EMPTY 以使得其资源仅仅只能由_variant_t的析构函数释放。

Microsoft 特殊处结束

_variant_t::Clear

Microsoft 特殊处→

```
void Clear() throw(_com_error);
```

注释

对封装的 VARIANT 对象调用 VariantClear。

Microsoft 特殊处结束

_variant_t::ChangeType

Microsoft 特殊处→

```
void ChangeType(VARTYPE vartype, const _variant_t* pSrc = NULL)
throw(_com_error);
```

参数

vartype

_variant_t对象的VARTYPE。

pSrc
一个指向要被转换的_variant_t 对象的指针。如果它的值是 NULL,则会适当地进行转换。
注释
此成员函数转换一个_variant_t 对象为被说明的 VARTYPE。如果 *pSrc* 是 NULL,会适当地进行转换,否则,此_variant_t 对象会从 *pSrc* 拷贝并进行转换。
Microsoft 特殊处结束

_variant_t::Detach

Microsoft 特殊处→
VARIANT Detach() throw(_com_error);
返回值
封装的 VARIANT
注释
提取并返回封装的 VARIANT,然后清除_variant_t 对象但并不消毁它。此成员函数从封装中移去 VARIANT 并把_variant_t 对象的 VARTYPE 设置为 VT_EMPTY。可通过调用 VariantInit 函数释放返回的 VARIANT。
Microsoft 特殊处结束

_variant_t::SetString

Microsoft 特殊处→
void SetString(const char* *pSrc*) throw(_com_error);
参数

pSrc
指向一个多字节字符串的指针。
注释
把一个多字节字符串转换为单代码 BSTR 对象,然后把它赋给 _variant_t 对象。
Microsoft 特殊处结束

运算符

_variant_t::operator =

Microsoft 特殊处→

```
_variant_t& operator=(const VARIANT& varSrc) throw(_com_error);
_variant_t& operator=(const VARIANT* pVarSrc) throw(_com_error);
_variant_t& operator=(const _variant_t& var_t_Src) throw(_com_error);
_variant_t& operator=(short sSrc) throw(_com_error);
_variant_t& operator=(long lSrc) throw(_com_error);
_variant_t& operator=(float fltSrc) throw(_com_error);
_variant_t& operator=(double dblSrc) throw(_com_error);
_variant_t& operator=(const CY& cySrc) throw(_com_error);
_variant_t& operator=(const _bstr_t& bstrSrc) throw(_com_error);
_variant_t& operator=(const wchar_t* wstrSrc) throw(_com_error);
_variant_t& operator=(const char* strSrc) throw(_com_error);
_variant_t& operator=(IDispatch* pDispSrc) throw(_com_error);
```

```
_variant_t& operator=(bool bSrc) throw(_com_error);
_variant_t& operator=(IUnknown* pSrc) throw(_com_error);
_variant_t& operator=(const DECIMAL& decSrc) throw(_com_error);
_variant_t& operator=(BYTE bSrc) throw(_com_error);
```

注释

这些运算符把一个新的值赋给 _variant_t 对象：

- operator=(*varSrc*) 把一个已存在的 VARIANT 赋给一个 _variant_t 对象。
- operator=(*pVarSrc*) 把一个已存在的 VARIANT 赋给一个 _variant_t 对象。
- operator=(*var_t_Src*) 把一个已存在的 _variant_t 赋给一个 _variant_t 对象。
- operator=(*sSrc*) 把一个短整型值赋给一个 _variant_t 对象。
- operator=(*lSrc*) 把一个长整型值赋给一个 _variant_t 对象。
- operator=(*fltSrc*) 把一个 float 数值赋给一个 _variant_t 对象。
- operator=(*dblSrc*) 把一个 double 数值赋给一个 _variant_t 对象。
- operator=(*cySrc*) 把一个 CY 对象赋给一个 _variant_t 对象。
- operator=(*bstrSrc*) 把一个 BSTR 对象赋给一个 _variant_t 对象。
- operator=(*wstrSrc*) 把一个 Unicode 字符串赋给一个 _variant_t 对象。
- operator=(*strSrc*) 把一个多字节字符串赋给一个 _variant_t 对

象。

- operator=(*bSrc*) 把一个 Bool 值赋给一个 _variant_t 对象。
- operator=(*pDispSrc*) 把一个 VT_DISPATCH 对象赋给一个 _variant_t 对象。
- operator=(*pIUnknownSrc*) 把一个 VT_UNKNOWN 对象赋给一个 _variant_t 对象。
- operator=(*decSrc*) 把一个十进制值赋给一个 _variant_t 对象。
- operator=(*bSrc*) 把一个字节值赋给一个 _variant_t 对象。

Microsoft 特殊处结束

_variant_t 关系运算符

Microsoft 特殊处→

```
bool operator==(const VARIANT& varSrc) const throw(_com_error);
bool operator==(const VARIANT* pSrc) const throw(_com_error);
bool operator!=(const VARIANT& varSrc) const throw(_com_error);
bool operator!=(const VARIANT* pSrc) const throw(_com_error);
```

参数

varSrc

一个要同 _variant_t 对象进行比较的 VARIANT

pSrc

指向要同 _variant_t 对象进行比较的 VARIANT 的指针

注释

把一个_variant_t 对象同一个 VARIANT 进行比较,检查是否相等。如果比较可以进行,返回真,否则返回假。
Microsoft 特殊处结束

_variant_t 提取符

Microsoft 特殊处→

```
operator short() const throw(_com_error);
operator long() const throw(_com_error);
operator float() const throw(_com_error);
operator double() const throw(_com_error);
operator CY() const throw(_com_error);
operator bool() const throw(_com_error);
operator DECIMAL() const throw(_com_error);
operator BYTE() const throw(_com_error);
operator _bstr_t() const throw(_com_error);
operator IDispatch*() const throw(_com_error);
operator IUnknown*() const throw(_com_error);
```

注释
从一个封装的 VARIANT 中提取原始数据。如果 VARIANT 还不是正确的类型,则采用函数 VariantChangeType 试图进行转换,若失败会产生一个错误。

- operator short() 提取一个短整型值。
- operator long() 提取一个长整型值。

- operator float() 提取一个 float 型数值。
- operator double() 提取一个 double 型数值。
- operator CY() 提取一个 CY 对象。
- operator bool() 提取一个 BOOL 型值。
- operator DECIMAL() 提取一个十进制数值。
- operator BYTE() 提取一个 BYTE 型数值。
- operator _bstr_t() 提取一个封装在_bstr_t 对象中的字符串。
- operator IDispatch*() 从一个封装的 VARIANT 中提取一个 dispinterface 指针。对结果指针调用了 AddRef 函数,故有责任调用 Release 去释放它。
- operator IUnknown*() 从一个封装的 VARIANT 中提取一个 COM 接口指针。对结果指针调用了 AddRef 函数,故有责任调用 Release 去释放它。

Microsoft 特殊处结束

附录 D　图　　表

本附录包含以下一些图表：

- ASCII 字符代码
- ASCII 多语言代码
- ANSI 字符代码
- 关键字代码

ASCII 字符代码

ASCII 字符代码表含有扩展 ASCII 字符集的十进制和十六进制值。这些扩展字符集包括 ASCII 字符集和 128 个其它为图形和画线的字符,通常叫作“IBM 字符集”。

- 图表 1(代码 0～127)
- 图表 2(代码 128～255)

ASCII 多语言代码

在 IBM 字符集上存在着很多不同的变型,称为“代码页”。在一些欧洲国家出售的系统中作用的多语言代码是代码页 850,其中含有较少的图形字符、较多的重音字符以及其它特殊字符。

ANSI 字符代码

ANSI字符代码表列出了在Windows程序中使用的扩展字符。ANSI字符集中从32～126是来自于 ASCII 字符集的可显示代码。显示为实心块的 ANSI 字符是未定义

的字符以及在输出设备上会表现得不同的字符。

关键字代码

有些键,如功能键、光标键以及 ALT+KEY 的组合键没有 ASCII 代码。当一个键被按下的时候,在键盘中的微处理器产生一个两字节的“扩展扫描码”。第一个字节(低位)包含了 ASCII 码(如果有的话)。第二个字节(高位)含有扫描码——当键盘的键被按下或者被释放时由键盘所产生的一个唯一码。因为此扩展扫描码比标准 ASCII 码用得更广泛,程序可以用它确定哪些键没有 ASCII 码。

ASCII 字符代码图表 1

ASCII Character Codes Chart 1

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	SLE
^Q	17	11	◄	CS1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SIB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	sp
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	/
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂

ASCII 代码 127 有代码 DEL。在 MS-DOS 下,这个代码与 ASCII8(BS)的作用相同。DEL 是由 CTRL+BKSP 按键产生的。

ASCII 字符代码图表 2

ASCII Character Codes Chart 2

Dec	Hex	Char
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ

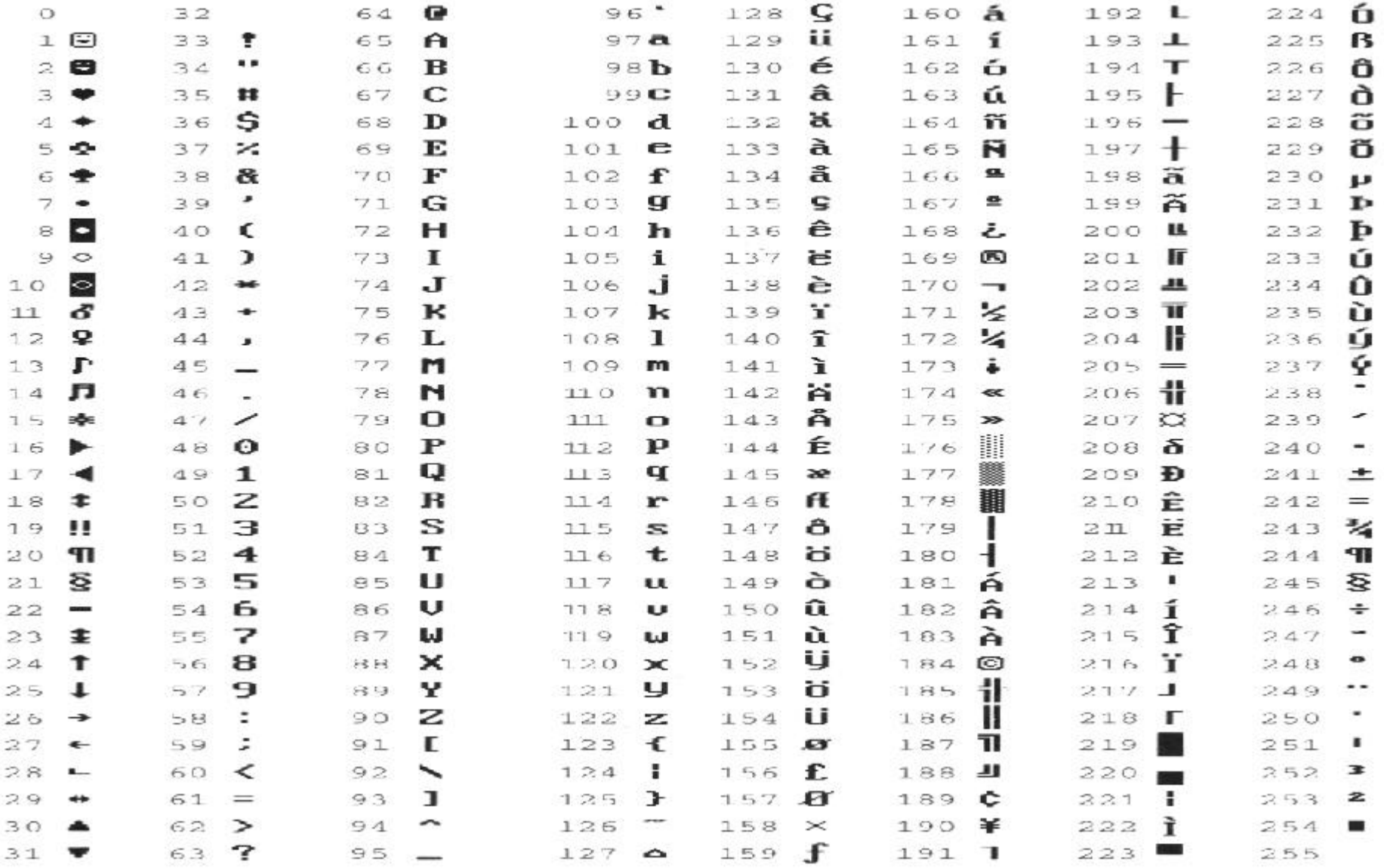
Dec	Hex	Char
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	▒
177	B1	▒
178	B2	▒
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐

Dec	Hex	Char
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	▒
220	DC	▒
221	DD	▒
222	DE	▒
223	DF	▒

Dec	Hex	Char
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	φ
238	EE	ε
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	n
253	FD	2
254	FE	■
255	FF	

ASCII 多语言代码图表

ASCII Multilingual Codes Chart



ANSI 字符代码图表

■ 指出这个字符不被 Windows 支持

T_T 指出这个字符仅在 TrueType 字体下可用

关键字代码图表 1

Key Codes Chart 1

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
ESC	1	01	27	1B	ESC	27	1B	ESC	27	1B	ESC	1	01	NUL§
1!	2	02	49	31	1	33	21	!				120	78	NUL
2@	3	03	50	32	2	64	40	@	3	03	NUL	121	79	NUL
3#	4	04	51	33	3	35	23	#				122	7A	NUL
4$	5	05	52	34	4	36	24	$				123	7B	NUL
5%	6	06	53	35	5	37	25	%				124	7C	NUL
6^	7	07	54	36	6	94	5E	^	30	1E	RS	125	7D	NUL
7&	8	08	55	37	7	38	26	&				126	7E	NUL
8*	9	09	56	38	8	42	2A	*				127	7F	NUL
9(	10	0A	57	39	9	40	28	(				128	80	NUL
0)	11	0B	48	30	0	41	29	)				129	81	NUL
-_	12	0C	45	2D	-	95	5F	_	31	1F	US	130	82	NUL
=+	13	0D	61	3D	=	43	2B	+				131	83	NUL
BKSP	14	0E	8	08		8	08		127	7F		14	0E	NUL§
TAB	15	0F	9	09		15	0F	NUL	148	94	NUL§	15	A5	NUL§
Q	16	10	113	71	q	81	51	Q	17	11	DC1	16	10	NUL
W	17	11	119	77	w	87	57	W	23	17	ETB	17	11	NUL
E	18	12	101	65	e	69	45	E	5	05	ENQ	18	12	NUL
R	19	13	114	72	r	82	52	R	18	12	DC2	19	13	NUL
T	20	14	116	74	t	84	54	T	20	14	SO	20	14	NUL
Y	21	15	121	79	y	89	59	Y	25	19	EM	21	15	NUL
U	22	16	117	75	u	85	55	U	21	15	NAK	22	16	NUL
I	23	17	105	69	i	73	49	I	9	09	TAB	23	17	NUL
O	24	18	111	6F	o	79	4F	O	15	0F	SI	24	18	NUL
P	25	19	112	70	p	80	50	P	16	10	DLE	25	19	NUL
[{	26	1A	91	5B	[	123	7B	{	27	1B	ESC	26	1A	NUL§
]}	27	1B	93	5D	]	125	7D	}	29	1D	GS	27	1B	NUL§
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF	28	1C	NUL§
ENTER£	28	1C	13	0D	CR	13	0D	CR	10	0A	LF	166	A6	NUL§
L CTRL	29	1D												
R CTRL£	29	1D												
A	30	1E	97	61	a	65	41	A	1	01	SOH	30	1E	NUL
S	31	1F	115	73	s	83	53	S	19	13	DC3	31	1F	NUL
D	32	20	100	64	d	68	44	D	4	04	EOT	32	20	NUL
F	33	21	102	66	f	70	46	F	6	06	ACK	33	21	NUL
G	34	22	103	67	g	71	47	G	7	07	BEL	34	22	NUL
H	35	23	104	68	h	72	48	H	8	08	BS	35	23	NUL
J	36	24	106	6A	j	74	4A	J	10	0A	LF	36	24	NUL
K	37	25	107	6B	k	75	4B	K	11	0B	VT	37	25	NUL
L	38	26	108	6C	l	76	4C	L	12	0C	FF	38	26	NUL
:;	39	27	59	3B	;	58	3A	:				39	27	NUL§
'"	40	28	39	27	'	34	22	"				40	28	NUL§
`~	41	29	96	60	`	126	7E	~				41	29	NUL§
L SHIFT	42	2A												
\\|	43	2B	92	5C	\	124	7C	|	28	1C	FS			
Z	44	2C	122	7A	z	90	5A	Z	26	1A	SUB	44	2C	NUL
X	45	2D	120	78	x	88	58	X	24	18	CAN	45	2D	NUL
C	46	2E	99	63	c	67	43	C	3	03	ETX	46	2E	NUL
V	47	2F	118	76	v	86	56	V	22	16	SYN	47	2F	NUL
B	48	30	98	62	b	66	42	B	2	02	STX	48	30	NUL
N	49	31	110	6E	n	78	4E	N	14	0E	SO	49	31	NUL
M	50	32	109	6D	m	77	4D	M	13	0D	CR	50	32	NUL
,<	51	33	44	2C	,	60	3C	<				51	33	NUL§
.>	52	34	46	2E	.	62	3E	>				52	34	NUL§

关 键 字 代 码 图 表 2

Key Codes Chart 2

Key	Scan Code		ASCII or Extended†			ASCII or Extended† with SHIFT			ASCII or Extended† with CTRL			ASCII or Extended† with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
/?	53	35	47	2F	/	63	3F	?				53	34	NUL§
GRAY /£	53	35	47	2F	/	63	3F	?	149	95	NUL	164	A5	NUL
R SHIFT	54	36												
*PRTSC	55	37	42	2A	*	PRTSC		††	16	10				
L ALT	56	38												
R ALT£	56	38												
SPACE	57	39	32	20	SPC	32	20	SPC	32	20	SPC	32	20	SPC
CAPS	58	3A												
F1	59	3B	59	3B	NUL	84	54	NUL	94	5E	NUL	104	68	NUL
F2	60	3C	60	3C	NUL	85	55	NUL	95	5F	NUL	105	69	NUL
F3	61	3D	61	3D	NUL	86	56	NUL	96	60	NUL	106	6A	NUL
F4	62	3E	62	3E	NUL	87	57	NUL	97	61	NUL	107	6B	NUL
F5	63	3F	63	3F	NUL	88	58	NUL	98	62	NUL	108	6C	NUL
F6	64	40	64	40	NUL	89	59	NUL	99	63	NUL	109	6D	NUL
F7	65	41	65	41	NUL	90	5A	NUL	100	64	NUL	110	6E	NUL
F8	66	42	66	42	NUL	91	5B	NUL	101	65	NUL	111	6F	NUL
F9	67	43	67	43	NUL	92	5C	NUL	102	66	NUL	112	70	NUL
F10	68	44	68	44	NUL	93	5D	NUL	103	67	NUL	113	71	NUL
F11£	87	57	133	85	E0	135	87	E0	137	89	E0	139	8B	E0
F12£	88	58	134	86	E0	136	88	E0	138	8A	E0	140	8C	E0
NUM	69	45												
SCROLL	70	46												
HOME	71	47	71	47	NUL	55	37	7	119	77	NUL			
HOME£	71	47	71	47	E0	71	47	E0	119	77	E0	151	97	NUL
UP	72	48	72	48	NUL	56	38	8	141	8D	NUL§			
UP£	72	48	72	48	E0	72	48	E0	141	8D	E0	152	98	NUL
PGUP	73	49	73	49	NUL	57	39	9	132	84	NUL			
PGUP£	73	49	73	49	E0	73	49	E0	132	84	E0	153	99	NUL
GRAY-	74	4A				45	2D	-						
LEFT	75	4B	75	4B	NUL	52	34	4	115	73	NUL			
LEFT£	75	4B	75	4B	E0	75	4B	E0	115	73	E0	155	9B	NUL
CENTER	76	4C				53	35	5						
RIGHT	77	4D	77	4D	NUL	54	36	6	116	74	NUL			
RIGHT£	77	4D	77	4D	E0	77	4D	E0	116	74	E0	157	9D	NUL
GRAY+	78	4E				43	2B	+						
END	79	4F	79	4F	NUL	49	31	1	117	75	NUL			
END£	79	4F	79	4F	E0	79	4F	E0	117	75	E0	159	9F	NUL
DOWN	80	50	80	50	NUL	50	32	2	145	91	NUL§			
DOWN£	80	50	80	50	E0	80	50	E0	145	91	E0	160	A0	NUL
PGDN	81	51	81	51	NUL	51	33	3	118	76	NUL			
PGDN£	81	51	81	51	E0	81	51	E0	118	76	E0	161	A1	NUL
INS	82	52	82	52	NUL	48	30	0	146	92	NUL§			
INS£	82	52	82	52	E0	82	52	E0	146	92	E0	162	A2	NUL
DEL	83	53	83	53	NUL	46	2E	.	147	93	NUL§			
DEL£	83	53	83	53	E0	83	53	E0	147	93	E0	163	A3	NUL

↑ 在初始字符时扩展的代码返回 0(NUL)或 E0(十进制 224)。这是键盘缓冲区中第二个(扩展的)带有可用的信号。
§ 这些组合键仅在扩展的键盘上识别。
£ 这些键仅在扩展的键盘上可用。大多数在光标/控制键盘区。如果从键盘端口(60h)读扫描码,它以两个字节(E0h)后跟正常扫描码方式出现。但是,当通过 BIOS 中断 16h 读 ENTER 和/键时,只看见 E0h,因为该中断仅给出一位扫描码。
↑↑在 MS-DOS 下,SHIFT+PRTSC 引起中断 5,它打印屏幕,除非已经定义一个中断处理器代替该缺省的中断 5 处理器。