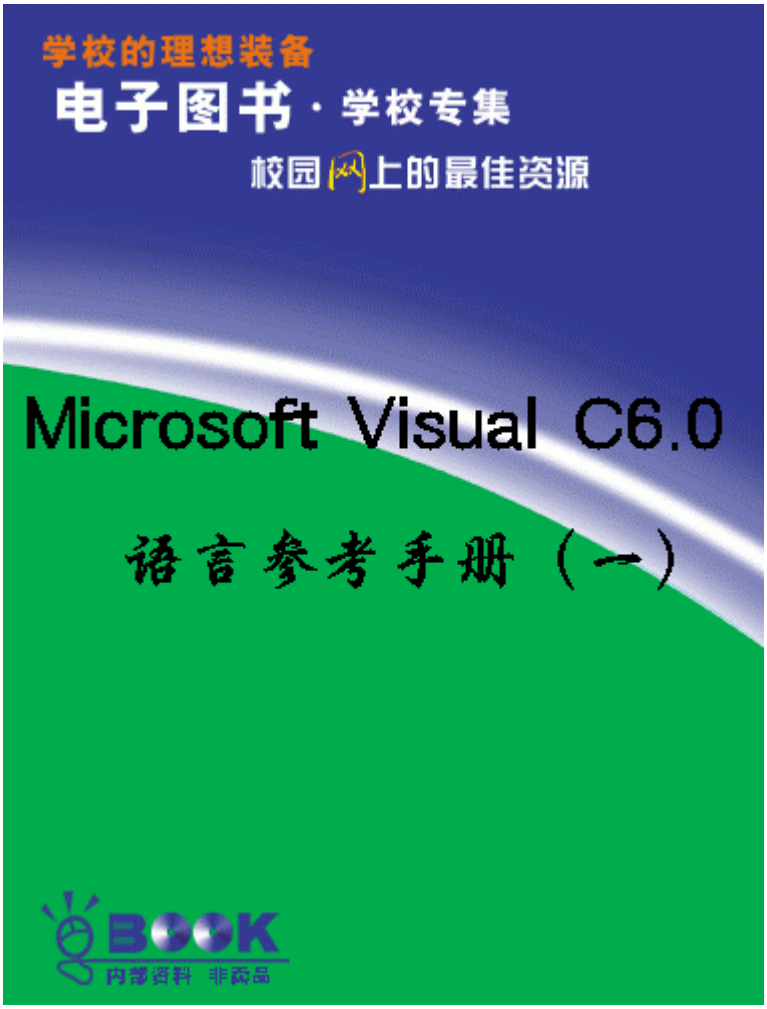
学校的理想装备
电子图书·学校专集
校园网上的最佳资源
Microsoft Visual C6.0
语言参考手册（一）
BOOK
内部资料 非卖品

目　　录

第一部分　Microsoft Visual C 6.0 语言参考手册

引言
第 1 章　C 的基本元素
第 2 章　程序结构
第 3 章　说明和类
第 4 章　表达式和赋值
第 5 章　语句
第 6 章　函数
附录 A　C 语言语法总结
附录 B　实现定义的行为

第二部分　Microsoft Visual C++ 6.0 语言参考手册

引言　C++语言参考手册的组织
第 1 章　词法规定
第 2 章　基本概念
第 3 章　标准转换
第 4 章　表达式
第 5 章　语句
第 6 章　说明
第 7 章　说明符

第 8 章　类
第 9 章　派生类
第 10 章　成员访问控制
第 11 章　特殊成员函数
第 12 章　重载
附录 A　语法总结
附录 B　Microsoft 特殊修饰符
附录 C　编译器 COM 支持类
附录 D 图表

第三部分 Microsoft Visual C++ 6.0 预处理器参考手册

引言
第 1 章　预处理器
第 2 章 编译指示指令
附录　语法总结

返回总目录

Microsoft Visual C 6.0
语言参考手册

第一部分目录

引　　言 6
C 语言参考手册的组织 6
本手册的范围 6
ANSI 一致性 7
第 1 章　C 的基本元素 8
语言符号 9
注　　释 10
关 键 词 13
标 识 符 14
常　　量 20
字符串文字 37
标点和特殊字符 41
第 2 章　程 序 结 构 43
源文件和源程序 43
main 函数和程序执行 51
命令行参量的分析 55
生存期、范围、可见性和连接 58
名称空间 65
第 3 章　说明和类型 68
说明概述 69
存储类 72

指示符 83
类型修饰符 87
说明符和变量说明 89
解释更复杂的说明符 122
基本类型的存储 136
不完整类型 143
typedef 说明 144
扩充的存储类属性 148
第 4 章 表达式和赋值 154
操作数和表达式 154
运算符 163
类型转换 205
第 5 章 语 句 219
语句概述 219
break 语句 220
复合语句 222
continue 语句 224
do-while 语句 225
表达式语句 226
for 语句 227
goto 和标号语句 229
if 语句 231

空语句 233
return 语句 234
switch 语句 236
try-except 语句 240
try-finally 语句 244
while 语句 245
第 6 章 函 数 247
函数概述 247
函数定义 249
函数原型 274
函数调用 277
附录 A C 语言语法总结 286
定义和约定 286
词法 287
短语结构语法 295
附录 B 实现定义的行为 308
转换：诊断 308
环 境 309
标识符的行为 310
字 符 311
整 数 315
浮点数学 319

数组的指针 320
寄存器:寄存器的可用性 321
结构、联合、枚举和位域 322
修饰符:访问 volatile 对象 325
说明符:最大个数 325
语句:switch 语句的限制 326
预处理器命令 326
库函数 330

引　　言

C 语言参考手册的组织

- C 的基本元素
- 程序结构
- 说明和类型
- 表达式和赋值
- 语句
- 函数
- C 语言语法总结
- 实现定义的行为

本手册的范围

C 是一种灵活的语言,它把很多程序设计决定交给你。为了实现这种想法,C 规定
了很少的限制,如类型转换。尽管这种语言特征使得程序设计工作更容易,但你必须知道这种语言,掌握程序的行为。本章提供了 C 语言组成成分的信息和 Microsoft 实现的特征。C 语言的语法来自 ANSI X3.159-1989 美国国家信息系统——程序设计语言——C 标准(简称为 ANSI C 标准),

但它不是 ANSI C 标准部分。附录 A“C 语言语法总结”提供了该语法以及如何读和使用该语法定义的描述。

本书没有讨论 C++的程序设计,有关 C++语言的信息参见“Microsoft Visual C++ 6.0 参考库”的“Microsoft Visual C++ 6.0 语言参考”卷。

ANSI 一致性

Microsoft C 符合 ANSI C 标准中宣布的 C 语言的标准,Microsoft 扩充了 ANSI C 标准,在本书的文本和语法中以及联机参考中另给以注记。因为这些扩充不是 ANSI 标准的一部分,这些使用可能限制了系统之间程序的移植性。缺省地,Microsoft 扩充是允许的,为了禁止这些扩充,指定/Za 编译器选项。使用/Za,非 ANSI 代码会产生错误或警告。

第 1 章　C 的基本元素

本章描述了 C 程序语言的组成元素,包括用于构造一个 C 程序的名称、数组和字符。ANSI C 语法称这些组成元素为“语言符号”,本章解释如何定义语言符号以及编译器如何对它们求值的。

包括如下主题:

- 语言符号
- 注释
- 关键词
- 标识符
- 常量
- 字符串文字
- 标点和特定字符

本章还包括三字母、浮点常量、整数常量和转义序列的参考表。

“运算符”是指定值如何操作的符号(有单字符和字符组合),每个符号都解释为单个单元,称为一个语言符号。有关详细信息参见第 4 章“表达式和赋值”。

语言符号

在 C 源程序中,编译器识别的基本元素是“语言符号”,一个语言符号是编译器不能分成组成成分元素的源程序文本。

语法

语言符号:

- 关键词
- 标识符
- 常量
- 字符串文字
- 运算符
- 标点字符

注意:有关 ANSI 语言规定的解释,参见本卷后面附录 A“C 语言语法总结”的引言部分。

在本章中描述的关键词、标识符、常量、字符串文字和运算符都是语言符号的例子。标点字符和中括号([])、花括号({})、圆括号(())和逗号(,)也是语言符号。

空白字符

空格、制表、换行、回车、换页、纵向制表(vertical tab)和换行(newline)字符都称为空白字符(white-space),因为它们具有相同的功能,即作为打印

页上单词和行之间的空白,使其更容易阅读。通过空白字符和其它语言符号，例如运算符和标点，从而对语言符号进行定界。当对代码进行词法分析时,C 编译器忽略空白字符,除非使用它们作为分隔符或者字符常量或字符串文字的组成成分。使用空白字符使程序更具可读性。注意编译器也把注释作为空白字符处理。

注　释

一个“注释”是以正向斜线/星号的组合(/*)(编译器处理作为单个空白字符,其它忽略)开头的字符序列。一个注释可以包含可表示字符集的字符的任何组合,包括换行字符,但不能有“结束注释”定界符(*/)。注释可以多于一行但不能嵌套。

注释可以出现在一个空白字符允许出现的任何地方,由于编译器把一个注释作为一个空白字符处理,你不能在语言符号中包含注释,编译器忽略注释中的字符。

使用注释文档化你的代码,这个例子是编译器接受的一个注释:

```
/* Comments can contain keywords such as
    for and while without generating errors. */
```

注释可以与一个代码语句出现在同一行:

```
printf("Hello\n"); /* Comments can go here */
```

你可以在函数或程序模块之前选择一个描述性的注释块:

```
/* MATHERR.C illustrates writing an error routine
```

```
 * for math functions.
 */
```

既然注释不能包含嵌套的注释,这个例子导致一个错误:

```
/* Comment out this routine for testing
/* Open file */
    fh=_open("myfile.c",_O_RDONLY);
    .
    .
    .
*/
```

这个错误出现是因为编译器识别第一个*/,单词 Open file 之后作为注释结束,它处理余下的文本,在发现注释外面的*/时产生一个错误。

当你使用注释表示某些代码行为了测试而不激活时,预处理器命令#if 和#endif 以及条件编译是另一种有用的方法。有关详细信息参见本卷后面“预处理器参考”中的“预处理器命令”。

Microsoft 特殊处→

Microsoft 编译器还支持以两个正向斜线(//)开头的单行注释。如果你用/Za(ANSI 标准)编译,这些注释产生错误,另外这些注释不能超过第二行。

```
// This is a valid comment
```

注释以两个正向斜线(//)开头,以下一个换行字符结束,不能在前加上一个转义字符。在下面的例子中,换行符有一个反向斜线(\),建立一个“转义序列”。这个转义序列导致编译器把下一行作为前面行的一部分进行处理(有关详细信息,参见本章后面的“转义序列”)。

```
// my comment \
  i++;
```

因此,i++;语句被注释了。

Microsoft C 缺省是启用 Microsoft 扩充的。使用/Za 关闭这些扩充。

Microsoft 特殊处结束

语言符号的求值

当编译器解释语言符号时,它在移到下一个语言符号之前在单个语言符号中包含尽可能多的字符。因为这种行为,编译器不可能在不用空白的字符分隔它们时来解释语言符号,对于如下表达式:

```
i+++j
```

在这个例子中,编译器首先从三个加号中产生最长可能的运算符(++),然后处理余下的加号作为一个加法运算符(+)。因此,该表达式解释为(i++)+(j),而不是(i)+(++j)。在这个类似的情况下,使用空白和括号避免模糊性,并确保正确的表达式计算。

Microsoft 特殊处→

C 编译器处理 CTRL+Z 字符为一个文件结尾指示器,它忽略 CTRL+Z 之后的任何文本。

Microsoft 特殊处结束

关 键 词

关键词是 C 编译器中具有特定意义的单词。在翻译阶段 7 和 8,一个标识符不能和 C 关键词具有相同的拼写和大小写(参见本卷后面“预处理器参考”中的转换阶段的描述;有关标识符的信息,参见下一节“标识符”)。C 语言使用如下关键词:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

你不能重新定义关键词,但你可以使用 C 预处理器命令在编译之前指定用于替换关键词的文本。

Microsoft 特殊处→

ANSI C 标准允许用两个下划线引导的标识符作为编译执行的保留字。因此,Microsoft 约定 Microsoft 的指定关键词之前带两个下划线。这些词不能用作标识符名称。对于 ANSI 命名标识符的描述,包括双下划线的使用,参见下一节“标识符”。

Microsoft C 编译器识别的关键词和指定标识符如下:

__asm	dllimport[2]	__int8		naked[2]
__based[1]	__except		__int16	__stdcall
__cdecl	__fastcall	__int32		thread[2]
__declspec	__finally	__int64		__try
dllexport[2]	__inline	__leave		

1. __based 关键词限制用于 32 位目标编译。

2. 这些是与 __declspec 一起使用时指定的标识符,它们用在其它上下文中没有限制。

缺省地 Microsoft 扩充是允许的,为了确保你的程序是完整可移植的,你在编译时指定/Za 选项(ANSI 兼容的编译)来关闭 Microsoft 扩充。当你这样做后,Microsoft 特定关键词是禁止的。

当允许 Microsoft 扩充时,你可以在程序中使用上面所列的关键词。为了与 ANSI 一致,这些大多数关键词加上了一个双下划线前缀。但有四个例外,即 dllexport, dllimport, naked 和 thread,仅与 __declspc 一起使用,因此不需要引导的两个下划线。为了向后兼容,也支持余下的关键字的一个下划线引导的版本。

Microsoft 特殊处结束

标 识 符

“标识符”或“符号”是你的程序中提供的变量、类型、函数和标号的名称。标识符在拼写和大小写上必须不同于关键词。你不能使用关键词(C 或 Microsoft)作为标识符,它们为了特定用途而被保留。你可以通过在

变量、类型或函数的说明中指定它来建立一个标识符。在以下例子中,result 是一个整数变量的标识符,而 main 和 printf 是函数的标识符名称:

```
void main()
{
    int result;

    if (result!=0)
        printf("Bad file handle\n");
}
```

一旦说明,你可以在后面的程序语句中使用该标识符引用关联的值。

一种特定类型的标识符,称为语句标号,它可以用在 goto 语句中(其说明在第 3 章“说明和类型”中描述。语句标号在第 5 章“语句”的“goto 和标号语句”中描述)。

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

一个标识符名称的第一个字符必须是非数字(也就是,第一个字符必须是一个下划线或者一个大写或小写字母)。ANSI 允许一个外部标识符的名称中有 6 个有效字符,在一个内部标识符(一个函数中)的名称中允许有 31 个字符。外部标识符(以全局范围说明或用存储类 extern 说明)要服从另外的命名限制,因为这些标识符可以被其它软件例如链接器处理。

Microsoft 特殊处→

虽然 ANSI 允许外部标识符名称中有 6 个有效字符,允许内部标识符(一个函数中)中有 31 个字符,但 Microsoft C 编译器允许一个内部或外部标识符名称中有 247 个字符。如果不关心与 ANSI 的兼容性,你可以使用/H 选项(限制外部名称的长度)修改这个缺省值为一个较小或较大的值。

Microsoft 特殊处结束

C 编译器把大写和小写字母当作不同的字符。这个特征称为“大小写敏感的”,允许你建立具有相同的拼写但有一个或多个不同的大小写的不同的标识符。例如,下面每个标识符都是唯一的:

add

ADD

Add

aDD

Microsoft 特殊处→

不要为标识符选择以两个下划线开头或以一个下划线加上一个大写字母

开头的名称。ANSI C 标准允许以这些字符组合开头的标识符名称保留为编译器使用。文件层范围的标识符也不要以一个下划线加上一个小写字母开头来命名。以这些字符开头的标识符也是保留的。Microsoft 约定使用一个下划线和一个大写字母开头的宏名称以及两个下划线开头的 Microsoft 特定关键字名称。为了避免任何命名冲突,总是选择不以一个或两个下划线开头的名称，或者以一个下划线加上一个大写字母开头的名称。

Microsoft 特殊处结束

下面的例子是满足 ANSI 和 Microsoft 命名限制的有效标识符:

```
j
count
temp1
top_of_page
skip12
LastNum
```

Microsoft 特殊处→

虽然源文件中的标识符缺省是大小写敏感的,但目标文件中的符号却不是。Microsoft C 在一个编译单元中处理标识符为大小写敏感的。

Microsoft 链接器是大小写敏感的,你必须根据大小写一致性地指定所有标识符。

“源字符集”是可以在源文件中出现的合法字符集。对于 Microsoft C,源字符集是标准 ASCII 字符集。包括 ASCII 字符的源字符集和执行字

符集用作转义序列。有关执行字符集的信息参见本章后面的“字符常量”。

Microsoft 特殊处结束

一个标识符具有“范围”,它是指被程序已知的和“连接”的程序范围,该“连接”确定另一个范围内同名称标识符是否是同一个标识符。这些主题在第 2 章“程序结构”中的“生存期、范围、可见性和连接”中解释。

多字节和宽字符

一个多字节字符是由一个或多个字节序列组成的一个字符。每个字节序列表示扩展字符集中的单个字符。多字节字符用在字符串例如汉字中。宽字符是总为 16 位宽的多种语言字符代码。字符常量的类型是 char,对于宽字符,其类型为 wchar_t。既然宽字符总是固定的长度,使用宽字符可以简化使用国际字符集的程序设计。

宽字符串文字 L"hello"变成一个类型 wchar_t 的 6 个整数的数组:

{L’h’,L’e’,L’l’,L’l’,L’o’,0}

单代码(Unicode)规格是宽字符的规格。多字节和宽字符之间进行转换的运行库例程包括 mbstowcs、mbtowc、wcstombs 和 wctomb。

三字母

C 源程序的源字符集被包含在 7 位 ASCII 字符集中,但不是 ISO 646-1983

Invariant Code Set的子集。三字母(trigraph)序列允许C程序仅使用ISO(国际标准组织) Invariant Code Set编写。三字母是编译器用对应的标点字符替换的三字符序列(以两个问号开头)。你可以在C源文件中使用三字母,该源文件的字符集不能包含某些标点字符的方便图形表示。

表1.1给出了9个三字母序列。所有源文件中出现的第一行的标点字符都被第2列中对应的字符所替换。

表1.1　三字母序列

三字母	标点字符
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

一个三字母总是作为单个源字符处理,在第一次转换阶段中,在识别字符串文字和字符常量中的转义字符之前进行三字母的转换。仅识别表1.1中列出的9个三字母,所有其它字符序列不作转换。

字符转义序列\?防止类似三字母的字符序列被误解释(有关转义序列的信

息,参见本章后面的“转义序列”)。例如,如果你试图用以下 printf 语句打印字符串 What??!:

```
printf(“ What??!\n");
```

打印的字符串 What|,因为??!是一个三字母,它被|字符所替换。正确打印这个字符串的语句如下:

```
printf(What?\?!\n");
```

在这个 printf 语句中,在第二个问号之前加上一个反斜杠转义字符防止??!作为一个三字母的误解释。

常　量

一个“常量”是一个可以在程序中作为值的一个数、字符,或者字符串。使用常量表示不能修改的浮点数、整数、枚举或字符值。

语法

常量:

- 浮点常量
- 整数常量
- 枚举常量
- 字符常量

常量通过具有一个值和一个类型来描述。浮点数、整数和字符常量在下面三个小节讨论。枚举常量在第 3 章“说明和类型”的“枚举说明”中描述。

浮点数常量

一个“浮点数常量”是一个表示带符号实数的十进制数。带符号实数的表示包括一个整数部分、一个小数部分和一个指数。使用浮点常量表示不能改变的浮点值。

语法

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: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

你可以省略小数点之前的数字(该值的整数部分),或小数点之后的数字(小数部分),但不能都省略。仅当你包括一个指数时才可以省去小数点。常量的数字和字符之间不需要空白字符分隔。

如下例子说明了几种浮点常量和表达式的格式:

```
15.75
1.575E1 /*=15.75*/
1575e-2 /*=15.75*/
-2.5e-3 /*=-0.0025*/
25E-4 /*=0.0025*/
```

浮点常量除非前面加有一个负号(-),否则是一个正数。在这种情况下,负号作为单目算术取反运算符处理。浮点常量具有类型 float、double、long 或 long double。

一个没有 f、F、l 或 L 后缀的浮点常量具有类型 double。如果有 f 或 F 后缀,该常量的类型为 float;如果有字母 l 或 L 后缀,其类型为 long double。例如:

```
100L /*有类型 long double*/
100F /*有类型 float*/
100D /*有类型 double*/
```

注意,Microsoft C 编译器把 long double 映射成类型 double。有关 double、float 和 long 类型的信息参见第 3 章“说明和类型”中的“基本类型存储”。

你可以省略浮点常量的整数部分,如以下例子所示,数值.75 可以采用多种

方式表示,包括如下:

.0075e2

0.075e1

.075e1

75e-2

浮点常量的限制

Microsoft 特殊处→

表 1.2 中给出了浮点常量值的限制,头文件 FLOAT.H 包含这些信息。

表 1.2 浮点常量的限制

常量	含义	值
FLT_DIG	q 个数字的数值,有 q 个十进制	6
DBL_DIG	数字的浮点数可以舍入成一个	15
LDBL_DIG	浮点表示,回过来时不丢失精度	15
FLT_EPSILON	最小正数 x,并且 x+1.0 不等于 1.0	1.192092896e-07F
DBL_EPSILON		2.220446049250313 1e-016
LDBL_EPSILON		2.220446049250313 1e-016
FLT_GUARD		0
FLT_MANT_DIG	在浮点有效数字中以 FLT_RADIX 指	24

续表

DBL_MANT_DIG	定基数的数字的数值。该基数为2;	53
LDBL_MANT_DIG	因此这些值指定位	53
FLT_MAX	最大可表示的浮点数	3.402823466e+38F
DBL_MAX		1.7976931348623158e+308
LDBL_MAX		1.7976931348623158e+308
FLT_MAX_10_EXP	最大整数,10 的这个数的乘方是一	38
DBL_MAX_10_EXP	个可表示的浮点数	308
LDBL_MAX_10_EXP		308
FLT_MAX_EXP	最大整数,FLT_RADIX 的这个数的乘	128
DBL_MAX_EXP	方是一个可表示的浮点数	1024
LDBL_MAX_EXP		1024
FLT_MIN	最小正数	1.175494351-38F
DBL_MIN		2.225073858507201 4e-308
LDBL_MIN		2.225073858507201 4e-308

续表

FLT_MIN_10_EXP	最小负整数,10 的这个数	-37
DBL_MIN_10_EXP	的乘方是一个可表示的浮	-307
LDBL_MIN_10_EXP	点数	-307
FLT_MIN_EXP	最小负整数,FLT_RADIX-	125
DBL_MIN_EXP	的这个数的乘方是一个可表-	1021
LDBL_MIN_EXP	示的浮点数	-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	1(近)
_DBL_ROUNDS	模式	1(近)
_LDBL_ROUNDS		1(近)

注意:表 1.2 中的信息可能在将来有不同的实现。

Microsoft 特殊处结束

整数常量

一“整数常量”是表示一个整数值的十进制(基数为 10)、八进制(基数

为 8)或十六进制(基数为 16)的数。使用整数常量表示不能改变的整数值。

语法

整数常量:

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

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 **长整数后缀** opt

长整数后缀 **无符号后缀** opt

无符号后缀:**如下之一**

u U

长整数后缀:**如下之一**

l L

64 **位整数后缀**:

i64

除非整数常量前面有负号(-),**否则是正数。负号被解释为单目算术取反运算符**(**有关这个运算的信息**,**参见第** 4 **章“表达式和赋值”中的“单目算术运算符”**)。

如果一个整数常量以 0x **或** 0X **字母开头**,**则它是十六进制的**;**如果以数字** 0 **开头**,**则为八进制数**;**否则**,**它假设为十进制数。

下面两行是等价的:

```
0x1C /* 十进制数 28 的十六进制表示 */
034/* 十进制数 28 的八进制表示 */
```

一个整数常量的数字之间不能用空白字符分开。以下例子是有效的十进制、八进制和十六进制常量:

/*十进制常量*/

10

132

32179

/*八进制常量*/

012

0204

076663

/*十六进制常量*/

0xa 或 0xA

0x84

0x7dB3 或 0x7DB3

整数类型

每个整数常量基于它的值和表示方式被给定一个类型。你可以通过在常量的末尾添加字母 l 或 L 来强制任何整数常量为 long 类型。通过添加 u 或 U 来强制它为 unsigned 类型。小写字母 l 可能与数字 1 混淆,应避免这样。几个 long 整数常量的格式如下:

/*长十进制常量*/

10L

79L

/*长八进制常量*/

012L

0115L

/*长十六进制常量*/

0xaL 或 0xAL

0X4fL 或 0x4FL

/*无符号长十进制常量*/

776745UL

778866LU

你赋给一个常量的类型依赖于该常量表示的值。一个常量的值必须在其类型可表示的值的范围内,一个常量的类型确定了当该常量用于一个表达式或应用一个负号(-)时执行的转换。这里列出了整数常量的转换规则:

- 没有后缀的十进制常量的类型有 int、long int 或 unsigned long int。在该常量的值可以表示的三种类型中第一种类型作为赋给该常量的类型。
- 没有后缀的八进制和十六进制常量赋给的类型根据常量的大小可以是 int、 unsigned int、long int 或 unsigned long int。
- 带 u 或 U 后缀的常量赋给的类型根据它们的大小可以是 unsigned int

或 unsigned long int。

- 带 l 或 L 后缀的常量赋给的类型根据它们的尺寸可以是 long int 或 unsigned long int。
- 带 u 或 U 和 l 或 L 后缀的常量赋给的类型是 unsigned long int。

整数限制

Microsoft 特殊处→

表 1.3 列出了整数类型的限制,这些限制定义在标准头文件 LIMITS.H 中。

Microsoft C 还允许说明可变大小的整数变量,可以是大小为 8 位、16 位或 32 位的整数类型,有关可变大小整数的信息,参见第 3 章“说明和类型”中的“可变大小整数类型”。

表 1.3　整数常量的限制

常量	含义	值
CHAR_BIT	不是一个位域的最小变量中的位数	8
SCHAR_MIN	类型 signed char 的变量的最小值	-128
SCHAR_MAX	类型 signed char 的变量的最大值	127
UCHAR_MAX	类型 unsigned char 的变量的最大值	225(0xff)
CHAR_MIN	类型 char 的变量的最小值	-128;如果使用/J 选项则为 0
CHAR_MAX	类型 char 的变量的最大值	127;如果使用/J 选项则为 255
MB_LEN_MAX	多字符常量中的最大字节数	2
SHRT_MIN	类型 short 的变量的最小值	-32768
SHRT_MAX	类型 short 的变量的最大值	32767
USHRT_MAX	类型 unsigned short 的变量的最大值	65535(0xffff)
INT_MIN	类型 int 的变量的最小值	-2147483647-1
INT_MAX	类型 int 的变量的最大值	2147483647
UINT_MAX	类型 unsigned int 的变量的最大值	4294967295(0xffffffff)
LONG_MIN	类型 long 的变量的最小值	-2147483647-1

续表

LONG_MAX	类型 long 的变量的最大值	2147483647
ULONG_MAX	类型 unsigned long 的变量的最大值	4294967295(0xffffffff)

如果一个值超过了最大整数表示,Microsoft 编译器产生一个错误。

Microsoft 特殊处结束

字符常量

一个“字符常量”是由可表示字符集中单个字符由单引号(' ')括起来构成的。字符常量可用来表示执行字符集中的字符。

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(\)或换行符之外的任何成员。

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

字符类型

一个整数字符常量不以字母 L 开头并具有 int 类型。一个包含单个字符的整数字符常量的值是作为一个整数解释的字符的数值。例如,字符 a 的数值是十进制 97 和十六进制 61。
从语法上看,一个“宽字符常量”是一个带前缀字母 L 的字符常量。一个宽字符常量有类型 wchar_t,一个整数类型定义在 STDDEF.H 头文件中,例如:

```
charschar='x';      /*一个字符常量*/
wchar_t wchar=L'x';/*一个同样字符的宽字符常量*/
```

宽字符常量有 16 位宽,指出扩充的执行字符集的成员。它们允许你采用因太大不能由类型 char 表示的字母表示字符。有关宽字符的更多信息,参见本章前面的“多字节和宽字符”。

执行字符集

本书经常提到“执行字符集”。执行字符集不必与编写 C 程序使用的源字符集完全相同,执行字符集包括源字符集中所有字符以及空字符、换行符、回退符、水平制表、垂直制表、回车和转义序列。源字符集和执行字符集在其它实现上可能不同。

转义序列

由一个反向斜杠(\)后跟一个字母或一个数字构成的字符组合称之为“转义序列”。为了表示换行符、单引号或某些其它字符常量,你必须使用转义序列。一个转义序列作为单个字符,因此是一个有效的字符常量。

转义序列一般用于指出动作,如在终端和打印机上的回车和制表移动,它们也用来提供非打印字符和有特定意义的字符,如双引号的文字表示。表 1.4 列出了 ANSI 转义序列及其表示。

注意,反向斜杠之后加一个问号(\?)指出一个文字问号,在这种情况中,该字符序列可能误解释为一个三字母,有关更多的信息,参见本章前面的“三字母”。

表 1.4　转义序列

转义序列	表示
\a	响铃(警告)
\b	回退
\f	换页
\n	换行
\r	回车
\t	水平制表
\v	垂直制表
\'	单引号
\"	双引号
\\	反向斜杠
\?	文字问号
\ooo	八进制表示的 ASCII 字符
\xhhh	十六进制表示的 ASCII 字符

Microsoft 特殊处→

如果一个字符之前的反向斜杠不出现在表 1.4 中,编译器处理该无定义的字符为该字符本身,例如,\x 作为一个 x 处理。

Microsoft 特殊处结束

转义序列允许你把非图形控制字符传送给一个显示设备,例如,ESC 字符

(\033)经常用作一个终端或打印机的控制命令的第一个字符。有些转义序列是设备特定的。例如,垂直制表和换页转义序列(\v 和\f)不能用于屏幕输出,而它们执行适当的打印机操作。

你可以使用反斜杠(\)作为一个续行符,当一个换行符(等价于按 RETURN)后立即跟一个反斜杠(\),编译器便忽略该反斜杠和换行符,把下一行作为前面一行的一部分。这主要用于超过一行的预处理定义,例如:

```
#define assert(exp) \
((exp) ? (void) 0:_assert(#exp,_ _FILE_ _,_ _LINE_ _))
```

八进制和十六进制字符规格

序列\ooo 表示你可以以一个三数字八进制字符代码指定 ASCII 字符集中的任何字符,该八进制整数的数值指定期待的字符或宽字符的值。

类似地,序列\xhhh 允许你以一个十六进制字符代码指定任何一个 ASCII 字符,例如,你可以以正常的 C 转义序列(\b)或\010(八进制)码或\x008(十六进制)码给出 ASCII 回退符。

在一个八进制转义序列中你只能使用 0～7 数字。八进制转义序列不能长于 3 个数字并以第一个不是一个八进制数字的字符结束。虽然你不需要使用所有三个数字,但必须至少使用一个,例如,ASCII 回退符的八进制表示是\10,字母 A 的八进制表示是\101,正如 ASCII 表给出的。

类似地,你必须为十六进制转义序列至少给出一个数字,但可以省略第二个和第三个数字。因此,你可以对回退符指出十六进制转义序列为\x8、\x08 或\x008。

八进制或十六进制转义序列的值必须在这样的可表示范围内,对于字符常量是类型 unsigned char 的可表示值范围;对于宽字符常量是类型 wchar_t 的可表示值范围。有关宽字符常量的信息参见本章前面的“多字节或宽字符”。

不同于八进制转义常量,一个转义序列中十六进制数字的数值是没有限制的,一个十六进制转义序列以第一个不属于十六进制数字的字符终止。因为十六进制数字包括字字母 a 到 f,小心确保该转义序列在指定的数字终止。为了避免混乱,你可以在一个宏定义中放置八进制或十六进制字符定义:

```
#define Bell  '\x07'
```

对于十六进制值,你可以分解该字符串以便清楚地显示正确的值:

```
"\xabc" /*一个字符*/
"\xab" "c" /*两个字符*/
```

字符串文字

一个“字符串文字”是来自源字符集中的字符用双引号(“”)括起来的字符序列。字符串文字用于表示以空格字符结尾的字符串的字符序列,对于宽字符串文字,其前必须加上前缀 L。

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:

源字符集中除双引号(")、反向斜杠(\)或换行符之外的任何成员

转义序列

如下例子是一个简单的字符串文字:

```
char amessage="This is a string literal.";
```

表 1.4 中列出的所有转义码都是有效的字符串文字。为了在一个字符串文字中表示一个双括号,使用转义序列\"。单引号(')没有转义序列也可以表示。反向斜杠(\)在一个字符串中的表示是其后必须跟第二个反向斜杠(\\)。当一个反向斜杠出现在一行的末尾时,它总是解释成一个续行符。

字符串文字的类型

字符串文字具有 char 的类型数组(也就是 char[])(宽字符串具有 wchar_t 的类型数组(也就是 wchar_t[]))。这样意味着一个字符串是一个具有类型 char 元素的数组。该数组中元素成员等于该字符串中字符个数加上 1 即结尾的空格字符。

字符串文字的存储

一个字符串文字的字符顺序存储在相邻的存储器位置。一个字符串文字中的一个转义序列(例如\\或\")都作为单个字符计数。一个空格字符(由\0转义序列表示)自动添加,作为每个字符串文字的结尾标志(这出现在转换阶段 7 中)。注意编译器不可能在不同的地址存储两个相同字符串。/Gf(删除重复字符串)编译器选项强制编译器把相同字符串拷贝放进可执行文件中。

Microsoft 特殊处→

字符串具有静态存储期。有关静态存储器的信息,参见第 3 章说明和类型中的“存储类”。

Microsoft 特殊处结束

字符串文字连接

为了构成占用多行的字符串文字,你可以连接两个字符串。为此,键入一个反向斜杠,然后按 RETURN 键。该反向斜杠导致编译器忽略后面的换行符。例如,字符串文字:

```
"Long strings can be bro\
  ken into two or more pieces."
```

与以下字符串文字是相同的:

```
"Long strings can be broken into two or more pieces."
```

字符串连接可以用于以前采用一个反向斜杠跟一个换行字符来输入多于

一行的字符串文字的任何地方。

为了在一个字符串文字中强制换行,在该字符串中要求分断该行的地方插入一个换行转义序列(\n),如下:

```
"Enter a number between 1 and 100\nOr press return"
```

因为字符串可以在源代码中的任何列开始,并且长字符串可以继续到下一行的任何列,你可以定位该字符串增加源代码的可读性,无论哪种情况输出的屏幕表示不受影响。例如:

```
printf("This is the first half of the string,"
       this is the second half");
```

字符串的每部分都包含在双引号中,各部分连接起来作为单个字符串来输出,这种连接在转换阶段指定的编译中根据事件的序列来执行。

```
"This is the first half of the string,this is the second half"
```

一个字符串指针,初始化为两个由空白分隔的不同字符串文字,而作为单个字符串进行存储(指针在第 3 章“说明和类型”中的“指针说明”中讨论)。适当地引用,正如下面的例子,结果与前面的例子是相同的:

```
char *string="This is the first half of the string,"
             "this is the second half";


 printf("%s",string);
```

在转换阶段 6 中,由任何相邻字符串文字或相邻宽字符串文字序列指定的多字节字符序列都连接成单个多字节字符序列。因此,不要设计这样的程序,允许在执行中修改字符串文字。ANSI C 标准指出修改一个字符

串的结果是不确定的。

最大字符串长度

Microsoft 特殊处→
ANSI 兼容性需要一个编译器在连接之后的一个字符串文字中至多可接受 509 个字符,在 Microsoft C 中允许的一个字符串文字的最大长度接近 2048 个字节。如果字符串文字由包含在双引号中的各部分组成,预处理器把这些部分连接成单个字符串,对于每个连接的行,还增加一个额外字节到字节总数中。
例如,假设一个字符串由 40 行组成,每行 50 个字符(共有 2000 个字符),另有一个 7 个字符的行,每行用双引号括起来,这样 2007 个字节加上结尾空格字符的一个字节,总共 2008 个字节。在连接时,每行增加一个额外字符到第一行,这样总共有 2048 个字节。注意,如果用续行符(\)代替双引号,预处理器不会为每行都增加一个额外的字符。
Microsoft 特殊处结束

标点和特殊字符

在 C 字符集中标点和特殊字符有各种用途,从组织程序文本到定义编译器或编译的程序的执行功能。它们不指定执行的操作。有些标点符号也是运算符(参见第 4 章“表达式和赋值”中的“运算符”)。编译器从上

下文确定它们的用途。

语法

标点:如下之一

[ ] ( ) { } * , : = ; ... #

这些字符在 C 中具有特定含义,它们的作用在本书中描述。#号(#)仅在预处理命令中出现。

第 2 章　程 序 结 构

本章给出 C 程序和程序执行的一个概述,也介绍了理解 C 程序和组成成分的重要术语和特征。讨论的主题包括:

- 源文件和源程序
- main 函数和程序执行
- 命令行参量分析
- 生存期、范围、可见性和连接
- 名称空间

因为本章是概述,讨论的主题仅包括一些引言材料,有关详细解释参见交叉引用的信息。

源文件和源程序

一个源程序可以分成一个或多个“源文件”或“转换单元”。输入到编译器被称为一个“转换单元”。

语法

转换单元:

　　外部说明

转换单元 外部说明

外部说明:

函数定义

说明

第 3 章“说明和类型”中的“说明概述”给出了说明非终结符的语法,本卷后面的“预处理器参考”解释了如何处理转换单元。

注意:有关 ANSI 语法规则的解释,参见本卷后面的附录 A“C 语言语法总结”的引言。

一个转换单元的组成成分是包括函数定义和标识符说明的外部说明。这些说明和定义可以在源文件、头文件、库和该程序需要的其它文件中,你必须编译每个转换单元并链接产生的目标文件来生成一个程序。

一个 C“源程序”是命令、编译指示、说明、定义、语句块和函数组成的,为了使一个 Microsoft C 程序生效,每一个组成成分必须具有本书中描述的语法,虽然它们可以以任何次序放在程序中(服从本书描绘的规则)。但这些组成成分在一个程序中的位置不影响一个程序中使用的变量和函数(有关更多信息,参见本章后面的“生存期、范围、可见性和连接”) 源文件不需要包含可执行语句,例如,你可能发现这种方式是有用的,在一个源文件中放置变量的定义,然后在另一个源文件说明引用这些变量。这种技术很容易找到和修改定义,同样的理由,常量和宏经常组成称为“包括文件”或“头文件”的分开文件中,需要时在源文件中引用它们,有关宏和包括文件的信息,参见本卷后面的“预处理器参考”。

预处理器的命令

一个“命令”指示 C 预处理器在编译之前对程序文本执行特定动作。预处理器命令在本卷后面的“预处理器参考”中进行充分的描述,本例使用预处理器命令 #define:

```
#define MAX 100
```

这个语句告诉编译器在编辑之前将 MAX 的每次出现均用 100 替换。C 编译器预处理器命令是:

#define	#endif	#ifdef	#line
#elif	#error	#ifndef	#pragma
#else	#if	#include	#undef

编译指示

Microsoft 特殊处→

一个“编译指示”指示编译器在编译时执行一个特殊动作。编译指示从编译器到编译器时发生改变。例如,你可以使用 optimize 编译指示设置你的程序要执行的优化。Microsoft C编译指示有:

alloc_text	data_seg	inline_recurs ion	setlocale
auto_inline	function	intrinsic	warning
check_stack	hdrstop	message	
code_seg	include_alias	ptimize	
comment	inline_depth	pack	

有关 Microsoft C 编译器编译指示的描述参见本卷后面“预处理器参考”中的第 2 章“编译指示命令”。
Microsoft 特殊处结束

说明和定义

一个“说明”在一个特定变量、函数或类型及其属性之间建立关联。第 3 章“说明和类型”中的“说明概述”给出了说明非终结符的 ANSI 语法,一个说明也指出访问一个标识符的地方和时间(标识符的“连接”)。有关连接的信息参见本章后面的“生存期、范围、可见性和连接”。
一个变量的“定义”建立与说明相同的关联,它还导致分配该变量的存储。
例如,main、find 和 count 函数以及 var 和 val 变量都依次定义在一个源文件中:

```
void main()
{
}

 int var=0;
 double val[MAXVAL];

 char find(fileptr)
```

```
{
}

int count(double f)
{
}
```

变量 var 和 val 用于 find 和 count 函数中,不需要进一步的说明,但它们的名字在 main 中是不可见的(不能访问)。

函数说明和定义

函数原型建立该函数的名称、其返回值、它的形参的类型和个数。一个函数定义包括函数体。

函数和变量说明可以出现在一个函数定义的内部或外部。一个函数定义中的任何说明称为“内部的”或“局部的”。所有函数定义外部的说明称为“外部的”、“全局的”或“文件范围”。变量定义像说明一样,可以在内部层(一个函数定义内部)或外部层(所有函数的外部)中进行。函数定义总是出现在外部层。函数定义的进一步讨论在第 6 章“函数”的“函数定义”中。函数原型在第 6 章的“函数原型”中讨论。

块

用花括号({})括起来的说明、定义和语句序列称为一个“块”。C 中有两种类型的块。复合语句是一个或多个语句组成的语句(参见第 5 章“语

句”中的“复合语句”),它是一种类型的块。另一种类型的块是“函数定义”,它由一个复合语句(函数体)加上该函数关联的“头”(该函数名称、返回类型和形参)。一个块中的另一个块称之为“嵌套”。
注意,所有复合语句都是用花括号括起来的,但并不是花括号中的任何东西都组成一个复合语句,例如,数组、结构或枚举元素的规格都出现在花括号中,但它们不是复合语句。

例子程序

如下 C 源程序由两个源文件组成,它给出一个 C 程序中各种可能的说明和定义的概述。本书后面的章节讨论如何编写这些说明、定义和初始化,以及如何使用 C 关键词如 static 和 extern。printf 函数在 C 头文件 STDIO.H 中说明。
main 和 max 函数假设在分开的文件中,程序的执行从 main 函数开始。在 main 之前不会显式执行用户函数。

```
/*********************************************************
                    FILE1.C - main function
*********************************************************/
#define ONE 1
#define TWO 2
#define THREE 3
#include <stdio.h>
```

```
int a=1;                                    /*定义外部变量的说明        */
int b=2;

extern int max(int a,int b);        /*函数原型                        */

int main()                                  /*main 函数的函数定义       */
{
      int c;                                /*两个未初始化的局部变量的定义*/
      int d;
      extern  int  u;                       /*重新引用其它地方定义的外部变量的说明
*/

      static int v;                         /*定义具有连续生存期的变量                  */

      int w=ONE,x=TWO,y=THREE;
      int z=0;
      z=max(x,y);                           /*可执行的语句*/
      w=max(z,w);
      printf("%d %d\n",z,w);
      return 0;
}
/*************************************************************
```

```
                    FILE2.C - defination of max function
**********************************************************************/
int max(int a,int b)                    /* 注意形式参数包括在函数头中      */
{
        if (a>b)
            return(a);
        else
            return(b);
}
```

FILE1.C 包含 max 函数的原型,这种说明有时称为“前向说明”,因为在该函数使用之前说明。main 函数的定义包括调用 max。

以#define 开头的行是预处理器命令。这些命令告诉编译器在 FILE1.C 中分别用 1、2 和 3 替换标识符 ONE、TWO 和 THREE。以#include 开头的行告诉编译器包括文件 STDIO.H,它包含 printf 函数的原型。预处理器命令在本卷后面的“预处理器参考”中解释。

FILE1.C 使用定义说明来初始化全局变量 a 和 b,局部变量 c 和 d 说明了但未初始化。所有这些变量都分配了存储。静态和外部变量 u 和 v 都自动初始化为 0。因此在说明时只有 a、b、u 和 v 包含有意义的值,因为它们显式或隐含地初始化。F1LE2.C 包含 max 的函数定义。这个定义满足在 FILE1.C 中调用 max。

标识符的生存期和可见性在本章后面的“生存期、范围、可见性和连接”中讨论。有关函数的更多信息,参见第 6 章“函数”。

main 函数和程序执行

每个 C 程序有一个必须命名为 main 的主(main)函数。如果你的代码服从单代码程序设计模式,你可以使用 main 的宽字符版本 wmain。main 函数作为程序执行的起始点。通常通过在程序中调用其它函数来控制程序的执行。一个程序执行到 main 的末尾会终止,但它可以由于各种原因在其它程序的地方终止。这时可能检测到某个错误,你要强制一个程序终止,为此,要使用 exit 函数。有关使用 exit 函数的例子和信息参见“ Microsoft Visual C++ 6.0 参考库 ”的“ Microsoft Visual C++ 6.0 运行库参考 ”卷。
在源程序中的函数实现一个或多个特定功能。main 函数调用这些函数实现它们的各自功能。当 main 调用另一函数时,它把执行控制传给该函数,因此从该函数的第一个语句开始执行。当执行 return 语句或到达该函数的末尾时,把控制返回给 main。
你可以说明任何函数包括 main 所具有的参数。术语“ 参数 ”或“ 形参 ”指的是接受一个传给该函数的值的标识符。有关传送参量给参数的信息参见第 6 章“ 函数 ”中的“ 参数 ”。当一个函数调用另一个函数时,被调用函数从调用函数获取它的参数值,这些值称为“ 参量 ”。你可为 main 说明形参,以便它使用如下格式从命令行接受参量:

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

当你要传送信息给 main 时,参数习惯上命名为 *argc* 和 *argv*,尽管 C 编译器不要求这些名称。*argc* 和 *argv* 由 C 语言定义。如果传给 main 第三个参数,习惯上命名这个参数为 *envp*。本章后面的例子说明了如何使用这

三个参数访问命令行参量。下面小节解释这些参数。
有关 main 宽字符版本的描述参见下一节“使用 wmain”。

使用 wmain

Microsoft 特殊处→
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。如果你要编写服从单代码程序设计模式的可移植代码,则使用 wmain 代替 main。
你可以使用类似于 main 的格式说明 wmain 的形式参数,然后传送宽字符参量和任选的宽字符环境指针给该程序。wmain 的 *argv* 和 *envp* 参数是 wchar_t * 类型,例如:

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

如果你的程序使用一个 main 函数,通过运行库在启动程序时建立多字节字符环境。当需要时(例如调用 _wgetenv 或 _wputenv 函数)只建立该环境的一个宽字符拷贝。如果一个 MBCS 环境已经存在时,第一次调用 _wputenv 或第一次调用 _wgetenv 时,则建立对应的宽字符串环境,然后由 _wenviron 全局变量指向它,该变量是 _environ 全局变量的宽字符版本。在这时,该环境的两个拷贝(MBCS 和单代码)同时存在,并且由操作系统在该程序生存期中进行维护。
类似地,如果你的程序使用一个 wmain 函数,在程序启动时建立一个宽字符环境并由 _wenviron 全局变量指向它。在第一次调用 _putenv 或 getenv 时建立一个 MBCS(ASCII)环境,并由 _environ 全局变量指向它。有关

MBCS 环境更多的信息,参见“ Microsoft Visual C++参考库 ”的“ Microsoft Visual C++6.0 运行库参考 ”卷。

Microsoft 特殊处结束

参量描述

main 和 wmain 函数的 *argc* 参数是一个整数,指出从命令行传给该程序的参量个数,由于程序名称也作为一个参量考虑,*argc* 的值至少为 1。

argv 参数是一个表示程序参量的以空格结尾的字符串的指针数组。该数组的每个元素指向一个传给 main(或 wmain)的参量的字符串(有关数组的信息,参见第 3 章“ 说明和类型 ”中的“ 数组说明 ”)。*argv* 参数可以作为一个类型 char 的指针数组(char *argv[])或者类型 char 的指针的指针(char **argv)来说明。对于 wmain,*argv* 参数可以作为类型 wchar_t 的指针数组(wchar_t *argv[])或者类型 wchar_t 的指针的指针(wchar_t **argv)来说明。第一个字符串(argv[0])是程序名称,最后指针(argv[argc])为 NULL(有关获取环境变量信息的另一种方法参见“ Microsoft Visual C++6.0 运行库参考 ”中的“ getenv ”)。

Microsoft 特殊处→

envp 参数是表示用户环境变量中值设置的以空格结尾的字符串的数组的指针。*envp* 参数可以说明为 char 的一个指针数组(char *envp[])或 char 的指针的指针(char **envp)。在 wmain 函数中,*envp* 参数可以说明为 wchar_t 的指针数组(wchar_t *invp[])或 wchar_t 的指针的指针(wchar_t **envp)。该数组的末尾由 NULL *指针指向。注意,传给 main 或 wmain 的环境块是当前环境的一个“ 冻

结的”拷贝。如果你后面经由调用_putenv 或_wputenv 改变该环境,当前环境(由getenv/_wgetenv 和_environ 或_wenviron 变量返回)将改变,但由 *envp* 所指的块不改变。*envp* 参量在 C 中是 ANSI 兼容的,但在 C++中不是。
Microsoft 特殊处结束

扩充通配符参量

Microsoft 特殊处→
当运行一个 C 程序时,你可以使用两个通配符即问号(?)和星号(*)指出命令行上的文件名和路径参量。
命令行参量由一个称为_setargv(在宽字符环境中称为_wsetargv)的例程处理,它缺省地不扩充 argv 字符串数组中的通配符为分离的字符串。你可以使用一个功能更强大的版本_setargv 通过与 setargv.obj 文件链接处理通配符来代替正常的_setargv。如果你的程序使用一个 wmain 函数,则与 wsetargv.obj 链接。
为了与 setargv.obj 或 wsetargv.obj 链接,使用/link 选项,例如:

```
cl typeit.c /link setargv.obj
```

该通配符与操作系统命令中的同样方式进行扩充(如果你不熟悉该通配符,参见你的操作系统用户指南)。在双引号中括起一个参量阻止通配符扩充。在一个括起的参量中,你可以通过在双引号之前加一个反向斜杠(\)来表示双引号文字。
如果该通配符参量没有找到匹配的,该参量作为文字传送。
Microsoft 特殊处结束

命令行参量的分析

Microsoft 特殊处→

在解释操作系统命令行给定的参量时，Microsoft C 启动代码使用如下规则：

- 由空白定界参量，它是一个空格或制表。
- 由双引号括起的一个字符串作为单个参量解释，忽略其中包含的空白。一个括起的字符串可以嵌入在一个参量中。注意，插入记号(^)不作为一个转义字符或定界符识别。
- 双引号前加一个反向斜杠(\)解释为一个文字双引号(")。
- 反向斜杠作为文字解释的，除非它们直接跟一个双引号。
- 如果偶数个反向斜杠后跟一个双引号，那么 *argv* 数组中的每对反向斜杠(\\)用一个反向斜杠代替，且将双引号(")解释为一个字符定界符。
- 如果奇数个反向斜杠后跟一个双引号，那么和 argv 数组中每对反向斜杠(\\)用一个反向斜杠代替，双引号被解释为剩下的一个反向斜杠的转义序列，导致在 *argv* 中放置一个文字双引号(")。

如下表说明了传递给 argv 几个命令行参量例子的解释结果，列出的第 2 列、第 3 列和第 4 列的结果来自后面的 ARGS.C 程序的输出。

命令行输出	argv[1]	argv[2]	argv[3]
"a b c" d e	a b c	d	e
"ab\"c" "\\" d	ab"c	\	d

```
 a\\\b d"e f"g h a|||bde  h
 fg
 a\\\"b c d                    a\"b          c          d
 a\\\\"b c" d e                a\\b c         d         e
/*ARGS.C 说明了如下用于访问命令行参量和环境变量:
 * argc argvenvp
 *的变量
 */
#include <stdio.h>

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

        /* 显示每个命令行参量*/
        printf("\nCommand-line arguments:\n");
        for (count=0;count<argc;count++);
              printf("argv[%d] %s\n",count,argv[count]);

        /*显示每个环境变量*/
```

```
        printf("\nEnvironment variables:\n");
        while (*envp!=NULL)
              printf(" %s\n",*(envp++));

        return;
}
```

这个程序的一个输出例子如下:

```
Command-line arguments:
      argv[0] C:\MSC\TEST.EXE

Environment variables:
      COMSPEC=C:\NT\SYSTEM32\CMD.EXE

  PATH=c:\nt;c:\binb;c:\binr;c:\nt;c:\system32;c:\word;c:\help;c:\msc;c:\;
  PROMPT=[$p]
  TEMP=c:\tmp
  TMP=c:\tmp
  EDITORS=c:\binr
  WINDIR=c:\nt
```

Microsoft 特殊处结束

定制命令行处理

如果你的程序不用命令行参量,你可以通过阻止使用实现命令行处理的例程来节省一些空间,这个例程称为_setargv(在宽字符环境中为_wsetargv),正如本章前面“扩充通配符参量”中描述的。为了阻止它的使用,在包含 main 函数的文件中定义一个不做任何事情的例程,也取名为_setargv(在宽字符环境中为_wsetargv)。那么调用_setargv 或_wsetargv 满足你定义的_setargv 或_westargv 而不加载它们的库版本。

类似地,如果你不通过 *envp* 参量访问该环境表,可以提供自己的空例程放在环境处理例程_setenvp(或_wsetenvp)中。

如果你的程序调用 C 运行库中的_spawn 或_exec 簇的例程,你不要阻止该环境处理例程,因为这个例程用于从繁育进程传递一个环境给新进程。

生存期、范围、可见性和连接

为了理解 C 程序如何工作的,你必须掌握程序中使用变量和函数的规则。几个概念对于掌握这些规则是重要的:

- 生存期
- 范围和可见性
- 连接

生存期

“生存期”是一个程序执行期间一个变量或函数存在的周期。标识符的存储期间确定了它的生存期。

用存储类指示符 static 说明的标识具有静态存储期。静态存储期(也称为“全局”)的标识符在一个程序期间具有存储和确定的值。在程序启动之前仅一次初始化就确定了存储和值。以外部或内部连接说明的标识符也具有静态存储期(参见本章后面的“连接”)。

不以 static 存储类指示符说明的标识符如果在一个函数内部说明,它具有自动存储期,具有自动存储期的标识符(一个局部的标识符)仅在说明或定义该标识符的块中具有存储和确定的值,一个自动标识符在程序每次进入该块时分配新的存储,当程序退出该块时会丢失它的存储(和它的值)。在一个函数中以非连接说明的标识符也具有自动存储期。

如下规则指出一个标识符是否具有全局(静态的)或局部的(自动的)生存期:

- 所有函数都具有静态生存期。因此它们存在于程序执行期间的任何时候。在外部层说明的标识符(也就是,程序中函数定义同层的所有块的外部)总是全局(静态)生存期。
- 如果一个局部变量有一个初始化器,每次建立该变量时都初始化它(除非它说明成 static)。函数参数也有局部生存期,你可以在一个块中通过在说明中包括 static 存储类指示符来为一个标识符指定全局生存期。一旦说明成 static,该变量保留它从该块的一个入口

到下一个入口的值。

虽然具有全局生存期的标识符在源程序的整个执行中都存在(例如,一个外部说明的变量或用 static 关键字说明的局部变量),它可能在程序的所有部分都是不可见的。有关可见性的信息参见“范围和可见性”。有关存储类指示符非终结符的讨论参见第 3 章“说明和类型”的中“存储类”。如果需要分配存储器(动态的),通过使用特定库例程如 malloc 来实现。既然动态存储器分配使用库例程,它不作为语言的一部分。参见“Microsoft Visual C++ 6.0 运行库参考”中的 malloc 函数。

范围和可见性

一个标识符的“可见性”确定了可以引用的程序部分即它的“范围”。一个标识符仅在它的“范围”所包括的程序部分是可见的(也就是可以使用的),它对包含它的文件、函数、块或函数原型可以是有限制的(以便增大限制)。一个标识符的范围是可以使用它的名称的程序部分。有时称之为“词法范围”。有四种类型的范围:函数、文件、块和函数原型。除标号外所有标识符都通过其说明所在层来确定它的范围。如下每一种类型的规则确定了一个程序中标识符的可见性:

文件范围

具有文件范围的说明或类型指示符出现在任何块或参数表的外面,可以在说明之后转换单元的任何位置访问。具有文件范围的标识符名称通常称为“全局的”或“外部的”。一个全局标识符的范围开始于它的定义或说明的地方,终止于该转换单位的末尾。

函数范围

一个标号是唯一具有函数范围的标识符。一个标号通过它使用的语句隐含地说明了。在一个函数中标号名称必须是唯一的(有关标号和标号名称的更多信息,参见第 5 章“语句”中的“goto 和标号语句”)。

块范围

具有块范围的标识符的说明或类型指示符出现在一个块内或一个函数定义的形式参数的说明表中。它仅在它说明和定义的地方到包含其定义或说明的块的末尾的范围内是可见的。它的范围限制为该块和该块嵌套调用的块到关闭该关联块的花括号。这样的标识符有时称为“局部变量”。

函数原型范围

具有函数原型范围的标识符的说明或类型指示符出现在一个函数原型中的参数说明表中(不是该函数的说明部分)。它的范围在该函数说明的末尾。

在其它源文件中的变量可见性的适当说明在第 3 章“说明和类型”中的“存储类”中描述。在外部层用 static 存储类指示符说明的变量和函数仅在定义它们的源文件中是可见的,所有其它函数是全局可见的。

生存期和可见性总结

表 2.1 是大多数标识符的生存期和可见性特性的总结。开头三列给出定义生存期和可见性的属性,具有开头三列属性的标识符的生存期和可见性在第 4 列和第 5 列中给出。但该表不包括所有可能的情况。有关更多信息参见第 3 章“说明和类型”中的“存储类”。

表 2.1　生存期和可见性总结

属性：				结果：
层	项	存储类指示符	生存期	可见性
文件范围	变量定义	static	全局的	出现它的源文件中的剩余部分
	变量说明	extern	全局的	在出现它的源文件中的剩余部分
	函数原型或定义	static	全局的	单个源文件
	函数原型	extern	全局的	源文件的剩余部分
块范围	变量说明	extern	全局的	块
	变量定义	static	全局的	块
	变量定义	auto 或 register	局部的	块

如下例子说明了块、嵌套和变量的可见性：

```
#include <stdio.h>
int i=1;     /*i 定义在外部层                                    */
int main() /*main 定义在外部层                                   */
{
        printf("%d\n",i);/*打印 1(外部层 i 的值)                  */
        {                          /* 开始第一个嵌套的块          */
```

```
        int i=2,j=3;                /* i 和 j 定义在内部层          */
        printf("%d %d\n",i,j); /*  打印 2,3                    */
    {                                  /*  开始第二个嵌套的块          */
            int i=0;                /* i 重新定义                    */
     printf("%d %d\n",i,j);      /*  打印 0,3                    */
     }                                 /*  结束第二个嵌套的块          */
     printf("%d\n",i);             /*  打印 2(恢复外面的定义)      */
     }                                 /*  结束第一个嵌套的块          */
     printf("%d\n",i);             /*  打印 1(恢复外层定义)        */

     return 0;
}
```

本例中,有四层可见性:外部层和三个块层。在屏幕上打印的值在每行的后面都有注释。

连　接

标识符的名称在不同的范围可以指不同的标识符,一个标识符在不同范围内说明或者在同一范围内说明多次可以通过称为“连接”的过程使之指向相同的标识符或函数。连接确定其中标识符可以引用的程序部分(它的“可见性”)。有三种类型的连接:内部的、外部的和无连接。

内部连接

如果一个对象或一个函数的一个文件范围标识符的说明包含存储类指示符 static，该标识符具有内部连接。否则,该标识符有外部连接。有关存储类标识符非终结符的讨论,参见第 3 章“说明和类型”中的“存储类”。在一个转换单元中,具有内部连接的标识符的每个实例表示同样的标识符或函数。内部连接的标识符对于一个转换单元是唯一的。

外部连接

如果在文件范围层的标识符的第一次说明没有使用 static 存储类指示符,该对象具有外部连接。
如果一个函数的一个标识符的说明没有存储类指示符。它的连接好象用存储类指示符 extern 说明来确定。如果一个对象的标识符说明具有文件范围且没有存储类指示符,则它的连接是外部的。
一个具有外部连接的标识符的名称与其它任何相同名称的外部连接的说明表示相同的函数或数据对象，这两个说明可能在同一转换单元中或不同的转换单元中。如果该对象或函数还具有全局生存期,那么该对象或程序由整个程序共享。

无连接的

如果在一个块中的标识符说明没有包括 extern 存储类指示符,该标识符是无连接的,对于这个函数是唯一的。
如下标识符是无连接的:

- 一个标识符说明成非对象或函数的其它任何事情。
- 一个标识符说明成一个函数参数。
- 一个对象的块范围标识符不以 extern 存储类指示符说明。

如果一个标识符是无连接的,在同一范围层的相同名称的多次说明(在一个说明或类型指示符中)会产生一个符号重新定义错误。

名称空间

编译器建立“名称空间”用于识别用于各种不同项的标识符。在每个名称空间中的名称必须是唯一的以避免冲突,但相同的标识符可以出现在多个名称空间中。这意味着你可以为两个或多个不同的项使用相同的标识符,提供的项在不同的名称空间中。编译器基于程序中标识符的句法上下文来解决引用。

注意:不要将有限制的 C 的名称空间表示与 C++“名称空间”特征相混淆。有关更多信息参见本卷后面“Microsoft Visual C++ 6.0 语言参考”中的第 6 章“说明”的“名称空间”。

下面列出了 C 中使用的名称空间的描述:

语句标号

命名的语句标号是语句部分。语句标号的定义总是跟着一个冒号但不是一个 case 标号部分。语句标号的使用总是跟在 goto 关键词的后面。语名标号不要与其它名称或其它函数的标号名称进行区分。

结构、联合和枚举标志

这些标志是结构、联合和枚举类型标识符的一部分,如果出现,总是直接跟随保留字 struct、union 或 enum。该标志名称必须与其它相同可见性的结构、枚举或联合标志相区别。

结构或联合的成员

在名称空间中分配与每个结构和联合类型相关联的成员名称,也就是,相同的标识符可以同时是不同结构或联合的成员的组成成分名称。组成成分的定义总是出现在结构或联合类型指示符中。组成成分名称的使用总是直接跟随成员选择运算符(->和.)。在结构或联合中,一个成员名称必须是唯一的,但为了与其它名称区别,要包含不同结构和联合的成员的名称或结构本身的名称。

普通标识符

名称空间中的所有其它名称包括所有变量、函数(包括形式参数和局部变量)和枚举常量。标识符名称具有嵌套的可见性,因此你可以在块中重新定义它们。

typedef 标识符

类型定义(typedef)名称不能在同一范围内用作标识符。

例如,因为结构标志、结构成员和变量名称在三种不同的名称空间中,本例中三个项都命名为 student 不会有冲突 。每个项的上下文允许正确地解释程序中 student 的每次出现(有关结构的信息,参见第 3 章“说明和类型”中的“结构说明”):

```
struct student {
    char strdent [20]
```

```
int class;
int id;
} student;
```

当 student 出现在 struct 关键字之后时,编译器识别它为结构标志。当 student 出现在一个成员选择地运算符(->或.)之后时,该名称指的是结构成员。在其它上下文中,student 指的是结构变量。但为了不造成模糊,建议不重载该标志名称空间。

第 3 章　说明和类型

本章描述变量、函数和类型的说明与初始化。C 语言包括一个标准数据类型的标准集。
你还可以增加自己的数据类型,称为“派生的类型”,即通过已经定义的类型说明新的类型。
讨论的主题如下:

- 说明概述
- 存储类
- 指示符
- 类型修饰符
- 说明符和变量说明
- 解释复杂说明
- 初始化
- 基本类型的存储
- 不完整的类型
- 类型定义说明
- 扩展存储类属性

说明概述

一个“说明“指出一组标识符的解释和属性。
导致为标识符的对象或函数保留存储的说明称之为“定义”。变量、函数和类型的 C 说明具有这样的语法:
语法
说明:
    说明指示符 初始化说明符表 $_{opt}$
说明指示符:
    存储类指示符 属性序列 $_{opt}$ 说明指示符 $_{opt}$
      /*属性序列 $_{opt}$ 是 Microsoft 特殊处*/
    类型指示符 说明指示符 $_{opt}$
    类型修饰符 说明指示符 $_{opt}$
初始化说明符表:
    初始化说明符
    初始化说明符表 初始化说明符
初始化说明符:
    说明符
    说明符=初始化器
注意:说明的语法在后面不再重复。后面使用的语句通常以说明符非终结符开头。
初始化说明符表中的说明包含命名的标识符;初始化是初始化器的简

称。初始化说明符表是用逗号分隔的说明符序列,每个都有附加类型信息或一个初始化器或两者都有。说明符包含任何说明的标识符。说明指示符非终结符由类型序列和指出连接、存储期的存储类指示符以及说明符表示的实体类型最小的部分组成。因此,说明由存储类指示符、类型指示符、类型修饰符、说明符和初始化器的某种组合组成。

说明可以包含属性序列中列出的一个或多个任选属性。这些 Microsoft 特定属性实现函数的变型,这在本书中详细讨论,对于属性列表,参见本卷后面的附录 A“C 语言语法总结”。

在一个变量说明的一般格式中,类型指示符给出该变量的数据类型。类型指示符可以是复合的,可以用 const 或 volatile 修饰。说明符给出了变量名称,说明一个数组或指针类型的修饰。例如:

```
int const *fp;
```

说明一个名称为 fp 的变量为不能修改的(const)int 值的指针。你可以使用多个以逗号分隔的说明符在一个说明中定义一个以上的变量。

一个说明必须至少有一个说明符,或者它的类型指示符必须说明一个结构标志、联合标志或一个枚举器的成员。一个说明符是一个这样的标识符,可以分别用方括号([])、星号(*)或圆括号(())修饰来说明一个数组、指针或函数。当你说明简单变量(例如字符、整数和浮点项)或一些简单变量的结构和联合时,该说明符也是一个标识符。有关更多信息,参见本章后面的“说明符和变量说明”。

所有定义都隐含地说明,但不是所有说明都是定义。例如,以 extern 存储类指示符开头的说明是“引用”,而不是“定义”说明。如果外部变量

在其定义之前被引用,或者如果它定义在另一个使用的源文件中,必须使用一个 extern 说明。“引用”说明不分配存储,在说明中变量也不能初始化。

在变量说明中需要一个存储类或一个类型(或两者都需要)。除了 __declspec 外,在一个说明中只允许有一个存储类指标符,不是所有的存储类指示符都允许在每个上下文中。

__declspec 存储类允许与其它存储类指示符一起使用,允许使用多次。一个说明的存储类指示符影响被说明的项是如何存储和初始化的、以及程序的哪部分可以引用该项。

定义在 C 中的存储类指示符终结符包括 auto、extern、register、static 和 typedef。另外,Microsoft C 包括存储类指示符终结符 __declspec。所有除 typedef 和 __declspec 外的存储类指示符终结符都在“存储类”中讨论。有关 typedef 的信息参见本章后面的“typedef 说明”。有关 __declspec 的信息参见“扩充的存储类属性”。

在一个源程序中说明的位置以及该变量是否存在其它说明对于确定变量的生存期是重要的因素。可以有多个重新说明,但只有一个定义。但一个定义可以出现在多个转换单元中。对于内部连接的对象,这个规则分别应用于每个转换单元,因为内部连接的对象对一个转换单元来说是唯一的。对于外部连接的对象,这个规则应用于整个程序。有关可见性的更多信息,参见第 2 章的“生存期、范围、可见性和连接”。

类型指示符提供了一些标识符数据类型的信息,缺省类型指示符是 int。有关更多信息,参见本章后面的“类型指示符”。类型指示符也可以定义

类型标志、结构和联合组成成分名称以及枚举常量。有关更多信息,参见本章后面的“枚举说明”、“结构说明”和“联合说明”。

有两个类型修饰符终结符:const 和 volatile。这些修饰符指出另外的类型特性,它们仅与访问通过 l 值确定类型的对象有关。有关 const 和 volatile 的更多信息,参见本章后面的“类型修饰符”。有关 l 值的定义,参见第 4 章“表达式和赋值”中的“L 值和 R 值表达式”。

存储类

一个变量的“存储类”确定该项是“全局的”生存期还是“局部的”生存期。C 调用这两个生存期:“static”和“automatic”。一个具有全局生存期的项在整个程序执行中都存在并有一个值。所有函数都有全局生存期。

自动变量或具有局部生存期的变量,在每次把控制传给定义它们的块时都分配新的存储。当执行返回时,该变量不再具有有意义的值。

C 提供如下存储类指示符:

语法

存储类指示符:

auto
register
static
extern

typedef
__declspec(扩充的说明修饰符序列)/*Microsoft 特定的*/
除了 __declspec 外,你在一个说明的说明指示符中只能使用一个存储类指示符。如果没有存储类指示符,在一个块中的说明建立自动对象。
用 auto 或 register 指示符说明的项具有局部生存期。用 static 或 extern 指示符说明的项具有全局生存期。
因为 typedef 和__declspec 语义上不同于其它 4 个存储类指示符终结符,把它们分开进行讨论。有关 typedef 的特定信息,参见“typedef 说明”。有关
__declspec 的特定信息,参见“扩充存储类属性”。
变量和函数说明在源文件中的位置也影响存储类和可见性。所有函数定义外面的说明被说成出现在“外部层”。函数定义内部的说明出现在“内部层”。
每个存储类指示符的精确含义依赖于两个因素:
该说明出现在外部层还是内部层。
该项是说明成变量还是函数。
下面两节“外部层说明的存储类指示符”和“内部层说明的存储类指示符”描述了每种说明中的存储类指示符终结符并解释了从变量中忽略存储类指示符的缺省行为。“函数说明的存储类指示符”讨论了在函数中使用存储类指示符。

外部层说明的存储类指示符

外部变量是文件范围的变量,它们定义在任何函数的外面,对于很多函数是可用的。函数只能在外部层定义,因此,不能嵌套。缺省地,所有同名称的外部变量和函数的引用指向同一对象,这意味着它们具有“外部连接”(你可以使用 static 关键词覆盖它,有关 static 的详细情况参见本章后面的信息)。

在外部层的变量说明是变量定义(“定义说明”)或其它地方定义的变量的引用(“引用说明”)。

一个初始化变量的外部变量说明(隐含或显式地)是该变量的一个定义说明。在外部层的定义可以有几种格式:

- 你用 static 存储类指示符说明的变量。你可以用一个常量表达式显式地初始化该 static 变量,正如“初始化”中描述的。如果你省略该初始化器,该变量缺省初始化为 0,例如,这里有两个定义变量 k 的语句。

  ```
  static int k=16
  static int k;
  ```

- 一个你在外部层显式初始化的变量。例如,int j=3;是变量 j 的一个定义。

在外部层说明变量时(也就是在所有函数的外面),你可以使用 static 或 extern 存储类指示符或整个地省略该存储类指示符。你不能在外部层使用 auto 和 register 存储类指示符终结符。

一旦一个变量在外部层定义了,它在余下的转换单元中是可见的。该变量在同一个源文件中其说明的前面是不可见的。在该程序的其它源文件中也是不可见的,除非有一个引用说明使它成为可见的,正如下面说明的。

static 的相关规则包括:

- 所有块外面的以不带 static 关键字方式说明的变量总是在整个程序中保留它们的值。为了限制它们对特定转换单元的访问,你必须使用 static 关键字,这给它们“内部连接”。为了使它们对整个程序而言是全局的,省略显式存储类或使用关键字 extern(参见下面表中的规则),这给它们“外部连接”。内部和外部连接在第 2 章“程序结构”中的“连接”中讨论。
- 你可以在一个程序中的外部层一个变量仅定义一次,可以在不同转换单元中用相同的名称和 static 存储类指示符定义另外的变量。由于每个 static 定义仅在它自己的转换单元中可见,因此不会出现冲突。这提供了这样的标识符名称的有用方式,该标识符在一个转换单元的函数中实现共享,但对其它转换单元是不可见的。
- static 存储类指示符也可以应用于函数。如果说明一个函数为 static,它的名称在被说明的文件之外是不可见的。

使用 extern 的规则是:

- extern 存储类指示符说明一个其它地方说明的变量的引用,你可以使用一个 extern 说明使其它源文件中的定义是可见的,或者使一个变量在相同源文件中其定义的前面是可见的。一旦你在外部层说明

了一个变量的引用,该变量在说明引用出现的转换单元的余下部分都是可见的。

- 对于一个有效的 extern 引用,所指的变量必须定义一次,而且在外部层仅一次。这个定义 (没有 extern 存储类)可以在任何构成程序的转换单元中。

例子

如下例子说明了外部说明:

```
/*************************************************
                    SOURCE FILE ONE
*************************************************/
extern int i;        /*引用 i,定义在下面*/
void next(void);    /*函数原型*/

void main()
{
   i++;
   printf("%d\n",i);    /*i 等于 4*/
   next();
}

int i=3;             /*i 的定义*/
```

```
void next(void)
{
   i++;
   printf("%d\n",i);   /*i 等于 5*/
   other();
}

/*************************************************
                    SOURCE FILE TWO
*************************************************/
extern int i;   /*引用 i,定义在下面*/
                /*i 在第一个源文件中*/
void other(void)
{
   i++;
   printf("%d\n",i);   /*i 等于 6*/
}
```

这个例子中的两个源文件包含 i 的三个外部说明。只有一项是“定义说明”,它为:

```
int i=3;
```

定义全局变量 i 并用初始值 3 对其进行初始化。在第一个源文件的顶部 i 的“引用”说明使用 extern 使该全局变量在这个文件中它的定义说明

之前是可见的。在第二个源文件中 i 的引用说明使它在这个源文件中是可见的。如果一个变量的定义实例在转换单元中没有提供,编译器假设有一个:

```
extern int x;
```

引用说明和一个定义引用:

```
int x=0;
```

出现在该程序的另外转换单元中。

所有三个函数 main、next 和 other 执行同样的功能;它们将 i 增 1,并打印它。打印值 4、5 和 6。

如果变量 i 还没有初始化,它自动设置为 0。在这种情况下,打印值 1、2 和 3。有关变量初始化的信息参见本章后面的“初始化”。

内部层说明的存储类指示符

你可以在内部层为变量说明使用任何四种存储类指示符终结符。当你从这样的说明省略存储类指示符时,缺省的存储类是 auto。因此在一个 C 程序中很少看见 auto 关键字。

auto 存储类指示符

auto 存储类指示符说明一个自动变量,即一个具有局部生存期的变量。一个 auto 变量仅在说明它的块中是可见的。说明 auto 变量包括初始化语句,正如“初始化”节中讨论的。由于具有 auto 存储类的变量不会自动初始化,你可以在说明它们时显式对其初始化或者在该块的语句中赋

给初始值。未初始化的 auto 变量的值是不确定的(对于 auto 或 register 存储类的局部变量,如果给出一个初始化语句,那么每次进入其范围内都进行初始化)。
一个内部 static 变量(一个带有局部或块范围的 static)可以被带有任何外部的或 static 项的地址初始化,而不能被带有其他 auto 项的地址初始化,因为 auto 项地址不是一个常量。

register 存储类指示符

Microsoft 特殊处→
Microsoft C/C++编译器并不赞成寄存器变量的用户要求。但为了兼容性,与 register 关键字关联的所有其它语义都被编译器允准。例如,你不能向一个寄存器对象应用单目地址运算符(&),也不能对数组使用 register 关键字。
Microsoft 特殊处结束

static 存储类指示符

在内部层用 static 存储类指示符说明的变量具有全局生存期,但仅在其说明的块中是可见的,对于常量字符串,使用 static 是有用的,因为它减小了在经常调用的函数频繁的初始化的额外开销。
如果你不显式初始化一个 static 变量,它缺省初始化为 0,在一个函数内部,static 导致分配存储并作为一个定义,内部静态变量提供的私有、持久存储仅对单个函数是可见的。

extern 存储类指示符

用 extern 存储类指示符说明的变量是定义在外部层该程序的任何源文件中同名变量的一个引用。内部 extern 说明用于使外部层变量定义在该块中是可见的。除非在外部层有其它说明,否则用 extern 关键字说明的一个变量仅在其说明的块是中可见的。

例子

这个例子说明了内部和外部层说明:

```
#include <stdio.h>
int i=1;
void other(void);

void main()
{
  /* 引用 i,定义在上面 */
   extern int i;

   /*初始化值是 0,a 仅在 main 中可见*/
   static int a;

   /* 如果可以,b 存储在一个寄存器中, */
   register int b=0;
```

```
  /* 缺省存储类是 auto */
  int c=0;

  /*打印的值是 1,0,0,0 */
  printf("%d\n%d\n%d\n%d\n",i,a,b,c);
  other();
  return;
}

void other(void)
{
  /* 全局 i 的地址赋给指针变量 */
  static int *external_i=&i;

  /* i 重新定义,全局 i 不再可见 */
  int i=16;

  /* a 仅在 other 函数内可见 */
  static int a=2;

  a +=2;
  /* 打印的值是 16,4,1 */
```

```
    printf("%d\n%d\n%d\n",i,a,*external_i);
}
```

本例中,变量 i 的定义在外部层,并有初值 1,在 main 函数中的一个 extern 说明用来说明该外部层 i 的一个引用。static 变量 a 缺省为 0,因为省略了初始化语句。调用 printf 打印值 1、0、0 和 0。

在 other 函数中,全局变量 i 的地址用来初始化该 static 指针变量 external_i。这个工作因为全局变量有 static 生存期,其地址在程序执行中不改变。接着,变量 i 重新定义为局部变量并有初始 16。这个重新定义不影响外部层 i 的值,它通过使用它的局部变量的名称而隐藏。全局变量 i 的值现在仅在这个块中通过指针 external_i 间接访问。试图把 auto 变量 i 的地址赋给一个指针是不行的,由于它可能在每次进行该块时都不相同。变量 a 作为一个 static 变量说明并初始化为 2。这不会与 main 中的 a 冲突,因为在内部层的 static 变量仅在它们说明的块中是可见的。

变量 a 增大 2,给出 4 作为结果。如果 other 函数在同一程序中再次调用,a 的初始值为 4。内部 static 变量在程序退出,然后重新进入说明它们的块中时保存它们的值。

函数说明的存储类指示符

你可以在函数说明中使用 static 或 extern 存储类指示符。函数总是有全局生存期。

Microsoft 特殊处→

在内部层的函数说明和外部层的函数说明具有相同的含义。这意味着一

个函数从说明的点到该转换单元的其余部分中都是可见的,甚至在局部范围中说明亦如此。

Microsoft 特殊处结束

函数的可见性规则有点不同于变量的可见性规则,如下:

- 说明为 static 的一个函数仅在定义它的源文件中是可见的。在相同源文件中的函数可以调用 static 函数,但在其它源文件中不能用其名称直接访问。你可以用相同的名称在不同的源文件中没有冲突地说明另一个 static 函数。
- 以 extern 说明的函数在该程序的所有源文件中都是可见的(除非你以后重新说明这样的一个函数为 static)。任何函数可以调用一个 extern 函数。
- 省略存储类指示符的函数说明缺省是 extern 的。

Microsoft 特殊处→

Microsoft 允许重新定义一个 extern 标识符为 static。

Microsoft 特殊处结束

指示符

在说明中类型指示符定义一个变量或函数说明的类型。

语法

类型指示符:

void

char
short
int
long
float
double
signed
unsigned
结构或联合指示符
枚举指示符
typedef 名称

signed char、signed int、signed short int 和 signed long int 类型与它们的 unsigned 对应部分以及 enum 都称之为“整型”。float、double 和 long double 类型指示符被称为“浮点型”。你可以在一个变量或函数说明中使用任何整型或浮点型。如果在一个说明中没有提供一个类型指示符,把它作为 int。

任选关键字 signed 和 unsigned 可以位于或紧跟任何整型,除 enum 之外,都可以单独作为类型指示符,这种情况下,它们分别作为 signed int 和 unsigned int。当单独使用时,关键字 int 假设为 signed。当单独使用时,关键字 long 和 short 作为 long int 和 short int。

枚举类型考虑为基本类型。枚举类型的类型指示符在本章后面的“枚举说明”中讨论。

关键字 void 有三种用途:指出一个函数返回值、指出一个不接受参量的函数的参量类型表以及指出一个非指定的类型的指针。你可以使用 void 类型说明没有返回值的函数或说明一个未指定类型的指针。有关 void 出现在一个函数名称之后的括号中的信息,参见第 6 章“函数”中的“参量”。

Microsoft 特殊处→

现在的类型检测是 ANSI 兼容的,这意味着类型 short 和 int 是不同的类型。例如,在 Microsoft C 编译器中的重新定义被以前的编译器版本接受。

```
int   myfunc();
short myfunc();
```

下面的例子还产生一个关于不同类型之间间接赋值的警告:

```
int *pi;
short *ps;

ps=pi;   /*现在产生警告*/
```

Microsoft C 编译器也产生关于不同符号的警告,例如:

```
signed int *pi;
unsigned int *pu;

pi=pu; /*现在产生警告*/
```

类型 void 表达式求值是为了副作用。你不能以任何方式使用(不存在的)一个有类型 void 的表达式的值,也不能把一个 void 表达式(隐含或显式转换)转换成除 void

之外的任何类型。
如果你在需要一个 void 表达式的上下文中使用一个其它类型的表达式，它的值被丢弃。
为了与 ANSI 规格一致，void **不能用作 int **。只有 void *可以用作一个未指定类型的一个指针。
Microsoft 特殊处结束
你可以用 typedef 说明建立另外的类型指示符，正如在本章后面“typedef 说明”中描述的有关每种类型的尺寸的信息，请参见“基本类型的存储”。

数据类型指示符和等价者

本书通常使用表 3.1 中列出的类型指示符的格式，而不是长格式，它假设 char 类型缺省是有符号的。因此，在本书中，char 等价于 signed char。

表 3.1 类型指示符和等价者

类型指示符	等价者
signed char[1]	char
signed int	signed, int
signed short int	short, signed short
signed long int	long, signed long
unsigned char	-
unsigned int	unsigned
unsigned short int	unsigned short

续表

类型指示符	等价者
unsigned long int	unsigned long
float	-
long double[2]	-

1. 当你使 char 类型缺省为无符号的(通过指定/J 编译器选项),你不能简写 signed char 为 char。

2. 在 32 位操作系统中,Microsoft C 编译器把 long double 映射成 double。

Microsoft 特殊处→

你可以指定/J 编译器选项改变缺省 char 类型从有符号的到无符号的。当这个任选项起作用时,char 与 unsigned char 相同,你必须使用 signed 关键字说明一个有符号字符值。如果一个 char 值显式说明为有符号的,/J 选项对它没影响,当宽度变为 int 类型时其值进行符号扩充。当宽度变为 ing 类型时,char 类型进行 0 扩充。

Microsoft 特殊处结束

类型修饰符

类型修饰符给一个标识符两种属性之一。const 类型修饰符说明一个对象是不可修改的。voatile 类型说明一个项,其值在有时超出它出现的程序的范围例如并发执行线程时可以改变。

两个类型修饰符 const 和 volatile 在一个说明中仅出现一个。类型修饰符可以与任何类型指示符一起使用,但它们不能出现在多项说明中的第

一个逗号之后。

例如,以下说明是合法的:

```
typedef volatile int VI;
const int ci;
```

以下说明不是合法的:

```
typedef int *i, volatile *vi;
float f, const cf;
```

类型修饰符仅当在表达式中作为 l 值访问标识符时是相关的。有关 l 值和表达式的信息参见第 4 章“表达式和赋值”的“L 值和 R 值表达式”。

语法

类型修饰符:

```
const
volatile
```

如下是合法的 const 和 volatile 说明:

```
int const *p_ci; /* 常量 int 的指针 */
int const (*p_ci),  /* 常量 int 的指针 */
int *const cp_i; /* int 的常量指针 */
int (*const cp_i);   /* int 的常量指针 */
int volatile vint;   /* volatile 整数 */
```

如果一个数组类型的规格包括类型修饰符,被修饰是其元素,而不是数组类型。如果一个函数类型的规格包括修饰符,其行为是不确定的。volatile 和 const 都不影响对象的值的范围或算术属性。

这里列出如何使用 const 和 volatile:

- const 关键字可以用于修饰任何基本的或集合类型,或者任何类型的对象的指针或者一个 typedef。如果一个项仅用 const 类型修饰符说明,它的类型作为 const int。一个 const 变量可以初始化或放在一个只读存储区域中。const 关键字对于说明 const 的指针是有用的,因为这需要该函数不以任何方式改变该指针。
- 编译器假设在程序的任何地方,一个 volatile 变量可以通过使用或修改它的值的未知进程来访问。因此,不管命令行指定的优化,必须生成每个赋值的代码或一个 volatile 变量引用的代码,甚至它的出现没有作用亦如此。

  如果单独使用 volatile,假设是 int。volatile 类型指示符用于提供对特定存储器位置可靠的访问。对于可以通过信号处理器,通过并发执行程序或通过特定硬件例如存储器映射的 I/O 控制寄存器来访问或改变的数据对象使用 volatile。你可以为生存期声明一个变量为 volatile,或者造型单个引用为 volatile。
- 一个项可以是 const 和 volatile,在这种情况下该项不会由它自己的程序合理修改,但可以由一些异步进程修改。

说明符和变量说明

本章其余部分描述了变量类型的格式和含义,其总结在如下表中。特别地,下面的几节解释如何说明如下内容。

变量类型	说明
简单变量	具有整型或浮点型的单个值变量
数组	由相同类型的元素集合组成的变量
指针	指向其它变量的变量,包含变量的位置,以地址格式存储代替值
枚举变量	具有整型的简单变量,保存一个命名的整数常量集合中的一个值
结构	由具有不同类型的值的集合组成的变量
联合	由同一存储空间中几个不同类型的值组成的变量

一个说明符是指出引入到程序中的名称的一个说明,它可以包括修饰符,例如*(指针)和任何 Microsoft 调用约定的关键字。

Microsoft 特殊处→

在说明符中:

```
__declspec(thread) char *var;
```

char 是类型指示符,__declspec 和*是修饰符,var 是标识符的名称。

Microsoft 特殊处结束

你可以使用说明符说明指定类型值的数组、指定类型值的指针和指定类型返回值的函数。出现在数组和指针说明中的说明符在本章后面描述。

语法

说明符:

指针 $_{opt}$ 直接说明符

直接说明符:

标识符

(说明符)

直接说明符[常量表达式 $_{opt}$]

直接说明符(参数类型表)

直接说明符(标识符表 $_{opt}$)

指针:

*类型修饰符表 $_{opt}$

*类型修饰符表 $_{opt}$ 指针

类型修饰符表:

类型修饰符

类型修饰符表 类型修饰符

注意:有关一个说明符的引用语法,参见本章开头的“说明概述”中的说明的语法或参见本卷后面的附录 A“C 语言语法总结”。

当一个说明符由一个非修饰符的标识符组成时,说明的项有一个基本类型。如果一个星号(*)出现在一个标识符的左边,该类型修改为一个指针类型。如果该标识符紧跟方括号[],该类型修改为一个数组类型。有关说明中优先级解释的更多信息,参见本章后面的“解释更复杂的说明符”。

每个说明符至少说明一个标识符。一个说明符必须包括一个完整说明的类型指示符。该类型指示符给出了一个数组类型的元素的类型、指针类型所指对象的类型或者一个函数的返回值。

数组和指针说明在本章后面有更详细的讨论。如下例子说明几种简单的

说明符格式:

int list[20] /*说明一个名称为 list,20 个整数的数组*/

char *cp; /*说明一个 char 值的指针*/

double func(void) /*说明一个名称为 func 的函数,没有参量,返回值为一个 double 值*/

int *aptr[10]; /*说明一个有 10 个指针的数组*/

Microsoft 特殊处→

Microsoft C 编译器不限制修饰一个算术、结构或联合类型的说明符的个数。该个数仅受限于可用的存储器空间。

Microsoft 特殊处结束

简单变量说明

简单变量的说明是直接说明符的最简单的格式,它指出变量的名称和类型,它还指出该变量的存储类和数据类型。

在变量说明中需要存储类或类型(或两者),没有类型的变量(例如 var;)产生警告。

语法

说明符:

指针 $_{opt}$ 直接说明符

直接说明符:

标识符

标识符:

非数字
标识符 非数字
标识符 数字

对于算术、结构、联合、枚举和 void 类型以及由 typedef 命名表示的类型,在一个说明中可以使用简单说明符,因为该类型指示符提供所有类型信息。指针、数组和函数类型需要更复杂的说明符。

你可以使用一个标识符表,标识符之间用逗号分隔,以便在同一说明中指出几个变量。在该说明中定义所有变量具有相同的基本类型。例如:

```
int x,y; /*说明两个 int 类型的简单变量*/
int const z=1; /*说明类型为 int 的变量值*/
```

变量 x 和 y 为了一个特殊实现保存 int 类型定义的集中的任何值。简单对象 z 初始化为 1,并且是不可修改的。如果 z 的说明是一个非初始化的静态变量或具有文件范围,它接受一个初始值 0 并也是不可修改的。

```
unsigned long reply,flag; /*说明两个命名为 reply 和 flag 的变量*/
```

在这个例子中,两个变量 reply 和 flag 都有 unsigned long 类型并保存无符号整数值。

枚举说明

一个枚举是由一个命名的整数常量组成的。一个枚举类型说明给出了枚举标志的名称(任选的)并定义命名的整数标识符的集合(称为“枚举集”、“枚举常量”、“枚举器”或“成员”)。一个具有枚举类型的常量存储该类型定义的枚举集的值之一。

enum 类型的变量可以用在指标表达式中,作为所有算术和关系运算符的操作数,枚举提供了替代#define 预处理器命令的另一种方法,利用它为你生成值和遵守正常范围规则的优点。

在 ANSI C 中,定义一个枚举常量的值的表达式总是具有 int 类型,因此与一个枚举变量关联的存储是单个 int 值需要的存储。一个枚举常量或一个枚举类型的值可以用在 C 语言允许一个整数表达式使用的任何地方。

语法

enum 指示符:

enum 标识符 $_{opt}${枚举器表}

enum 标识符

可选标识符命名由枚举器表定义的枚举类型。这个标识符经常称为该表指定的枚举的“标志”。一个类型指示符的格式为:

enum 标识符{枚举器表}

说明该标识符是由枚举器表指定的枚举的标志。该枚举器表定义了该“枚举器内容”。该枚举器表在下面详细描述。

如果一个标志的说明是可见的,后续使用该标志而省略枚举器表的说明指出以前说明的枚举的类型。该标志必须指的是一个定义的枚举类型,该枚举类型必须在当前范围中。因为枚举类型定义在其它地方,该枚举器表不出现在这个说明中。从枚举派生的类型的说明和枚举类型的typedef 说明可以在该枚举类型定义之前使用该枚举标志。

语法

枚举器表:

枚举器

枚举器表,枚举器

枚举器:

枚举常量

枚举常量=常量表达式

枚举常量:

标识符

在枚举器表中的每个枚举器常量命名该枚举集的值。缺省地,第一个枚举常量与值 0 关联,该表中的下一个枚举常量与值(常量表达式+1)的值关联,除非你显式地将其与其它值关联。一个枚举常量的名称等价于它的值。

你可以使用枚举常量=常量表达式来覆盖值的缺省序列。因此,如果枚举常量=常量表达式出现在该枚举器表中,该枚举常量与常量表达式给定的值关联。该常量表达式必须有 int 类型且可以为负数。

如下规则应用于一个枚举集的成员:

- 一个枚举集可以包含重复的常量值。例如,你可以用两个不同的标识符与 0 关联,它们是同一个集中,可能是 mull 和 zero。
- 枚举集中的标识符必须不同于同一范围中具有相同可见性的其它标识符,包括普通变量名称和其它枚举表中的标识符。
- 枚举标志遵守一般范围规则,它们必须不同于相同可见性的其它枚举、结构和联合标志。

例子

这些例子说明了枚举的说明:

```
enum DAY /* 定义一个枚举类型 */
{
   saturday,/*命名 day 并说明具有该类型的变量 workday */
   sunday=0,
   monday,
   tuesday,
   wednesday,   /*wednesday 与 3 关联 */
   thursday,
   friday
} workday;
```

缺省地值 0 与 saturday 关联。标识符 sunday 显式设置为 0。余下的标识符根据缺省给出 1 到 5 的值。在以下例子中,一个集 DAY 的值赋给变量 today:

```
enum DAY today=wednesday;
```

注意，枚举常量的名称用作赋值。由于 DAY 枚举类型是以前说明的,只有枚举标志 DAY 是必要的。

为了显式地把一个整数值赋给一个枚举的数据类型的变量,使用一个类型造型:

```
workday=(enum DAY) (day_value-1);
```

这个造型在 C 中推荐,但是不需要的。

```
enum BOOLEAN/*说明一个称为 BOOLEAN 的枚举数据类型*/
```

```
 {
false,/*false=0,true=1 */
true
 };
```

```
  enum BOOLEAN end_flag,match_flag;   /* BOOLEAN 类型的两个变量*/
```

这个说明也可以指定为:

```
enum BOOLEAN {false,true} end_flag,match_flag;\
```

或

```
enum BOOLEAN {false,true} end_flag;
enum BOOLEAN match_flag;
```

使用这三个变量的例子如下:

```
if (match_flag == false)
  {
 .
 .   /*语句*/
 .
  }
  end_flag = true;
```

可以说明未命名的枚举器数据类型。省略数据类型的名称,但可以说明变量。变量 response **是这样定义的类型的变量**:

```
enum {yes, no} response;
```

结构说明

一个“结构说明”命名一个类型并指定具有不同类型变量值(称为结构的“成员”或域)的序列。一个称为“标志”的任选标识符给出结构类型的名称,可以用在后面对结构类型的引用中,一个结构类型的变量保存该类型定义的整个序列。在 C 中结构类似于其它语言中众所周知的“记录”。

语法

结构或联合标识符:

结构或联合标识符 $_{opt}$ {结构说明表}

结构或联合标识符

结构或联合:

struct

union

结构说明表:

结构说明

结构说明表 结构说明

结构的内容定义为:

结构说明:

指示符修饰符表 结构说明符表

指示符修饰符表:

类型指示符 指示符修饰符表 $_{opt}$

类型修饰符 指示符修饰符表$_{opt}$

结构说明表:

结构说明符

结构说明符表,结构说明符

结构说明符:

说明符

一个结构类型的说明不为一个结构设置另外空间,它只是一个模块,便于后面说明结构变量。

前面定义的标识符(标志)可以用于指向其它地方定义的一个结构类型。在这种情况下,结构说明表不能重复直到它的定义是可见的。结构的指针的说明和结构类型的 typedef 可以在该结构类型被定义之前使用该结构标志。但必须在其任何域的尺寸的实际使用之前遇到该结构定义。这是类型和类型标志的不完整定义。为了完成这个定义,必须在相同范围的后面出现一个类型定义。

结构说明表指出结构成员的类型和名称。一个结构说明表参量包含一个或多个变量或位域说明。

在结构说明表中说明的每个变量定义为该结构类型的一个成员,在结构说明表中的变量说明和本章讨论的其它变量的说明具有相同的格式,除了该说明不能包含存储类指示符或初始化语句外,该结构成员可以具有除 void 之外的任何类型、一个不完整的类型或一个函数类型。

一个成员不能说明成为具有包含该成员的结构的类型,但一个成员可以说明成为包含该成员的结构的指针,该结构类型有一个标志。这允许你

建立结构链表。
结构与其它标识符遵循同样的范围规则。结构标识符必须不同于具有相同可见性的其它结构、联合和枚举标志。
在结构说明表中的每个结构说明在该表中必须是唯一的。但一个结构说明表中的标识符名称不必与其它普通变量名或其它结构说明表中的标识符相区分。
通过在文件范围层说明,还可以访问嵌套的结构。例如,给出这样的说明:

```
struct a
  {
int x;
struct b
    {
  int y
    } var2;
  } var1;
```

如下说明都是合法的:

```
struct a var3;
struct b var4;
```

例子
这些例子说明了结构的说明:

```
struct employee/*定义一个命名为 temp的结构变量*/
  {
```

```
char name[20];
int id;
long class;
 } temp;
```

employee 结构有三个成员,name、id 和 class。name 成员是一个 20 个元素的数组;id 和 class 分别是 int 和 long 类型的简单成员。标识符 employee 是结构标识符。

```
struct employee student, faculty, staff;
```

这个例子定义了三个结构变量:student、faculty 和 staff。每个结构都有三个成员的相同表。这些成员被说明为有结构类型 employee,定义在前面的例子中:

```
struct/*定义一个无名的结构和 a*/
{
 float x,y;
} complex;   /*结构变量命名为 complex*/
```

complex 结构具有两个 float 类型的成员 x 和 y,该结构类型没有标志,因此是未命名或无名的。

```
struct sample /*定义一个命名为 x 的结构*/
{
   char c;
   float *pf;
   struct sample *next;
```

} x;

该结构开头两个成员是一个 char 变量和一个 float 值的指针。第三个成员 next 说明为定义的结构类型(sample)的指针。

在不需要命名的标志是无名结构是有用的。这是一个说明定义所有结构实例的情况。

例如:

```
struct
{
int x;
int y;
} mystruct;
```

嵌入的结构通常是无名的。

```
struct somestruct
{
   struct /*无名的结构*/
   {
  int x,y;
   } point;
   int type;
} w;
```

Microsoft 特殊处→

编译器允许一个无尺寸的或 0 尺寸的数组作为一个结构的最后成员,如

果一个常量数组的尺寸在不同情况下也不同时,这是有用的。这样的一个说明如下:

struct 标识符

{

说明集

类型 数组名称[];

};

无尺寸的数组只能作为一个结构的最后成员。包含无尺寸的数组说明的结构可以嵌套在其它结构中,以及在包含它的结构中没有更多成员需要说明。不允许这样结构的数组,当 sizeof 运算符应用于这种类型的变量或这个类型本身时,该数组的尺寸假设为 0。

在该结构是另一个结构或联合的成员时,没有一个说明符也可以指定该结构说明。域名称提升到包括它的结构中。例如,一个较少命令的结构如下:

```
struct s
{
float y;
struct
{
int a,b,c;
};
char str[10];
```

```
} *p_s;
.
.
.
p_s->b = 100;   /*s 结构中一个域的引用*/
```

有关结构引用的信息,参见第 4 章“表达式和赋值”中的“结构和联合成员”。

Microsoft 特殊处结束

位域

除了说明一个结构或联合的成员外,一个结构说明符还可以指定位的个数,称为一个位域,其长度由说明符的域名后的冒号来设置。一个位域解释为一个整型。

语法

结构说明符:

    说明符
    类型指示符 说明符$_{opt}$:常量表达式

该常量表达式指出域的位域。说明符的类型指示符必须是 unsigned int、signed int 或 int,且常量表达式必须是非负整数。如果其值为 0,该说明不是说明符。不允许有位域的数组、位域的指针和返回位域的函数。任选说明符命名位域。位域只能说明为一个结构的一部分。取地址运算符(&)不能应用于位域的组成成分。

无名的位域不能被引用,它们的内容在运行时是不可预料的。它们为了赋值目的可以用作“哑元”域。一个其宽度指定为 0 的无名位域保证在该结构说明表中紧跟它的成员从一个 int 边界开始存储。
位域也必须足够长以便包含位模式,例如,这样的两个语句是非法的:

```
short a:17,   /*非法的!*/
int long y:33/*非法的!*/
```

如下例子定义一个二维结构数组并命名为 screen:

```
struct
{
      unsigned short icon : 8;
      unsigned short color : 4;
      unsigned short underline : 1;
      unsigned short blink : 1;
 } screen[25][80];
```

该数组包含 2000 个元素,每个元素是一个包含四个位域成员:icon、color、underline 和 blink 的结构。每个结构的尺寸是两个字节。
位域与整数类型具有相同的语义。这意味着一个表达式中使用一个位域的方式与相同基本类型的变量使用的方式完全相同,而不论在位域中有多少位。
Microsoft 特殊处→
以 int 定义的位域处理为有符号的。一个 Microsoft 扩充 ANSI C 标准允许位域有 char 和 long 类型(signed 和 unsigned)。具有基本类型 long、short

或 char(signed 或 unsigned)的无名的位域强制赋给适合于该基本类型的边界。

在一个整数从最低有效数字位到最高有效数字位都分配位域,在以下代码中:

```
struct mybitfields
 {
 unsigned short a : 4;
 unsigned short b : 5;
 unsigned short c : 7;
 } test;

 void main(void)
 {
 test.a=2;
 test.b=31;
 test.c=0;
 }
```

这些位排列如下

00000001 11110010

ccccccccb bbbbaaaa

由于 8086 处理器簇在高位字节之前存储整数的低位字节,上面的整数 0x01F2 在物理存储器中先存储 0xF2,接着是 0x01。

Microsoft 特殊处结束

结构的存储和赋值

Microsoft 特殊处→

结构成员以它们说明的顺序存储:第一个成员具有最低的存储器地址,最后的成员具有最高的存储器地址。

每个数据对象有一个对齐要求。对于结构,该要求是它的成员最大的要求值。每个对象分配一个偏移量以便:

偏移量 % 对齐要求 == 0

如果整数类型具有相同的尺寸,且如果下一个位域适合于当前分配单元且通过该位域的公共对齐要求的统一而没有边界交叉,则调整位域为相同的 1、2 或 4 字节分配单元。

为了节省空间或使现存数据结构一致,你可以更紧凑或更松散地存储结构。/Zp[*n*]编译器选项和#pragma pack 控制结构数据如何“紧凑”进入存储器。当你使用/Zp[*n*]选项时,这里 n 是 1、2、4、8 或 16,第一个结构成员之后的每个结构成员存储一个字节边界,该边界是域的对齐要求或紧凑尺寸(n)。这里是很小的,表示成一个公式,字节边界是:

min(n,sizeof(项))

这里的 n 是/Zp[*n*]选项表示的紧凑尺寸,项是结构成员。缺省的紧凑尺寸是/Zp8。为了使用 pack 编译指示以指出紧凑,该紧凑是非命令行中为一个特定结构所指的紧凑,给出 pack 编译指示,这里紧凑尺寸在该结构之前取值为 1、2、4、8 或 16。为了在命令行重新安装该紧凑,指出不带参量

的 pack 编译指示。
对于 Microsoft C 编译器,位域缺省为 long 尺寸。结构成员在该类型的尺寸或/Zp[*n*]尺寸上对齐,这里是较小的。缺省尺寸为 4。
Microsoft 特殊处结束

联合说明

一个“联合说明”指出变量值的集合,它是任选的,一个标志命名该联合。该变量值称为联合的成员,可以有不同类型,联合类似于其它语言中的“可变记录”。

语法

结构或联合指示符:
    结构或联合 指示符 $_{opt}$ {结构说明表}
    结构或联合 标识符

结构或联合:
    struct
    union

结构说明表:
    结构说明
    结构说明表 结构说明

联合内容定义为:

结构说明:
    指示符修饰符表 结构说明符表;

指示符修饰符表:

类型指示符 指示符修饰符表 $_{opt}$

类型修饰符 指示符修饰符表 $_{opt}$

结构说明符表:

结构说明符

结构说明符表,结构说明符

具有 union 类型的变量存储这个类型定义的值之一。相同的规则控制结构和联合说明。联合也可以有位域。

联合的成员不能有一个不完整的类型、类型 void 或函数类型。因此成员不能是该联合的一个实例但可以是说明的联合类型的指针。

一个联合类型说明只是一个模板,存储器不保存直到说明该变量。

注意:如果说明一个有两个类型的联合,但该联合用其它类型访问,其结果是不可靠的。例如,说明一个 float 和 int 的联合,存储一个 float 值,但以后作为 int 访问它的值。在这种情况下,该值依赖于 float 值的内部存储。其整数值是不可靠的。

例子

如下是联合的例子

```
union sign/*一个定义和一个说明*/
{
   int svar;
   unsigned uvar;
} number;
```

这个例子用 sign 类型定义了一个联合,说明一个命名为 number 的变量,它有两个成员:

svar 是一个有符号整数和一个无符号整数 uvar。这个说明允许 number 的当前值作为一个有符号的或无符号的值存储。与这个联合类型关联的标志是 sign。

```
 union /*定义一个名称为 screen 的一个二维数组*/
 {
 struct
 {
unsigned int icon : 8;
unsigned color : 4;
 } window1;
 int screenval;
} screen[25][80];
```

screen 数组包含 2000 个元素,该数组的每个元素是具有两个成员:Window1 和 screenval 的单个联合。window1 成员是具有两个位域的成员:icon 和 color 的结构。screenval 成员是一个 int。在任何时候,每个联合成员保存 screenval 表示的 int 或 window1 表示的结构。

Microsoft 特殊处→

当一个联合是另一个结构或联合的成员时可以无名地说明嵌套的联合。如下是一个无名联合的例子:

```
struct str
```

```
{
   int a,b;
   union   /*无名的联合*/
   {
  char c[4];
  long l;
  float f;
   };
   char c_array[10];
} my_str;
.
.
.
my_str.l == 0L; /*my_str 联合中一个域的引用*/
```

联合经常嵌套在一个结构中,该结构包括这样的一个域,在任何特定时间,给出包含在该联合中数据的类型。下面是这样的一个联合的说明的例子:

```
struct x
{
   int type_tag;
   union
   {
  int x;
```

```
  float y;
   }
 }
```

有关引用联合的信息参见第 4 章“表达式和赋值”中的“结构和联合成员”。

Microsoft 特殊处结束

联合的存储

与一个联合变量关联的存储是该联合中最大成员所需要的存储。当存储较小成员时,该联合变量可以包含未用的存储器空间。所有成员存储在相同存储器中并起始于相同的地址。

每次把一个值赋给不同成员时,原来存储的值被覆盖。例如:

```
union /*定义一个名称为 x 的联合*/
{
 char *a,b;
 float f[20];
  } x;
```

x 联合的成员根据它们说明的次序依次是一个 char 值的指针、一个 char 值和一个 float 值的数组。x 分配的存储是具有 20 个元素的数组 f 所需要的存储,因为 f 是最长的联合成员。

由于该联合没有关联的标志,它的类型是未命名或“无名的”。

数组说明

一个“数组说明”命名数组和指定其元素的类型,它还定义了数组中的元素个数。具有数组类型的变量是数组元素类型的指针。

语法

说明

说明指示符 初始化说明符表$_{opt}$;

初始化说明符表:

初始化说明符

初始化说明符表,初始化说明符

初始化说明符:

说明符

说明符=初始化器

说明符:

指针 opt 直接说明符

直接说明符:

直接说明符[常量表达式$_{opt}$]

因为常量表达式是任选的,其语法有两种格式:

- 第一种格式定义一个数组变量。方括号中的常量表达式参量指出该数组中的元素个数,该常量表达式如果出现必须具有整数类型,并且具有一个大于 0 的值。每个元素具有类型指示符给定的类型。它可以是非 void 之外的任何类型。一个数组元素不能是一个函数类型。

- 第二种格式说明一个在其它地方定义的变量。它省略方括号中的常量表达式,但不是方括号。只有在以前初始化该数组并说明它为一个参量或在程序中其它地方显式定义的一个数组的引用时才可使用这种格式。

在两种格式中,直接说明符命名该变量并可以修改该变量的类型。直接说明符后的方括号([])修改该说明符为一个数组类型。

类型修饰符可以应用在一个数组类型对象的说明中,但该修饰符应用于元素而不是数组本身。

你可以在数组说明符之后跟随一个括起的常量表达式表来说明一个数组的数组(一个“多维”数组),格式为:

类型指示符 说明符[常量表达式][常量表达式]...

括号中的每个常量表达式定义给定维数的元素个数:二维数组有两个括起的表达式、三维数组有三个括起的表达式等等。如果你已经初始化该数组,说明它是一个参数或者说明它作为程序中其它地方显式定义的一个数组的引用,那么你可以省略第一个常量表达式。

你可以通过使用复杂说明符定义各种类型的指针数组,正如本章后面“解释更复杂的说明符”中描述的。

数组以行方式存储,例如,如下数组由两行组成,每行三列:

```
char A[2][3];
```

首先存储第一行的三个列,接着是第二行的三个列。这意味着最后一个下标更快速改变。

为了引用一个数组的单个元素,使用一个下标表达式,正如第 4 章“表达

式和赋值”中的“后缀运算符”中描述的。

例子

这些例子说明了数组的说明;

```
float matrix[10][15];
```

该二维数组命名为 matrix,有 150 个元素,每个元素具有 float 类型。

```
struct{
  float x,y;
  }complex[100];
```

它是一个结构数组的说明。这个数组有 100 个元素,每个元素是包含两个成员的结构。

```
extern char *name[];
```

这个语句说明一个指向 char 的数组的类型和名称。name 的实际定义出现在其它地方。

Microsoft 特殊处→

整数的类型需要保存一个数组的最大尺寸是 size_t 的尺寸,它定义在 STDDEF.H 头文件中。size_t 是一个 unsigned int,其范围是 0x00000000 到 0x7CFFFFFF。

Microsoft 特殊处结束

数组的存储

与一个数组类型关联的存储是其所有元素需要的存储。一个数组的所有元素从第一个元素到最后一个元素以相邻方式存储并逐步增大存储器位

置。

指针说明

一个“指针说明”命名一个指针变量并指定该变量指向的对象的类型。以指针说明的一个变量保存一个存储器地址。

语法

说明符:

指针 opt 直接说明符

直接说明符:

标识符

(说明符)

直接说明符[常量表达式 opt]

直接说明符(参数类型表)

直接说明符(标识符表 opt)

指针:

*类型修饰符表 opt

*类型修饰符表 opt 指针

类型修饰符表:

类型修饰符

类型修饰符表 类型修饰符

类型指示符给出该对象的类型,它可以是任何基本的、结构或联合类型。指针变量也可以指向函数、数组和其它函数(有关说明和解释更复杂指

针类型的信息,参见本章后面的“解释更复杂的说明符”)。
通过使类型指示符为 void,你可以推迟该类型规格到指针所指地方。这样的一个项作为“void 的指针”写成 void*。一个说明为 void 的指针的变量可以用于任何类型的对象的指针。但为了在指针上或所指对象上执行更多操作,对于每种操作,所指的类型必须显式指定(类型 char *和类型 void *的变量是赋值兼容的,而不使用类型造型)。这样的转换与一个类型造型是一致的(有关更多信息,参见第 4 章“表达式和赋值”中的“类型造型转换”)。
类型修饰符可以是 const 或 volatile,或者两者。它们分别指出该指针不能被程序本身修改(const)或者该指针可以被某个超出程序控制的进程合理地修改(volatile)(有关更多信息参见本章前面的“类型修饰符”)。
说明符命名变量,可以包括一个类型修饰符。例如,如果说明符表示一个数组,该指针的类型可以修改为一个数组的指针。
你可以在你定义结构、联合或枚举器类型之前说明一个结构、联合或枚举器类型的指针。你可以通过使用如下例子所示的结构或联合标志来说明指针。这样的说明是允许的,因为编辑器不需要知道该指针变量分配的结构或联合的空间的尺寸。
例子
如下例子说明指针的说明:

```
char *message;   /*说明 message 的指针变量*/
```

message 指针指向 char 类型的变量。

```
int *pointers[10];/*说明一个指针数组*/
```

pointers 数组有 10 个元素,每个元素是 int 类型变量的指针。

```
int (*pointer)[10];   /*说明有 10 个元素的数组的指针*/
```

pointer 变量指向一个有 10 个元素的数组,该数组中的每个元素具有 int 类型。

```
int const *x; /*说明一个指针变量 x,指向一个常量值*/
```

指针 x 可以修改为指向不同的 int 值,但它指向的值是不能修改的。

```
const int some_object = 5;
int other_object = 37;
int *const y = &fixed_object;
const volatile *const z = &some_object;
int *const volatile w = &some_object;
```

在这个说明中的变量 y 说明为一个 int 值的常量指针。它指向的值可以修改,但指针本身必须总是指向相同的位置:fixed_object 的地址。类似地,z 是一个常量指针,但它也说明为一个 int 的指针,其值不能由程序修改。另一个指示符 volatile 指出虽然由 z 所指的 const int 的值不能由程序修改,但可以由与程序并发运行的进程合理地修改。w 的说明指出该程序不能改变所指的值,程序不能修改该指针。

```
struct list *next, *previous; /*使用 list 的标志*/
```

这个例子说明两个指针变量 next 和 previous,指向结构类型 list。这个说明可以出现在 list 结构类型定义之前(参见下一个例子),只要 list 类型定义与该说明具有同样的可见性。

```
struct
```

```
{
   char *token;
   int count;
   struct list *next,
} line;
```

变量 line 具有 list 命名的结构类型,list 结构类型有三个成员:第一个是 char 值的指针、第二个是 int 值、第三个是另一个 list 结构的指针。

```
struct id
{
   unsigned int id_no;
   struct name *pname;
} record;
```

变量 record 具有结构类型 id。注意 pname 说明为另一个结构类型 name 的指针,这个说明可以出现在 name 类型定义之前。

地址存储

一个地址所需的存储数量和地址的含义取决于编译器的实现。不同类型的指针不保证具有相同的长度。因此,sizeof(char *)不必等于 sizeof(int *)。

Microsoft 特殊处→

对于 Microsoft C 编译器,sizeof(char *)等于 sizeof(int *)。

Microsoft 特殊处结束

基指针

Microsoft 特殊处→

对于 Microsoft32 位 C 编译器,一个基指针是距离一个 32 位指针基 32 位偏移量。基地址对于练习控制分配对象的段是有用的,由此减小可执行文件的长度和提高执行速度。通常地,指出一个基指针的格式为:

类型 __based(基)说明符

基地址的“基指针”的变化能使一个指针的规格像一个基一样。而基指针是开始于指针开头的存储器段的一个偏移量。基于指针地址的指针是 32 位编译中有效__based 关键字仅有的格式。在这样的编译中,它们是距离一个 32 位基的 32 位位移。

基于指针的指针的一个用途是为了包含指针的持久(persistent)标识符。由基于一个指针的指针组成一个链表可以存储到磁盘中,然后用保持有效的指针重新加载到存储器的另一个地方。

如下例子给出了一个基本指针的指针。

```
void *vpBuffer;
 struct llist_t
 {
void __based(vpBuffer) *vpData;
struct llist_t __based(vpBuffer) *llNext;
 };
```

指针 vpBuffer 被赋给在程序中某个后面的点分配的存储器的地址。该

链表相对于 vpBuffer 的值进行重新定位。
Microsoft 特殊处结束

抽象说明符

一个抽象说明符(abstract declarator)是没有一个标识符的说明符,由一个或多个指针、数组或函数修饰符组成。指针修饰符(*)总是位于一个说明符中标识符的前面;数组([])和函数(())修饰符紧跟标识符。知道这些,你可以确定标识符出现在一个抽象说明符中的位置并相应地解释该说明符。有关复杂说明符的另外信息和例子,参见下一节“解释更复杂的说明符”。通用的 typedef 可以用来简化说明符,参见本章后面的“typedef 说明”。
抽象说明符可以是复杂的,一个复杂抽象说明符中的圆括号指出一个特定的解释,正如说明中它们为复杂说明符所做的那样。
这些例子说明抽象说明符:

```
int *       /*一个 int 类型的指针的类型名称*/

int *[3]          /*三个 int 指针的数组*/

int (*)[5] /*五个 int 的数组的指针*/

int *()           /*没有参数并返回 int 的指针的函数*/
```

```
/* 一个没有参数并返回 int 的指针的函数的指针 */
int (*)(void)

/* 一个未指定元素个数的数组,其元素是这样的函数的指针,每个函数有一个 unsigned int
   参数和其它未指定个数的参数并返回 int */
int (*count[]) (unsigned int,...)
```

注意:抽象说明符不允许由一个空括号集()组成,因为这是模糊的。不可能确定该隐含的标识符是否属于括号内(这种情况中它是一个不能修改的类型)或是否在括号之前(这种情况下它是一个函数类型)。

解释更复杂的说明符

你可以在括号中包含任何说明符来指出一个“复杂说明符”的特殊解释。一个复杂说明符是由多个数组、指针或函数修饰符修饰的标识符。你可以将各种数组、指针和函数修饰符的组合应用于单个标识符。通常, typedef **可以用来简化说明。参见本章后面的“** typedef **说明”。**

在解释复杂说明符中,方括号和圆括号(也就是该标识符右边的修饰符)比星号(也就是该标识符左边的修饰符)更优先。方括号和圆括号具有相同的优先级且从左到右结合。在说明符完全解释后,最后应用类型指示符。通过使用圆括号你可以覆盖缺省的结合次序并强制一个特定的解释。但标识符名称不使用圆括号,这可能误解释为参数表。

解释复杂说明符的简单方法是使用如下四个步骤“从里向外”读:

1. 从标识符开始,直接查找右边的方括号和圆括号(如果有)。
2. 解释这些方括号或圆括号,然后查找右边的星号。
3. 如果在任何时候遇到一个右圆括号,回到前面对圆括号里的内容应用规则 1 和规则 2。
4. 应用类型指示符。

```
char *( *(*var)() )[10];
 ^    ^  ^ ^ ^     ^ ^
 7    6  4 2 1     3 5
```

在这个例子中,步骤按顺序标出,并解释如下:

1. 标识符 var 说明
2. 一个指针
3. 一个函数返回
4. 一个指针
5. 一个有 10 个元素的数组
6. 一个指针
7. char 值

例子

以下例子说明其它复杂的说明以及圆括号如何影响一个说明的含义的。

```
int *var[5];/*指向 int 值的数组*/
```

数组修饰符比指针修饰符具有更高的优先级,因此 var 说明为一个数组。指针修饰符应用于该数组元素的类型;因此,该数组元素是 int 值的指针。

```
int (*var)[5];   /* int 值的数组的指针 */
```

在这个 var 的说明中,圆括号给出的指针修饰符比数组修饰符的优先级高,var 说明成 5 个 int 值的数组的指针。

```
long *var (long,long); /*返回 long 指针的函数  */
```

函数修饰符的优先级也比指针修饰符的级别高。因此这个 var 的说明用来说明 var 是一个函数,其返回一个 long 值的指针。该函数说明为两个 long 值作为参量。

```
long(*var)(long,long), /*返回 long 的函数的指针 */
```

这个例子类似于前面的一个,圆括号给出的指针修饰符比函数修饰符具有更高的优先级,var 被说明为一个返回 long 值的函数的指针。同样,该函数有两个 long 参量。

```
struct both/*函数指针的数组*/
  {             /*返回结构*/
int a;
char b;
  } (*var[5])(struct both,struct both);
```

一个数组的元素不能是函数,但这个说明证明了如何说明函数指针的数组。在这个例子中,var 说明成一个函数的 5 个指针的数组,该函数返回具有 2 个成员的结构。该函数的参量说明成具有相同结构类型 both 的两个结构。注意,把*var[5]括起来的圆括号是必须的,没有它们,该说明是非法的,试图说明一个函数的数组,如下:

```
/*非法的*/
```

```
struch both *var[5](struct both,struch both);
```

如下语句说明一个指针数组:

```
unsigned int *(*const *name[5][10])(void);
```

name 数组有 50 个元素,以多维数组的形式组织,其元素是一个指针的指针,它是一个常量,这个常量指针指向一个函数,该函数没有参数并返回一个无符号类型的指针。

下一个例子是一个函数,它返回一个具有 3 个 double 值的数组的指针:

```
double(*var(double(*)[3]))[3];
```

在这个说明中,函数返回一个数组的指针,因为返回数组的函数是非法的。这里 var 说明成一个函数,该函数返回具有三个 double 值的数组的指针。该函数 var 有一个参量,该参量象返回值,它是具有三个 double 值的数组的指针。该参量类型是由抽象说明符给出的。

在参量类型中星号外的圆括号是需要的,没有它们,该参量类型是三个 double 值的指针的数组。有关抽象说明符的讨论和例子,参见本章前面的“抽象说明符”。

```
union sign   /*指针数组的数组*/
 {              /*联合的指针*/
     int x;
     unsigned y;
 } **var[5][5];
```

本例说明了如何放置圆括号来改变说明的含义。在这个例子中,var 是联

合指针的 5 元素数组的指针的 5 元素数组。对于如何使用 typedef 避免复杂说明的例子,参见“ typedef 说明”。

初始化

一个“初始化器”是赋给说明变量的值的序列。你可以通过在变量说明中将一个初始化器应用于说明符中来设置一个变量的初值。把值或初始化器的值赋给变量。

下面几小节描述如何初始化标量、集合和字符串类型的变量。“标量类型”包括所有算术类型加上指针。“集合类型”包括数组、结构和联合。

初始化标量类型

当初始化标量类型时,赋值表达式的值赋给该变量。应用赋值的转换规则(有关转换规则的信息,参见第 4 章“类型转换”)。

语法

说明:

说明指示符 初始化说明符表 $_{opt}$;

说明指示符:

存储类指示符 说明指示符 $_{opt}$

类型指示符 说明指示符 $_{opt}$

类型修饰符 说明指示符 $_{opt}$

初始化说明符表:

初始化说明符

初始化说明表,初始化说明符

初始化说明符:

说明符

说明符=初始化器/*对于标量初始化*/

初始化器:

赋值表达式

你可以初始化任何类型的变量,要遵守如下规则:

- 在文件范围层说明的变量可以初始化。如果你不显式初始化外部层的变量,它缺省地初始化为 0。
- 一个常量表达式可以用于初始化任何用 static 存储类指示符说明的全局变量,被说明为 static 的变量在程序执行开始被初始化。如果你不显式初始化一个全局 static 变量,它缺省地初始化为 0,每个具有指针类型的成员都赋值一个空指针。
- 用 auto 或 register 存储类指示符说明的变量在每次执行控制传给说明该变量的块时都进行初始化。如果你在一个 auto 或 register 变量说明中省略一个初始化器,该变量的初始化值是不确定的。对于自动和寄存器值,初始化器并不限制是一个常量,它可以是包含以前定义值的任何表达式,甚至是函数调用。
- 外部变量说明的初始化值和所有 static 变量的初始值,不论是外部还是内部的,都必须是一个常量表达式(有关更多信息,参见第 4 章中的“常量表达式”)。由于任何外部说明的或静态变量的地址是常量,它可以用于初始化一个内部说明的 static 指针变量。但 auto 变量的地址不能用于一个静态初始化器,因为对于该块的每次执行

它可能有不同的值。你可以使用常量或变量初始化 auto 和 register 变量。

- 如果一个标识符的说明有块范围,且该标识符有外部连接,该说明不能有初始化。

例子

以下例子说明初始化:

```
int x=10
```

整数变量 x 初始化为常量表达式 10。

```
register int *px=0;
```

指针 px 初始化为 0,产生一个“空”指针。

```
const int c=(3*1024);
```

这个例子使用一个常量表达式(3*1024)初始化 c 为一个不能修改的常量值,因为有 const 关键字。

```
int *b=&x;
```

该语句用另一个变量 x 的地址初始化指针 b。

```
int *const a=&z;
```

指针 a 用变量 z 的地址进行初始化。由于它为 const,因此变量 a 只能初始化,不能修改。它总是指向相同的位置。

```
int GLOBAL;


int function(void)
{
```

```
    int LOCAL;
    static int *lp=&LOCAL;/*非法初始化*/
    static int *gp=&GLOBAL;   /*合法初始化*/
    register int *rp=&LOCAL; /*合法初始化*/
}
```

全局变量 GLOBAL 在外部层说明,因此它有全局生存期。局部变量 LOCAL 有 auto 存储类,仅在说明它的函数执行期间有地址。因此,试图用 LOCAL 的地址初始化 static 指针变量 lp 是不允许的。static 指针变量 gp 可以初始化为 GLOBAL 的地址,因为该地址总是相同的。类似地,*rp 可以初始化,因为 rp 是一个局部变量,具有非常量的初始化器,每次进入该块,LOCAL 有新地址,然后把它赋给 rp。

一初始化集合类型

个“集合”类型是一个结构、联合或数组类型。如果一个集合类型包含集合类型的成员,可以递归应用初始化规则。

语法

初始化器:

{初始化器表} /*对于集合初始化*/

{初始化器表,}

初始化器表:

初始化器

初始化器表,初始化器

初始化器表是由逗号分隔的初始化器的表,该表中每个初始化器是一个常量表达式或一个初始化器表。因此,初始化器是嵌套的。这个格式对于初始化一个集合类型的集合成员时很有用,正如下一小节的例子说明的。但如果一个自动标识符的初始化器是单个表达式,它不需要是一个常量表达式,对于标识符的赋值只不过需要相应的类型。

对于每个初始化器表,常量表达式的值依次赋给该集合变量的对应成员。

如果初始化器表的值比一个集合类型的少,对于外部和静态变量,该集合类型的余下成员初始化为 0。一个自动标识符不显式初始化,则其初始值是不确定的。如果初始化器表的值比一个集合类型的多,则产生一个错误。这些规则应用于每个嵌入的初始化器表,以及整个集合。

一个结构的初始化器是一个相同类型的表达式或包含在花括号({})中的成员的初始化器的表。无名的位域成员不进行初始化。

当初始化一个联合时,初始化器表必须是单个常量表达式。该常量表达式的值赋给联合的第一个成员。

如果一个数组有未知的尺寸,初始化器的成员确定该数组的尺寸,其类型也变成完整的。在 C 中没有方法指出一个初始化器的重复,或者初始化一个数组中间的元素而不提供其前面的值。如果你在程序中需要这种操作,在汇编语言中编写该例程。注意:初始化器的成员可以设置数组的尺寸:

```
int x[]={0,1,2}
```

但如果你指出尺寸和给出错误的初始化器个数,编译器会产生一个错

误。

Microsoft 特殊处→

一个数组的最大尺寸由 size_t 定义,它定义在 STDDEF.H 头文件中。size_t 是从 0x00000000 到 0x7CFFFFFF 范围的一个 unsigned int 值。

Microsoft 特殊处结束

例子

这个例子给出了一个数组的初始化器:

```
int p[4][3]=
  {
  {1,1,1},
  {2,2,2},
  {3,3,3,},
  {4,4,4,},
  };
```

这个语句说明 p 为一个 4×3 数组,并初始化它的第一行的元素为 1,第二行的元素为 2,如此等等直到第四行。注意第三行和第四行的初始化器表在最后常量表达式之后包含逗号。最后的初始化器表({4,4,4,},)也跟有逗号,这些额外的逗号是允许的但并不需要。分隔常量表达式只需要一个逗号,分隔初始化器表也只需要一个逗号。

如果集合成员没有嵌入的初始化器,值简单地依次赋给子集合的每个元素。因此,前面例子中的初始化与下面的语句是等价的:

```
int p[4][3]=
```

```
{
   1,1,1,2,2,2,3,3,3,4,4,4
};
```

初始化器表中的单个初始化器可以用花括号括起来,这样有助于明确上述例子。当你初始化一个集合变量时,你必须小心适当地使用花括号和初始化器表。如下的例子更详细地说明了编译器对花括号的解释。

```
typedef struct
{
      int n1,n2,n3;
} triplet;

triplet nlist[2][3] =
{
      {{1,2,3},{4,5,6},{7,8,9}},                    /*行 1*/
      {{10,11,12},{13,14,15},{16,17,18}}            /*行 2*/
};
```

在这个例子中,nlist 说明为 2×3 结构数组,每个结构有 3 个成员,该初始化器的第 1 行赋给 nlist 的第一行,如下:

1. 行 1 的第一个左花括号指示该编译器开始初始化 nlist 的第一个集合成员(也就是 nlist[0])。
2. 第二个左花括号指出开始初始化 nlist[0]的第一个集合成员(也就是 nlist[0][0]处的结构)。

3. 第一个右花括号终止结构 nlist[0][0]的初始化,下一个左花括号开始 nlist[0][1]的初始化。

4. 该过程继续直到该行结束,这时封闭的右花括号终止 nlist[0]的初始化。行 2 以同样的方式把值赋给 nlist 的第二行。注意,行 1 和行 2 封闭初始化器的最外面的花括号是需要的。如下例省略了外面的花括号,构造时产生一个错误:

```
triplet nlist[2][3]=                    /*这引起一个错误*/
{
   {1,2,3},{4,5,6},{7,8,9},             /*行 1*/
   {10,11,12},{13,14,15},{16,17,18}     /*行 2*/
   };
```

在这个构造中,行 1 的第一个左花括号开始初始化 nlist[0],它是三个结构的一个数组。

值 1、2 和 3 赋给第一个结构的三个成员。当遇到下一个右花括号(值 3 之后的)时,完成 nlist[0]的初始化,该三结构数组中的两个余下结构自动初始化为 0,类似地,{4,5,6}初始化 nlist 的第二行中的第一个结构。nlist[1]的余下两个结构设置为 0。当编译器遇到下一个初始化器表({7,8,9}),它试图初始化 nlist[2]。由于 nlist 只有两行,这个试图导致一个错误。

在下面的例子中,X 的三个 int 成员分别初始化为 1,2 和 3。

```
struct list
{
   int i,j,k;
```

```
    float m[2][3];
} x = {
    1,
    2,
    3,
    {4.0,4.0,4.0}
  };
```

在上述 list 结构中,m 的第一行中的三个元素初始化为 4.0;m 的下行的元素缺省初始化为 0.0。

```
union
  {
      char x[2][3];
      int i,j,k;
  } y={{
      {'1'},
      {'4'}
  }
};
```

在这个例子中,联合变量 y 被初始化。该联合的第一个元素是一个数组,因此,该初始化器是一个集合初始化器。初始化器表 {'1'}把值赋给该数组的第一行。由于该表中只有一个值,第一列中的元素初始化为字符 1,该行中余下两个元素缺省初始化为 0。类似地,x 的第二行的第一个元素

初始化为字符 4,该行中余下的两个元素初始化值 0。

初始化字符串

你可以用一个字符串文字(或宽字符串文字)初始化一个字符(或宽字符)数组,例如:

```
char code[]="abc";
```

初始化 code 为一个 4 元素的字符数组,第 4 个元素是空字符,以终止所有字符串文字。

一个标识符表的长度只能等于初始化标识符的个数。如果你指定一个数组的尺寸小于字符串的长度,额外的字符被忽略。例如,如下说明初始化 code 为一个三元素的字符数组:

```
char code[3]="abcd";
```

仅将初始化器的开头三个字符赋给 code。字符 d 和该字符串结尾的空格字符被丢弃。注意将建立一个非终结的字符串(也就是,没有一个 0 值标志结尾)并产生一个诊断消息指出这个条件。

说明:

```
char s[]="abc", t[3]="abc";
```

等价于:

```
char s[]={'a', 'b', 'c', '\0'},
     t[3]={ 'a', 'b', 'c'};
```

如果字符串短于指定的数组尺寸,该数组余下的元素初始化为 0。

Microsoft 特殊处→

在 Microsoft C 中,字符串文字的长度可达 2048 个字节。
Microsoft 特殊处结束

基本类型的存储

表 3.2 总结了与每个基本类型关联的存储。

表 3.2 基本类型的尺寸

类型	存储
char,unsigned char, signed char	1 字节
short, unsigned short	2 字节
int,unsigned int	4 字节
long,unsigned long	4 字节
float	4 字节
double	8 字节
long double	8 字节

C 数据类型属于通用目录。“整数”类型包括 char、int、short、long、signed、unsigned 和 enum。“浮点”类型包括 float、double 和 long double。“算术”类型包括所有浮点和整数类型。

类型 char

char 类型用于存储可表示字符集的一个成员的整数值。这个整数值是对

应指定字符的 ASCII 码。
Microsoft 特殊处→
一个 unsigned char 的字符值具有从 0 到 0xFF 十六进制值的范围。一个 signed char 具有从 0x80 到 0x7F 的范围。这些范围分别转换为 0 到 255 十进制和 -128 到 +127 十进制。编译器选项 /J 改变缺省值从 signed 变为 unsigned。
Microsoft 特殊处结束

类型 int

一个带符号或无符号 int 项的尺寸是特殊机器上一个整数的标准尺寸,例如,在 16 位操作系统中,int 类型通常是 16 位或 2 个字节。在 32 位操作系统中,int 类型通常是 32 位或 4 个字节。因此,int 类型根据目标环境等价于 short int 或 long int 类型,unsigned int 类型等价于 unsigned short 或 unsigned long 类型。int 类型都表示有符号的值,除非有其它指定。
类型指定符 int 和 unsigned int(或简称 unsigned)定义 C 语言的某种特征(例如,enum 类型)。在这些情况下,特殊实现的 int 和 unsigned int 的定义确定了实际存储。
Microsoft 特殊处→
有符号整数以 2 的补码格式表示。对于负数,最高位保存符号 1,对于正数和 0,它为 0。值的范围在表 1.3 中给出,它们来自 LIMITS.H 头文件。
Microsoft 特殊处结束
注意:int 和 unsigned int 类型在 C 程序中广泛使用,因为它们允许一个特

定机器以其最有效的方式处理整数值。但由于 int 和 unsigned int 类型的尺寸是变化的,依赖于特定 int 尺寸的程序不能移植到其它机器中。为了使程序更具移植性,你可以使用一个 sizeof 运算符的表达式代替硬码数据尺寸(正如第 4 章“表达式和赋值”中“sizeof 运算符”讨论的)。

带尺寸的整数类型

Microsoft 特殊处→
Microsoft 特殊处支持带尺寸的整数类型。你可以使用 __intn 类型指示符说明 8、16、32 或 64 位整数变量,这里 n 是整型变量的尺寸,以位为单位。n 的值可以是 8、16、32 或 64。下面的例子说明了四种带尺寸的整型:

```
__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
```

开头三个带尺寸整型是具有相同尺寸的 ANSI 类型的同义词,这对于编写多平台交叉移植的代码是有用的。注意 __int8 数据类型与 char 类型是同义词,
__int16 与 short 类型是同义词,__int32 与 int 类型是同义词。__int64 类型没有等价的 ANSI 对应部分。
Microsoft 特殊处结束

float 类型

浮点数使用 IEEE(电子电气工程师协会)格式,单精度浮点类型有 4 个字节,由一个符号位,一个 8 位超过 127 二进制指数和一个 23 位尾部。尾部表示一个 1.0 到 2.0 之间的数。由于尾数的高端位总是 1,它不存储在这个数中。这种表示给出浮点类型近似从 3.4E-38 到 3.4E+38 的范围。

你根据应用的需要说明一个变量为浮点数或双精度数。这两种类型之间的差别是它们能表示、它们需要存储的有效位数和它们的范围。表 3.3 说明了它们在有效位数和存储要求方面的差异。

表 3.3 浮点类型

类型	有效数字	字节个数
浮点	6-7	4
双精度	15-16	8

浮点变量用一个尾部表示,它包含该数值和指数的值,指数包含该数值的大小次序。表 3.4 说明了每个浮点类型分配给尾部和指数的位数。任何浮点或双精度数最重要的位总是符号位,如果它为 1,该数值认为是负数;否则,该数值认为是正数。

表 3.4 指数和尾部的长度

类型	指数长度	尾部长度
浮点	8 位	23 位
双精度	11 位	52 位

因为指数以无符号格式存储,指数用它的可能值的一半进行偏置。对于浮点类型,该偏置为 127;对于双精度类型,该偏置为 1023。你可以通过从指数值中减去偏置值来计算实际的指数值。

尾数以大于或等于 1 且小于 2 的二进制分数进行存储。对于浮点和双精度类型,在尾数中最重要的位置中隐含一个引导的 1,因此,尾部实际上分别是 24 和 53 位长,即使该最重要的位没有存储在存储器中。

代替上面描述的存储方法,浮点软件包以反向规格化的数值存储二进制浮点数值,“反向规格化的数值”是具有保留指数值的非 0 浮点数值,尾数最重要的位总是 0。使用反向规格化格式,一个浮点数值的范围可以在精度上进行扩充。你不能控制一个浮点数是以规格化的格式还是反向规格化的格式表示。浮点软件包不使用反向规格化的格式,除非指数小于规格化的格式中能表示的最小值。

表 3.5 给出了每种浮点类型的变量中能存储的最小值和最大值。这个表中列出的值仅应用于规格化的浮点数值。反向规格化的浮点数值具有一个更小的最小值。注意 80x87 寄存器中保留的数值总是以 80 位规格化的格式表示的。仅当存储 32 位或 64 位浮点变量(float 类型和 long 类型的变量)时数值可以以反向规格化的格式进行表示。

表 3.5 浮点类型的范围

类型	最小值	最大值
浮点	1.175494351E-38	3.402823466E+38
双精度	2.2250738585072014E-308	1.7976931348623158E+308

如果精度不是很重要的,考虑为浮点变量使用浮点类型;相反如果精度是最重要的标准,使用双精度类型。
浮点变量可以提升为一种更大有效位数的类型(从浮点类型到双精度类型)。当你在浮点变量上执行算术运算时这种提升经常出现。算术运算总是取最高精度作为变量的高精度。
例如,考虑如下类型说明:

```
float f_short;
double f_long;
long double f_longer;


f_short=f_short*f_long;
```

在上述例子中,变量 f_short 提升为双精度类型,且由 f_long 相乘。那么结果在赋给 f_short 之前舍入为浮点类型。
在下面例子中(使用前面例子中的说明),算术运算在变量的浮点(32 位)精度上进行的。然后把结果提升为双精度类型:

```
f_longer=f_short*f_short;
```

double 类型

具有 double 类型的双精度值有 8 个字节。该格式类似于 float 格式,除了它有 11 位超过 1023 指数和一个 52 位尾数加上隐含的高端 1 位外。该格式给出 double 类型的近似范围是从 1.7E-308 到 1.7E+308。
Microsoft 特殊处→

double 类型包含 64 位:1 个符号位,11 个指数位和 52 个尾数位。它的范围是+/-1.7E308,至少有 15 个数字的精度。
Microsoft 特殊处结束

long double 类型

一个变量的值的范围由给定位数内部表示的最小值和最大值定界。但因为 C 的约定规则(在第 4 章“表达式和赋值”中的“类型转换”中详细讨论),你不能在表达式中为一个特定类型的常量总是使用最大值或最小值。
例如,常量表达式-32768 由算术负运算符(-)应用于常量 32768 组成的,由于 32768 太大不能作为一个 short int 数表示,它被给以 long 类型。结果,常量表达式-32768 为 long 类型。
你只能通过类型造型强制为 short 类型来以 short int 表示-32768。在类型造型中没有信息丢弃,因为-32768 可以以 2 字节进行内部表示。在十进制表示中,值 65000 作为一个有符号的常量,它给以 long 类型,因为 65000 不适合于 short。这样的 65000 值只能通过类型造型强制为 unsigned short 类型或给定该值以八进制或十六进制表示,或者指定它为 65000U 来作为 unsigned short 类型表示。你可以造型强制这个值为 unsigned short 类型而不丢失信息,因为作为一个无符号数值,65000 适合于以 2 个字节表示。
Microsoft 特殊处→
long double 包含 80 个位:1 位表示符号,15 位表示指数,64 位表示尾数。

它的范围是+/-1.2E4932,其精度至少为 19 个数字。虽然 long double 和 double 是分开的类型,但 long double 和 double 的表示是一致的。
Microsoft 特殊处结束

不完整类型

一个不完整类型是描述一个标识符而缺乏确定该标识符尺寸所需信息的类型。一个“不完整类型”可以是:

- 一个没有指出其成员的结构类型。
- 一个没有指出其成员的联合类型。
- 一个没有指出维数的数组类型。

void 类型是一个不能完成的不完整类型。为了完成一个不完整类型,指出缺少的信息,如下例子说明如何建立和完成不完整的类型。

- 为了建立一个不完整结构类型,说明一个结构类型不指出它的成员。在如下例子中,PS 指针指向一个称为 student 的不完整结构类型:

```
struct student *ps;
```

- 为了完成一个不完整结构类型,在后面同一范围内用其指定的成员说明相同的结构类型:

```
struct   student
{
 int unm;
```

```
}                    /*student 结构现在完整了*/
```

- 为了建立一个不完整的数组类型,说明一个数组类型而不指定它的重复计数,例如:

```
char a[];   /*a 有不完整类型*/
```

- 为了完成一个不完整的数组类型,在后面同一范围内用指定的重复计数说明相同的名称:

```
char a[25];   /*a 现在有一个完整的类型*/
```

typedef 说明

一个 typedef 说明是一个用 typedef 作为存储类的说明。该说明符变成一个新类型。你可以使用 typedef 说明为 C 已经定义的类型或你已说明过的类型构造更短或更有意义的名称。

typedef 名称允许你封装可以改变的详细的实现。

一个 typedef 说明的解释方式与变量或函数说明的解释方式相同,但该标识符代替该说明指定的假设类型,变成该类型的同义词。

语法

说明:

说明指示符 **初始化说明符表** $_{opt}$

说明指示符:

存储类指示符 **说明指示符** $_{opt}$

类型指示符 **说明指示符** $_{opt}$

类型修饰符 说明指示符 opt

存储类指示符:

typedef

类型指示符:

void

char

short

int

long

float

double

signed

unsigned

结构或联合指示符

枚举指示符

typedef 名称

typedef 名称:

标识符

注意，一个 typedef 说明不能建立类型,它为现存类型或以其它方式指定类型的名称建立一个同义词。当一个 typedef 名称用作一个类型指示符时,它可以用某些类型指示符进行组合,但不能是其它的。可接受的修饰符包括 const 和 volatile。

typedef 名称与普通标识符共享名称空间(有关更多信息,参见第 2 章“程序结构”中的“名称空间”),因此,一个程序可以有一个 typedef 名称和相同名称的局部范围标识符。例如:

```
typedef char FlagType;

int main()
{
}

int myproc(int)
{
   int FlagType;
}
```

当说明一个与 typedef 相同名称的局部范围标识符时,或当在同一范围或更内层范围中说明一个结构或联合的成员时,必须指出类型指示符。如下例子说明了这种约束:

```
typedef char FlagType;
const FlagType x;
```

为了为一个标识符,一个结构成员或者一个联合成员重新使用 FlagType 名称,必须提供其类型:

```
const int FlagType;   /*需要类型指示符*/
```

如下这样是不足够的:

```
const FlagType;     /*不完整规格*/
```

因为 FlagType 作为该类型的一部分,不是一个重新说明的标识符。如下说明是一个非法的说明:

```
int ;               /*非法说明*/
```

你可以用 typedef 说明任何类型,包括指针、函数和数组类型。你可以在定义一个结构或联合类型之前说明该结构或联合类型的指针的 typedef 名称,以及该定义与该说明具有相同的可见性。

typedef 名称可以用于提高代码的可读性。所有如下的三个 signal 说明都精确指出相同类型,开头一个没有使用任何 typedef 名称:

```
typedef void fv(int),(*pfv)(int);   /*typedef 说明*/


void (*signal(int,void(*)(int)))(int);
fv *signal(int,fv*);   /*使用 typedef 类型*/
pfv signal(int,pfv);   /*使用 typedef 类型*/
```

例子

下面的例子说明了 typedef 说明:

```
typedef int WHOLE;   /*说明 WHOLE 为 int 的同义词*/
```

注意 WHOLE 现在可以用在变量说明中,例如 WHOLE i;或 const WHOLE i;但说明 long WHOLE i;是非法的。

```
typedef struct club
{
   char name[30];
```

```
    int size,year;
}GROUP;
```

这个语句说明 GROUP 是一个具有三个成员的结构类型。因为也指定了一个结构标志 club,typedef 名称(GROUP)或结构标志可以用在说明中,你必须用标志使用结构关键字,不能用 typedef 名称使用结构关键字。

```
typedef GROUP *PG;   /*使用前面 typedef 名称说明一个指针*/
```

类型 PG 说明为 GROUP 类型的一个指针,它又被定义为一个结构类型。

```
typedef void DRAWF(int,int );
```

这个例子提供 DRAWF 类型,它作为一个函数,没有返回值,有两个 int 参量。例如,说明:

```
DRAWF box;
```

等价于说明:

```
void box(int,int);
```

扩充的存储类属性

Microsoft 特殊处→

扩充的属性语法简化和标准化 C 语言的 Microsoft 特殊处扩充。使用扩充的属性的存储类属性包括 thread、naked、dllimport 和 dllexport。

特定存储类信息的扩充属性语法使用 __declspec 关键字,它指出一个给定类型的实例，采用 Microsoft 特殊存储类属性(thread、naked、dllimport 或 dllexport)进行存储。其它存储类修饰符的例子包括 static 和 extern 关

键字,但这些关键字是 ANSI C 部分,因此不能被扩充属性语法覆盖。

语法

存储类指示符:

__declspec(扩充的说明修饰符序列)/*Microsoft 特殊处*/

扩充的说明修饰符序列:

扩充的说明修饰符 opt

扩充的说明修饰符序列 扩充的说明修饰符

扩充的说明修饰符:

thread

naked

dllimport

dllexport

用空白分隔说明修饰符。注意扩充的说明修饰符序列可以为空,在这种情况下,__declspec 没有作用。

thread、naked、dllimport 和 dllexport 存储类属性仅是它们应用的数据或函数的说明的特性(property);它们不能重新定义函数本身的类型属性。thread 属性只影响数据。naked 属性只影响函数。dllimport 和 dllexport 属性影响函数和数据。

Microsoft 特处殊结束

DLL 输入和输出

Microsoft 特殊处→

dllimport 和 dllexport 存储类修饰符是 C 语言的 Microsoft 特殊处扩充。这些修饰符定义 DLL 的客户界面(可扩充文件或另外的 DLL)。有关使用这些修饰符的特殊信息,参见第 6 章“函数”中的“DLL 输入和输出函数”。
Microsoft 特殊处结束

Naked

Microsoft 特殊处→
naked 存储类属性是 C 语言的 Microsoft 特殊处扩充。编译器为用 naked 存储类属性说明的函数生成没有序言和结尾部分的代码。当你需要使用内联函数代码编写自己的序言/结尾部分代码序列时 naked 函数很有用。naked 函数对于编写虚拟设备驱动程序是有用的。
有关使用 naked 函数的特定信息,参见第 6 章“函数”中的“naked 函数”。
Microsoft 特殊处结束

thread 局部存储

Microsoft 特殊处→
thread 局部存储(TLS)是在一个给定的多线程进程中每个线程分配特定线程数据的机制。在标准多线程程序中,数据在一个给定进程的所有线程中共享,而线程局部存储是分配每个线程数据的机制。有关线程完整的讨论,参见联机“平台 SDK”文档中的“进程和线程”。

Microsoft C 语言包括扩充的存储类属性,thread 与__declspec 关键字一起使用说明一个线程局部变量。例如,如下代码说明一个整数线程局部变量并用一个值对其初始化:

```
_ _declspec(thread)int tls_i=1;
```

当你说明静态约束线程局部变量时必须遵守这些指南:

- _ _declspec(thread)的使用可以与 DDL 输入的延迟加载进行交互。
- 你只能把 thread 属性应用于数据说明和定义。它不能用于函数说明或定义。

例如,如下代码产生一个编译器错误:

```
#define thread __declspec(thread)
thread void func();                /*错误*/
```

- 你只能在具有静态存储期的数据项上指定 thread 属性。这包括全局数据(static 和 extern)和局部静态数据。不能用 thread 属性说明自动数据。例如,如下代码产生编译器错误;

```
#define thread __declspec(thread)
void funcl()
{
   thread int tls_i; /*错误*/
}

int func2(thread int tls_i);    /*错误*/
{
```

```
    return tls_i;
}
```

- 你必须为线程局部数据的说明和定义使用 thread 属性,而无论该说明和定义是出现在相同文件或分开文件中,例如,如下代码产生一个错误:

```
#define thread __declspec(thread)
extern int tls_i;           /*这产生一个错误,因为*/
int thread tls_i;           /*说明和定义不相同*/
```

- 你不能使用 thread 属性作为一个类型修饰符。例如,如下代码产生一个编译器错误:

```
char *ch_ _declspec(thread);/*错误*/
```

- 一个线程局部变量的地址不能作为常量,涉及到该地址的任何表达式不能作为一个常量表达式。这意味着你不能使用一个线程局部变量的地址作为一个指针的初始化器。例如,编译器标志的如下代码有一个错误:

```
#define thread __declspec(thread)
thread int tls_i;
int *p=&tls_i; /*错误*/
```

- C 允许用涉及自身引用的表达式初始化一个变量,但仅对非静态范围的对象,例如:

```
#define thread __declspec(thread)
thread int tls_i=tls_i;                 /*错误*/
```

```
int j=j;                                    /*错误*/
thread int tls_i=sizeof(tls_i)    /*正确*/
```

注意一个初始化变量的 sizeof 表达式不构成对自身的引用,因而是允许的。

- __declspec(thread)的使用可以与 DLL 输入的延迟交互。

有关使用线程属性的更多信息,参见联机“Microsoft Visual C++6.0 程序员指南”中的“多线程主题”。

Microsoft 特殊处结束

第 4 章　表达式和赋值

本章描述如何在 C 语言中构造表达式和对其赋值。常量、标识符、字符串和函数调用都是在表达式中操作的操作数。C 语言具有所有常用的语言运算符。本章讨论这些运算符以及对 C 或 Microsoft 是唯一的运算符。讨论的主题包括:

- 操作数和表达式
- 运算符
- 类型转换

操作数和表达式

一个“操作数”是一个运算符作用的一个实体。一个“表达式”是执行这些动作的任何组合的运算符和操作数序列:

- 计算一个值
- 设计一个对象或函数
- 产生副作用

在 C 中的操作数包括常量、标识符、字符串、函数调用、下标表达式、成员选择表达式和通过运算符组合操作数或在圆括号中包括操作数而形

成的复杂表达式。这些操作数的语法在下一节“基本表达式”中给出。

基本表达式

表达式中的操作数称为“基本表达式”。

语法

基本表达式:

标识符

常量

字符串文字

(表达式)

表达式:

赋值表达式

表达式,赋值表达式

基本表达式中的标识符

标识符有整数、float、enum、struct、union、数组、指针或函数类型。一个标识符是一个基本表达式,它被说明用于设计一个对象(这种情况下它是一个 l 值)或一个函数(这种情况下它是一个函数指示符)。有关 l 值的定义参见“L 值和 R 值表达式”。

由一个数组标识符表示的指针值不是变量,因此,一个数组标识符不能形成一个赋值运算的左边操作数。因此不是一个可修改的 l 值。

一个作为函数说明的标识符表示一个其值是该函数地址的指针。该指针

地址指向一个返回指定类型值的函数。因此,函数标识符也不能是赋值操作中的 l 值。有关更多信息,参见第 1 章“C 的基本元素”中的“标识符”。

基本表达式中的常量

一个常量操作数有它给定常量值的值和类型。一个字符常量有 int 类型。一个整数常量有 int、long、unsigned int 或 unsigned long 类型,这取决于该整数的尺寸和指定值的方式。有关更多信息参见第 1 章“C 的基本元素”中的“常量”。

基本表达式中的字符串文字

一个“字符串文字”是一个字符、宽字符或用双引号括起的相邻字符序列。由于它们不是变量,因此字符串文字或任何它们的元素都不能作为一个赋值操作的左边操作数。一个字符串文字的类型是 char 的数组(或宽字符串文字的 wchar_t 数组)。在表达式中数组都转换成指针。有关字符串的更多信息,参见第 1 章“C 的基本元素”的“字符串文字”。

圆括号中的表达式

你可以在圆括号中括起任何操作数而不改变该括起的表达式的类型或值。例如,在表达式中:

```
(10+5)/5
```

圆括号中的 10+5 意味着首先对 10+5 求值,它变成除法(/)运算符的左操

作数。(10+5)/5 的结果是 3。没有圆括号的 10+5/5 的结果是 11。

虽然圆括号影响一个表达式中组合操作数的方式,它们不能保证在所有情况下的一个特殊求值次序。例如,如下表达式的圆括号和从左到右的组合都不能保证 i 在每个子表达式中是什么样的值。

(i++ +1)*(2+i)

编译器自由以任何次序计算乘法两边的值。如果 i 的初值为 0,整个表达式以如下两个语句求值:

(0+1+1)*(2+1)

(0+1+1)*(2+0)

副作用导致的异常在本章后面“副作用”中讨论。

L 值和 R 值表达式

指的是存储器位置的表达式称为“l 值”表达式,一个 l 值表示一个存储区域的“定位器”值或“左边”值,它隐含出现在等号(=)的左边。l 值经常是标识符。

指的是右修改的位置的表达式称为“可修改的 l 值”。一个可修改的 l 值不能有一个数组类型,一个不完整类型或具有 const 属性的类型,对于可以是可修改 l 值的结构和联合,它们必须没有 const 属性的成员。标识符的名称指示一个存储位置,而该变量的值是存储在该位置的值。

如果一个标识符指向一个存储器位置,其类型是算术、结构、联合或指针,则该标识符是一个可修改的 l 值。例如,如果 ptr 是一个存储区域的指针,那么 *ptr 是一个可修改的 l 值,指示该存储区域由 ptr 指向。

如下任何 C 表达式可以是 l 值表达式:

- 一个整数、浮点、指针、结构或联合类型的标识符。
- 一个不对数组求值的下标([])表达式。
- 一个成员选择表达式(->或.)
- 一个不指向一个数组的单目间接访问(*)表达式。
- 一个圆括号中的 l 值表达式。
- 一个 const 对象(一个不能修改的 l 值)。

本语“r 值”有时用来描述一个表达式的值以区别于一个 l 值。所有 l 值都是 r 值,但不是所有 r 值都是 l 值。

Microsoft 特殊处→

C 包括 ANSI C 标准的一个扩充是允许 l 值造型用作 l 值,以及该对象的尺寸不会通过该造型而伸长(有关更多信息,参见本章末尾的“类型造型转换”)。如下例子说明了这种特征:

```
char *p;
short i;
long l;

(long *) p=&l;      /*合法造型*/
(long) i=l;         /*非法造型*/
```

C 缺省能使 Microsoft 扩充。使用/Za 编译器选项禁止这种扩充。

Microsoft 特殊处结束

常量表达式

一个常量表达式在编译时求值,而不是在运行时求值,可以使用在一个常量可以被使用的任何地方。常量表达式必须求值为一个在该类型可表示值的范围内的常量。一个常量表达式的操作数可以是整数常量、字符常量、浮点常量、枚举常量、类型造型、sizeof 表达式和其它常量表达式。

语法

常量表达式:

    条件表达式

条件表达式:

    逻辑 OR 表达式

    逻辑 OR 表达式 ? 表达式 : 条件表达式

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:如下之一:

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

结构说明符、枚举器、直接说明符、直接抽象说明符和标号语句的非终结符包含常量表达式非终结符。

一个整数常量表达式必须用于指定一个结构位域成员的尺寸、一个枚举常量的值、一个数组的尺寸或一个 case 常量的值。

用在预处理器命令中的常量表达式满足一些附加限制。结果它们是众所周知的“限制的常量表达式”。一个限制的常量表达式不能包含 sizeof 表达式、枚举常量、到任何类型的类型造型或浮点类型常量。但它可以包含定义的特定常量表达式(标识符)。

表达式求值

涉及赋值、单目增 1、单目减 1 或调用一个函数的表达式具有伴随它们求值的结果(副作用),当到达一个“顺序点”时,该顺序点之前的任何事情包括任何副作用都保证在开始计算该顺序点之后的任何事情之前进行求值。

通过对一个表达式求值导致改变“副作用”。副作用出现在一个变量的值被一个表达式求值所改变的任何时候。所有赋值操作都有副作用。如果函数调用通过直接赋值或使用一个指针间接赋值来改变一个外部的可见项的值,则该函数调用也有副作用。

副作用

一个表达式的求值次序是由指定的实现确定的,除了当该语言保证一个特殊的求值次序外(正如本章后面的“优先级和求值次序”中描述的)。例如,在如下函数调用中出现副作用:

```
add(i+1,i=j+2);
```

```
myproc(getc(),getc());
```

一个函数调用的参量可以以任何次序求值。表达式 i+1 可以在 i=j+2 之前求值,或者 i=j+2 可以在 i+1 之前求值,每种情况下的结果是不同的,同样,不可能保证哪个字符实际传送给 myproc。由于单目增 1 和减 1 操作涉及到赋值,如下例子中的操作可能导致副作用:

```
x[i]=i++;
```

在这个例子中,被修改的 x 的值是不可预料的。下标的值可以是 i 的新值或旧值。其结果可能在不同编译器下或不同优化层都是不同的。由于 C 不确定副作用的求值次序,上面讨论的求值方法都是正确的,也可以实现。

为了确保你的代码是可移植和清楚的,避免使用其副作用依赖于特殊求值次序的语句。

顺序点

连贯的“顺序点”之间,一个对象的值仅能被一个表达式改变一次。C 语言定义如下顺序点:

- 逻辑 AND 运算符(&&)的左操作数。逻辑 AND 运算符的左操作数在继续之前完成求值和完成所有的副作用。如果该左操作数求值为假(0),不对其它操作数求值。
- 逻辑 OR 运算符(||)的左操作数。逻辑 OR 运算符的左操作数在继续之前完成求值和完成所有的副作用。如果该左操作数求值为真(非 0),不对其它操作数求值。

- 逗号运算符的左操作数。逗号运算符的左操作数在继续之前完成求值和完成所有的副作用。逗号运算符的两个操作数总是求值。注意在一个函数调用中的逗号运算符不保证求值的次序。
- 函数调用运算符。在进入一个函数之前对该函数的所有参量求值和完成所有副作用,在参量中不指定求值的次序。
- 条件运算符的第一个操作数。在继续之前完成该条件运算符的第一个操作数的求值并完成所有的副作用。
- 一个完整初始化表达式结束(也就是,一个不是另一个表达式部分的表达式,例如,在一个说明语句中一个初始化的结果)。
- 一个表达式语句中的表达式。表达式语句由一个任选表达式后跟一个分号(;)。该表达式为了副作用而求值,并紧跟这个求值求一个顺序点。
- 在一个选择(if 或 switch)语句中的控制表达式。在依赖于该选择的代码执行之前完成该表达式的求值,并完成所有的副作用。
- 一个 while 或 do 语句的控制表达式。在 while 或 do 循环的下一次迭代中的任何语句执行之前完成该表达式的求值,并完成所有的副作用。
- 一个 for 语句的所有三个表达式。在 for 循环的下一次迭代中的任何语句执行之前完成该表达式的求值,并完成所有的副作用。
- 一个 return 语句中的表达式。在控制返回到调用函数之前完成该表达式的求值,并完成所有的副作用。

运算符

有三种类型的运算符。单目表达式由一个单目运算符加到一个操作数组成,或者 sizeof 关键字跟一个表达式,该表达式可以是一个变量的名称或一个造型表达式。如果该表达式是一个造型表达式,它必须包括在圆括号中。一个双目表达式由两个操作数加上连接的双目运算符组成。一个三目表达式由三个操作数加上连接的条件表达式运算符组成。

C 中包括如下单目运算符:

符号	名称
- ~ !	取反和取非运算符
* &	间接访问和取地址运算符
sizeof	尺寸运算符
+	单目加法运算符
++ --	单目增 1 和减 1 运算符

双目运算符从左向右结合。C 提供如下双目运算符:

符号	名称
* / %	乘法运算符
+ -	加法运算符
<< >>	移位运算符
<> <= >= == !=	关系运算符
& | ^	按位运算符
&& ||	逻辑运算符

, 顺序求值运算符

条件表达式运算符比双目表达式具有较低的优先级,且在右结合上不同于它们。带运算符的表达式也包括赋值表达式,它使用单目或双目赋值运算符。单目赋值运算符是增 1(++)和减 1(--)运算符;双目赋值运算符是简单赋值运算符(=)和复合赋值运算符。

每个复合赋值运算符是另外的双目运算符与简单赋值运算符的组合。

优先级和求值次序

C 运算符的优先级和结合律影响表达式中操作数的组合和求值。一个运算符的优先级仅在出现更高或更低优先级的另外运算符时才有意义。具有高优先级运算符的表达式首先求值。优先级也可以通过词“绑定”(binding)来描述。具有更高优先级的运算符被说成具有更紧密的联结。

表 4.1 总结了 C 运算符的优先级和结合律(运算符求值的次序),以优先级从高到低列出。而几个运算符只能一起出现,它们具有相等的优先级并根据它们的结合律进行求值。该表中的运算符在以“后缀运算符”开头的小节中。本节的余下部分给出了优先级和结合律一般的信息。

表 4.1　C 运算符的优先级和结合律

符号[1]	操作类型	结合律
[] () . -> 后缀++和后缀--	表达式	从左向右
前缀++和前缀-- sizeof & + _ ~ !	单目	从右向左
类型造型	单目	从右向左
* / %	乘法	从左向右
+ -	加法	从左向右
<< >>	位移	从左向右
<> <= >=	关系	从左向右
== !=	相等	从左向右
&	按位 AND	从左向右
^	按位异或	从左向右
|	按位 OR	从左向右
&&	逻辑 AND	从左向右
||	逻辑 OR	从左向右
?:	条件表达式	从右向左
= *= /= %=	简单和复合赋值[2]	从右向左
+= -= <<= >>=		
&= ^= |=		

续表

,	顺序求值	从左向右

1. 运算符以优先级递降次序列出。如果同一行或同一组中有几个运算符出现,它们具有相同的优先级。

2.所有简单和复合赋值运算符具有相同的优先级。

一个表达式可以包含几个具有相同优先级的运算符。当在一个表达式的同一层中出现几个这样的运算符时,根据这些运算符的结合律从右向左或从左向右进行求值。求值的方向不影响这样的表达式的结果,该表达式在同层中可以包含多个乘法(*)、加法(+)或双目按位(&|^)运算符。优先级的次序不是由语言定义的。编译器如果能够保证一致的结果,可以以任意次序对表达式求值。

只有顺序求值(,)、逻辑 AND(&&)、逻辑 OR(||)、条件表达式(?:)和函数调用运算符设立顺序点,因此保证这些运算符的特殊求值次序。函数调用运算符是该函数标识符之后的一组圆括号。顺序求值运算符(,)保证从左向右对它的运算符求值(注意,在一个函数调用中逗号运算符号不同于顺序求值运算符,不提供任何这样的保证)。有关更多信息,参见本章前面的“顺序点”。

逻辑运算符也保证它们的运算符从左向右求值,但它们求值需要确定该表达式结果的最少的运算符个数。这称之为“短路”求值。因些,该表达式的一些运算符可能不求值,例如,在表达式中:

x && y++

只有 x 为真(非 0),第二个运算符 y++才被求值,因此,如果 x 为假(0),则 y

不增 1。

例子

下表说明一个编译器如何自动绑定几个基本表达式。

表达式	自动联结
a&b||c	(a&b)||c
a=b||c	a=(b||c)
q&&r||s--	(q&&r)||s--

在第一个表达式中,按位 AND 运算符的优先级比逻辑 OR 运算符(||)的优先级高,因此,a&b 形成了逻辑 OR 运算的第一个操作数。

在第二个表达式中,逻辑 OR 运算符(||)比简单赋值运算符(=)有更高的优先级,因此 b||c 在赋值中作为右边运算符进行组合。注意赋给 a 的值是 0 或 1。

第三个表达式说明一个正确地组成的表达式可能产生一个不可预料的结果。逻辑 AND 运算符(&&)比逻辑 OR 运算符(||)的优先级高,因此,q&&r 结合成一个操作数。由于逻辑运算符保证从左向右求值,q&&r 在 s--之前被求值。如果 q&&r 求值为非 0 值,s --不再求值,s 不减 1。

如果 s 不减 1,可能导致程序出问题,s--应在表达式中作为第一个操作数出现,或者 s 在一个分开的操作中减 1。

下面的表达式是非法的,在编译时产生一个诊断消息:

非法表达式	缺省组合
p==0?p+=1:p+=2	(p==0?p+=1:p)+=2

在这个表达式中,相等运算符(==)有最高的优先级,因此 p==0 组合成一个

操作数。条件表达式运算符(?:)有次高的优先级,它的第一个操作数是p==0,第二个操作数是 p+=1。但该条件表达式运算符的最后操作数被认为是 p 而不是 p+=2,这是由于 p 与该条件表达式运算符的结合比与该复合赋值运算符的绑定更紧密。因为+=2 没有左边运算符,出现一个语法错误。

你可以使用圆括号防止这种错误出现并产生更可读的代码。例如,你可以使用如下圆括号以改正前面的例子并使之更清楚:

```
(p==0)?(p+=1):(p+=2)
```

常用的算术转换

大多数 C 运算符执行类型转换把一个表达式的操作数转换成共同的类型或扩充短的值为机器操作中使用的整数尺寸。C 运算符执行的转换依赖于特定运算符和操作数的类型或操作数。但很多运算符在整数和浮点类型的操作数上执行相似的转换,这些转换就是众所周知的“算术转换”。一个操作数值转换成一个兼容的类型使其值不发生改变。

下面总结的算术转换称为“常用的算术转换”。这些步骤仅应用于算术类型的双目运算符以及两个操作数不是相同类型的情况。其目的是产生一个公共的类型,它也是最后结果的类型。为了确定实际发生的转换,编译器把如下算法应用于表达式中的双目操作。下面步骤的次序不是按优先级给出的:

1. 如果操作数之一类型为 long double,则其它操作数转换成 long double 类型。

2. 如果上述条件不满足,且操作数之一类型为 double,则其它操作数转换成 double 类型。

3.如果上述两个条件不满足,且操作数之一为 float 类型,则其它操作数转换成 float 类型。

4.如果上述三个条件都不满足(没有操作数为浮点类型),那么在操作数上执行如下整数转换:

- 如果操作数之一为 unsigned long 类型,则其它操作数转换成 unsigned long 类型。
- 如果上述条件不满足,且操作数之一为 long 类型,其它为 unsigned int 类型,则操作数都转换成 unsigned long 类型。
- 如果上述两个条件都不满足,且操作数之一为 long 类型,则其它操作数转换成 long 类型。
- 如果上述三个条件都不满足,且操作数之一为 unsigned int,则其它操作数转换成 unsigned int 类型。
- 如果上述条件都不满足,则操作数都转换成 int 类型。

以下例子说明了这些转换规则:

```
float    fVal;
double dVal;
inti      Val;
unsigned long ulVal;
    dVal = iVal * ulVal;    /*iVal 使用步骤 4 转换为 unsigned long
                              *乘法的结果转换成 double */
```

```
    dVal = ulVal + fVal;   /*ulVal使用步骤 3 转换为 float
  *加法的结果转换成 double */
```

后缀运算符

后缀运算符在表达式值中具有最高的优先级(最紧密的绑定)。

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[参量表达式表 $_{opt}$]
- 后缀表达式.标识符
- 后缀表达式->标识符
- 后缀表达式++
- 后缀表达式--

在这个优先级层中的运算符有数组下标、函数调用、结构和联合成员以及后缀增 1 和后缀减 1 运算符。

一维数组

一个后缀表达式后跟一个方括号([])中的表达式是一个数组对象的一个元素的下标表示。一个下标表达式表示这样地址的值,其地址是该后缀表达式加上表达式中指定值的位置,表示为:

后缀表达式[表达式]

通常,由后缀表达式表示的值是一个指针值,例如一个数组标识符和一个表达式是一个整数值。但所有语法上需要的是一个为指针类型的表达式,另一个为整数类型。因此,整数值可以在后缀表达式位置,指针值可以在方括号的表达式中或“下标”位置。例如,如下代码是合法的:

```
int sum *ptr,a[10];
int main
{
   ptr=a;
   sum=4[ptr];
}
```

下标表达式通常用于指向数组元素,但你可以将下标应用于任何指针。无论值的次序,表达式必须包括在方括号([])中。

下标表达式通过把整数值加上指针值,然后在结果上应用间接访问运算符(*)求值(有关间接访问运算符的讨论,参见本章后面的“间接访问和取地址运算符”)。实际上,如下四个表达式是相等的,假设 a 是一个指针,b 是一个整数:

```
a[b]
*(a+b)
*(b+a)
b[a]
```

根据加法运算符的转换规则(在“加法运算符”中给出),该整数值通过将它乘以指针地址类型的长度来将其转换成一个地址偏移量。

假如,假设标识符 line 指向一个 int 类型的数组,如下过程用于计算下标表达式 ling[i]的值:

1. 整数值 I 被一个 int 项长度确定的字节个数相乘。i 的转换值表示 i 个 int 位置。

2. 这个转换的值加上最初指针值(line)产生一个地址,它是距离 line 偏移 i 个 int 的位置。

3. 将间接访问运算符应用于该新地址。其结果是该位置数组元素的值(直观地为 line[i])。

下标表达式 line[0]表示 line 的第一个元素的值,由于它偏移 line 所表示的地址为 0。类似地,一个表达式例如 line[5]指的是偏移 line 为 5 个位置的元素或该数组的第 6 个元素。

多维数组

一个下标表达式也可以有多个下标,如下:

expression1[expression2][expression3]...

下标表达式从左向右结合。最左下标表达式 expression1[expression2]首先求值。其地址加上 expression1 和 expression2 构成一个指针表达式;expression3 加上这个指针表达式构成一个新的指针表达式,如此等等直到最后下标表达式已加上了。在最后下标表达式求值之后应用间接访问运算符(*),除非最后的指针值是一个数组类型的地址(参见下面的例子)。

具有多个下标的表达式指的是“多维数组”。一个多维数组是一个其元

素是数组的数组。例如,一个三维数组的第一个元素是一个两维数组。

例子

对于如下例子中,一个命名为 prop 的数组用三个元素说明,每个是一个 int 类型的 4×6 数组。

```
int prop[3][4][6];
int i,*ip,(*ipp)[6];
```

如下引用 prop 数组:

```
i=prop[0][0][1];
```

上面例子说明了如何引用 prop 的第 2 个 int 元素。数组以行存储,因此最后的下标最快地变化;表达式 prop[0][0][2]指的是该数组的下一个(第 3 个)元素,等等。

```
i=prop[2][1][3];
```

这个语句更加复杂地引用 prop 的单个元素。该表达式求值如下:

1. 第 1 个下标为 2,由 4×6 的 int 数组的尺寸相乘,再加上 prop 的指针值,其结果指向 prop 的第 3 个 4×6 数组。
2. 第 2 个下标为 1,由 6 个元素 int 数组的尺寸相乘,再加上 prop[5]表示的地址。
3. 该 6 元素数组的每一个元素是一个 int 值,因此,最后下标 3,在加到 prop[2][1]之前由一个 int 的尺寸相乘。结果指针是该 6 元素数组的第 4 个元素。
4. 将间接访问运算符应用到该指针值。其结果是该地址处的 int 元素。

如下两个例子说明了不应用间接访问运算符的情况:

ip=prop[2][1];

ipp=prop[2];

在上述语句的第一个语句中,表达式 prop[2][1]是该三维数组 prop 的有效引用。它指的是第一个 6 元素数组(上述说明的)。由于该指针值是一个数组的地址,不能应用间接访问运算符。

类似地,在第二个语句 ipp=prop[2]中,表达式 prop[2]的结果是一个二维数组的地址的指针值。

函数调用

一个“函数调用”是一个包含被调用函数的名称或函数指针值或者传送给该函数的任选参量的表达式。

语法

后缀表达式:

后缀表达式(参量表达式表 $_{opt}$)

参量表达式表:

赋值表达式

参量表达式表,赋值表达式

后缀表达式必须计算一个函数地址(例如,一个函数标识符或一个函数指针的值),参量表达式表是一个其值(“参量”)传送给该函数的表达式表(用逗号分隔表达式)。参量表达式表参量可以为空。

一个函数调用表达式具有该函数返回的值和类型,一个函数不能返回一个数组类型的对象,如果该函数的返回类型为 void(也就是,该函数说明为

没有返回值),则该函数调用表达式也有 void 类型(有关更多信息,参见第 6 章“函数”中的“函数调用”)。

结构和联合成员

一个“成员选择表达式”指的是结构和联合的成员,这样的一个表达式具有选择的成员的值和类型。

语法

后缀表达式.标识符

后缀表达式->标识符

这个表描述了成员选择表达式的两种格式:

1. 在第一种格式中,后缀表达式表示一个 struct 或 union 类型的值,标识符命名该指定结构或联合的一个成员。该操作的值就是该标识符的值,如果后缀表达式是一个 l 值,则该操作的值也是一个 l 值。有关更多信息,参见本章前面的“L 值和 R 值表达式”。

2. 在第二种格式中,后缀表达式表示一个结构或联合的一个指针,标识符命名该指定结构或数组的一个成员。其值是该标识符的值,且是一个 l 值。

成员选择表达式的两种格式具有类似的作用。事实上,涉及成员选择运算符(->)的表达式是这样的表达式的速记版本,这个表达式使用句点且该句点之前的表达式由作用于一个指针值的间接访问运算符组成。因此:

表达式->标识符

等价于:

(*表达式).标识符

这里表达式是一个指针值。

例子

如下例子指的是这种结构说明,有关在这些例子中使用间接访问运算符(*)的信息,参见本章后面的“间接访问和取地址运算符”。

```
struct pair
{
   int a;
   int b;
   struct pair *sp;
} item,list[10];
```

item 的一个成员选择表达式如下:

```
item.sp=&item
```

在上述例子中,将 item 结构的地址赋给该结构的 sp 成员。这样 item 包含自身的一个指针。

```
(item.sp)->a = 24;
```

在这个例子中,指针表达式 item.sp 与成员选择运算符(->)一起使用把一个值赋给 a:

```
list[8].b = 12;
```

这个语句说明如何从一个结构数组选择单个结构成员。

后缀增 1 和减 1 运算符

后缀增 1 和减 1 运算符的操作数是可修改的 l 值的标量类型。

语法

后缀表达式:

后缀表达式++

后缀表达式--

后缀增 1 或减 1 操作的结果是该操作数的值。在获得结果后,该操作数增 1(或减 1)。如下代码说明该后缀增 1 运算符。

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

本例中,变量 var 是与 0 进行比较,如果 var 在增 1 之间是正数,则执行下一个语句。首先,由 q 所指对象的值赋给由 q 所指的对象,然后,q 和 p 都增 1。

单目运算符

单目运算符出现它的操作数之前并从右向左结合。

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 造型表达式

sizeof 单目表达式

sizeof(类型名称)

单目运算符:如下之一

& * + - ~ !

前缀增 1 和减 1 运算符

当增 1 和减 1 运算符出现在操作数之间时，单目运算符(++和--)被称为“前缀”增 1 和减 1 运算符。

后缀增 1 和减 1 的优先级比前缀增 1 和减 1 运算符的优先级高。操作数必须是整数、浮点或指针类型以及必须是一个可修改的 l 值表达式(一个没有 const 属性的表达式)。其结果是一个 l 值。

当运算符出现在操作数之前时,该操作被增 1 或减 1,其新值是该表达式的结果。

一个整数或浮点类型的操作数增大或减小整数值 1,其结果的类型与操作数类型相同。

一个指针类型的操作数增大或减小它地址所指对象的尺寸,一个增 1 的指针指向下一个对象,一个减 1 的指针指向前一个对象。

例子

这个例子说明单目前缀减 1 运算符:

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

在这个例子中,变量 i 在作为 line 的下标之前减 1。

间接访问和取地址运算符

间接访问运算符(*)通过一个指针间接访问一个值。该操作值必须是一个值。该操作数必须是一个指针值。其操作结果是该操作数所指的值,也就是该操作数所指地址处的值。其结果类型是该操作数所指地址的类型。

如果操作数指向一个函数,其结果是一个函数指示符,如果它指向一个存储位置,其结果是一个 l 值指示该存储位置。

如果该指针值是无效的,其结果是不确定的。如下表包括无效指针值的一些最普遍的条件:

- 该指针是一个空指针。
- 该指针指出一个在引用时不可见的局部项的地址。
- 该指针指出一个不适合赋给该对象所指类型的地址。
- 该指针指出一个执行程序不能使用的地址。

取地址运算符(&)给出它的操作数的地址。取地址运算符的操作数可以是一个函数指示符或一个 l 值,该 l 值指示一个不是位域的对象,且没有用 register 存储类指示符说明。

取地址运算符的结果是该操作数的指针,该指针所指地址的类型是操作数的类型。

取地址运算符只能应用于在文件范围层说明的基本类型、结构或联合类型或者下标数组的引用。在这些表达式中,在该地址表达式中可以加上

或减去一个不包括取地址运算符的常量表达式。

例子

下面的例子使用三个说明：

```
int *pa,x;
int a[20];
double d;
```

下面的语句使用取地址运算符:

```
pa=&a[5];
```

该取地址运算符(&)取数组 a 的第 6 个元素的地址,把结果存储在指针变量 pa 中。

```
x=*pa;
```

在这个例子中使用间接访问运算符(&)访问存储在 pa 地址处的 int 值,把该值赋给整数变量 x。

```
if (x == *&x)
    printf("True\n");
```

这个例子打印单词 True,从而证实把间接访问运算符应用于 x 的地址的结果与 x 是相同的。

```
int roundup(void); /*函数说明 */
int *proundup=roundup;
int *pround=&roundup;
```

一旦说明了函数 roundup,该 roundup 的两个指针也说明和初始化了。第一个指针 proundup 仅使用该函数的名称初始化,而第二个指针 pround 在

初始化中使用取地址运算符。这两种初始化是相同的。

单目算术运算符

在下表中讨论 C 单目加、算术取反、按位取反和逻辑非运算符。

运算符	说明
+	单目加运算符位于圆括号中一个表达式的前面强制组合括起的操作。它用在涉及多个结合或交换双目运算符的表达式中。该操作数必须是算术类型。其结果是操作数的值。一个整数操作数要进行整数提升,结果的类型是提升的操作数的类型。
-	算术取反运算符产生该操作数的取反值(2 的补码)。该操作数必须是整数或浮点值。这个运算符执行常用的算术转换。
～	按位取反(或按位 NOT)运算符产生其操作数的按位取反。该操作数必须是整数类型。这个运算符执行常用的算术转换,其结果具有转换后的操作数的类型。
!	逻辑非(逻辑 NOT)运算符在操作数为真(非 0)时产生值 0,在操作数为假(0)时产生值 1。其结果具有 int 类型。该操作数必须是一个整数、浮点或指针值。

指针上的单目算术操作是非法的。

例子

如下例子说明单目算术运算符:

```
short x=987;
```

```
x=-x;
```
在上述例子中,x 的新值是 987 的算术取反即-987。
```
unsigned short y=0xAAAA;
y=~y;
```
在这个例子中,y 的新值是无符号值 0xAAAA 的按位取反即 0x5555。
```
if (!(x<y))
```
如果 x 大于或等于 y,该表达式的结果是 1(真)。如果 x 小于 y,其结果是 0(假)。

sizeof 运算符

sizeof 运算符以字节为单位给出存储该操作数的类型的对象所需要的存储数目。这个运算符允许你在程序中避免指定依赖于机器的数据尺寸。

语法

sizeof 单目表达式

sizeof(类型名称)

该操作数是一个为单目表达式或类型造型表达式(也就是,一个类型指示符放在圆括号中)的标识符。该单目表达式不能表示一个位域对象、一个不完整类型或一个函数指示符。其结果是一个无符号整数参量。标准头文件 STDDEF.H 定义这个类型为 size_t。

当你把该 sizeof 运算符应用到一个数组标识符时,其结果是整个数组的尺寸而不是该数组标识符表示的指针的尺寸。

当你把该 sizeof 运算符应用到一个结构或联合类型名称,或者一个结构

或联合类型的标识符时,其结果是该结构或联合的字节个数,包括内部的和尾部的填充。这个尺寸可以包括用于在存储器边界上对齐结构或联合的成员而使用的内部和尾部填充。因此,其结果可能不对应于所有成员所需存储相加的结果。

如果一个没有尺寸的数组是一个结构的最后元素,则 sizeof 运算符返回没有该数组的结构的尺寸。

```
buffer=calloc(100,sizeof(int));
```

这个例子使用 sizeof 运算符传送一个 int 的尺寸,它在不同的机器中可能不同,作为一个参量传送给名称为 calloc 的运行函数。由这个函数返回的值存储在 buffer 中。

```
static char *strings[] = {
    "this is string one",
    "this is string two",
    "this is string three",
    };
const int string_no = (sizeof strings) / sizeof strings[0]);
```

在这个例子中,strings 是一个 char 的指针的数组。该指针的成员是数组中的元素个数,但不是指定的。使用 sizeof 运算符计算该数组中的元素个数容易确定指针个数。const 整数值 string_no 初始化为这个数。因为这是一个 const 值,string_no 不能修改。

造型运算符

一个类型造型提供了在特定状态下显式转换一个对象类型的方法。

语法

造型表达式:

单目表达式

(类型名称)造型表达式

在类型造型强制转换之后，编译器处理造型表达式为类型名称所指的类型。造型可以把任何标量类型的对象转换成另一种标量类型的对象。显式类型造型也遵守本章后面“赋值转换”中的讨论的隐含转换规则。造型上的另外限制来自指定类型的实际尺寸或表示。有关整数类型的实际尺寸的信息,参见第 3 章“说明和类型”中的“基本类型的存储”。有关类型造型的更多信息,参见本章后面的“类型造型转换”。

乘法运算符

乘法运算符执行乘法(*)、除法(/)和余数(%)操作。

语法

乘法表达式:

造型表达式

乘法表达式 * 造型表达式

乘法表达式 / 造型表达式

乘法表达式 % 造型表达式

余数运算符(%)的操作数必须是整数。乘法(*)和除法(/)运算符可以使用整数或浮点类型操作数,操作数的类型可以不同。

乘法运算符对操作数执行常用的算术转换,其结果的类型是转换后操作数的类型。

注意:由于乘法运算符执行的转换不提供上溢出和下溢出条件,如果一个乘法运算的结果不能以转换后的操作数的类型表示,则可能丢失信息。

c 乘法运算符的描述如下:

运算符	说明
*	该乘法运算符导致它们两个操作数相乘
/	该除法运算符导致第一个操作数被第二个操作数相除 如果两个整数相除且结果不是一个整数,则根据如下规则截除它: ● 根据 ANSI C 标准,除 0 的结果是不确定的,C 编译器在编译时或运行时产生一个错误 ● 如果两个操作数是正数或无符号数,其结果向 0 进行截除 ● 如果有一个操作数为负数,该运算的结果是小于或等于代数商的最大整数或 者大于或等于代数商的最小整数,这被实现所定义(参见下面的 Microsoft 特殊处小节)

续表

运算符	说明
%	余数运算符的结果是第一个操作数除以第二个操作数所得的余数。当除法不精确时,其结果由如下规则确定： ● 如果右边操作数是 0,其结果是不确定的 ● 如果两个操作数都是正数或无符号数,其结果为正数 ● 如果有一个操作数为负数,其结果是不精确的,它们被实现所定义(参见下面 的 Microsoft 特殊处小节)

Microsoft 特殊处→

在有一个操作数为负数的除法中,截除的方向朝向 0。

如果使用余数运算符的除法中有一个操作数为负数,其结果与被除数(表达式中的第一个操作数)的符号相同。

Microsoft 特殊处结束

例子

下面的说明使用以下例子:

```
int i=10,j=3,n;
double x=2.0,y;
```

这个语句使用乘法运算符:

```
y=x*i;
```

在这种情况下,x 乘以 i 得到值 20.0,其结果为 double 类型。

```
n=i/j;
```

在这个例子中,10 被 3 相除,结果朝向 0 进行截除,产生整数 3。

n=i%j;
这个语句赋给 n 为 10 除以 3 的余数即 1。
Microsoft 特殊处→
余数的符号与被除数的符号相同,例如:
50%-6=2
-50%6=-2
以下情况下,50 与 2 具有相同的符号。
Microsoft 特殊处结束

加法运算符

加法运算符执行加法(+)和减法(-)

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

注意: 虽然加法表达式的语法包括乘法表达式,这并不表示一定需要使用乘法。有关乘法表达式、造型表达式和单目表达式,参见本书后面的附录 A“C 语言语法总结”。

操作数可以是整数或浮点值。有些加法运算也可以在指针值上执行,正如每个运算符的讨论中概述的。

加法运算符在整数和浮点操作数上执行常用的算术转换。其结果的类型

是转换后操作数的类型。由于加法运算符执行的转换不提供上溢出或下溢出条件,如果一个加法运算的结果不能由转换后操作数的类型表示,则可能丢失信息。

加法(+)

加法运算符(+)导致两个操作数相加。两个操作数可以是整数或浮点类型,或者一个操作数且另一个操作数是一个整数。

当把一个整数加上一个指针时,整数值(i)通过将它乘以该指针所指值的尺寸来转换。在转换之后,整数值表示 i 存储器位置,而每个位置都有其指针类型指定的长度。不转换的整数值加到该指针值时,其结果是一个新的指针值以表示距离最初地址 i 个位置的地址。该新的指针值指向与最初指针值相同类型的值,因此与数组指标相同(参见本章前面的“一维数组”和“多维数组”)。如果和的指针指向超过了该数组范围,除了在超过最高端的第一个位置外,其结果是不确定的。有关更多信息参见本章后面的“指针算术”。

减法(-)

减法运算符(-)从第一个操作数减去第二个操作数。两个操作数可以是整数或浮点类型,或者一个操作数可以是一个指针且另一个操作数是一个整数。

当两个指针相减时,其差通过除以该指针所指类型的值的尺寸来转换成一个有符号整数值。该整数值的尺寸由包括在标准头文件 STDDEF.H

中的类型 ptrdiff_t 定义。其结果是表示两个地址之间相差的存储器位置的个数,该结果仅对相同数组的两个元素才是有意义的,正如“指针算术”中讨论的。

当从一个指针值减去一个整数时,减法运算符通过该整数值(i)乘以该指针所指值的尺寸来转换,在转换之后,该整数值表示 i 存储位置,而每个位置都有该指针类型所指的长度。当从该指针值减去该转换的整数值时,其结果是最初地址的前 i 个存储器位置。新的指针指向最初指针值所指的类型的值。

使用加法运算符

下例说明了加和减运算符,使用了这些说明:

```
int i=4,j;
float x[10];
float *px;
```

如下语句是等价的:

```
px=&x[4+i];
px=&x[4]+i;
```

i 的值乘以一个 float 的长度并加到&x[4]中,其结果指针值是 x[8]的地址。

```
j=&x[i]-&x[i-2];
```

在这个例子中,从 x 的第五个元素的地址(由 x[i]给出)减去 x 的第三个元素的地址(由 x[i-2]给出)。其差除以一个 float 的长度,结果为整数 2。

指针算术

涉及到一个指针和一个整数的加法运算只存在该指针操作数指向一个数组的元素以及该整数产生同一数组中的偏移量时才有意义。当该整数值转换成一个地址偏移量时,编译器假设只有相同尺寸的存储器位置位于最初地址和该地址加上偏离量的结果之间。

对于数组成员这个假设是有效的。通过定义,一个数组是相同类型的值的序列,它的元素存储在相邻存储器位置。但除数组外,其它任何类型的存储不能保证由相同类型的标识符填充,也即,存储器位置之间可能出现空格,即使相同类型的位置也是如此。因此,除数组元素外任何值的地址加或减的结果都是不确定的。

类似地,当两个指针值相减时,其转换假设只有相同类型的值,没有空格,且在操作数给定的地址空间。

按位位移运算符

位移运算符左移(<<)或右移(>>)第一个操作数的位置个数,由第二个操作数指定。

语法

位移表达式:

加法表达式

位移表达式 << 加法表达式

位移表达式 >> 加法表达式

两个操作数必须是整数值,这些运算符执行常用的算术转换,其结果的类型是转换后左操作数的类型。
对于向左移位移,腾出的右边位设置为 0。对于向右位移,腾出的左边位基于转换后第一个操作数的类型进行填充。如果该类型是 unsigned,它们设置为 0;否则,它们用符号位的拷贝进行填充。对于向左位移运算符,没有上溢出,语句:

```
expr1 << expr2
```

等价于乘以 2^{expr2}。对于右移运算符:

```
expr1 >> expr2
```

如果 expr1 是无符号的或有一个非负值,它等价于除以 2^{expr2}。
如果第二个操作数是负数,或者如果右操作数大于或等于提升的左操作数中的位为单位的宽度,则一个位移运算的结果是不确定的。由于位移运算符执行的转换没有提供上溢出或下溢出条件,如果一个位移运算的结果不能由转换后第一个操作数的类型所表示,则可能丢失信息。

```
unsigned int x,y,z;


x=0x00AA;
y=0x5500;


z=(x<<8)+(y>>8);
```

在这个例子中,x 左移 8 个位置,y 右移 8 个位置。位移的值相加,结果为 0xAA55 并赋给 z。将一个负数右移一位产生的结果是其绝对值舍入的

一半,例如,-253(二进制 1111111100000011)右移一位产生-127(二进制 1111111110000001)。一个正数 253 右移一位产生+126。
右移保留符号位。当一个有符号整数右位移时,保留设置最重要的位。当一个无符号整数右位移时,该最重要位被清除。
如果 0xF000 是无符号数,其结果是 0x7800。如果 0xF0000000 是有符号的,右移一位产生 0xF8000000。右移一个正数 32 次产生 0xF0000000。右移一个负数 32 次产生 0xFFFFFFFF。

关系和相等运算符

双目关系和相等运算符将其第一个操作数与第二个操作数进行比较以测试指定关系的有效性。如果测试的关系为真,则该关系表达式的结果为 1,如果为假则为 0。其结果的类型为 int。

语法:

关系表达式:
    位移表达式
    关系表达式 < 位移表达式
    关系表达式 > 位移表达式
    关系表达式 <= 位移表达式
    关系表达式 >= 位移表达式

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

关系和相等运算符测试如下关系：

运算符	测试的关系
<	第一个操作数小于第二个操作数
>	第一个操作数大于第二个操作数
<=	第一个操作数小于或等于第二个操作数
>=	第一个操作数大于或等于第二个操作数
==	第一个操作数等于第二个操作数
!=	第一个操作数不等于第二个操作数

上表中的开头四个运算符比相等运算符(==和!=)具有更高的优先级。参见表 4.1 中的优先级信息。

操作数可以是整数、浮点或指针类型。操作数的类型可以不同。关系运算符在整数和浮点类型操作数上执行常用的算术转换。另外,你可以用关系和相等运算符构成如下操作数类型的组合：

- 任何关系或相等运算符的两个操作数可以是相同类型的指针。对于等于(==)和不等于(!=) 运算符,比较的结果指出两个指针是否指向相同存储器位置。对于其它关系运算符(<、>、<=和>=),比较的结果指出所指对象的两个存储器位置之间的关系。关系运算符仅比较偏移量。

  只对相同对象的部分定义了指针比较。如果该指针指向一个数组的成员, 该比较等价于对应下标的比较。第一个数组元素的地址“小于”最后一个元素的地址。在结构的情况下,后面说明的结构成员

的指针“大于”前面说明的结构成员的指针。对于同一联合成员的指针是相等的。

- 一个指针值可以与常量值 0 比较是相等(==)或不相等(!=)。具有 0 值的指针称为“空”指针,也就是,它不指向任何有效的存储器位置。
- 相等运算符遵守与关系运算符相同的规则,但允许另外的可能:一个指针可以与一个具有 0 值的常量整数表达式或与 void 的指针进行比较。如果两个指针都为空指针,它们是相等的。相等运算符比较两个段和偏移量。

例子

如下例子说明关系和相等运算符

```
int x=0,y=0;
if (x<y)
```

因为 x **和** y **相等,这个例子中的表达式产生值** 0。

```
char array[10];
char *p;

for (p=array;p<&array[10];p++)
      *p='\0';
```

本例中的段设置 array **的每个元素为一个空字符常量。**

```
enum color {red,white,green} col;
  .
  .
```

```
if (col==red)
```

这些语句用标志 color 说明一个名称为 col 的枚举变量。在任何时候,该变量可以包含 0、1 或 2 的一个整数值,它表示枚举集 color:分别为红、白和绿之一。当执行 if 语句时如果 col 包含 0,则执行依赖该 if 的任何语句。

按位运算符

按位运算符执行按位 AND(&)、按位异或(^)和按位 OR(|)运算。

语法

AND 表达式:

相等表达式

AND 表达式 & 相等表达式

异或表达式:

AND 表达式

异或表达式 ^ AND 表达式

OR 表达式:

异或表达式

OR 表达式 | 异或表达式

按位运算符的操作数必须有整数类型,但它们的类型可能不同。这些运算符执行常用的算术转换。其结果的类型是转换后操作数的类型。

C 按位运算符描述如下:

运算符	说明
&	按位 AND 运算符将第一个操作数的每个位与第二个操作数的对应位进行比较,如果两个位都为 1,对应的结果位设置为 1;否则,对应的结果位设置为 0。
^	按位异或运算符将第一个操作数的每个位与第二个操作数的对应位进行比较,如果一个位为 0 而另一位为 1,对应的结果位设置为 1,否则,对应的结果位设置为 0
|	按位 OR 运算符将第一个操作数的每个位与第二个操作数的对应位进行比较,如果有一个位为 0,对应的结果位设置为 1;否则,对应的结果位设置为 1

例子

这些说明用于如下三个例子:

```
short i=0xAB00;
short i=0xABCD;
short n;

n=i&j;
```

在第一个例子中赋给 n 的结果与 i 的值(十六进制 0xAB00)相同。

```
n=i|j;

n=i^j;
```

在第二个例子中按位 OR 产生结果 0xABCD(十六进制数),而在第三个例子中的按位异或产生 0xCD(十六进制数)。

Microsoft 特殊处→

在有符号整数上按位运算的结果是根据 ANSI C 标准实现定义的。对于 C 编译器,在有符号整数上的按位运算与在无符号整数上的按位运算相同。例如,-16&99 可以以二进制表示为:

```
  11111111  11110000
& 00000000  01100011
  ----------------
  00000000  01100000
```

该按位 AND 的结果是十进制 96。

Microsoft 特殊处结束

逻辑运算符

逻辑运算符执行逻辑 AND(&&)和逻辑 OR(||)运算。

语法

逻辑 AND 表达式

OR 表达式

逻辑 AND 表达式 && OR 表达式

OR 表达式:

逻辑 AND 表达式

逻辑 AND 表达式 || 逻辑 AND 表达式

逻辑运算符不执行常用的算术转换。代替地,它们根据它的 0 的等值对每个操作数进行求值,一个逻辑运算的结果为 0 或 1,其结果的类型为 int。C 逻辑运算符描述如下:

运算符	说明
&&	如果两个操作数都为非 0 值,则逻辑 AND 运算符产生值 1。如果有一个操作数为 0,其结果为 0。如果一个逻辑 AND 运算的第一个操作数等于 0,第二个操作数不求值
||	逻辑 OR 运算符在其操作数上执行一个 OR 运算。如果两个操作数都为 0 值,其结果为 0。如果有一个操作数为非 0 值,其结果为 1。如果一个逻辑 OR 运算符的第一个操作数为非 0 值,则第二个操作数不求值

逻辑 AND 和逻辑 OR 表达式的操作数从左到右求值。如果第一个操作数的值能够足够确定该运算的结果,则第二个结果数不进行求值,这称为“短路求值”。在第一个操作数之后有一个顺序点。有关更多信息参见本章前面的“顺序点”。

例子

如下例子说明逻辑运算符:

```
int w,x,y,z;


if (x<y && y<z)
   printf("x is less than z\n");
```

在这个例子中,如果 x 小于 y 且 y 小于 z 则调用 printf 函数打印一个消息。

如果 x 大于 y,第二个操作数(y<z)不求值且不打印该消息。注意，在这种情况可能出现问题,第二个操作数由于某些其它原因而有副作用。

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

在这个例子中,如果 x 等于 w 和 y 或 z,printf 函数的第二个求值为真时打印 1;否则,求值假时打印 0。只要条件之一求值为真,则求值停止。

条件表达式运算符

C 有一个三元运算符:条件表达式运算符(?:)。

语法

条件表达式:

逻辑 OR 表达式

逻辑 OR 表达式 ? 表达式 : 条件表达式

逻辑 OR 表达式必须具有整数、浮点或指针类型。根据等值于 0 的方式进行求值。一个顺序点紧跟逻辑 OR 表达式。操作数求值过程如下:

- 如果逻辑 OR 表达式不等于 0 表达式求值。该结果由非终结符表达式给出(这意味着表达式仅在逻辑 OR 表达式为真时求值)。
- 如果逻辑 OR 表达式等于 0,条件表达式求值。该结果是其条件表达式的值(这意味着仅在逻辑 OR 表达式为假时对条件表达式求值)。

注意,表达式或条件表达式之一求值,但不是两者。

一个条件表达式结果的类型依赖于表达式或条件表达式操作数的类型如下:

- 如果表达式或条件表达式具有整数或浮点类型(它们的类型可以不

同),该运算符执行常用的算术转换。该结果的类型是转换后操作数的类型。

- 如果表达式和条件表达式具有相同的结构、联合或指针类型,该结果类型是同样的结构、联合或指针类型。
- 如果两个操作数都具有 void 类型,该结果是 void 类型。
- 如果操作数之一为任何类型的一个对象的指针,另一个操作数是一个 void 的指针,该对象的指针转换成一个 void 的指针,该结果是一个 void 的指针。
- 如果表达式或条件表达式之一是一个指针,另一个操作数是一个具有 0 值的常量表达式。该结果的类型是 void 的指针。

在指针的类型比较中,该指针所指的类型中的任何类型修饰符(const 或 volatile)都是不重要的。但结果类型以两个条件的组成成分继承修饰符。

例子

如下例子说明条件运算符的使用

```
j=(i<0) ? (-i) : (i);
```

这个例子把 i 的绝对值赋给 j。如果 i 小于 0,则把-i 赋给 j;如果 i 大于或等于 0,i 赋给 j。

```
void f1(void);
void f2(void);
int x;
int y;
      .
```

.

.

(x == y) ? (f1()) : (f2());

在这个例子中,说明了两个函数 f1 和 f2 以及两个变量 x 和 y。在程序后面,如果这两个变量具有相同的值,则调用 f1 函数;否则调用 f2 函数。

赋值运算符

一个赋值运算把右边操作数的值赋给左边操作数命名的存储位置。因此,一个赋值运算的左边操作数必须是可修改的 l 值。在赋值之后,一个赋值表达式具有左边操作数的值,但不是一个 l 值。

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:**如下之一**

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

C 中的赋值运算符可以在单个运算中进行转换和赋值。C 提供了如下赋值运算符:

运算符	执行的运算
=	简单赋值
*=	乘法赋值
/=	除法赋值
%=	余数赋值
+=	加法赋值
-=	减法赋值
<<=	左位移赋值
>>=	右位移赋值
&=	按位 AND 赋值
^=	按位异或赋值
|+	按位或赋值

在赋值中,右边值的类型转换成左边值的类型，并在赋值发生后把该值存储在左边操作数中。左边操作数不能是数组、函数或常量。在本章后面的“类型转换”中详细讨论了指定的转换路径,它依赖于两种类型。

简单赋值

简单赋值运算符把它的右操作数赋给左操作数。右操作数的值转换成赋值表达式的类型,并替换存储在左操作数指定的对象中的值。应用赋值的转换规则(参见本章后面的“赋值转换”)。

```
double x;
```

```
int y;


x=y;
```

在这个例子中,y 的值转换成 double 类型并赋给 x。

复合赋值

复合赋值运算符组合简单赋值运算符和其它双目运算符。复合赋值运算符执行另外运算符指定的运算,然后把结果赋给左操作数。例如,这样的一个复合赋值表达式:

```
expression1+=expression2
```

可以理解为

```
expression1=expression1+expression2
```

但复合赋值表达式不等于扩充的版本,因为在加法运算中和在赋值运算中，复合赋值表达式仅求值 expression1 一次,而扩充的版本求值 expression1 两次。

一个复合赋值运算符的操作数必须是整数或浮点类型。每个复合赋值运算符执行对应的双目运算符执行的转换，并相应地限制它的操作数的类型。加法赋值(+=)和减法赋值(-=)运算符也有一个指针类型的左操作数,在这种情况下右边操作数必须是整数类型。一个复合操作数的结果具有左操作数的值和类型。

```
#define MASK 0xff00
n&=MASK;
```

在这个例子中,在 n 和 MASK 上执行一个按位 AND 运算,并把结果赋给 n。显式常量 MASK 用一个#define 预处理器命令定义。

顺序求值运算符

顺序求值运算符也称为“逗号运算符”,从左向右顺序计算它的两个操作数。

语法

表达式:

赋值表达式

表达式,赋值表达式

顺序求值运算符的左操作数作为一个 void 表达式求值。该运算的结果具有和右操作数相同的值和类型,每个操作数可以是任何类型。顺序求值运算符在它的操作数之间不执行任何类型转换,它不产生 l 值。在第一个操作数之后产生一个顺序点,这意味着从左操作数求值产生的副作用在右操作数求值开始之前均已完成。有关更多信息,参见本章前面的“顺序点”。

顺序求值运算符通常用于对上下文中两个或多个表达式求值,而这里仅允许一个表达式。

逗号在某些上下文中用作分隔符,但你必须小心不要将它作为分隔符使用和作为运算符使用相混淆,这两种使用是完全不同的。

例子

这个例子说明顺序求值运算符:

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

在这个例子中,for 语句的第三个表达式的每个操作数都分开求值。左操作数 i+=i 首先求值,然后是右操作数 j--求值。

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

在 func_one 函数调用中,三个用逗号隔开的参量分别传送给:x,y+2 和 z。在 func_two 函数调用中,圆括号强制编译器解释第一个逗号为顺序求值运算符。这个函数调用传送两个参量给 func_two,第一个参量是顺序求值运算(x--,y+2)的结果,它具有 y+2 表达式的值和类型,第二个参量是 z。

类型转换

类型转换依赖于指定的运算符和操作数或运算符的类型。类型转换在如下情况下执行:

- 当一个类型的值赋给不同类型的变量时,是或者一个运算符在执行该运算之前转换它的一个操作数或多个操作数的类型时。
- 当一个类型的值显式造型转换为一个不同的类型时。
- 当一个值作为一个参量传送给一个函数或者当从一个函数返回一个类型时。
- 一个字符、一个短整数、一个整数位域、所有有符号的或无符号的或者枚举类型的一个对象都可以用在表达式能够使用整数的地方。如果一个 int 可以表示所有最初类型的值,那么该值转换成 int,否

则,它转换成 unsigned int。这个过程称为“整数提升”。整数提升保持值,也就是,在提升之后的值保证与提升前的值相同。有关更多信息参见本章前面的“常用的算术转换”。

赋值转换

在赋值运算中,赋值的类型转换成接受赋值的变量的类型。C 允许整数和浮点类型之间通过赋值进行转换,即使在转换中选择信息也如此。使用的转换方法依赖于赋值中涉及的类型,正如“常用的算术转换”和如下小节描述的:

- 从有符号整数类型转换
- 从无符号整数类型转换
- 从浮点类型转换
- 从指针类型转换和转换到指针类型
- 从其它类型转换

类型修饰符不影响转换的允许性,虽然一个 const l 值不能用作赋值的左边。

从有符号整数类型转换

当一个有符号整数用相等或更大的尺寸转换成一个无符号整数,且该有符号整数不是负数时,该值不改变。通过符号扩充该有符号整数来进行转换,一个有符号整数通过高端的位转换成一个更短的有符号整数,其结果解释为一个无符号整数,如下例子:

```
int i=-3;
unsigned short u;


u=i;
printf("%hu\n",u);     /*打印 65533*/
```

当一个有符号整数转换成一个浮点类型时,没有信息丢失,除当一个 long int 或 unsigned long int 值转换成一个 float 值可能丢失某些精度外。

表 4.2 总结了有符号整数类型的转换。这个表假设缺省 char 类型为有符号的。如果你使用一个编译选项改变缺省 char 类型为无符号的,这个转换在表 4.3 中给出,即应用 unsigned char 类型代替表 4.2 中的转换。

表 4.2 有符号整数类型的转换

从	到	方法
char[1]	short	符号扩充
char	long	符号扩充
char	unsigned char	保留模式,丢失高端作为符号的位
char	unsigned short	符号扩充为 short,转换 short 为 unsigned short
char	unsigned long	符号扩充为 long,转换 long 为 unsigned long
char	float	符号扩充为 long,转换 long 为 float
char	double	符号扩充为 long,转换 long 为 double
char	long double	符号扩充为 long,转换 long 为 double

续表

从	到	方法
short	Char	保留低端字节
short	long	符号扩充
short	unsigned char	保留低端字节
short	unsigned short	保留模式,丢失高端作为符号位的位
short	unsigned long	符号扩充为 long,转换 long 为 unsigned long
short	float	符号扩充为 long,转换 long 为 float
short	double	符号扩充为 long,转换 long 为 double
short	long double	符号扩充为 long,转换 long 为 double
long	char	保留低端字节
long	short	保留低端字
long	unsigned char	保留低端字节
long	unsigned short	保留低端字
long	unsigned long	保留模式,丢失高端作为符号位的位
long	float	作为 float 表示,如果 long 不能精确表示,丢失精度
long	double	作为 double 表示,如果 long 不能作为一个 double 精确表示,丢失精度
long	long double	作为 double 表示,如果 long 不能作为一个 double 精确表示,丢失精度

1. 所有 char 项假设 char 类型缺省是有符号的。

Microsoft 特殊处→
对于一个 Microsoft 32 位 C 编译器,一个整数等价于 long。一个 int 值转换处理与 long 的相同。
Microsoft 特殊处结束

从无符号整数类型转换

一个无符号整数通过截除高端的位转换成一个更短的无符号或有符号整数,或者通过 0 扩充(参见表 4.3)转换成一个更长的无符号或有符号整数。当整数类型的值用更小的尺寸降级为一个有符号整数,或者一个无符号整数转换成它对应的有符号整数时,如果它能在新类型中表示则不改变它的值。但如果设置符号位,则它表示的值发生改变,如下例子:

```
int j;
unsigned short k=65533;


j=k;
printf("%hd\n",j);   /*打印-3*/
```

如果它不能表示,其结果是实现定义的。有关 C 编译器处理整数降级的更多信息,参见本章后面“类型造型转换”。从整数类型转换或从造型整数类型的转换的结果相同。
无符号值只能以保留它们的值的方式进行转换,在 C 中是不可直接表示的。唯一的异常是从 unsigned long 转换到 float,它丢失了最重要的低端位。否则保留值,无论是有符号的还是无符号的。当一个整数类型的值

转换成浮点时,该值超过了表示范围,其结果是不可预料的(有关整数和浮点类型的范围的信息,参见第 3 章“说明和类型”中的“基本类型的存储”)。

表 4.3 总结了从无符号整数类型的转换。

表 4.3 从无符号整数类型的转换

从	到	方法
unsigned char	char	保留位模式,高端位变成符号位
unsigned char	short	0 扩充
unsigned char	long	0 扩充
unsigned char	unsigned short	0 扩充
unsigned char	unsigned long	0 扩充
unsigned char	float	转换为 long,转换 long 为 float
unsigned char	double	转换为 long,转换 long 为 double
unsigned char	long double	转换为 long,转换 long 为 double
unsigned short	char	保留低端字节
unsigned short	short	保留位模式,高端位变成符号位
unsigned short	long	0 扩充
unsigned short	unsigned char	保留低端字节
unsigned short	unsigned long	0 扩充
unsigned short	float	转换为 long,转换 long 为 float
unsigned short	double	转换为 long,转换 long 为 double
unsigned short	long double	转换为 long,转换 long 为 double
unsigned long	char	保留低端字节

续表

unsigned long	Short	保留低端字
unsigned long	long	保留位模式,高端位变成符号位
unsigned long	unsigned char	保留低端字节
unsigned long	unsigned short	保留低端字
unsigned long	float	转换为 long,转换 long 为 float
unsigned long	double	直接转换为 double
unsigned long	long double	转换为 long,转换 long 为 double

Microsoft 特殊处→

对于 Microsoft 32 位 C 编译器,unsigned int 类型等价于 unsigned long 类型。一个 unsigned int 值的转换的方式与一个 unsigned long 的转换方式相同。如果转换的值大于最大正的有符号 long 值,那么从 unsigned long 值转换到 float 是不准确的。

Microsoft 特殊处结束

从浮点类型转换

一个 float 值可以转换成一个 double 或 long double,或者一个 double 转换成一个 long double,不改变其中的值。如果可能,一个 double 值转换成一个准确表示的 float 值。如果该值不能准确表示,则可能丢失精度。如果结果超出了范围,其行为是不可确定的。有关浮点类型的范围,参见第 1 章“C 的基本元素”中的“浮点常量的限制”。

通过把一个浮点值先转换成一个 long,然后把该 long 值转换成指定的整数值,正如表 4.4 所说明的,把该浮点值转换成一个整数值,在转换成 long

中,丢弃该浮点值的小数部分。如果仍太大不适合 long 表示,则该转换的结果是不确定的。

Microsoft 特殊处→

当把一个 double 或 long double 浮点数转换成一个较小的浮点数时,如果发生下溢出则朝向 0 截除该浮点数的值。一个上溢出导致一个运行错误。注意,C 编译器把 long double 映射成 double。

Microsoft 特殊处结束

表 4.4 总结了从浮点类型的转换。

表 4.4 从浮点类型的转换

从	到	方法
float	char	转换为 long;转换 long 为 char
float	short	转换为 long;转换 long 为 short
float	long	在小数点处截除。如果结果太大不能表示成 long,则其结果是不确定的
float	unsigned short	转换为 long,转换 long 为 unsigned short
float	unsigned long	转换为 long,转换 long 为 unsigned long
float	double	改变内部表示
float	long chouble	改变内部表示
double	char	转换为 float,转换 float 为 char

续表

从	到	方法
double	short	转换为 float,转换 float 为 short
double	long	在小数点处截除。如果该结果太大不能用 long 表示,则结果是不确定的
double	unsigned short	转换为 long,转换 long 为 unsigned short
double	unsigned long	转换为 long,转换 long 为 unsigned long
double	float	表示为 float。如果 double 值不能以 float 类型准确地表示,会出现精度丢失。如果该值太大不能作为 float 表示,其结果是不确定的
long double	char	转换为 float,转换 float 为 char
long doubl	short	转换为 float,转换 float 为 short
long double	long	在小数点处截除。如果结果太大不能用 long 表示,则结果是不确定的
long double	unsigned short	转换为 long;转换 long 为 unsigned short
long double	unsigned long	转换为 long;转换 long 为 unsigned long

续表

long double	float	表示为 float。如果 double 值不能以 float 类型准确地表示,会出现精度丢失。如果该值太大不能作为 float 表示,其结果是不确定的
long double	double	该 long double 值作为 double 处理

如果被转换的值大于最大正数 long 值,则从 float、double 或 long double 值到 unsigned long 的转换是不准确的。

转换到指针类型和从指针类型转换

一个值类型的指针可以转换成一个不同类型的指针。虽然这个结果可能因为对齐要求和不同类型的存储尺寸而导致其结果是不确定的。一个对象的指针可以转换成一个其类型需要较少或完全相同的对象的指针,反过来没有改变。

一个 void 的指针可以转换成任何类型的一个指针,或者从任何类型的指针可以转换成一个 void 的指针。如果其结果被回过来转换成最初的类型,则恢复最初的指针。

如果一个指针转换成另一个指针,后者有相同的类型,但有不同的或另外的修饰符,该新的指针与老的指针相同,除了强制加上新的修饰符之外。

一个指针值也可以转换成一个整数值。根据如下规则,其转换路径依赖于该指针的尺寸和整数类型的尺寸:

- 如果指针的尺寸大于或等于该整数的尺寸,该指针的行为在转换中

像一个无符号值一样,除了它不能转换成一个浮点值外。

- 如果该指针小于整数类型,首先把该指针转换成一个与整数类型相同尺寸的指针,然后再转换成整数类型。

相反地,一个整数类型也可以根据如下规则转换成一个指针类型:

- 如果该整数类型与指针类型具有相同的尺寸,简单地转换该整数值作为一个指针(一个无符号整数)处理。
- 如果该整数的尺寸不同于该指针类型的尺寸,首先把整数类型转换成该指针的尺寸,这里使用表 4.2 和表 4.3 中给出的转换路线;然后把它作为一个指针值处理。

一个具有 0 型的整数常量表达式或一个强制造型为类型 void *的表达式可以通过一个类型造型、赋值或与任何类型的指针进行比较来转换。这产生一个等于另一个相同类型空指针的空指针,但这个空指针不等于一个函数或一个对象的任何指针。不为常量 0 的其它整数也可以转换成指针类型,但其结果不是可移植的。

从其它类型转换

由于一个 enum 值是一个定义的 int,从一个 enum 值转换或转换到 enum 值与 int 类型的方式相同。对于 C 编译器,一个整数与一个 long 是相同的。

Microsoft 特殊处→

结构或联合类型之间不允许转换。

任何值可以转换成类型 void,但这个转换的结果仅用在一个表达式值被

丢弃的上下文中,例如在一个表达式语句中。

由定义，void 类型没有值,因此它不能转换成任何其它类型,其它类型也不能通过赋值转换成 void,然而你可以显式造型转换一个值为类型 void,正如下一节“类型造型转换”中讨论的。

Microsoft 特殊处结束

类型造型转换

你可以使用类型造型显式转换类型。

语法

造型表达式:

单目表达式

(类型名称)造型表达式

类型名称:

指示符修饰符表 抽象说明符$_{opt}$

类型名称是一个类型,造型表达式是要转换成这个类型的值。一个带有类型造型的表达式不能是一个 l 值。造型表达式是被转换赋给类型名称指定类型的变量。赋值的转换规则(在前面“赋值转换”中描述)也适用于类型造型。

表 4.5 说明了可以造型转换为给定类型的类型。

表 4.5 合法的类型造型

目的类型	可能的源
整数类型	任何整数类型、浮点类型或一个针对对象的指针
浮点类型	任何算术类型
一个针对对象的指针或(void*)	任何整数类型、(void*)、一个针对对象的指针或一个函数指针
函数指针	任何整数类型、一个针对对象的指针或一个函数指针
一个结构、联合或数组	无
void 类型	任何类型

任何标识符都可以造型转换为 void,但如果在一个类型造型表达中的指定的类型不是 void,那么该标识符造型转换成不是一个 void 表达式的类型。任何表达式都可以造型为 void,但一个类型为 void 的类型不能造型转换为任何其它类型。例如,一个具有 void 类型的函数不能有另一个类型的返回值。

注意,一个 void 表达式有一个 void 的类型指针,而不是类型 void。如果一个对象类型为 void 类型,其结果表达式不能赋给任何项。类似地,一个类型造型对象不是一个可接受的 l 值,因此不能赋给一个类型造型对象。

Microsoft 特殊处→

只要该标识符的尺寸不发生改变,一个类型造型可以是一个 l 值表达式。有关 l 值表示式的信息,参见本章开头的“L 值和 R 值表达式”。

Microsoft 特殊处结束

你可以用一个造型把一个表达式转换为类型 void,但结果表达式只能用于不需要一个值的地方。一个转换为 void *的对象指针,在回过来转换为最初的类型时将返回它的最初值。

函数调用转换

在一个函数调用中参量上执行的转换类型依赖于用于说明该被调用函数的参量的函数原型(向前说明)。

如果有一个函数原型上包括说明的参量类型,编译器执行类型检测(参见第 6 章“函数”)。

如果没有函数原型,则在函数调用中参量上执行常用的算术转换。这些转换在调用中的每个参量上独立执行。这意味着一个 float 值可以转换为一个 double 值;一个 char 或 short 值可以转换为一个 int;一个 unsigned char 或 unsigned sort 可以转换为一个 unsigned int。

第 5 章　语　　句

一个 C 程序的语句控制一个程序执行的流程。在 C 中,和其它程序设计语言中一样,有几种类型的语句可以用来执行循环、选择其它执行的语句以及转换控制。后面简要概述语句语法,本章以字母排序次序描述这些 C 语句:

break 语句　　if 语句
复合语句　　空语句
continue 语句　　return 语句
do-while 语句　　switch 语句
表达式语句　　try-except 语句
for 语句　　try-finally 语句
goto 和标号语句　　while 语句

语句概述

C 语句由语句符号、表达式和其它语句组成。一个构成另一个语句的组成成分的语句被称为“体”。本章讨论了如下语法给出的每种语句类型。

语法

语句:

标号语句
复合语句
表达式语句
选择语句
迭代语句
跳转语句
try-except 语句　　/* Microsoft 特殊处 */
try_finally 语句　　/* Microsoft 特殊处 */

语句体频繁地是作为“复合语句”,一个复合语句由其它包括关键字的语句组成。复合语句通过花括号({})限定。所有其它 C 语句以一个分号(;)结尾。一个分号是一个语句结束符。

表达式语句包含一个 C 表达式,该 C 表达式包含第 4 章“表达式和赋值”中介绍的算术和逻辑运算符。null 语句是一个空语句。

任何 C 语句可以以一个标识的标号开头,该标号由一个名称加一个冒号组成。由于只有 goto 语句识别语句标号,语句标号与 goto 一起讨论,有关更多信息参见本章后面的“goto 和标号语句”。

break 语句

break 语句用于终止包含它的 do、for、switch 或 while 语句的执行。控制传递给该终止语句的后续语句。

语法

跳转语句:

```
    break;
```

break 语句频繁地使用在一个 switch 语句中以终止一个特殊情况的处理。缺少包含它的迭代语句或 switch 语句时产生一个错误。

在嵌套的语句中,break 语句仅终止直接包含它的 do、for、switch 或 while 语句。

你可以使用一个 return 或 goto 语句把控制转向该嵌套结构的外面的地方。

如下例子说明了 break 语句:

```
for (i=0,i<LENGTH;i++)   /*当执行 break 语句时控制返回到这里*/
{
  for(j=0,j<WIDTH;j++)
  {
      if (lins[i][j]==' \0 ' )
      {
          lengths[i]=j;
          break;
      }
  }
}
```

这个例子处理一个存储在 lines 中的可变长度字符串的数组。break 语句

导致在每个字符串中找到结尾的空格字符('\0')后退出里层的 for 循环并把该位置存储在 lengths[i]中。
在 break 导致从里层循环退出时变量 j 不增大。然后控制返回到外层 for 循环。i 增大且重复这个过程直到 i 大于或等于 LENGTH。

复合语句

一个复合语句(也称为一个“块”)一般作为另一语句例如 if 语句的语句体出现。第 3 章“说明和类型”描述了出现在一个复合语句开头的说明的格式和含义。

语法

复合语句:
    {说明表 $_{opt}$ 语句表 $_{opt}$}

说明表:
    说明
    说明表 说明

语句表:
    语句
    语句表 语句

如果有说明,它们必须放在任何语句之前。每个在一个复合语句开头说明的标识符的范围从其说明点开始延伸到该块结束。它在该块中是可见的,除非在一个更里层的块中存在相同名称的标识符的说明。

在一个复合语句中的标识符都假定是 auto,除非用 register、static 或 extern 进行显式说明,除了函数之外,因为函数只能是 extern。你可以在函数说明中省略 extern 指示符,该函数仍然是 extern 的。

如果一个变量或函数在一个复合语句中用存储类 extern 说明,则不分配存储,不能进行初始化。该说明指的是在其它地方定义的一个外部变量或函数。

在一个块中用 auto 或 register 关键字说明的变量被重新说明,如果必要,在每次进入该复合语句时进行初始化。这些变量在该复合语句退出时就没有定义了。如果一个变量在一个块内说明并具有 static 属性,在程序执行开始时初始化该变量并在整个程序中保存它的值。有关 static 的信息参见第 3 章“说明和类型”中的“存储类”。

这个例子说明一个复合语句:

```
if (i>0)
{
    line[i]=x;
    x++;
    i--;
}
```

在这个例子中,如果 i 大于 0,那么该复合语句中的所有语句依次执行。

continue 语句

continue 语句把控制传递给包含该语句的 do、for 或 while 语句的下一次迭代,绕过 do、for 或 while 语句体的任何余下语句。 continue 的一般使用是从一个深的嵌套循环中返回到一个循环的开头。

语法

跳转语句:

```
continue;
```

确定一个 do、for 或 while 语句的下一次迭代如下:

- **在一个 do 或一个 while 语句中,通过重新 do 或 while 语句中表达式的求值来开始下一次迭代。**
- **在一个 for 语句中的 continue 语句导致 for 语句的第一个表达式被求值,然后编译器重新条件表达式的求值,根据其结果,终止或迭代该语句体。有关 for 语句和它的非终结符的更多信息参见本章后面的“ for 语句 ”。**

如下是 continue 语句的例子:

```
while(i-->0)
{
   x=f(i);
   if(x==1)
      continue;
   y+=x*x;
```

```
}
```

在这个例子中,当 i 大于 0 时执行该语句体。首先把 f(i)赋给 x,然后如果 x 等于 1,则执行 continue 语句,忽略语句体中其余的语句,执行回到该循环的顶部,求值该循环的测试。

do-while 语句

do-while 语句让你重复一个语句或复合语句直到一个指定的条件变成假。

语法

迭代语句:

do 语句 while(表达式);

在一个 do-while 语句中的表达式在执行该循环体之后求值,因此该循环体至少执行一次。

该表达式必须是算术或指针类型。执行过程如下:

1. 执行语句体。
2. 接着对表达式求值,如果该表达式为假,该 do-while 语句终止,控制转向程序中的下一个语句,如果表达式为真(非 0),重复该过程,从第 1 步开始。

当执行语句体中的一个 break、goto 或 return 语句时也终止该 do-while 语句。

如下是 do-while 语句的例子:

```
do
{
```

```
    y=f(x);
    x--;
} while(x>0);
```

在这个 do-while 语句中,执行 y=f(x);和 x--;两个语句,不论 x 的初值如何。然后求值 x>0,如果 x 大于 0,则又执行该语句体和重新求值 x>0,只要 x 保持大于 0,则重复执行该语句体。当 x 变成 0 或负数时终止该 do-while 语句的执行,该循环体至少执行一次。

表达式语句

当执行一个表达式语句时,根据第 4 章“表达式和赋值”中给出的规则对该表达式求值。

语法

表达式语句:

表达式 $_{opt}$

一个表达式求值的所有副作用都在执行下一个语句之前完成。一个空表达式语句称为一个空语句。有关更多信息参见本章后面的“空语句”。

这些例子说明了表达式语句:

```
x=(y+3);              /* 把 y+3 的值赋给 x*/
x++;                   /*x 增 1*/
x=y=0                  /*把 x 和 y 都初始化为 0 */
proc(arg1,arg2);   /*函数调用返回 void */
```

```
y=z=(f(x)+3);        /*一个函数调用表达式 */
```

在最后一个函数调用表达式语句中,该表达式的值包括该函数返回的任何值再增大 3,然后赋给变量 y 和 z。

for 语句

for 语句让你重复一个语句或复合语句指定的次数。一个 for 语句体执行 0 次或多次直到一个任选的条件变成假。你可以在 for 语句中使用任选表达式并在 for 结构的执行中初始化和改变值。

语法

迭代语句:

for (初始表达式$_{opt}$;条件表达式$_{opt}$;循环表达式$_{opt}$) 语句

一个 for 语句的执行过程如下:

1. 如果有,则求值初始表达式,这指出该循环的初始化。初始表达式在类型上没有限制。
2. 如果有,则求值条件表达式,这个表达式必须是算术或指针类型。它在每次迭代之前求值,有三种可能的结果:

- 如果条件表达式为真(非 0),则执行该语句。然后如果有循环表达式,则对其求值。在每次迭代之后都对循环表达式求值。其类型没有限制。副作用将依次执行。然后又开始求值条件表达式的过程。
- 如果省略了条件表达式,条件表达式被认为是真,执行过程和前面段中描述的过程完全一样。一个没有条件表达式的 for 语句仅在执行

语句体中的 break 或 return 语句时终止,或者在执行一个(跳出 for 语句体)goto 语句时终止。

- 如果条件表达式为假(0),for 语句的执行终止,并把控制传递给程序中的下一个语句。

一个 for 语句也在执行语句体中的 break、goto 或 return 语句时终止。一个 for 循环中的 continue 语句导致求值循环表达式。当执行一个 for 循环中的 break 语句时,循环表达式不求值或执行。如下语句:

```
for(;;);
```

是一个通常的产生无限循环的方式,它只能用 break、goto 或 return 语句退出。

如下例子说明 for 语句:

```
for(i=space=tab=0;i<MAX,i++)

    if (line[i]== ' ')
        space++;
    if (line[i]== '\t')
    {
        tab++;
        line[i]= ' ';
    }
 }
```

这个例子对名称为 line 的字符数组中的空格(' ')和制表键('\t')进行计

数,并用一个空格替换每个制表键,首先 i、space 和 tab 都初始化为 0。然后 i 和常量 MAX 进行比较;如果 i 小于 MAX,则执行该语句体。根据 line[i] 的值,执行或不执行 if 语句的语句体。然后 i 增 1 并又与 MAX 比较,只要 i 小于 MAX 则重复地执行该语句体。

goto 和标号语句

goto 语句转换控制到一个标号。给定的标号必须在相同的函数中,而且只能出现在相同函数中的一个语句的前面。

语法

语句:

    标号语句
    跳转语句

跳转语句:

    goto **标识符**

标号语句:

    标识符:**语句**

一个语句标号仅对 goto 语句有意义,在任何其它上下文中,一个标号语句的执行忽略该标号。

一个跳转语句必须保留在相同的函数中,只能出现在相同函数中一个语句之前。跟随一个 goto 的标识符名称有它自己的名称空间,因此该名称与其它标识符无关。标号不能重新说明。有关更多信息参见第 2 章“程

序结构”中的“名称空间”。
一种好的程序设计风格是在可能出现 goto 的地方优先使用 break、continue 和 return 语句。一个 break 语句只能从一个循环层退出,而一个 goto 可以从深的嵌套循环中退出。
如下例子说明了 goto 语句的作用:

```
void main()
{
    int i,j;

    for (i=0,i<10;i++)
    {
        printf("Outer loop executing. i=%d\n",i);
        for(j=0;j<3;j++)
    {
            printf("Inner loop executing. j=%d\n",j),
            if (i==5)
                goto stop;
      }
   }
   /* 这个消息不打印 */
   printf ("Loop exited. i=%d\n",i);
   stop:printf("Jumped to stop,i=%d\n",i);
```

```
}
```

在这个例子中,在 i 等于 5 时一个 goto 语句把控制转向括号为 stop 的语句。

if 语句

if 语句控制条件分支。如果其中表达式的值为非 0 则执行一个 if 语句体,if 语句的语法有两种格式。

语法

选择语句:

if (表达式) 语句

if (表达式) 语句 else 语句

在 if 语句的两种格式中,其中的表达式可以具有除结构外的任何类型,它被求值,包括所有副作用。在语法的第一种格式中,如果表达式为真(非 0),则执行其中的语句;如果该表达式为假(0),则忽略该语句。在该语法的第二种格式中,它使用了 else,如果表达式的值为假,则执行第二个语句。使用这两种格式,控制都从 if 语句转向程序中的下一个语句,除了其中语句之一包含一个 break、continue 或 goto 之外。

如下是 if 语句的例子:

```
if(i>0)
    y=x/i;
else
{
```

```
    x=i;
    y=f(x);
}
```

在这个例子中,如果 i 大于 0 则执行 y=x/i;语句;如果 i 小于或等于 0,则把 i 赋给 x,且把 f(x)赋给 y。注意形成 if 子句的语句以一个分号结尾。

当嵌套 if 语句和 else 子句时,使用花括号来组合语句。形成一个更清楚的复合语句。如果不出现在括号,编译器通过把 else 与最近的缺少 else 的 if 进行关联来解决模糊性。

```
if (i>0)                /*没有花括号 */
    if(j>i)
        x=j;
    else
       x=i;
```

在这个例子中,else 子句与里层的 if 语句关联。如果 i 小于或等于 0,则没有值赋给 x。

```
if (i>0)
{               /*有花括号*/
    if (j>i)
        x=j;
}
else
    x=j;
```

在这个例子中里层的 if 语句用花括号括起来,使 else 子句与外层 if 语句匹配。如果 i 小于或等于 0,把 i 赋给 x。

空语句

一个“空语句”是一个仅包含分号的语句。它可以出现在一个语句能够出现的任何地方。
当执行一个空语句时不发生任何事件。编写一个空语句的正确方式是:
语法

```
;
```

例如 do、for、if 和 while 语句需要一个可执行的语句出现在其语句体中,在不需要一个实际的语句体的情况下要求空语句满足语法要求。
正如其它任何 C 语言一样。你可以在一个空语句之前包括一个标号。
为了给一个不是语句的项加上标号时,例如一个复合语句的闭花括号,你可以给一个空语句加上标号,把它直接插入到获得相同效果的项之前。
这个例子说明了空语句的作用:

```
for (i=0;i<10;ling[i++]=0)
    ;
```

在这个例子中,for 语句的循环表达式 line[i++]=0 初始化 line 的开头 10 个元素为 0,该语句作为空语句,不必须有后续语句。

return 语句

return 语句终止一个函数的执行并将控制返回调用函数。在调用函数中该调用点之后恢复执行。一个 return 语句也返回一个值给调用函数,有关更多信息参见第 6 章“函数”中的“返回类型”。

语法

跳转语法

return 表达式$_{opt}$

如果出现表达式,则把该表达式的值返回给调用函数;如果省略该表达式,该函数的返回值是不确定的。如果出现表达式,该表达式转换成函数返回的类型。如果该函数用返回类型 void 说明,一个包含表达式的 return 语句产生一个警告并不对该表达式求值。

如果一个函数定义中没有 return 语句出现,在被调用函数的最后一个语句执行之后控制自动返回给调用函数,在这种情况下,被调用函数的返回值是不确定的。如果不需要返回值,说明该函数具有 void 返回类型;否则,缺省的返回类型是 int。

很多程序员使用圆括号括起该 return 语句的表达式参量。但 C 不需要圆括号。这个例子说明 return 语句的作用:

```
void draw(int I,long L);
long sq(int s);
int main()
{
```

```
    long y;
    int x;

    y=sq(x);
    draw(x,y);
    return();
}

long sq(int s)
{
    return (s * s);
}

void draw( int I, long L)
{
     /*这里的语句定义画图功能 */
     return;
}
```

在这个例子中,main 函数调用两个函数:sq 和 draw。sq 函数返回 x*x 的值到 main,这里的返回值赋给 y。draw 函数说明为一个 void 函数,并不返回值。试图赋给 draw 返回值会导致发生一个诊断消息。

switch 语句

switch 和 case 语句帮助控制复杂条件和分支操作。switch 语句把控制转向该语句体中的一个语句。

语法

选择语句:

switch(**表达式**)**语句**

标号语句:

case **常量表达式**:**语句**

default:**语句**

控制转向与 switch(**表达式**)**的值匹配的** case **常量表达式的语句。**switch **语句可以包含任何个数的** case **实例,但在同一个** switch **语句中不能有两个** switch **常量具有相同的值。语句体的执行开始于选择的语句并继续执行直到该语句体结束或者直到一个** break **语句转移控制到该语句体之外。**

switch **语句的使用通常象这样的**:

```
switch(表达式)
{
    说明
    .
    .
    .
```

case 常量表达式
如果表达式等于这个常量表达式的值而执行的语句
.
.
.
break
default:
如果表达式"不等于任何情况常量表达式时而执行的语句"
}
你可以使用 break 语句终止 switch 语句中一个特殊情况的处理并跳转到该 switch 语句的末尾,没有 break,该执行继续到下一个 case,执行语句直到一个 break 或到达该语句末尾,在有些情况下,这个继续是希望的。
如果没有 case 常量表达式等于 switch(表达式)的值,则执行 default 语句。
如果省略了 default 语句且没有 case 匹配,则不执行 switch 体中的语句。
最多只能有一个 default 语句,default 语句不一定放在最后,可以放在 switch 语句体的任何地方。实际上,经常放在 switch 语句开头更有效。
一个 case 或 default 标号只能出现在一个 switch 语句内部。
switch 表达式和 case 常量表达式的类型必须是整数。每个 case 常量表达式的值在语句体中必须是唯一的。
switch 语句体的 case 和 default 标号仅在最初测试用于确定该语句体的执行开始的位置。
switch 语句可以嵌套,在执行转向任何 switch 语句之前初始化任何静态

变量。

注意:说明可以出现在该复合语句的开头形成 switch 体,但不执行包括在说明中的初始化。switch 语句把控制直接转向该语句体中一个可执行的语句,绕过包含初始化的行。

如下例子说明了 switch 语句的作用:

```
switch (c)
{
  case ' A ' :
     capa++;
  case ' a ' :
     lettera++;
  default:
     total++;
}
```

本例中 switch 体的所有三个语句在 c 等于 ' A ' 时都执行,因为在每个 case 之前没有出现一个 break 语句。执行控制转向第一个语句(capa++),并继续依次通过语句体中余下的语句。如果 c 等于 ' A ',则 lettera 和 total 都增 1。如果 c 不等于 ' A ' 或 ' a ',则只有 total 增 1。

```
switch(i)
{
    case -1:
       n++;
```

```
            break;
        case 0:
            z++;
            break;
        case 1:
            p++;
            break;
}
```

在这个例子中,switch 体中每个语句后跟一个 break 语句。break 语句在执行一个语句后强制从该语句体退出。如果 i 等于-1,只有 n 增 1,跟随 n++ 语句后面的 break 语句导致执行控制转出该语句体,而避开其余的语句。类似地,如果 i 等于 0,只有 z 增 1;如果 i 等于 1,只有 p 增 1。最后的一个 break 语句不必要求,因为控制在复合语句的末尾就转出体外,但为了一致性包括了该语句。

单个语句可以加上多个 case 标号,正如如下例子:

```
case 'a':
case 'b':
case 'c':
case 'd':
case 'e':
case 'f':hexcvt(c);
```

在这个例子中,如果常量表达式等于'a'到'f'的任何字母,则调用

hexcvt 函数。
Microsoft 特殊处→
Microsoft C 不限制一个 switch 语句中 case 子句的个数,其个数仅受限于可用的存储器。
ANSI C 在一个 switch 语句至多允许 257 个 case 标号。
Microsoft C 缺省能使 Microsoft 扩充,使用/Za 编译器选项禁止这些扩充。
Microsoft 特殊处结束

try-except 语句

Microsoft 特殊处→
try-except 语句是 C 语言的 Microsoft 扩充,它能使一个应用在出现正常终止执行事件时获取程序控制,这样的事件称为异常,处理异常的机制被称为结构的异常处理。
异常可以基于硬件或基于软件的,甚至在应用不能从硬件或软件异常完全恢复时,结构的异常处理使它可能显示错误信息并捕捉该应用的内部状态帮助诊断新问题。这对于处理不容易重现的间歇性问题特别有用。

语法

try-except 语句:

```
__try 复合语句
__except(表达式)复合语句
```

在 __try 语句之后的复合语句是保护段。__except 子句之后的复合语句是

异常处理器。该处理器指出在保护段执行期间发生一个异常时所执行的一组动作。执行过程如下:

1. 执行保护段。

2. 如果在保护段执行期间没有异常发生,继续执行__except 子句之后的语句。

3. 如果在保护段执行期间出现一个异常或者在保护段调用的例程中出现一个异常,则__except 表达式被求值,其返回值确定如何处理异常。这里有三个值:

EXCEPTION_CONTINUE_SEARCH:异常不能识别,继续搜索栈查找一个处理器,首先搜索包含 try-except 语句的处理器,然后搜索下一个最高优先级的处理器。

EXCEPTION_CONTINUE_EXECUTION:异常被识别但不予考虑。继续在出现异常的点执行。

EXCEPTION_EXECUTE_HANDLER:异常被识别。通过执行__except 复合语句把控制转向异常处理器,然后在出现异常的点继续执行。因为__except 表达式作为一个 C 异常被求值,它限制为单个值、条件表达式运算符或逗号运算符。如果需要更广泛的处理,该表达式可以调用一个例程返回上述所列的三个值之一。

注意:结构的异常处理与 C 和 C++源文件一起工作。虽然它不能特别为 C++设计,你能够通过使用 C++异常处理确保你的代码具有更好的移植性。C++异常处理机制也具有更好的灵活性,它能够处理任何类型的异常。

对于 C++程序,C++异常处理可以用于代替结构的异常处理。有关更多信息,参见本书后面“Microsoft Visual C++6.0 语言参考手册”部分第 5 章“语句”中的“异常处理”。

一个应用中的每个例程都有自己的异常处理器。__except 表达式在__try 语句体的范围内执行。这意味着它可以访问这里说明的任何局部变量。

__leave 关键字在一个 try-except 语句块中是有效的。__leave 的作用是跳到该 try-except 块的末尾。在该异常处理器结束之后恢复执行。虽然一个 goto 语句可以用于完成同样的功能,但一个 goto 语句导致栈的消除内务操作。

__leave 语句更有效,因为它不涉及到栈的消除内务操作。

使用 longjmp 运行函数退出一个 try-except 语句认为是异常终止。跳进一个__try 语句是非法的,但跳出一个__try 语句是合法的。如果一个进程在执行一个 try-except 语句中被删除则不调用该异常处理器。

例子

如下是一个异常处理器和一个终止处理器的例子。有关终止处理器的更多信息参见下一节“try-finally 语句”。

.

.

•

```
puts("hello");
__try {
   puts("in try");
```

```
   __try {
   puts("in try");
      RAISE_AN_EXCEPTION();
} __finally {
      puts("in finally");
}
} __except(puts("in filter"), EXCEPTION_EXECUTE_HANDLER) {
   puts("in except");
}
puts("world");
```

如下是上例的输出结果,在右边加有注释:

```
hello
in try          /*进入 try    */
in try          /*进入嵌套的 try*/
in filter       /*执行过滤器;因此接受返回 1 */
in finally      /*消除嵌套的 finally*/
in except       /*转移控制到选择的处理器      */
world             /*退出处理器      */
```

Microsoft **特殊处结束**

try-finally 语句

Microsoft 特殊处→

try-finally 语句是 C 语句的 Microsoft 扩充,能使应用保证在执行一个代码块中断时执行清理(cleanup)代码。清理由这样的功能组成,如取消存储器分配关闭文件和释放文件句柄。try-finally 语句对于这样的例程特别有用,该例程有几个位置检测导致过早地从该例程返回的错误。

语法

try-finally 语句:

__try 复合语句
__finally 复合语句

__try 子句后的复合语句是保护段。__finally 子句后的复合语句是终止处理器。该处理器指出在退出保护段时执行的一组动作,无论该保护段是由异常(非正常终止)或由标准下行(正常终止)退出的。

通过简单的顺序执行(下行)控制到达一个__try 语句。当控制进入该__try 语句时,它关联的处理器变成活动的。执行过程如下:

1. 执行保护段。
2. 调用终止处理器。
3. 当终止处理器完成后,继续执行__finally 语句之后的语句,无论该保护段如何终止(例如,经由 goto 语句转出保护段外面或至由一个 return 语句)在控制流移出保护段之前执行该终止处理器。

__leave 关键字在一个 try-finally 语句块中是有效的。__leave 的作用是跳

转到该 try-finally 块的末尾,直接执行该终止处理器,虽然一个 goto 语句可以用于完成相同的功能,但一个 goto 语句导致栈的消除内务操作。__leave 语句更有效,因为它不涉及到栈的消除内务操作。

使用一个 return 语句或 longjmp 运行函数退出一个 try-finally 语句被认为是非正常终止。跳进一个__try 语句是非法的,但从中跳转出来是合法的。从启动点到目的之间所有活动的__finally 语句必须运行,这称之为“局部展开”。

如果在执行 try-finally 语句时一个进程被删去,则不调用该终止处理器。

注意:结构的异常处理与 C 和 C++源文件一起工作。虽然它不能特别为 C++设计,你能够通过使用 C++异常处理确保你的代码具有更好的移植性。C++异常处理机制也具有更好的灵活性,它能够处理任何类型的异常。

对于 C++程序,C++异常处理可以用于代替结构的异常处理。有关更多信息,参见本书“Microsoft Visual C++ 6.0 语言参考手册”部分第 5 章“语句”中的“异常处理”。

参见 try-except 语句的例子看 try-finally 语句是如何工作的。

Microsoft 特殊处结束

while 语句

while 语句让你重复一个语句直到指定的表达式变成假。

语法

迭代语句:

while(表达式) 语句

该表达式必须具有算术或指针类型。执行过程如下:

1. 对表达式进行求值。

2. 如果表达式起初为假,不执行 while 语句体。控制从 while 语句转向程序中的下一个语句。

如果表达式为真(非 0),执行语句体并从第 1 步开始重复这一过程。

当执行该语句体中的一个 break、goto 或 return 时也终止 while 语句。

使用 continue 语句终止一次迭代而不退出 while 循环,continue 语句把控制转向 while 语句的下一次迭代。

如下是一个 while 语句的例子:

```
while(i>=0)
{
    string1[i] = string2[i];
    i--;
}
```

这个例子把字符从 string2 拷贝到 string1。如果 i 大于或等于 0,把 string2[i] 赋给 string[i]并将 i 减 1。当 i 到达 0 或小于 0 时,终止该 while 语句的执行。

第 6 章　函　　数

函数是 C 中的基本模块单元。一个函数设计用于执行特定的功能,它的名称通常反映其功能。一个函数包含说明和语句。本章描述如何说明、定义和调用函数的。其它讨论的主题有:

- 函数概述
- 函数定义
- 函数原型
- 函数调用

函数概述

函数必须有一个定义和一个说明,如果说明出现在函数被调用之前,一个定义可以作为一个说明,函数定义包括函数体即在该函数被调用时执行的代码。

一个函数说明建立名称、返回值和在程序其它地方定义的函数的属性。一个函数说明必须位于该函数调用的前面。这是为什么在你的代码中调用一个运行函数之前包括包含该运行函数说明的头文件的原因。如果该说明具有类型和参数个数的信息,该说明是原型。

有关更多信息参见本章后面的“函数原型”。
编译器使用该原型将后面调用中的参量类型与具有该函数参数的函数进行比较,必要时把参量类型转换成参数的类型。
一个函数调用将执行控制从调用函数传递到被调用函数。如果有参量,参量通过被调用函数的值传递。在被调用函数中执行一个 return 语句返回控制和一个可能的值到调用函数。

已废除的函数说明和定义格式

旧式函数说明和定义使用在说明参数上有点不同于 ANSI C 标准推荐的语法。首先,旧式说明没有参数表;第二,在函数定义中列出参数,但在参数表中不说明它们的定义。类型说明位于组成该函数体的复合语句的前面。旧式语法已废除了,不在新代码中使用。但仍支持使用旧式语法的代码。如下例子给出了说明和定义的已废除的格式:

```
double old_style();          /*已废除的函数说明*/

double alt_style(a,real)   /*已废除的函数说明*/
      double *real;
      int a;
{
      return (*real+a);
}
```

返回一个整数或与一个 int 具有相同尺寸的指针的函数不需要有一个说

明,虽然建议有该说明。

为了遵守 ANSI C 标准,旧式函数说明使用一种省略方式,现在使用/Za 选项时会产生一个错误;当使用/Ze 选项时产生一个第 4 层的警告。例如:

```
void functl(a,...)        /*在/Ze 下产生一个警告*/
int a;                    /*或在/Za 下产生一个错误*/
{
}
```

你可以作为原型重写这个说明:

```
void funct1(int a,...)
{
}
```

如果你在后面用一个省略或一个不同于其提升类型的类型参数来说明或定义相同函数,则旧式函数说明也产生警告。

下一节“函数定义”说明了函数定义的语法,包括旧式语法。在旧式语法中参数表的非终结符是标识符表。

函数定义

一个函数指出函数的名称、它接受的参数类型和个数以及它的返回值。一个函数定义还包括说明它的局部变量以及确定该函数所做事情的函数体。

语法

转换单元:

外部说明

转换单元 外部说明

外部说明: /* 仅允许在外部(文件)范围内 */

函数定义

说明

函数定义: /* 这里的说明符是函数说明符 */

说明指示符$_{opt}$ 属性序列$_{opt}$ 说明符 说明表$_{opt}$ 复合语句

/* 属性序列是 Microsoft 特殊处 */

原型参数是:

说明指示符:

存储类指示符 说明指示符$_{opt}$ 类型指示符$_{opt}$ 类型指示符 说明指示符$_{opt}$

类型修饰符 说明指示符$_{opt}$

说明表:

说明

说明表 说明

说明符:

指针$_{opt}$ 直接说明符

直接说明符: /* 一个函数说明符 */

直接说明符(参数类型表) /* 新式说明符 */

直接说明符(标识符表$_{opt}$) /* 已废除的说明符 */

在一个定义中参数表使用以下语法:

参数类型: /* 参数表 */
    参数表
    参数表,...
参数表:
    参数说明
    参数表,参数说明
参数说明:
    参数指示符 说明符
    说明指示符 抽象说明符$_{opt}$
在一个旧式函数定义中的参数表使用以下语法:
标识符表: /* 已废除的函数定义和说明 */
    标识符
    标识符表,标识符
函数体的语法:
复合语句: /* 函数体 */
    {说明表$_{opt}$ 语句表$_{opt}$}
可以用于修饰一个函数说明的存储类指示符只有 extern 和 static。extern 指示符指示该函数可以从其它文件引用,也就是,该函数名称被输出到链接器。static 指示符指示该函数不能从其它文件引用,也就是,该函数名称不输出到链接器。如果一个函数定义中不出现存储类,则假设是 extern。在任何情况下,该函数从定义点到文件结尾就是可见的。
任选说明指示符和强制性说明符一起指出该函数的返回类型和名称。说

明符是命名该函数的标识符和紧跟函数名的圆括号的组合。传送的属性序列非终结符是定义在“函数属性”中的 Microsoft 特殊特征。

直接说明符(在说明符语法中)指出定义的函数的名称和它的参数的标识符。如果直接说明符包括一个参数表,该表指出所有参数的类型。这样的一个说明符也作为后面调用该函数的一个函数原型。

在函数定义中的说明表中的一个说明不能包含 register 之外的一个存储类指示符。只有在为一个 int 类型的值指定 register 存储类时,其说明指示符语法中的类型指示符可以省略。

复合语句是包含局部变量说明、引用外部的说明项和语句的函数体。

“函数属性”、“存储类”、“返回值”、“参数”和“函数体”等节详细地描述了函数定义的组成份。

函数属性

Microsoft 特殊处→

任选的属性序列非终结符允许你在每一个函数基础上选择一个调用约定。你也可以指定函数为 __fastcall 或 __inline。

Microsoft 特殊处结束

指定调用约定

Microsoft 特殊处→

有关调用约定的信息,参见联机“Microsoft Visual C++ 6.0 程序员指南”中的“调用约定主题”。

Microsoft 特殊处结束

联编函数

Microsoft 特殊处→

__inline 关键字告诉编译器在一个函数调用的每个实例中用该函数的定义替换该代码。但替换仅出现在编译器的决定中。例如,如果一个函数的地址被占用或它太大,则编译器不联编该函数。

对一个认为是联编的候选者的函数,它必须使用新式函数定义。

使用这个格式指出一个联编函数:

__inline 类型 opt 函数定义;

使用联编函数产生更快的代码,有时产生比相同函数调用产生更小的代码,其原因如下:

- 它节省需要执行函数调用的时间。
- 小的联编函数,可能三行或更少,因为编译器不产生处理参量和返回值的代码而生成比相同的函数调用更少的代码。
- 生成的联编函数是属于不能正常使用的优化代码,因为编译器不执行内部过程的优化。

使用__inline 的函数不要与联编汇编器混淆,有关更多信息参见“联编汇编器”。

Microsoft 特殊处结束

联编汇编器

Microsoft 特殊处→

联编汇编器让你在你的 C 源程序中直接嵌入汇编语言指令而不要额外的汇编和链接步骤。

联编汇编器构建在编译器中,你不需要单独的汇编器例如 Microsoft 宏汇编器(MASM)。

因为该联编汇编器不需要单独的汇编和链接步骤,它比一个单独的汇编器更方便。联编汇编代码可以使用任何 C 变量或函数名称,因此它容易与你的程序的 C 代码集成在一起。又因为汇编代码可以与 C 语句混合,因此它可以完成单独用 C 很麻烦的或不可能的功能。

__asm 关键字调用联编汇编器,它可以出现在任何合法的 C 语句中,但不能出现自身的语句中。之后必须跟一个汇编指令、一个用花括号括起的指令组或者至少一个空花括号对。

术语“__asm 块”在这里指的是任何指令或指令组,不论是否出现在花括号中。

下面的代码是一个花括号括起的简单__asm 块(这个代码是一个空制函数):

```
__asm
{
   push ebp
   mov ebp,esp
   sub esp, __LOCAL_SIZE
```

```
}
```

另一种方式,你可以在每个汇编指令前面放置一个__asm:

```
__asm push ebp
__asm mov ebp,esp
__asm sub esp, __LOCAL_SIZE
```

由于__asm 关键字是一个语句分隔符,你也可以把汇编指令放在同一行中:

```
__asm push ebp __asm mov ebp,esp   __asm sub esp, __LOCAL_SIZE
```

Microsoft 特殊处结束

DLL 输入和输出函数

Microsoft 特殊处→

dllimport 和 dllexport 存储类修饰符是 C 语言的 Microsoft 特殊处扩充。这些修饰显式定义了 DLL 的客户界面(可执行的文件或另外的 DLL)。说明为 dllexport 的函数消除了一个模块定义(.DLL)文件的需要。你可以为数据和对象使用 dllimport 和 dllexport 修饰符。

dllimport 和 dllexport 存储类修饰符必须与扩充的属性语法关键字__declspec 一起使用,下面是这样的例子:

```
#define DllImport __declspec(dllimport)
#define DllExport __declspec(dllexport)

DllExport void func();
```

```
Dllexport int i = 10;
DllExport int j;
DllExport int n;
```

有关扩充的存储类修饰符的语法的指定信息,参见第 3 章“说明和类型”中的“扩充的存储类型属性”。

Microsoft 特殊处结束

定义和说明

Microsoft 特殊处→

DLL 界面指的是系统中某个程序中输出的所有已知项(函数和数据);也就是所有说明为 dllimport 或 dllexport 的所有项。包括在 DLL 界面中的所有说明必须指定为 dllimport 或 dllexport 属性。但该定义只能指定 dllexport 属性。例如,如下函数定义生成一个编译器错误:

```
#define DLLImport __declspec(dllimport)
#define DLLExport __declspec(dllexport)

DLLImport int func()/*错误:在定义中禁止 dllimport*/
{
    return 1;
}
```

下面代码也产生一个错误:

```
#define DllImport __declspec(dllimport)
```

```
#define DllExport __declspec(dllexport)

DllImport int i=10;     /*错误:这是一个定义*/
```
但如下是正确的语法:
```
#define DllImport __declspec(dllimport)
#define DllExport __declspec(dllexport)

DllExport int i=10;     /*正确:这是一个输出定义*/
```
dllexport 的使用隐含一个定义,而 dllimport 隐含一个说明。你必须对 dllexport 使用 extern 关键字强制为一个说明;否则,隐含是一个定义。
```
#define DllImport __declspec(dllimport)
#define DllExport __declspec(dllexport)

extern DllImport int k;     /*这是正确的并隐含一个说明*/
Dllimport int j;
```
Microsoft 特殊处结束

用 dllexport 和 dllimport 定义联编函数

Microsoft 特殊处→

你可以用 dllexport 属性定义一个联编函数,在这种情况下,该函数总是被实例化和被输出,无论程序中的任何模块引用该函数。该函数假定是被另一程序输入。

你也可以用 dllimport 属性说明一个函数为联编函数,在这种情况下,该函数可以被伸展(从属于/Ob(联编)编译器选项规格)但不能被实例化。在特殊情况中,如果一个联编输入的函数的地址被占用,该函数的地址保留在返回的 DLL 中。这个行为和占用一个非联编输入的函数的地址相同。
在联编函数中的静态局部数据和字符串在 DLL 和象在单个程序中似的客户(也就是,一个没有 DLL 界面的可执行文件)之间维护相同的标识符。
在进行提供输入的联编函数的练习时要小心,例如,如果你修改 DLL,不要假设该客户使用该 DLL 的改变的版本。为了保证你加载适当的 DLL 版本,重新建立该 DLL 的客户。
Microsoft 特殊处结束

dllimport/dllexport 的规则和限制

Microsoft 特殊处→

- 如果你说明一个函数没有 dllimport 或 dllexport 属性,该函数不认为是 DLL 界面的部分。因此,该函数的定义必须出现在该模块中或相同程序的另一个模块中。为了使该函数成为 DLL 界面部分,必须在其它模块中以 dllexport 说明该函数的定义;否则,在建立客户时产生一个链接器错误。
- 如果你的程序的单个模块包含相同函数的 dllimport 和 dllexport 说明,那么 dllexport 属性的优先级比 dllimport 属性的优先级高。但编译器产生一个警告。例如:

```
#define DLLimport __declspec(dllimport)
#define DLLexport __declspec(dllexport)

    DllImport   void   func1(void);
    DllExport   void   func1(void);/*警告:dllexport 更优先*/
```

- 你不能用一个以 dllimport 属性说明的数据对象的地址初始化一个静态数据指针。例如,如下代码产生一个错误:

```
#define DllImport __declspec(dllimport)
#define DllExport __declspec(dllexport)

    DllImport int i ;
    .
    .
    .
    int *pi=&i;  /* 错误 */
    void func2()
    {
       static int *pi=&i; /* 错误 */
     }
```

- 用一个 dllimport 说明的函数的地址初始化一个静态函数指针,设置该指针为该 DLL 输入形实替换程序(一个转换控制到该函数的代码块)而不是该函数的地址。如下赋值不产生错误消息:

```
#define DllImport __declspec(dllimport)
#define DllExport __declspec(dllexport)

   DllImport void func1(void)
   .
   .
   .
   static void (*pf)(void)=&func1;/* 没有错误 */

   void func2()
   {
      static void (*pf)(void)=&func1;/* 没有错误 */
   }
```

- 因为在一个对象的说明中包括 dllexport 属性的程序必须提供这个对象的定义,你可以用一个 dllexport 函数的地址初始化一个全局或局部静态函数指针。类似地,你可以用一个 dllexport 数据对象的地址初始化一个全局或局部静态数据指针。例如:

```
#define DllImport __declspec(dllimport)
#define DllExport __declspec(dllexport)

   DllImport void func1(void);
   DllImport int i;
```

```
DllExport void func1(void);
DllExport int i;
.
.
.
int *pi=&i;     /* 正确 */
static void(*pf)(void) = &func1;   /* 正确 */

void func2()
{
    static int *pi=i; /* 正确 */
    static void (*pf)(void) = &func1; /* 正确 */
}
```

Microsoft 特殊处结束

naked 函数

Microsoft 特殊处→

naked 存储类属性是 C 语言的一个 Microsoft 特殊处扩充。对于用 naked 存储类属性说明的函数,编译器生成没有序言和结尾部分的代码。你可以利用这个特征，在使用联编写虚拟设备驱动程序中特别有用。

因为 naked 属性仅与一个函数的定义有关,它不是一个类型修饰符,naked

函数使用第 3 章“说明和类型”中的“扩充的存储类属性”中描述的扩充的属性语法。
下面的例子用 naked 属性定义了一个函数:

```
__declspec(naked)int func(formal_parameters)
{
    /* 函数体 */
}
```

或者,另一种方式是:

```
#define naked __declspec (naked)

naked int func(formal_parameters)
{
    /* 函数体 */
}
```

naked 属性只影响编译器生成该函数的序言和结尾部分序列的代码,它不影响调用这样函数生成的代码。因此,naked 属性不认为是该函数的类型部分,函数指针不能有 naked 属性。而且 naked 属性不能应用于一个数据定义。例如,如下代码产生错误:

```
__declspec(naked) int i;     /* 错误-naked 属性不允许用在数据说明中 */
```

naked 属性只能与函数的定义相关,而不能在函数的原型中指出,下面的说明产生一个编译器错误:

```
__declspec(naked) int func();   /* 错误-naked 属性不允许在函数说明中 */
```

Microsoft 特殊处结束

使用 naked 函数的规则和限制

Microsoft 特殊处→

- 在一个 naked 函数中不允许使用 return 语句。但可以把该返回值移到 RET 指令之前的 EAX 寄存器。
- 在一个 naked 函数中不允许使用结构的异常处理指令,因为该指令必须在栈框架中执行取消内务操作。
- 在一个 naked 函数中不允许使用 setjmp 运行函数,因为它必须在栈框架中执行取消内务操作。但 longjmp 运行函数是允许的。
- 在一个 naked 函数中不允许使用_alloc 函数。
- 为了确保在序言之前不出现局部变量的初始化代码,在函数范围不允许初始化的局部变量。
- 不推荐使用框架指针优化(/Oy 编译器选项),但它为了 naked 函数而自动取消。

Microsoft 特殊处结束

编写序言/结尾部分代码的考虑

Microsoft 特殊处→

在你编写自已的序言和结尾部分代码序列之前,重要的是了解栈框架的布局,它对于了解如何使用__LOCAL_SIZE 预定义的常量也是重要的。

栈框架布局

这个例子说明了在一个 32 位函数中可以出现的标准序言代码:

```
push    ebp                 ;存储 ebp
mov     ebp,esp             ;设置栈框架指针
sub     esp,localbytes      ;为局部变量分配空间
push    <registers>         ;存储 registers
```

localbytes 变量表示在栈上局部变量需要的字节数,registers 变量是表示在栈上保存寄存器表的位置占用者。在下推寄存器后,你可以把任何其它适当的数据放在栈中。如下是对应的结尾部分代码:

```
pop   <registers>     ;恢复 registers
mov   esp ebp         ;恢复栈指针
pop   ebp             ;恢复 ebp
ret                   ;从函数返回
```

栈总是向下推(从高到低存储器地址)。基指针(ebp)指向 ebp 的推入的值,局部变量区域起始于 ebp-2。为了访问该局部变量,通过 ebp 减适当的值计算从 ebp 的偏移量。

__LOCAL_SIZE 常量

编译器为联编汇编器块中使用函数序言代码而提供了一个常量 __LOCAL_SIZE。这个常量用在定制序言代码中用于在栈上为局部变量分配空间。

编译器确定 __LOCAL_SIZE 的值,该值是所有用户定义的局部变量和编译

器生成的临时变量的总的字节个数。__LOCAL__SIZE 仅用作一个立即操作数;它不能用在一个表达式中。
你不能改变或重新定义该常量的值。例如:

```
mov   eax, __LOCAL_SIZE          ; 立即操作数-正确
mov   eax, [ebp-__LOCAL_SIZE]    ; 错误
```

如下一个 naked 函数的例子包含定制序言和结尾部分序列:

```
__declspec( naked ) func()
{
    int i;
    int j;
    __asm    /* 序言 */
     {
     push ebp
     mov ebp,esp
     sub esp,__LOCAL_SIZE
     }
/* 函数体 */
__asm   /* 结尾部分 */
    {
    mov esp,ebp
    pop ebp
    ret
```

```
    }
}
```

Microsoft 特殊处结束

存储类

在函数定义中的存储类给出该函数为 extern 或 static 存储类。

语法:

函数定义:

说明指示符 $_{opt}$ 属性序列 $_{opt}$ 说明符 说明表 $_{opt}$ 复合语句

/* 属性序列是 Microsoft 特殊处 */

说明指示符:

存储类指示符 说明指示符 opt

类型指示符 说明指示符 opt

类型修饰符 说明指示符 opt

存储类指示符: /* 对于函数定义 */

extern

static

如果一个不包括存储类指示符的函数定义,该存储类缺省为 extern。你可以显式说明一个函数为 extern,但它可以不需要。

如果一个函数说明包含存储类指示符 extern,该标识符和任何文件范围的标识符的可见说明具有相同的连接。如果没有文件范围的可见说明,该标识符具有外部的连接。如果一个标识符有文件范围且没有存储类指示

符,那么该标识符具有外部的连接。外部的连接意味着该标识符的每个实例指示相同的对象或函数。有关连接和文件范围的更多信息参见第 2 章“程序结构”中的“生存期、范围、可见性和连接”。

用一非 extern 的存储类指示符说明一个块范围的函数会产生错误。

具有 static 存储类的一个函数仅在定义它的源文件中是可见的。所有其它函数,不论它们是显式还是隐含地给出 extern 存储类,它们在程序中的所有源文件中都是可见的。如果期待是 static 存储类,必须在该函数说明(如果有)的第一次出现或者该函数的定义上说明它为该存储类。

Microsoft 特殊处→

当允许 Microsoft 扩充时,最初说明没有一个存储类(或有 extern 存储类)的函数,如果该函数的定义在同一源文件中,且该定义显式指出 static 存储类,则该函数给出为 static 存储类。

当用/Ze 编译器选项编译时,在一个块中用 extern 关键字说明的函数具有全局可见性。

当用/Za 编译时不是这样的。如果考虑源代码的移植性,这个特征是不可靠的。

Microsoft 特殊处结束

返回类型

一个函数的返回值建立该函数返回值的尺寸和类型,对应于下面语法中的类型指示符:

语法

函数定义:

说明指示符 opt 属性序列 opt 说明符 说明表 opt 复合语句

/* 属性序列是 Microsoft 特殊处 */

说明指示符:

存储类指示符 说明指示符 opt

类型指示符 说明指示符 opt

类型修饰符 说明指示符 opt

类型指示符:

void

char

short

int

long

float

double

signed

unsigned

结构或联合指示符

枚举指示符

typedef 名称

类型指示符可以指出任何基本的、结构或联合类型。如果你不包括类型指示符,返回的类型假设是 int。

在函数定义中给出的返回值必须与在程序中其它地方该函数说明的返回值相匹配。当执行包含一个表达式的 return 语句时一个函数返回一个值。对该表达式求值时,如果需要,要转换该返回值的类型,并返回到该函数被调用的地方。如果一个函数被说明为返回类型是 void,则一个包含表达式的返回语句会产生一个警告,并不对其中的表达式进行求值。
如下例子说明了函数的返回值:

```
typedef struct
{
    char name[20];
    int id;
    long class;
} STUDENT;

/* 返回的类型是 STUDENT */
STUDENT sortstu(STUDENT a,STUDENT b)
{
    return((a.id < b.id) ? a : b);
}
```

这个例子用一个 typedef 说明定义了 STUDENT 类型,并定义函数 sortstu 具有 STUDENT 返回类型。该函数选择并返回它的两个结构参量之一。在该函数的后续调用中,编译器进行检测以确保参量类型为 STUDENT。
注意:通过传送该结构的指针而不是整个结构可以提高效率。

```
char *smallstr(char s1[],char s2[])
{
    int i;

    i=0;
    while (s1[i]!='\0'  && s2[i]!= '\0')
        i++;
    if (s1[i]== '\0')
        return(s1);
    else
        return(s2);
}
```

这个例子定义了一个这样的函数,该函数返回一个字符数组的指针。该函数有两个字符数组的(字符串)参量并返回两个字符串中较短的指针,一个数组的指针指向第一个数组元素并具有它的类型;因此,该函数的返回类型是 char 类型的指针。

对于返回 int 类型的函数,你不需要在调用它们之前说明它的 int 返回类型,虽然建议使用原型以便对参量和返回值进行正确的类型检测。

参　数

参量是通过一个函数调用传给该函数的值的名称,参数是该函数期待接受的值。在一个函数原型中,紧跟函数名称后面的圆括号包含该函数的

参数及其类型的完整表。
参数说明指出存储在参数中的值的类型、尺寸和标识符。

语法

函数定义:

    说明指示符 opt 属性序列 opt 说明符 说明表 opt 复合语句
    /* 属性序列是 Microsoft 特殊处 */

说明符:

    指针 opt 直接说明符

直接说明符: /* 一个函数说明符 */

    直接说明符(参数类型表)/*新式说明符*/

参数类型表: /* 一个参数表 */

    参数表
    参数表,...

参数表:

    参数说明
    参数表,参数说明

参数说明:

    说明指示符,说明符
    说明指示符,抽象说明符 opt

参数类型表是一个由逗号分隔的参数说明序列。在参数表中每个参数的格式如下:

[register] 类型指示符 [说明符]

用 auto 属性说明函数参数会产生错误,该参数的标识符用在函数体中指的是送给该函数的值。你不能在原型中命名参数,除非这些名称超出该说明结尾的范围。因此参数名称可以与函数定义中相同方式或不同方式赋值。这些标识符在函数体的最外层的块中不能重新定义,但它们可以在里层、嵌套的块中好像参数表是一个括起来的块重新定义。

在参数类型表中的每个标识符之前必须有适当的类型指示符,正如下面的例子指示的:

```
void new (double x, double y, double z)
{
    /* 这里是函数体 */
}
```

如果在参数表中至少出现一个参数,该表可以以一个逗号跟三个句点(,...)结尾。这个构造称为“省略用法”,指出该函数可变的参量个数(有关更多信息,参见本章后面的“用可变个数参量调用”)。但调用该函数的参量个数必须至少有逗号之前的参数个数多。

如果没有参量传送给该函数,该参数表由关键字 void 代替,这个 void 的使用不同于作为一个类型指示符的使用。

参数的次序和类型包括任何省略用法,在所有函数说明(如果有)和函数定义中必须都是相同的。在常用算术转换之后参量的类型必须与对应参数的类型进行兼容赋值(对于算术转换的信息参见第 4 章“表达式和赋值”中的“常用的算术转换”),省略号后的参量不进行检测。一个参数可以有任何基本的、结构、联合、指针或数组类型。

如果必要,编译器在每个参数和每个参量上执行独立的常用算术转换。在转换之后,没有短于 int 的参数,没有参数具有 float 类型,除非该参数类型在原型中显式指定为 float。例如,这意味着说明一个参数为 char 与说明它为 int 具有相同的作用。

函数体

一个“函数体”是包含指定该函数执行的语句的复合语句。

语法

函数定义:

说明指示符 $_{opt}$ 属性序列 $_{opt}$ 说明符 说明表 $_{opt}$ 复合语句

/* 属性序列是 Microsoft 特殊处 */

复合语句: /* 函数体 */

{说明表 $_{opt}$ 语句表 $_{opt}$}

在一个函数体中说明的变量、“局部变量”除非指定其它的存储类型,否则具有 auto 存储类。当一个函数被调用时,为局部变量建立存储并执行局部初始化。执行控制传给该复合语句中的第一个语句并继续执行直到遇到一个 return 语句或遇到该函数体结束。然后控制返回到函数被调用的地方。

如果该函数有一个返回值,必须执行包含一个表达式的 return 语句。如果没有 return 语句执行或者 return 语句不包含表达式,该函数的返回值是不确定的。

函数原型

一个函数说明位于函数定义之前,指出一个函数的名称、返回类型、存储类型和其它属性。为了作为一个原型使用,该函数说明也必须建立该函数参量的类型和标识符。

语法

说明:

说明指示符 属性序列$_{opt}$ 初始说明符表$_{opt}$

/* 属性序列$_{opt}$是 Microsoft 特殊处 */

说明指示符:

存储类指示符 说明指示符$_{opt}$

类型指示符 说明指示符$_{opt}$

类型修饰符 说明指示符$_{opt}$

初始说明符表:

初始说明符

初始说明符表,初始说明符

初始说明符:

说明符

说明符=初始化器

说明符:

指针$_{opt}$ 直接说明符

直接说明符: /* 一个函数说明符 */

直接说明符(参数类型表)　/* 新式说明符 */

直接说明符(标识符表$_{opt}$)　/* 已废除的说明符 */

该原型和函数定义具有相同的格式,除了一个分号直接跟在圆括号之后终止原型外,因此原型没有语句体。在每种情况下,返回值必须与函数定义中指定的返回值一致。

函数原型具有如下重要用途:

- 它们建立函数返回非 int 类型的返回类型,虽然返回 int 值的函数不需要原型,但建议有原型。
- 没有完成的原型,要进行标准转换,但不要试图用参数个数检测参量的类型和个数。
- 原型在函数定义之前用于初始化该函数的指针。
- 参数表用于对函数定义中的参数与函数调用中对应的参量进行检测。

每个参数转换的类型确定了栈中放置的函数调用参量的解释。参量和参数之间的一个类型不匹配导致栈中的参量被误解释。例如,在一个 16 位计算机中,如果一个 16 位指针作为一个参量传送,那么说明作为一个 long 参数,栈中的开头 32 位解释为一个 long 参数。

这个错误不仅对该 long 参数产生问题,而且对于随后的任何参数也产生问题。你可以通过所有函数说明完整的函数原型来检测这种错误。

一个原型建立一个函数的属性,以便在其定义(或出现在其它源文件中)之前的函数调用检测参数类型和返回类型的不匹配。例如,如果你在一个原型中指出 static 存储表指示符,你必须在函数定义中指定 static 存储类。

完整参数说明(int a)可以在同一说明中与抽象说明符(int)混合使用。例如,如下说明是合法的:

```
int add (int a,int);
```

该原型可以包括两种形式,一种是类型,一个是标识符,每个表达式作为一个参量传送。

但这样的标识符的范围只有到该说明结束为止。原型也可以反映参量个数是可变的事实,或者没有传送的参量。没有这样的一个表,不会反映出不匹配,因此编译器不能生成有关它们的诊断消息,有关类型检测的更多信息,参见本章后面的“参量”。

现在 Microsoft C 编译器的原型范围在使用/Za 编译器选项编译时是与ANSI一致的。这意味着如果你在一个原型中说明一个 struct 或 union 标志,该标志进入这个范围而不是一个全局范围。例如,当用/Za 与 ANSI 一致性的方式编译时,你不调用这个函数不会得到一个类型不匹配的错误:

```
void func1(struct s*);
```

为了修正你的代码,在该函数原型之前以全局范围定义或说明为 struct 或 union:

```
struct s;
void func1(struct s*);
```

在/Ze 下,该标志仍进入全局范围中。

函数调用

一个函数调用是一个传送控制和参量(如果有)给一个函数的表达式,它有格式:

表达式(表达式表 opt)

这里的表达式是一个函数名称或一个函数地址的求值,表达式表是一个表达式(用逗号分隔)的表。这些表达式的最后值是传送给该函数的参量。如果该函数不返回一个值,那么你说明它为一个返回 void 的函数。

如果一个说明在函数调用之前存在,但没有给出有关参数的信息,任何未说明的参量简单地经历常用的算术转换。

注意:函数参量表中的表达式可以以任何次序求值。因此其值被其它参量的副作用所改变的参量具有不确定的值。函数调用运算符所定义的顺序点仅保证参量中的所有副作用在控制传送给被调用函数之前被求值(注意,参量推入栈的次序是另一个事情)。有关更多信息参见第 4 章“表达式和赋值”中的“顺序点”。

在任何函数调用中的唯一要求是圆括号之前的表达式必须求值为一个函数地址。这意味着一个函数可以通过任何函数指针表达式调用。

例子

这个例子说明函数调用,它从一个 switch 语句调用函数:

```
void main()
{
    /* 函数原型 */
```

```
    long lift(int),step(int),drop(int);
    void work(int number,long (*function)(int i));
    int select,count;
    .
    .
    .
    select =1;
    switch(select)
   {
        case 1:work(count,lift);
                break;

        case 2:work(count,step);
                break;

        case 3:work(count,drop);
              /* 进入到下一个 case */
        default:
              break;
   }
}
```

```
/* 函数定义 */
void work(int number,long (*function)(int i))
{
    int i;
    long j;

  for (i=j=0;i<number;i++)
       j += (*function)(i);
}
```

在这个例子中,函数在 main 中调用:

work (count, lift),

它传送一个整数变量 count 和函数 lift 的地址给函数 work。注意该函数的地址通过给出的函数标识符简单地传送,由于一个函数标识符求值为一个指针表达式。为了以这种方式使用一个函数标识符,函数必须在该标识符被使用之前说明或定义;否则,不认识该标识符。在这种情况下,work 的一个原型在 main 函数的开头给出。

work 中的参数 function 说明为一个这样的函数的一个指针,该函数有一个 int 参量和返回一个 long 值。括起参数名称的圆括号是需要的;没有它们,该说明指出一个返回 long 值指针的函数。

函数 work 从 for 循环中通过作用如下函数调用方式调用选择的函数:

(*function)(i);

一个参量 i 传送给被调用的函数。

参　量

在一个函数调用中的参量具有这种格式:

表达式(表达式表$_{opt}$)　/* 函数调用 */

在一个函数调用中,表达式表是一个表达式的表(由逗号分隔)。这些表达式的最终值传送给函数的参量。如果该函数没有参量,表达式包含关键字 void。

一个参量可以是任何具有基本的、结构、联合或指针类型的值。所有参量都通过值传送。这意味着把该参量的拷贝赋给对应的参数。该函数不知道传送的参量的实际存储器位置。该函数使用这个拷贝不影响最初导出的变量。

虽然你不能作为参量传送数组或函数,你可以传送它们的指针。指针提供了一个函数通过引用访问一个值的方式。由于一个变量的指针保存变量的地址,函数使用这个地址访问该变量的值。指针参量允许一个函数访问数组和函数,尽管数组和函数不能作为参量传送。

参量求值的次序在不同编译器下和不同的优化层中可能发生变化。但在进入函数之前参量和任何副作用都必须完整地求值。有关副作用的信息参见第 4 章“表达式和赋值”中的“副作用”。

在一个函数调用中的表达式被求值并在函数调用中每个参量上执行常用的算术转换。如果一个原型是可用的,结果参量类型与原型对应的参数进行比较。如果它们不匹配,执行一次转换或发生一个诊断消息。参数

也经历常用的算术转换。
在表达式表中的表达式个数必须与参数个数相匹配。除非该函数的原型或定义显式指出参量个数是可变的,在这种情况下,编译器检测参数表中类型名称那样多的参量,如果必要,对它进行转换。有关更多信息,参见下一节的“用可变个数参量调用”。
如果该原型的参数表仅包含关键字 void,编译器在函数调用中期待 0 个参量,在定义中期待 0 个参数。如果发现了任何参量,编译器发出一个诊断消息。
例子
这个例子使用指针作为参量:

```
void main()
{
    /* 函数原型 */

    void swap(int *num1,int *num2);
    int x,y;
    .
    .
    .
    swap(&x,&y);   /* 函数调用 */
}
    /* 函数定义 */
```

```
void swap(int *num1,int *num2)
{
    int t;

    t=*num1;
    *num1=*num2;
    *num2=t;
}
```

在这个例子中,swap 函数在 main 中说明,它有两个参量,分别用标识符 num1 和 num2 表示,两个都是 int 值的指针。在原型定义中参量 num1 和 num2 也都说明为 int 类型值的指针。

在函数调用中:

```
swap (&x,&y);
```

x 的地址存储在 num1 中,y 的地址存储在 num2 中,现在在同样的位置存在两个名称或“别名”。swap 中 *num1 和 *num2 的引用有效地引用 main 中的 x 和 y。swap 中的赋值实际上交换 x 和 y 的内容。因此,不必要使用 return 语句。

编译器在 swap 的参量上执行类型检测,swap 的原型包括每个参数的参量类型。在该原型和定义中圆括号中的标识符可以是相同或不相同的。重要的是在原型和定义中参量的类型必须与参数其中的类型相匹配。

用可变个数参量调用

部分参数表可以通过省略标记终止。一个逗号后跟三个句点(,...)指出可以有更多的参量传送给函数,但它们没有给出更多信息。类型检测不是在每个参量上执行。在省略标识的前面必须至少有一个参数,省略标记必须是参数表中最后一个语言符号。没有省略标记,如果它接受除了参数表中说明的参数外的参数时,该函数的行为是不确定的。

为了用一个可变个数的参量调用一个函数,在函数调用中简单地指出任何个数的参量。

一个例子是 C 运行库的 printf 函数,该函数调用对于参数表或参量类型中说明的每个类型名称必须包含一个参量。

在函数调用中指定的所有参量都放在栈中,除非指定 _fastcall 调用约定。函数说明的参数个数确定从栈中取多少参量并赋给这些参数。你要负责从栈中接受任何另外的参量并确定出现多少参量。STDARGS.H 文件包含访问具有可变参量个数的函数的参量的 ANSI 方式宏。

同样,也支持包含在 VARARGS.H 中的 XENIX 方式宏。

这个样本说明是一个调用可变参量个数的函数:

```
int averagc(int first,...)
```

Microsoft 特殊处→

为了维护与 Microsoft C 以前版本的兼容性,ANSI C 标准的一个 Microsoft 扩充是允许参数表的末尾有一个没有句点的逗号(,)指出可变的参量个数。但建议把这个代码改变为加上省略标记。

Microsoft 特殊处结束

递归函数

在 C 程序中的任何函数可以递归调用,也就是,它可以调用本身。递归调用的次数受限于栈的尺寸。有关设置栈尺寸的链接器选项的信息,参见联机“Microsoft Visual C++ 6.0 程序员指南”中的“栈分配”(/STACK)链接器选项。每次调用该函数时,为其参数和 auto 及 register 变量分配新的存储以便它们前面的未完成的调用不被覆盖。参数只能直接访问建立它们的函数的实例。以前的参数不能直接访问后面的函数实例。
注意有 static 存储说明的变量在每次递归调用时不需要新的存储。它们的存储存在于程序的生存期中,这样的变量的每次引用都访问相同的存储区域。
例子
这个例子说明了递归调用:

```
int factorial(int num);/* 函数原型 */

void main()
{
   int result,number;
   .
   .
   .
```

```
   result=factorial(number);
}

int factorial(int num)     /* 函数定义 */
{
   .
   .
   .
   if ((num>0)   ||   (num<=10))
       return (num*factorial(num-1));
}
```

附录 A　C 语言语法总结

本附录给出了 C 语言和 Microsoft C 语言特征的完整描述。你可以使用本附录中的说明确定任何语言组成成分的准确语法。在本手册讨论主题的章节中解释了本节出现的语法。
注意:这个语法总结不是 ANSI C 标准部分,仅包括它的信息。Microsoft 特殊的语法在后面语法的注释中作了说明。

定义和约定

终结符是语法定义中的终点,不可能有其它分解。终结符包括一组保留字和用户定义标识符。
非终结符在语法中是位置占用者,在这个语法总结中的其它地方定义。定义可以是递归的。
一个任选的组成成分可以通过加上 opt 下标来指出,例如
{表达式 $_{opt}$}
指出在花括号中括起一个任选的表达式。
语法约定对语法的不同组成成分使用不同的字体属性,其符号和字体如下:

属性	说明
非终结符	斜体类型指出非终结符
const	粗体类型的终结符是必须输入的保留字和符号。在这个上下文中的字符总是大小写敏感的
opt	后有带有 opt 的非终结符总是任选的
缺省字体	以这种字体描述或列出的字符可以作为 C 语句中的终结符

一个非终结符后跟一个冒号(:)指出它的定义。另一种定义以分行形式列出,除了前有“如下之一”的词外。

词法

- 语言符号
- 关键字
- 标识符
- 常量
- 字符串文字
- 运算符
- 标点

语言符号总结

语言符号：
    关键字
    标识符
    常量
    字符串文字
    运算符
    标点
预处理器语言符号：
    头名称
    标识符
    pp 数
    字符常量
    字符串文字
    运算符
    标点
    每个不是上述之一的非空白字符
头名称：
    <路径指示符>
    “路径指示符”
路径指示符：

合法的文件路径

pp数：

数字

.数字

pp数 数字

pp数 非数字

pp数 e 符号

pp数 E 符号

pp数.

关键字总结

关键字:如下之一

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

标识符总结

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

常量总结

常量：
    浮点常量
    整数常量
    枚举常量
    字符常量
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: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

整数常量：

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

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 十六进制常量
0X 十六进制常量
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
无符号后缀:如下之一
u U
长整数后缀:如下之一
l L
字符常量:
'c 字符序列'
L'c 字符序列'
整数后缀:

无符号后缀 长整数后缀 opt
长整数后缀 无符号后缀 opt

c字符：
源字符集中除单引号(')、反斜杠(\)或换行符外的任何成员
转义序列

转义序列：
简单转义序列
八进制转义序列
十进制转义序列

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

字符串文字总结

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：
源字符集中除双引号(")、反斜杠(\)或换行符外的任何成员
转义序列

运算符

运算符:如下之一
[ ] ( ) . ->
++ -- & * + - ~ ! sizeof
/ % << >> < > <= >= == != ^ | && !!
? :
= *= /= %= += -= <<= >>= &= ^= |=
, # ##

赋值运算符:如下之一
= *= /= %= += -= <<= >>= &= ^= |=

标　点

标点:如下之一

[ ] ( ) { } * , : = ; ... #

短语结构语法

- 表达式
- 说明
- 语句
- 外部定义

表达式总结

基本表达式

标识符

常量

字符串文字

(表达式)

表达式:

赋值表达式

表达式,赋值表达式

常量表达式：
    条件表达式
条件表达式：
    逻辑 OR 表达式
    逻辑 OR 表达式 ? 表达式 : 条件表达式
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
后缀表达式：
    基本表达式
    后缀表达式[表达式]
    后缀表达式(参量表达式表 $_{opt}$)
    后缀表达式.标识符
    后缀表达式->标识符
    后缀表达式++
    后缀表达式--
赋值表达式表：
    赋值表达式
    赋值表达式表,赋值表达式
单目表达式：
    后缀表达式
    ++单目表达式

--单目表达式
单目表达式 造型表达式
sizeof 单目表达式
sizeof(类型名称)
单目表达式:如下之一
& * + - ~ !
造型表达式:
单目表达式
(类型名称)造型表达式
乘法表达式:
造型表达式
乘法表达式 * 造型表达式
乘法表达式 / 造型表达式
乘法表达式 % 造型表达式
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
位移表达式:
加法表达式
位移表达式<<加法表达式
位移表达式>>加法表达式

关系表达式：
    位移表达式
    关系表达式<位移表达式
    关系表达式>位移表达式
    关系表达式<=位移表达式
    关系表达式>=位移表达式
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
AND表达式：
    相等表达式
    AND表达式 & 相等表达式
异或表达式：
    AND表达式
    异或表达式 ^ AND表达式
或表达式：
    异或表达式
    或表达式 | 异或表达式
逻辑AND表达式：
    或表达式
    逻辑AND表达式 && 或表达式

逻辑 OR 表达式：
    逻辑 AND 表达式
    逻辑 OR 表达式 || 逻辑 AND 表达式

说明总结

    说明：
    说明指示符 属性序列$_{opt}$ 初始说明符表$_{opt}$;
                /* 属性序列是 Microsoft 特殊处 */
说明指示符：
    存储类指示符 说明指示符$_{opt}$
    类型指示符 说明指示符$_{opt}$
    类型修饰符 说明指示符$_{opt}$
属性序列:/* 属性序列是 Microsoft 特殊处 */
    属性 属性序列$_{opt}$
    属性:如下之一   /* Microsoft 特殊处 */
    __asm   __fastcall
    __based __inline
    __cdecl __stdcall
初始说明符表：
    初始说明符
    初始说明符表,初始说明符
初始说明符：

```
    初始说明符
    初始说明符=初始化器    /* 对于标量初始化 */
存储类指示符:
    auto
    register
    static
    extern
    typedef
    __declspec(扩充的说明修饰符序列)    /* Microsoft 特殊处 */
类型指示符:
    void
    char
    short
    int
    __int8  /* Microsoft 特殊处 */
    __int16 /* Microsoft 特殊处 */
    __int32 /* Microsoft 特殊处 */
    __int64 /* Microsoft 特殊处 */
    long
    float
    double
    signed
```

unsigned
结构或联合指示符
枚举指示符
typedef 名称
类型修饰符：
const
volatile
说明符：
指针 opt 直接说明符
直接说明符：
标识符
(说明符)
直接说明符 [常量表达式 opt]
直接说明符(参数类型表) /* 新式说明符 */
直接说明符(标识符表 opt) /* 已废除的说明符 */
指针：
*类型修饰符表 opt
*类型修饰符表 opt 指针
参数类型表： /* 参数表 */
参数表
参数表,...
参数表：

参数说明
参数表,参数说明
类型修饰符表：
类型修饰符
类型修饰符表 类型修饰符
枚举指示符：
enum 标识符$_{opt}$ {枚举器表}
enum 标识符
枚举器表：
枚举器
枚举器表,枚举器
枚举器：
枚举常量
枚举常量=常量表达式
枚举常量：
标识符
结构或联合指示符：
结构或联合标识符$_{opt}$ {结构说明表}
结构或联合标识符
结构或联合：
struct
union

结构说明表：
    结构说明
    结构说明表 结构说明
结构说明：
    指示符修饰符表 结构说明表
指示符修饰符表：
    类型指示符 指示符修饰符表 $_{opt}$
    类型修饰符 指示符修饰符表 $_{opt}$
结构说明符表：
    结构说明符
    结构说明符表,结构说明符
结构说明符：
    说明符
    类型指示符 说明符 $_{opt}$ ：常量表达式
参数说明：
    说明指示符 说明符 /* 命名的说明符 */
    说明指示符 抽象说明符 $_{opt}$ /* 无名称的说明符 */
标识符表： /* 对于旧式说明符 */
    标识符
    标识符表,标识符
抽象说明符： /* 与无名称的说明符一起使用 */
    指针

    指针 opt 直接抽象说明符
直接抽象说明符:
    (抽象说明符)
    直接抽象说明符 opt [常量表达式 opt]
    直接抽象说明符 opt (参数类型表 opt)
初始化器:
    赋值表达式
    {初始化器表}   /* 对于集合的初始化 */
    {初始化器表,}
初始化器表:
    初始化器
    初始化器表,初始化器
类型名称:
    指示符修饰符表 抽象说明符 opt
typedef 名称:
    标识符
扩充的说明符修饰符序列: /* Microsoft 特殊处 */
    扩充的说明符修饰符 opt
    扩充的说明符修饰符序列 扩充的说明符修饰符
    扩充的说明符修饰符: /* Microsoft 特殊处 */
    thread
    naked

dllimport
dllexport

语句总结

语句:
    标号语句
    复合语句
    表达式语句
    选择语句
    迭代语句
    跳转语句
    try-except 语句   /* Microsoft 特殊处 */
    try-finally 语句   /* Microsoft 特殊处 */
跳转语句:
    goto 标识符;
    continue;
    break;
    return 表达式 opt;
复合语句:
    {说明表 opt 语句表 opt}
说明表:
    说明

```
    说明表 说明
语句表:
    语句
    语句表 语句
表达式语句:
    表达表opt;
迭代语句:
    while (表达式) 语句
    do 语句 while (表达式)
    for (表达式opt ; 表达式opt ; 表达式opt) 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
标号语句:
    标识符:语句
    case 常量表达式 : 语句
    default : 语句
try-except语句:    /* Microsoft特殊处 */
    __try 复合语句
    __except (表达式)  复合语句
try-finally语句:   /* Microsoft特殊处 */
```

```
    __try 复合语句
    __finally   复合语句

外部定义

转换单元：
    外部说明
    转换单元 外部说明
外部说明：   /* 仅允许在外部(文件)范围 */
    函数定义
    说明
函数定义：   /* 这里的说明符是函数说明符 */
   说明指示符 opt 说明符 说明表 opt 复合语句
```

附录 B　实现定义的行为

ANSI X3.159-1989,美国国家信息系统标准——程序设计语言——C,包含一个称为“移植性问题”的附录,这个 ANSI 附录列出了 ANSI 保持开放的每个特殊实现的 C 语言的区域。本附录描述了 Microsoft C 如何处理这些 C 语言的实现定义的区域。

本附录遵循 ANSI 附录同样的次序。每个项包括 ANSI 章节解释实现定义的行为的引用。

注意:本附录仅描述了 C 编译器的 U.S.英语版本,其它语言的 Microsoft 实现可能有点不同。

转换：诊断

ANSI 2.1.1.3 如何标识一个诊断

Microsoft C 以如下格式产生错误消息:

filename(line-number):diagnostic Cnumber message

这里 filename 是遇到错误的源文件的名称;line-number 是编译器检测到错误的行号;diagnostic 是“错误”或“警告”;number 是一个唯一的标识错误或警

告的 4 数字编号(前面有一个像该语法中标记的 C);message 是一个解释的消息。

环　境

- main 的参量
- 交互设备

main 的参量

ANSI 2.1.2.2.1 main 的参量的语义

在 Microsoft C 中,程序启动时调用的函数称为 main。main 没有说明的原型,它可以定义为 0, 2 或 3 个参数:

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

上述第三行 main 接受三个参数,它是 Microsoft 对 ANSI C 标准的扩充。第 3 个参数 envp 是环境变量的指针数组。envp 数组以一个空格指针标志结尾。有关 main 和 envp 的更多信息参见第 2 章“程序结构”中的“main 函数和程序执行”。
变量 argc 不会是负数。
字符串数组以 argv[argc]结尾,它包含一个空格指针。
argv 数组的所有元素都是字符串的指针。

没有命令行参量调用的一个程序将接受为 1 的 argc,可执行文件名称放在 argv[0]中(在 MS-DOS 3.0 以前版本中,该可执行文件名称是不用的,在 argv[0]中放置字母“C”)。argv[1]到 argv[argc-1]所指的字符串表示该程序的参数。
在程序启动到程序终止之间,参数 argc 和 argv 是可修改的并保持它们最后存储的值。

交互设备

ANSI 2.1.2.3 哪些构成一个交互设备

Microsoft C 定义键盘和显示器为交互设备。

标识符的行为

- 没有外部连接的有效字符个数
- 有外部连接的有效字符个数
- 大写和小写

没有外部连接的有效字符个数

ANSI 3.1.2 没有外部连接的有效字符个数

标识符有效字符可达 247 个,编译器不限制一个标识符中使用的字符个数,但它

忽略超过这个限制的任何字符。

有外部连接的有效字符个数

ANSI 3.1.2 有外部连接的有效字符个数

在程序中说明的 extern 标识符用 Microsoft C 编译时可达 247 个有效字符。你可以使用 /H(限制外部名称的长度)选项修改这个缺省值为较小的值。

大写和小写

ANSI 3.1.2 区分大小写

Microsoft C 在一个编译单元中把标识符处理为大小写敏感的。
Microsoft C 链接器是大小写敏感的,你必须根据大小写一致性地指定所有标识符。

字　符

- ASCII 字符集
- 多字节字符
- 每个字符的位数
- 字符集

- 不能表示的字符常量
- 宽字符
- 转换多字节字符
- 字符值的范围

ASCII 字符集

ANSI 2.2.1 源和执行字符集的成员

源字符集是可以出现在源文件中合法的字符串。对于 Microsoft C,源字符集是标准 ASCII 字符集。
警告:因为键盘和控制台驱动程序可以重新映射字符集。用于国际发布的程序要检测国家代码。

多字节字符

ANSI 2.2.1.2 多字节字符的位移状态

多字节字符被某些实现包括 Microsoft C 使用来表示不能以基字符集表示的外国语言字符。但 Microsoft C 不支持任何依赖于国家的编码,因此没有位移状态。更多的信息参见第 1 章,“C 的基本元素”中的“多字节和宽字符”。

每个字符的位数

ANSI 2.2.4.2.1 一个字符中的位数

一个字符中的位数由显式常量 CHAR_BIT 表示。LIMITS.H 文件定义 CHAR_BIT 为 8。

字符集

ANSI 3.1.3.4 映射源字符集的成员

源字符集和执行字符集包括表 B.1 中所列的 ASCII 字符,转义序列也在这个表中列出。

表 B.1 转义序列

转义序列	字符	ASCII 值
\a	报警/响铃	7
\b	回退	8
\f	换页	12
\n	换行	10
\r	回车	13
\t	水平制表	9
\v	垂直制表	11

\"	双引号	34
\'	单引号	39
\\	反斜杠	92

不能表示的字符常量

> ANSI 3.1.3.4 包含一个不能在基本执行字符集中或者对于宽字符常量不能在扩充的字符集中表示的字符或转义序列的整数字符常量的值

所有字符常量或转义序列能够在扩充的字符集中表示。

宽字符

ANSI 3.1.3.4 包含一个以上字符的整数字符常量的值或者包含一个以上多字节字符的宽字符常量
出现的字符常量'ab'的整数值(int)0x6162。当有一个以上字节时,前面读的字节通过 CHAR_BIT 的值向左位移,下一个字节使用按位 OR 运算与 CHAR_BIT 低位进行比较。在该多字节字符常量中的字节数不能超过 sizeof(int),对于 32 位目标代码它为 4。
该多字节字符常是量像上述那样读,使用 mbtowc 运行函数转换为一个宽字符常量,如果该结果不是一个有效的宽字符常量,则发生一个错误。在任何事件mbtowc函数检查的字节个数受限于 MB_CUR_MAX 的值。

转换多字节字符

ANSI 3.1.3.4 使用当前场合把多字节字符转换成一个宽字符常量对应的宽字符(代码)

当前场所缺省为“C”场所。可以使用 setlocale 库例程改变。当前场所的 LC_CTYPE 目录设置当前的工作代码页,它确定多字节和宽字符集之间的对应和转换。mbstowcs，wcstombs，mbtowc 和 wctomb 库例程提供了多字节和宽字符集之间的直接映射。很多流例程，例如 print，scan，get 和 put 簇也自动提供这两个字符集之间的映射。

字符值的范围

ANSI 3.2.1.1 一个“普通的”char 是否和一个 signed char 或一个 unsigned char 值具有相同的范围

所有有符号值范围从 -128 到 127。所有无符号字符值范围从 0 到 255。
/J 编译器选项把缺省值从 signed 改变为 unsigned。

整　数

- 整数值的范围

- 整数的提升
- 有符号的按位运算
- 余数
- 向右位移

整数值的范围

ANSI 3.1.2.5 各种整数类型的表示和值集

整数包含 32 个位(4 个字节)。有符号整数以 2 的补码形式表示。对于负数最重要的有效位为 1;对于正数和 0 为 0。该值列出如下:

类型	最小值和最大值
unsigned short	0 到 65535
signed short	-32768 到 32767
unsigned long	0 到 4294967295
signed long	-2147483648 到 2147483647

整数的提升

ANSI 3.2.1.2 把一个整数转换成一个较短的有符号整数,或者把一个无符号整数转换成相等长度的有符号整数,如果转换的值不能表示,那么结果如何

当一个 long 整数造型转换为一个 short,或者一个 short 造型转换为一个 char,则保留低位的有效字节。
例如,这样的行:

```
short x=(short)0x12345678L;
```

把 0x5678 赋给 x,这样的行:

```
char y=(char)0x1234;
```

把值 0x34 赋给 y。
当把有符号变量转换成无符号值,位模式保持相同,反之亦然。例如,造型-2(0xFE)转换为一个无符号值产生 254(也就是 0xFE)。

有符号按位运算

ANSI 3.3 有符号整数上按位运算的结果

在有符号整数上的按位运算与在无符号整数上的按位运算相同。例如,-16&99 以二进制表示为:

```
  11111111 11110000
```

```
& 00000000 01100011
  --------------------------
  00000000 01100000
```

该按位 AND 的结果为 96。

余　数

ANSI 3.3.5 整数除法中余数的符号

余数的符号与被除数的符号相同。例如：

```
50/-6 == -8
50%-6 == 2
-50/6 == -8
-50%6 == -2
```

向右位移

ANSI 3.3.7 一个有符号整数类型的负数向右位移的结果

一个负数向右位移产生其绝对值向下舍入的一半。例如-253(二进制 11111111 00000011)向右位移一位产生-127(二进制 11111111 10000001)。一个正的 253 向右位移产生+126。

右位移保留符号位。当一个有符号整数右位移时，最高有效位保留设置。当一

个无符号整数右位移时，最高有效位被清除。
如果 0xF000 是无符号的,其结果是 0x7800。
如果 0xF0000000 是有符号的,向右位移产生 0xF8000000。向右位移一个正数 32 次产生 0xF0000000;向右位移一个负数 32 次产生 0xFFFFFFFF。

浮点数学

- 值
- 造型转换整数为浮点值
- 浮点值的截除

值

ANSI 3.1.2.5 各种浮点数类型的表示和值集

float 类型包含 32 位:1 个符号位,8 个指数位和 32 个尾数位。它的范围是+/-3.4E38,至少有 7 个精度数字。
double 类型包含 64 位:1 个符号位,11 个指数位和 52 个尾数位。它的范围是+/-1.7E308,至少有 15 个精度数字。
long double 类型包含 80 位:1 个符号位,15 个指数位和 64 个尾数位。它的范围是+/-1.2E4932,至少有 19 个精度数字。使用 Microsoft C 编译器,long double 类型的表示与 double 类型一致。

造型转换整数为浮点值

ANSI 3.2.1.3 当一个整数围转换成一个浮点数时,且不能准确地表示最初的值,截除该整数的方向

当一个整数造型转换成一个不能准确表示该值的浮点值时,该值被舍入(向上或向下)到最接近适合的值。
例如,造型转换一个 unsigned long(有 32 个精度位)为一个 float(其尾数有 32 个精度位),舍入该数为最接近 256 的倍数。long 值 4294966913 到 4294967167 转换为 float 值都舍入为 4294967040。

浮点值的截除

ANSI 3.2.1.4 当一个浮点数转换成一个更窄的浮点数时,截除或舍入的方向

当出现一个下溢出时,一个浮点变量的值向下舍入为 0。一个上溢出导致一个运行错误,根据指定的优化。可能产生一个不可预料的值。

数组的指针

- 最大数组尺寸

- 指针相减

最大数组尺寸

ANSI 3.3.3.4，4.1.1 需要一个整数类型保存一个数组的最大尺寸,也就是 size_t 的尺寸

size_t 类型定义是一个范围从 0x00000000 到 0x7CFFFFFF 的 unsigned int。

指针相减

ANSI 3.3.6，4.1.1 需要一整数类型保存相同数组两个元素之间的差,即 ptrdiff_t

ptrdiff_t 是一个范围从 -4294967296 到 4294967295 的 signed int。

寄存器:寄存器的可用性

ANSI 3.5.1 通过使用 register 存储类指示符实际放在寄存器中对象的范围

32 位编译器能赞成用户使用 register 变量。而是在优化时做自己的选择。

结构、联合、枚举和位域

- 联合的不适当访问
- 结构成员的填充和对齐
- 位域的符号
- 位域的存储
- enum 类型

联合的不适当访问

ANSI 3.3.2.3 使用一个不同类型的成员访问一个联合对象的成员

如果说明了一个联合的两个类型,且存储了一个值,但用另一个类型来访问该联合,其结果是不可预料的。
例如,一个联合以 float 和 int 说明了,存储了一个 float 值,但程序后来作用 int 来访问它,在这种情况下,该值依赖于 float 值的内部存储。该整数值不是可靠的。

结构成员的填充和对齐

ANSI 3.5.2.1 结构的成员的填充和对齐以及一个位域是否跨立一个存储单元的界限

结构成员以它们说明的次序进行存储,第一个成员具有最低的存储器地址,最后成员具有最高的存储器地址。
每个数据对象都有对齐要求。除结构、联合和数组外的所有数据的对齐要求是该对象的尺寸或者紧凑尺寸(用/Zp 或 pack 编译指示指定,它更少些)。对于结构、联合和数组,对齐要求是它的最大对齐要求。
每个对象分配一个偏移量以便:
偏移量 % 对齐要求==0
如果整数类型有相同尺寸,且如果通过位域的公共对齐要求强制下一个位域适合当前分配单元而不交叉界限,则相邻的位域都紧凑成相同的 1, 2 或 4 字节分配单元。

位域的符号

ANSI 3.5.2.1 一个“普通的”int域是处理为 signed int位域还是处理为 unsigned int位域

位域可以是 signed或 unsigned。普通位域处理为有符号的。

位域的存储

ANSI 3.5.2.1 在一个 int 中位域的分配次序

在一个整数中的位域分配从最低位到最高位。在如下代码中：

```
struct mybitfields
{
  unsigned a : 4;
  unsigned b : 5;
  unsigned c : 7;
} test;

void main(void)
{
  test.a=2;
  test.b=31;
  test.c=0;
}
```

位排到如下：

```
00000001 11110010
cccccccb bbbbaaaa
```

由于 80x86 处理器在高位字节之前存储低位字节,上面的整数 0x01F2 在物理存储器中的存储是先存储 0xF2 后跟 0x01。

enum 类型

ANSI 3.5.2.2 选择整数类型表示一个枚举类型的值

一个以 enum 说明的变量是一个 int。

修饰符:访问 volatile 对象

ANSI 3.5.5.3 什么构成对具有 volatile 修饰的类型的对象的访问

任何对一个 volatile 修饰的类型的引用是一个访问。

说明符:最大个数

ANSI 3.5.4 修饰一个算术的、结构或联合类型的说明符的最大个数

Microsoft C 不限制说明符的个数,该个数仅受限于可用的存储器空间。

语句:switch 语句的限制

ANSI 3.6.4.2 一个 switch 语句中 case 值的最大个数

Microsoft C 不限制一个 switch 语句中 case 值的个数,该个数仅受限于可用的存储器空间。

预处理器命令

- 字符常量和条件包含
- 包括括起的文件名
- 包括引用的文件名
- 字符序列
- 编译指示
- 缺省的日期和时间

字符常量和条件包含

ANSI 3.8.1 常量表达式中一个单字符字符常量的值控制条件包含是否与执行字符集中相同字符常量的值匹配。这样的一个字符常量是否可以有一个负值。在预处理器语句中使用的字符集与执行字符集相同。预处理器识别负字符值。

包括括起的文件名

ANSI 3.8.2 定位包括的源文件的方法

对于用角括号括起的文件规格,预处理器不搜索父文件的目录。一个“父”文件是其中已经有#include 命令的文件。而且从编译器命令行中/I 之后指定的目录开始搜索。如果/I 选择没有出现或失败,该预处理器使用 INCLUDE 环境变量查找任何角括号中括起的文件。

INCLUDE 环境变量中可以包含由分号(;)分隔的多个路径。如果一个以上的目录出现在/I 选项之中或 INCLUDE 环境变量之中,那么预处理器以它们出现的次序进行搜索它们。

包括引用的文件名

ANSI 3.8.2 支持包括的源文件中引用的名称

如果你在两个双引号("")之间为包括的文件指出一个完整的、不模糊的路径规格,该预处理器只搜索该路径规格而忽略标准目录。
对于指定为#include"path-spec"的包括的文件,目录搜索开始于父文件的目录,然后继续通过任何父文件的目录进行搜索。因此,搜索相对于包含当前源文件的目录开始。如果没有父文件和该文件未找到,继续搜索。好像该文件被括在角括号中。

字符序列

ANSI 3.8.2 源文件字符序列的映射

除了不支持转义序列外,预处理器使用和源文件语句中相同的字符集。
因此,为了指定一个包括文件的路径,仅使用一个反斜杠:

```
#include "path1/path2/myfile"
```

在源代码中,必须使用两个反斜杠:

```
fil=fopen("path1\\path2\\myfile","rt");
```

编译指示

ANSI 3.8.6，每个识别的#pragma命令上的行为

Microsoft C编译器定义了如下编译指示：

alloc_text	data_seg	include_alias	setlocale
auto_inline	function	intrinsic	warning
check_stack	hdrstop	message	
code_seg	inline_depth	optimize	
comment	inline_rescursion	pack	

缺省日期和时间

ANSI 3.8.8 当转换的日期和时间不可用时，_DATE_和_TIME_分别定义

当操作系统不支持转换的日期和时间时，_DATE_和_TIME_R 缺省值是“may 03 1957”和“17:00:00”。

库函数

- NULL 宏
- assert 函数打印的诊断
- 字符测试
- 域错误
- 浮点值的下溢出
- fmod 函数
- signal 函数
- 缺省信号
- 结束换行字符
- 空行
- 空格字符
- 添加模式中的文件位置
- 截除文本文件
- 文件缓冲
- 0 长度文件
- 文件名
- 文件访问限制
- 删除打开的文件
- 用一个存在的名称更名
- 读指针值

- 读范围
- 文件位置错误
- perror 函数产生的消息
- 分配 0 存储器
- abort 函数
- atexit 函数
- 环境名称
- system 函数
- strerror 函数
- 时区
- clock 函数

NULL 宏

ANSI 4.1.5 宏 NULL 伸展为空指针常量

几个包括文件定义 NULL 宏为((void*)0)。

assert 函数打印的诊断

ANSI 4.2 assert 函数打印的诊断和终止行为

assert 函数打印一个诊断消息,如果该表达式为假(0)则调用 abort 例程。诊断

消息格式为：
assertion failed:expression,file filename, line linenumber
这里 filename 是源文件的名称,linenumber 是该源文件中断言失败的行号。如果该表达式 expression 为真(非 0),则不发生动作。

字符测试

ANSI 4.3.1 isalnum、isalpha、iscntrl、islower、isprint 和 isupper 函数测试的字符集

下表描述了由 Microsoft C 编译器实现的函数。

函数	测试条件
isalnum	字符 0-9、A-Z、a-z、ASCII 48-57、65-90、97-122
isalpha	字符 A-Z、a-z、ASCII 65-90、97-122
iscntrl	ASCII 0-31、127
islower	字符 a-z、ASCII 97-122
isprint	字符 A-Z、a-z、0-9、标点、空格、ASCII 32-126
isupper	字符 A-Z、ASCII 65-90

域错误

ANSI 4.5.1 数学函数的返回值存在域错误

ERRNO.H 文件定义域错误常量 EDOM 为 33。

浮点值的下溢出

ANSI 4.5.1 在下溢出范围错误时,数学函数设置表达式 errno 为宏 ERANGE 的值

一个浮点下溢出不设置表达式 errno 为 ERANGE,当一个值接近 0 并最后下溢出,这个值设置为 0。

fmod 函数

ANSI 4.5.6.4 当 fmod 函数有第二个 0 参量时出现一个域错误或返回 0

当 fmod 函数有第二个 0 的参量时,该函数返回 0。

signal 函数

ANSI 4.7.1.1 signal 函数的信号集

传给 signal 的第一个参量必须是“Microsoft Visual C++ 6.0 参考库”的一卷“MicrosoftVisual C++6.0 运行库参量”中的信息，还列出了每种信号支持的操作模式。这些常量也定义在 SIGNAL.H 中。

缺省信号

ANSI 4.7.1.1 如果 signal 的等价者(sig,SIG_DFL)不在调用一个信号处理器之前执行,则执行该信号的分块

在一个程序开始运行时,信号设置为它们的缺省状态。

结束换行字符

ANSI 4.9.2 一个文本流的最后行需要一个结束换行字符

流函数识别换行或文件结尾作为一行的结束字符。

空　行

ANSI 4.9.2 当读入时出现一个换行字符之前空行是否立即写入文本流

空字符被保留。

空格字符

ANSI 4.9.2 可以添加到写入二进制流中的数据的空字符个数

任意个数的空格字符可以添加到一个二进制流中。

添加模式中的文件位置

ANSI 4.9.3 一个添加模式的文件位置指示符初始时是否处于该文件的结尾

当一个文件以添加模式打开时,文件位置指示符初始地指向文件的结尾。

截除文本文件

ANSI 4.9.3 在一个文本流上写是否导致关联文件超过这个点的部分被截除

在一个文本流上写不会截除超过这个点的部分。

文件缓冲

ANSI 4.9.3 文件缓冲特性

通过标准 I/0 函数访问的磁盘文件都是全缓冲的。缺省地,该缓冲区为 512 个字节。

0 长度文件

ANSI 4.9.3 一个 0 长度的文件是否实际存在

0 长度的文件是允许的。

文件名

ANSI 4.9.3 组成有效文件名的规则

一个文件格式可以包括一个任选的驱动器字母(后跟一个符号)、一系列任选目录名称(用反斜杠分隔)和一个文件名。
文件名和目录名可以包含至多 8 个字符后跟一个句点以及三个字符的扩展名。
文件名是大小写无关的。在名称或扩展名中不允许有通配符*和?。

文件访问限制

ANSI 4.9.3 同一文件是否可以多次打开

不允许打开一个已打开的文件。

删除打开的文件

ANSI 4.9.4.1 remove 函数在一个打开文件上的作用

remove 函数删除一个文件,如果该文件是打开的,这个函数失败并返回-1。

用一个存在的名称更名

ANSI 4.9.4.2 如果调用 rename 函数之前和新文件名同名的文件已存在,其结果如何

如果你试图使用一个已存在的名称更名一个文件,rename 函数失败并返回一个错误代码。

读指针值

ANSI 4.9.6.2 在 fscanf 函数中输入%p 的转换。

当指定%p 格式字符时,fscanf 函数把指针从十六进制 ASCII 值转换成正确的地址。

读范围

ANSI 4.9.6.2 在 fscanf 函数中%[转换的扫描表中第一个或最后一个破折号(——)字符的解释

如下行:

```
fscanf(fileptr,"%[A-Z]",strptr);
```

读 A-Z 范围内的任何数目的字符到 strptr 所指的字符串中。

文件位置错误

ANSI 4.9.9.1, 4.9.9.4 在失败时 fgetpos 或 ftell 函数设置的宏 errno 的值

当 fgetpos 或 ftell 失败时,如果位置是无效的,errno 设置为 EINVAL;如果文件号是坏的,则 errno 设置为 EBADF。这些显式常量定义在 ERRNO.H 中。

perror 函数产生的消息

ANSI 4.9.10.4 由 perror 函数产生的消息

perror 函数产生这些消息:

```
0   Error 0
1
2   No such file or directory
3
4
5
6
7   Arg list too long
```

```
8   Exec format error
9   Bad file number
10
11
12  Not enough core
13  Permission denied
14
15
16
17  File exists
18  Cross-device link
19
20
21
22  Invalid argument
23
24  Too many open files
25
26
27
28  No space left on device
29
```

```
30
31
32
33 Math argument
34 Result too large
35
36 Resource deadlock would occur
```

分配 0 存储器

ANSI 4.10.3 如果请求的尺寸为 0,calloc、malloc 或 realloc 函数的行为

calloc、malloc 和 realloc 函数接受 0 为参量。不分配实际的存储器,但返回一个有效指针,该存储器块可以在后面由 realloc 修改。

abort 函数

ANSI 4.10.4.1 abort 函数在打开的和临时文件上的行为

abort 函数不关闭打开的或临时的文件。它不刷新流缓冲区。

atexit 函数

ANSI 4.10.4.3 **如果参量的值为非 0 值即** EXIT_SUCCESS **或** EXIT_FAILURE,atexit **函数的返回状态**

如果成功,atexit 函数返回 0;如果不成功,返回非 0 值。

环境名称

ANSI 4.10.4.4 **环境名称集和通过** getenv **函数使用的改变环境表的方法**

环境名称集是无限制的。
为了从一个 C 程序改变环境变量,调用 _putenv **函数。为了从 Windows 95 或 Windows NT 中的命令行改变环境变量,使用 SET 命令(例如,SET LIB=D:\LIBS)。从一个 C 程序设置环境变量仅在该操作系统命令外壳的主拷贝被运行(在 Windows NT 中为 CMD.EXE,在 Windows 95 中为 COMMAND.COM)时存在。例如,以下行:**

```
system(SET LIB=D:\LIBS);
```

将运行 Windows NT 命令外壳(CMD.EXE)的一个拷贝,设置环境变量 LIB,返回到 C 程序,并退出 CMD.EXE 的第二个拷贝。退出 CMD.EXE 的拷贝时删除临时环境变量 LIB。
这个例子也可以在 Windows 95 平台上运行。

同样,_putenv 函数所作的改变仅延续到该程序终止。

system 函数

ANSI 4.10.4.5 system 函数执行的字符串的内容和模式

system 函数在一个 C 程序中而不是命令行执行一个内部操作系统命令或.EXE、.COM(在 Windows 95 中为.CMD)或.BAT 文件。
system函数查找命令解释器,在 Windows NT 操作系统中为 CMD.EXE,在 Windows 95 中为 COMMAND.COM。system 函数然后把参量字符串传给该命令解释器。

strerror 函数

ANSI 4.11.6.2 strerror 函数返回的错误消息字符串的内容

strerror 函数产生这些消息:
0 Error 0
1
2 No such file or directory
3
4
5
6

```
7  Arg list too long
8  Exec format error
9  Bad file number
10
11
12 Not enough core
13 Permission denied
14
15
16
17 File exists
18 Cross-device link
19
20
21
22 Invalid argument
23
24 Too many open files
25
26
27
28 No space left on device
```

29
30
31
32
33 Math argument
34 Result too large
35
36 Resource deadlock would occur

时　区

ANSI 4.12.1 本地时区和夏令时

本地时区为太平洋标准时间。Microsoft C 支持夏令时。

clock 函数

ANSI 4.12.2.1　clock 函数的时间

clock 函数的时间起始(从 0 开始)于 C 程序开始执行。它返回以 1/CLOCKS_PER_SEC(对于 Microsoft C 它等于 1/1000)为单位的时间。