1 C 的标准化过程

C 语言自诞生到现在，期间经历了多次标准化过程，主要分成以下几个阶段：

1.1 Traditional C

此时的 C 语言还没有标准化，来自“*C Programming Language*, First Edition, by Brian W. Kernighan, Dennis M. Ritchie. Prentice Hall PTR 1978”的 C 描述可算作“正式”的标准，所以此时的 C 也称为“K&R”C。

期间 C 语言一直不断的发生细微的变化，各编译器厂商也有自己的扩展，这个过程一直持续到 20 世纪 80 年代末。

1.2 C89

考虑到标准化的重要，ANSI(American National Standards Institute)制定了第一个 C 标准，在 1989 年被正式采用(American National Standard X3.159-1989)，故称为 C89，也称为 ANSI C。

该标准随后被 ISO 采纳，成为国际标准(ISO/IEC 9899:1990)。

C89 的主要改动：

- 定义了 C 标准库；
- 新的预处理命令和特性；
- 函数原型(prototype)；
- 新关键字：const、volatile、signed；
- 宽字符、宽字符串和多字节字符；
- 转化规则、声明(declaration)、类型检查的改变。

1.3 C95

这是对 C89 的一个修订和扩充，称为“C89 with Amendment 1”或 C95，严格说来并不是一个真正的标准。

C95 的主要改动：

- 3 个新标准头文件：iso646.h、wctype.h、wchar.h；
- 一些新的标记(token)和宏(macro)；
- 一些新的 printf/scanf 系列函数的格式符；
- 增加了大量的宽字符和多字节字符函数、常数和类型。

1.4 C99

这是目前最新的标准，由 ISO 制定于 1999 年(ISO/IEC 9899:1999)，故称为 C99。

C99 的主要改动：

- 复数(complex)；
- 整数(integer)类型扩展；
- 变长数组；
- Boolean 类型；
- 非英语字符集的更好支持；
- 浮点类型的更好支持；
- 提供全部类型的数学函数；
- C++风格注释(//)。

2 C 标准文档

2.1 C99

这是一个 pdf 文件：c99.pdf。

2.2 C89

C99 已经替代 C89 成为标准，所以 C89 文档已经很难找了。

下面是书籍“*C Programming Language*, Second Edition, by Brian W. Kernighan, Dennis M. Ritchie. Prentice Hall PTR (April 1, 1988), 0131103628.”附录 A 的一份拷贝，在此作为 C89 标准以供需要时查阅。

同时也提供中文版本，内容来自该书对应的中译版“《C 程序设计语言》，徐宝文等译，机械工业出版社出版，ISBN 7111075897”。

文档仅供个人参考使用(建议以英文版为主)：

英文版　　中文版

为方便起见，提供一份标点符号中英对照表，希望有用：标点符号中英对照表。

3 C 标准的选择

选择标准依赖于编译器的支持和对可移植性的要求。

C99 是当前的标准，但它仍未得到广泛支持，虽然标准发布已经多年。C99 对 C89(C95)的改动非常大，如果编写 C99 的代码，那么可移植性必然受到限制。此外，个人认为 C99 的一些新特性在大多数程序设计中并不是必须的。

C89(包括 C95)是目前使用最广泛的，并得到所有主流编译器的支持。

Traditional C 现在只会在一些非常老的代码中才能见到了，除非你在维护旧代码，否则不应该再使用它。

所以，个人觉得当前还是以 C89(包括 C95)标准为主。

1 引言

本手册描述的 c 语言是 1988 年 10 月 31 日提交给 ANSI 的草案，批准号为“美国国家信息系统标准—C 程序设计语言，X3.159-1989”。尽管我们已非常小心，以便这个手册的介绍可以信赖，但它毕竟不是标准本身，而是对标准的一个解释。这个手册的安排基本与标准相似，也与本书的第 1 版相似，但是对细节的组织是不同的。本手册给出的语法与标难是一样的，只是有少量产生式有所修改，词法元素和预处理器的定义也非形式化。注释部分说明了 ANSI 标准 C 与本书第 1 版介绍的或其他编译器所支持的语言的细微差别。

2 词法规则

一个程序由存储在文件中的一个或多个翻译单元织成，程序的翻译分几个阶段完成，这将在 12 节中介绍。翻译的第一阶段完成低级的词法转换，执行由字符# 开始的行所引入的指令，并进行宏定义和宏扩展。当预处理(将在 12 节中介绍)完成后，程序就被归约成一个单词序列。

2.1 单词

共有 6 类单词；标识符、关键字、常量、字符串字面值、运算符和其他分隔符。空格、横向和纵向制表符. 换行符、换页符和注解(合称空白符)在程序中仅用来分隔单词，因此将被略过。空白符用来分开相邻的标识符、关键字和常量。

如果到某一字符为止的输人流被分成若干单词.那么下一个单词就是可能组成单词的最长的字符串。

2.2 注解

注解以字符/*开始，以*/结束。注解不可以嵌套，也不可以出现在字符串中或字符字面值中。

2.3 标识符

标识符是一个字母和数字的序列，其第一个字符必须是一个字母，下划线_也被当做字母。大写和小写字母组成的标识符是不同的。标识符可以任意长。对于内部标识符，至少前 31 个字母是有意义的，在某些实现中这个值可以更大。内部标识符包括预处理的宏名和其他没有外部连接(见 11.2 节)的名字。有外部连接的标识符的限制要多一些，其实现可能只认为前 6 个字符是有意义的. 而且有可能忽略大小写的不同。

2.4 关键字

以下标识符被保留为关键字，它们不能用做别的用途：

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				

有些实现还把单词 fortran 和 asm 保留为关键字。

- 关健字 const、signed 和 volatile 是 ANSI 标准中新增加的，enum 和 void 是第 1 版后新增如的，entry 曾经被保留为关键字，但现在已不是了。

2.5 常量

共有几种类型的常量，它们每一种都有一个数据类型，基本类型将在 4.2 节讨论。

常量:

整数常量

字符常量

浮点常量

枚举常量

2.5.1 整数常量

整数常量由一串数字序列组成。如果它以 0(数字 0)开始，那么是八进制数．否则就是十进制数。八进制常量不包括数字 8 和 9。以 0x 和 0X(数字 0)开始的数字序列是十六进制数，十六进制数包含到从 a～f 或从 A～F 的字母，它们分别表示 10～15。

一个整数常量可以以宁母 u 或 U 为后缀，表示它是一个无符号数；也可以以字母 l 或 L 为后缀，表示它是一个长整数。

一个整数常量的类型依赖于它的形式、值和后缀(类型的讨论见 4 节)。如果它没有后缀且是十进制的，那么它的类型很可能是 int、long int 或 unsigned long int。如果它没有后缀且是八进制的或十六进制的，那么它的类型很可能是 int、unsigned int、long int、unsigned long int。如果它的后缀为 u 或 U，那么

它的类型很可能是 unsigned int 或 unsigned long int。如果它的后缀为 l 或 L，那么它的类型很可能是 long int 或 unsigned long int。

- 整数常量类型的确定比第 1 版要详细得多；在第 1 版中，大的整数常量仅被看做是 long 类型的。U 后缀是新增加的。

2.5.2 字符常量

字符常量是由单引号括住的一个或多个字符的序列，如'x'。单字符常量的值是执行时机器的字符集中的此字符的数值，多字符常量的值由实现定义。

字符常量不包括字符'和换行符，为了表示它们和某些其他的字符，可以使用以下的转义序列(换码序列)：

newline	NL (LF)	\n	换行符
horizontal tab	HT	\t	横向制表符
vertical tab	VT	\v	纵向制表符
backspace	BS	\b	回退符
carriage return	CR	\r	回车符
formfeed	FF	\f	换页符
audible alert	BEL	\a	响铃符
backslash	\	\\	反斜线
question mark	?	\?	问号
single quote	'	\'	单引号
double quote	"	\"	双引号
octal number	ooo	\ooo	八进制数
hex number	hh	\xhh	十六进制数

转义序列\ooo 由反斜杠后跟 1、2 或 3 个用来确定对应字符的值的八进制数字组成。一个普通的例子是\0(其后没有数字)，它表示字符 NUL。转义序列\xhh 由反

斜杠开始，后跟 x，其后是十六进制数字，用来确定对应字符的值。数字的个数没有限制，但如果对应的字符的值超过最大的字符的值，那么该行为是未定义的。对于八进制或十六进制转义字符，如果实现中将类型 char 看做是有符号的，那么将对字符值进行符号扩展，就好像它被强制转换为 char 类型一样。如果\后面的字符不是以上所说明的，那么其行为是未定义的。

在 C 语言的某些实现中，有一个扩展的字符集，扩展的部分不能用 char 类型表示。在该扩展集中，常量是由一个前导 L 开始(如：L'x')，叫做宽字符常量。这种常量的类型为 wchar_t。这是一个整数类型，定义在标准头文件<stddef.h>中。与通常的字符常量一样．可以使用八进制和十六进制的转义序列；但是，如果值超过 wchar_t 可以表示的范围，那么结果是未定义的。

- 某些转义序列是新增加的，特别是十六进制字符的表示。扩展的字符也是新增加的。通常美国和西欧所用的字符集可以用 char 编码；增加 wchar_t 的主要意图是为了表示亚洲的语言。

2.5.3 浮点常量

一个浮点常量包含有一个整数部分、一个小数点、一个小数部分、一个 e 或 E，一个可选的有符号整数类型的指数和一个可选的表示类型的后缀(即 f、F、l 或 L)。整数和小数部分均由数字序列组成。可以没有整数部分或小数部分(但不能二者都没有)。小数点部分或者 e 和指数部分可以没有(但不能二者都没有)。浮点常量的类型由后缀确定，F 或 f 后缀表示它 float 类型；1 或 L 后缀表明它是 long double 类型；若没有后缀则是 double 类型。

- 浮点常量的后缀是新增加的。

2.5.4 枚举常量

定义为枚举符的标识符是 int 类型的常量(见 8.4 节)。

2.6 字符串字面值

字符串字面值也叫字符毒常量，是由双引号括起来的一个字符序列，如"..."。字符串的类型为“字符数组”，存储类为static(见4节)，由给定的字符来初始化。相同的字符串字面值是否看做是不同的取决于具体的实现。如果程序试图改变字符串字面值，那么该行为是未定义的。

我们可以把相邻的字符串字面值连接为一个单一的字符串。任何连接之后，一个空字节\0被加到字符串的后面以使程序在扫描字符串时知道已到达字符串的末尾。字符串字面值不包含换行符和双引号符；但可以用与字符常量相同的转义序列来表示它们。

与字符常量一样，扩展字符集中字符串字面值以前导字符L表示，如L"..."。宽字符的字符串字面值的类型为wchar_t的数组，将普通字符串和宽字符的字符串字面值进行连接是未定义的。

- 如下说明都是ANSI C标准中新增加的：字符串字面值不必相区别、紫止修改字符串字面值以及相邻字符串字面值的连接。宽字符的字符串字面值也是新增加的。

3 语法符号

在本手册用到的语法符号中，语法类别由斜体字表示。字面值单词和字符以打字机字体表示。可选类别通常列在不同的行中，但在少数情况下，一长串短的可选项可以表示在一行中，以短语“之一”(one of)标识。任选的终结符或非终结符带有下标“opt”。例如：

表达式$_{opt}$

表示一个括在花括号中的任选的表达式。语法概要在 13 节中给出。

- 与本书第 1 版给出的语法不同，本书给出的语法使表达式运算符的优先级和结合性是显式的。

4 标识符的含义

标识符也叫名字，可以指代很多实体：函数、结构标记、联合和枚举，结构或联合的成员，枚举常量，类型定义名字以及对象。一个对象，有时也称为变量，是一个存储区域。它的解释依赖于两个主要属性：它的存储类和它的类型。存储类决定了与该标识的对象相关联的存储区域的生命周期，类型决定了该对象的值的含义。一个名字还有一个作用域和一个连接，作用域即程序中可见此名字的区域，连接决定另一区域中的同一个名字是否指代的是同一个对象或函数。作用域相连接将在 11 节今讨论。

4.1 存储类

存储类共有两类：自动存储类和静态存储类。一个对象说明的几个关键字与上下文一起确定了对象的存储类。自动对象对于一个分程序(见 9.3 节)来说是局部的，在退出分程序时该对象将消失。如果没有提到存储类说明．或者如果使用了 auto 区分符，那么分程序中的说明生成的都是自动对象。说明为 register 的对象也是自动的，并且(在可能时)存储在机器的快速寄存器中。

静态对象可以局部于某个分程序，或者对所有分程序来说是外部的。不论是哪一种情况，在退出和再进入函数和分程序时其值不变。在一个分程序包括提供函数代码的分程序内，静态对象用关键字 static 说明。在所有分程序外部说明的且与函数定义在同一级的对象总是静态的，可以通过使用 static 关键字使它们局

部于一个特定的翻译单元，这使得它们有内部连接。通过略去显式的存储类或通过使用关键字 extern，可以使这些对象全局于整个程序，并有外部连接。

4.2 基本类型

有几种基本类型。在 C 标准库中描述的标准头文件<limits.h>定义了在局部实现中每种类型的最大值和最小值。C 标准库给出的数表示最小的可接受限度。

说明为字符(char)的对象要大到足以存储执行字符集(execution character set)中的任何字符。如果字符集中的某个字符存储在一个 char 象中，那么该对象的值等于字符的整数码，并且是非负的。其他量也可存储在 char 变量中；但其取值范围，特别是其值是否有符号．依赖于具体的实现。

以 unsigned char 说明的无符号字符与普通字符占用同样的空间，但其值总是非负的。以 signed char 显式说明的有符号字符也与普通字符占用同样大的空间。

- 在本书的第 1 版中没有 unsigned char 类型，但它的用途很广泛。signed char 是新增加的。

除了 char 类型外，还有 3 种不问大小的整数类型：short int、int 和 long int。普通 int 对象的大小与主机的自然结构一样大，其他大小的整数类型都有特殊的用途。较长的整数至少要占有与较短整数一样的存储空间；但是具体的实现可以便一般整数(int)有与短整数(short int)或长整数(long int)有同样的大小；除非特别说明，整数类型都表示有符号数。

以关键字 unsigned 说明的无符号整数遵守算术模 2n 的规则，其中 n 是相应整数表示的位数。这样对无符号数的算术运算永远不会溢出。可以存储在带符号对象中的非负值的集合是可以存储在相应的无符号对象中的值的子集，并且这两个集合的重叠部分的表示是一样的。

单精度浮点数 float)、双精度浮点数(double)和多精度浮点数(long double)中任何类型可能是同义的，但精度由前到后是上升的。

- long double 是新增加的类型，在第 1 版中 long float 与 double 类型等价，但现在已不再相同。

枚举是具有整型值的一个独特的类型。与每个枚举相关联的是一个有名常量的集合(见 8,4 节)。枚举类型类似于整数类型。但是，如果某个特定枚举类型的对象被赋于的值不是其常量中的一个，或者被赋于的不是一个同类型的表达式，那么枚举类型通常用于编译器以产生警告信息。

因为以上这些类型的对象可以被解释为数字，所以统称它们为算术类型。char 类型、int 族类型，不论大小如何，是否有符号. 都统称为整数类型。类型 float、double 和 long double 统称为浮点类型。

void 类型说明值的一个空集合，它被用来说明那些不产生任何值的函数的类型。

4.3 派生类型

除了基本类型外，我们还可以通过以下几种方法构造派生类型，这些派生类型从概念上说有无限多个：

给定类型的对象的数组；
近日给定类型的对象的函数；
指向给定类型的对象的指针；
包含一系列不同类型的对象的结构；
包含不同类型的几个对象中任意一个的联合。

一般地，在构造对象时可以递归地使用这些方法。

4.4 类型限定符

对象的类型可以有附加的限定符。说明为 const 的对象表明此对象的值不可以改变。说明为 volatile 的对象表明它有与优化相关的特殊属性。限定符既不影响对象的值的范围也不影响其算术属性。限定符将在 8.2 节讨论。

5 对象和左值

对象是一个指名的存储区域，左值是指向某个对象的表达式。左值表达式的一个明显的例子是一个有合适类型与存储类的标识符。某些运算符可以产生左值。例如，如果 E 是一个指针类型的表达式，那么*E 是一个左值表达式，它指代由 E 指向的对象。名字“左值”来源于赋值表达式 E1 = E2，其中左运算分量 E1 必须是一个左值表达式。对每个运算符的讨论说明了此运算符是否需要一个左值运算分量以及它是否产生一个左值。

6 转换

依据运算分量的不同，某些运算符会引起运算分量的值由某个类型转换为另一个类型。本节解释这种转换所产生的结果。6.5 节将讨论大多数普通运算符所需的转换；对每个运算符的讨论会在需要时做补充。

6.1 整提升

在一个表达式中，凡是可以使用整数的地方都可以使用有符号或无符号的字符、短整数和整数的位字段及枚举类型的对象。如果原来类型的所有值都可用 int 类型表示，那么原来类型的值就被转换为 int 类型；否则就被转换为 unsigned int 类型。这一过程称为整提升。

6.2 整数转换

任何整数转换为某个给定的无符号类型的方法是：找出与此整数同余的最小的非负值，其模数为该无符号类型能够表示的最大值加 1。在二进制补码表示中，如果该无符号类型的位模式较窄，那么这就相当于左截取：如果该无符号类型的位模式较宽，那么这就相当于对有符号值进行符号扩展和对无符号值填 0。

当任何整数被转换成有符号类型时，如果它可以在新类型中表示出来则其值不变，否则它的值由具体实现定义。

6.3 整数和浮点数

当把浮点类型的值转换为整数类型时，其小数点部分将被丢弃掉。如果结果值不能用此整类型来表示，那么其行为是未定义的。特别地，将负的浮点数转换为无符号整类型的结果没有指定。

当把整类型的值转换为浮点类型时，如果该值在该浮点类型可表示的范围内但不能精确表示，那么结果可以是下一个较高的或下一个较低的可表示值。如果该值超过可表示的范围，那么其行为是未定义的。

6.4 浮点类型

当一个精度较低的浮点值被转换为有相同或更高精度的浮点类型时，它的值不变。当把一个有较高精度的浮点类型的值转换为精度较低的浮点类型时，如果它的值在可表示范围内，那么结果可以是下一个较高的或下一个较低的表示值。如果结果在范围之外，那么其行为是未定义的。

6.5 算术转换

许多运算符都会以相似的方式在运算过程中引起转换并产生结果类型。其效果是将所有运算分量转换为同一类型，并以此作为结果的类型。这种方式的转换称为普通算术转换。

1. 首先，如果一个运算分量为 long double 类型，那么另一个也被转换为 long double 类型。
2. 否则，如果一个运算分量为 double 类型，那么另一个也被转换为 double 类型。
3. 否则，如果一个运算分量为 float 类型，那么另一个也被转换为 float 类型。
4. 否则，同时对两个运算分量进行整提升，然后，如果一个运算分量为 unsigned long int 类型，那么另一个也被转换为 unsigned long int 类型。
5. 否则，如果一个运算分量为 long int 类型且另一个运算分量为 unsigned int 类型，那么结果依赖于 long int 类型是否可以表示所有的 unsigned int 类型的值。如果可以，那么 unsigned int 类型的运算分量转换为 long int；如果不可以，那么两个运算分量均转换为 unsigned long int 类型。
6. 否则，如果一个运算分量为 long int 类型，那么另一个也被转换为 long int 类型。
7. 否则，如果一个运算分量为 unsigned int 类型，那么另一个也被转换为 unsigned int 类型。
8. 否则，两个运算分量均为 int 类型。

- 这里有两个变化。第一，对 float 运算分量的算术运算可以只用单精度而不是双精度；而在第 1 版中指定所有的浮点运算都是双精度。第二，当

较短的无符号类型与较长的有符号类型一起运算时，不将无符号类型的属性传递给结果类型；而在第 1 版中无符号类型总是处于支配地位。新规则稍微有点复杂，但减少了当无符号数与有符号数混合使用时的麻烦。但当一个无符号表达式与一个同样大小的有符号表达式相比较时仍会得到不期望的结果。

6.6 指针和整数

指针值可以加上或减去一个整数类型的表达式，在这种情况下，整数表达式的转换按照对加法类运算符的讨论进行(见 7.7 节)。

两个指向同一数组中同一类型的对象的指针可以进行减法运算. 其结果被转换为整数；转换方式按对减法类运算符的讨论进行(见 7.7 节)。

值为 0 的整常量表达式或强制转换为类型 void *的表达式可通过强制转换、赋值或比较转换为另一种类型的指针。其结果将产生一个空指针，此空指针等于同一类型的另一空指针，但不等于任何指向函数或对象的指针。

某些其他涉及指针的转换也可进行，但其结果依赖于具体的实现。这些转换必须由一个显式的类型转换运算符或强制类型转换(见 7.5 节和 8.8 节)来指定。

指针可以转换为整数类型，只要此类型足够大；所要求的大小依赖于具体的实现。映射函数也依赖于实现。

一个整类型对象可以显式地转换为指针类型。映射总是使一个足够宽的从指针转换来的整数转换回到同一个指针，否则其结果依赖于实现。

指向某一类型的指针可以被转换为指向另一类型的指针，但是如果该指针不指向在存储区域中适当对齐的对象，那么结果指针可能会导致地址异常。指向某对象的指针在转换成一个指向其类型有更少或相同的存储对齐方式的限制的对象时，可以保证原封不动地再转换回来时。“对齐”的概念依赖于实现，但 char 类型

的对象有最少的对齐限制。如将在 6.8 节中讨论的，指针也可以转换为 void * 类型，并可原封不动地转换回来。

一个指针可以转换为同样类型的另一个指针.除了增加或删除该指针所指的对象类型的限定符(见 4.4 节和 8.2 节)。如果增加了限定符，那么新指针与原指针等价，不同的是多了由限定符带来的限制。如果删除了限定符，那么对基本对象的运算仍受它实际说明中的限定符的限制。

最后，指向一个函数的指针可以转换为指向另一个函数类型的指针，调用转换后指针所指的函数的效果依赖于实现。但是，如果转换后的指针被重新转换为原来的类型，则结果与原来的指针一致。

6.7 空类型 void

一个 void 对象的(不存在的)值不可以以任何方式使用，也不能被显式或隐式地转换为一非空类型。因为一个空表达式表示一个不存在的值，这样的表达式只可使用在不需要值的地方，例如作为一个表达式语句(见 9.2 节)或作为逗号运算符的左运算分量(见 7.18 节)。

可以通过强制类型转换将表达式转换为 void 类型。例如，在表达式语句中一个空的强制类型转换将丢掉函数调用的返回值。

- void 没有在本书的第 1 版中出现，但是从本书第 1 版出版后，就一直被广泛使用着。

6.8 指向空类型 void 的指针

指向任何对象的指针可以被转换为 void *类型而不会丢失信息。如果将结果再转换为初始指针类型，那么初始指针被恢复。与在 6.6 节中讨论的、一般需要显

式的强制转换的指针到指针的转换不同，指针可以被赋值为 void *类型指针．也可以赋值给 void *类型指针．并和 void *类型指针进行比较。

- 对 void *指针的解释是新增加的，以前 char *指针扮演通用指针的角色。ANSI 标准特别允许 void *类型指针和其他对象指针在赋值和关系表达中混用，而对其他的指针的混合使用则要求有显式的类型转换。

7 表达式

表达式运算符的优先级与本节中各小节的先后次序相同，即最高优先级的运算符最先介绍。例如，作为加法运算符+(见 7.7 节)的运算分量的表达式是在 7.1 节至 7.6 节定义的那些表达式。在每一小节中，各个运算符具有相同的优先级。在每个小节中也讨论了该节所讨论的运算符的左、右结合律。13 节给出的语法结合了运算符的优先级和结合律。

运算符的优先级和结合律是明确规定的，但是表达式的求值次序除少数例外情况外是没有定义的，尽管子表达式会有副作用。也就是说，除非一个运算符的定义保证了其运算分量以一特定顺序求值，否则具体的实现可以自由地选择任一求值次序，甚至可以交替求值。但是，每个运算符以与它所出现的表达式的句法分析兼容的方式将其运算分量产生的值结合起来。

- 这个规则取消了以前具有在数学上满足交换律和结合律的运算符的表达式可以任意排列的自由，但可能会在计算时不满足结合津。这个改变仅影响浮点数在接近其精确度限度的计算以及可能发生溢出的情况。

C 语言没有定义在表达式求值过程中的溢出、除法检查和其他异常的处理。大多数现有 C 语言的实现在进行有符号整数表达式的求值时以及在赋值时忽略溢出异常，但并不是所有实现都这样做。对除数为 0 和所有浮点异常的处理，不同的实现有不同的方式，有时候可以用非标准库函数进行调整。

7.1 指针生成

对于某类型 T，如果某表达式或子表达式的类型为“T 的数组”。那么此表达式的值是指向数组中第一个对象的指针，并且此表达式的类型被转换为“指向 T 的指针”。如果此表达式是一元运算符&、++、--或 sizeof 的运算分量．或是赋值类运算符或圆点运算符.的左运算分量．那么转换不会发生。类似地，类型为“返回 T 的函数”的表达式被转换为类型“指向返回 T 的函数的指针”，除非此表达式被用作&运算符的运算分量。

7.2 初等表达式

初等表达式是标识符、常量、字符串或带括号的表达式。

初等表达式
  标识符
  常量
  字符串
  (表达式)

一个标识符只要是按下面所讨论的方式适当说明的就是初等表达式。其类型由说明指定。如果一个标识符指定一个对象(见 5 节)且其类型是算术、结构、联合或指针类型，那么它是一个左值。

一个常量是一个初等表达式，其类型依赖于它的形式，见 2.5 节的讨论．

一个字符串字面值是一个初等表达式。它的初始类型是 char 数组类型(对于宽字符字符串，则为 wchar_t 数组类型)，但遵循 7.1 节给出的规则。它通常被修改

为指向 char 类型(wchar_t 类型)的指针，从而结果是指向字符串中第一个字符的指针。在一些初始化程序中不能进行这样的转换，详见 8.7 节。

用括号括起来的表达式是一个初等表达式，它的类型和值与无括号的表达式一致。此表达式是否是左值不受括号的影响。

7.3 后缀表达式

后缀表达式中的运算符遵循从左到右的结合规则。

后缀表达式:

初等表达式

后缀表达式 [*表达式*]

后缀表达式 (*变元表达式表*$_{opt}$)

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 --

变元表达式表:

赋值表达式

变元表达式表, *赋值表达式*

7.3.1 数组引用

带下标的数组由一个后缀表达式后跟一个括在方括号中的表达式来表示。这两个表达式中要有一个的类型必须为“指向 *T* 的指针”，其中 *T* 是某种类型；另一个表达式的类型必须为整数。下标表达式的类型为 *T*。表达式 E1[E2]，在定义上等同于*((E1)+(E2))。有关数组引用的更多讨论见 8.6.2 节。

7.3.2 函数调用

函数调用由一个后缀表达式(称为函数命名符)后跟由圆括号括起来的包含一个可能为空的、由逗号分隔的赋值表达式表组成，这些表达式就是函数的变元。如果后缀表达式包含一个在当前作用域中不存在的标识符，那么此标识符就被隐式地说明，就好像说明

```
extern int 标识符();
```

在包含此函数调用的最内层分程序中被给出一样。这个后缀表达式(在可能的隐式说明和指针生成之后，见 7.1 节)必须有类型“指向返回 *T* 的函数的指针”，其中 *T* 为某个类型，且函数调用的值的类型为 *T*。

- 在第 I 版中，这个类型被限制为函数类型，并且在通过指向函数的指针来调用此函数时必须有一个显式的*运算符，ANSI C 标准允许现存的一些编译程序用同样的语法来进行函数调用和通过指向函数的指针来进行函数调用。旧的语法仍然可用。

术语变元用来表示传递给函数调用的表达式，而术语参数则用来表示由函数定义或函数说明所接收的输入对象(或其标识符)，通常也可用术语“实际变元(参数)”和“形式变元(参数)”来区分它们。

在准备调用函数时，要对它的每个变元进行复制，所有的变元传递严格地按值进行。函数可能会改变其参数对象的值(即变元表达式值的拷贝)，但这个改变不会影响变元的值。然而，可以将指针作为变元传递，以使函数可以改变指针所指向的对象的值。

函数可以用两种方式说明；在新的方式中，参数的类型是作为函数类型的一部分显式指定的，这种说明称为函数原型。在旧的方式中，参数类型没有说明。函数说明在 8.6.3 节和 10.1 节讨论。

如果在一个函数说明的作用域中函数是以旧方式说明的.那么按以下方式对每个变元进行缺省变元提升:对每个整类型变元进行整提升(见 6.1 节)，将每个 float

类型的变元转换为 double 类型。如果调用时变元的数目与函数定义中参数的数目不等，或者某个变元的类型提升后与相应的参数类型不一致，那么函数调用的结果是未定义的。类型一致性依赖于函数定义是以新方式进行的还是以旧方式进行的。如果是旧方式的，那么类型一致性检查将在提升过的调用的变元类型和提升过的参数类型之间进行；如果定义是新方式的，那么提升过的变元类型必须与没有提升过的参数本身的类型一致。

如果在函数调用的作用域中函数说明是以新方式进行的，那么变元将被转换为函数原型中的相应参数类型，就像是赋值一样。变元数目必须与显式说明的参数数目相同，除非函数说明的参数表以省略号(, ...)结束。在这种情况下，变元的数目必须等于或超过参数的数目；其后无显式指定类型的参数与之对应的变元要进行缺省的变元提升，如前面段落中所述。如果函数定义是以旧方式进行的，那么在调用中可见的原型中的每个变元类型必须与相应函数定义中的参数类型一致(函数定义中的参数类型已进行过变元提升)。

- 这些规则特别复杂，因为必须要考虑到新旧方式函数的混合使用。应尽可能避免新旧方式混合使用。

变元的求值次序没有指定。不同的编译器的实现方式各不相同。然而，在进入函数前变元和函数命名符是完全求值的，包括所有的副作用。对任何函数都可以进行递归调用。

7.3.3 结构引用

一个后续表达式后跟一个圆点和一个标识符仍是一个后缀表达式。第一个运算分量表达式的类型必须是一个结构或联合，标识符必须是结构或联合的成员名字。结果值是结构或联合的指名的成员，其类型是对应成员的类型。如果第一个表达式是一个左值且第二个表达式的类型不是数组类型．那么整个表达式是一个左值。

一个后缀表达式后跟一个箭头(由-和>组成)和一个标识符仍是一个后缀表达式。第一个运算分量表达式必须是一个指向结构或联合的指针，标识符必须指名结构

或联合的一个成员，结果指向指针表达式所指向的结构或联合的指名成员，结果类型是对应成员的类型。如果成员类型不是数组类型那么整个表达式是一个左值。

这样，表达式 E1->MOS 与(*E1).MOS 等价。结构和联合将在 8.3 节讨论。

- 在本书的第 1 版中. 已经规定了在这样的表达式中，成员的名字必须属于后缀表达式所指定的结构或联合，但是这个规则并没有强制实行。最新的编译程序和 ANSI 强制规定这一点。

7.3.4 后缀加一与减一运算符

一个后续表达式后跟一个++或--运算符仍是一个后缀表达式。表达式的值是运算分量的值。当执行完此表达式后，运算分量的值加 l(++〕或减 l(--)。这个运算分量必须是一个左值。对运算分量的限制和运算细节的详细讨论见加法类运算符(7.7 节)和赋值类运算符(7.17 节)。其结果不是左值。

7.4 一元运算符

表达式中的一元运算符遵循从右到左的结合原则。

一元表达式:
后缀表达式
++ *一元表达式*
-- *一元表达式*
一元运算符 强制转换表达式
sizeof *一元表达式*
sizeof(*类型名*)

一元运算符: 任意一个
& * + - ~ !

7.4.1 前缀加一与减一运算符

以运算符++或--为前缀的一元表达式仍是一个一元表达式。运算分量将被加1(++)或减1(--)，整个表达式的值是经过加减以后的值。该运算分量必须是一个左值。对运算分量的限制和运算细节的讨论详见加法类运算符(见7.7节)和赋值类运算符(见7.17节)。结果不是左值。

7.4.2 地址运算符

一元运算符&用于计算运算分星的地址。该运算分量必须是一个既不能指向位字段，也不能指向说明为 register 的对象的左值或函数类型。结果值是一个指针，指向由左值所指的对象或函数。如果运算分量的类型为 *T*，那么结果的类型为指向 *T* 的指针。

7.4.3 间接寻址运算符

一元*运算符表示间接寻址，它返回其运算分量所指向的对象或函数。如果它的运算分量是一个指针且所指向的对象是算术、结构、联合或指针类型，那么它是一个左值。如果表达式的类型为“指向 *T* 的指针”，那么结果类型为 *T*。

7.4.4 一元加运算符

一元+运算符的运算分量必须是算术类型，其结果是运算分量的值。如果运算分量是整类型，那么就要进行整提升．结果类型是经过提升后的运算分量的类型。

- 一元+运算符是 ANSI C 标准新增加的，增加它是为了与一元-运算符对称。

7.4.5 一元减运算符

一元-运算符的运算分量必须是算术类型，结果为运算分量的负值。如果运算分量是整类型，那么就要进行整提升。有符号数的负值的计算方式为：将提升所得

到的类型中的最大值减去提升过的运算分量的值，然后加 1；但 0 的负值仍为 0。结果类型为提升过的运算分量的类型。

7.4.6 二进制求反运算符

一元~运算符的运算分量必须是整类型，结果为运算分量的二进制反码。在运算过程中要对运算分量进行整提升。如果运算分量是无符号类型的，那么结果是通过由提升后的类型的最大值减去运算分量的值得到的值。如果运算分量是有符号的，那么结果的计算方式为：将提升后的运算分量转换为相应的无符号类型，进行二进制求反运算，再将结果转换为有符号类型。结果的类型为提升后的运算分量的类型。

7.4.7 逻辑非运算符

运算符!的运算分星必须是算术类型或是一个指针。如果运算分量等于 0，那么结果为 1，否则结果为 0。结果类型为 int。

7.4.8 sizeof 运算符

sizeof 运算符用于求存储其运算分量类型的对象所需要的字节数。运算分量或者为一个未求值的表达式，或者为一个由括号括起的类型名字。当 sizeof 被用于 char 类型时，其值为 1。当用于数组时，其值为数组中字节的总数。当用于结构或联合时，结果是对象中的字节数，包括任何使对象平铺为数组所需要的填充空间：有 *n* 个元素的数组的大小是一个元素大小的 *n* 倍。此运算符不能用于函数类型和不完全类型的运算分量，也不能用于位字段。结果是一个无符号整形常量，具体的类型由实现定义。在标准头文件<stddef.h>(见 C 标准库)中，这一类型被定义为 size_t 类型。

7.5 强制转换

以括号括起来的类型名开头的一元表达式将导致表达式的值被转换为指名的类型。

强制转换表达式：
一元表达式
(类型名字) 强制转换表达式

这个结构称为强制转换。类型名字将在 8.8 节描述。转换的结果已在 6 节讨论过。包含强制转换的表达式不是左值。

7.6 乘法类运算符

乘法类运算符*、/和%遵循从左到右的结合规则。

乘法类表达式：
强制转换表达式
*乘法类表达式 * 强制转换表达式*
乘法类表达式 / 强制转换表达式
乘法类表达式 % 强制转换表达式

*和/的运算分量必须为算术类型，%的运算分量必须为整类型。对这些运算分量要进行常规算术转换，并预测结果类型。

二元运算符*表示乘法。

二元运算符/求得第一个运算分量被第二个运算分量除所得的商，而运算符%求得相应的余数。如果第二个运算分量为 0，那么结果没有定义。其他情况下(a/b)*b + a%b 等于 a 永远成立。如果两个运算分量均为非负，那么余数是非负的且小于除数，否则，仅可保证余数的绝对值小于除数的绝对值。

7.7 加法类运算符

加法类运算符+和-遵循从左到右的结合规则。如果运算分量有算术类型，那么要进行常规的算术转换。对于每个运算符有更多的可能类型。

加法类表达式:
乘法类表达式
加法类表达式 + *乘法类表达式*
加法类表达式 - *乘法类表达式*

运算符+作用的结果为两个运算分量的和。数组中指向一个对象的指针可以和一个任何整类型的值相加，后者将通过乘以所指对象的大小被转换为地址偏移量。相加的和是一个指针，它与初始指针有相同的类型，并指向同一数组中的另一个对象，此对象与初始对象之间有合适的偏移量。因此，如果 P 是一个指向数组中某个对象的指针，那么表达式 P+1 是指向数组中下一个对象的指针。如果相加的和所指定的指针不在数组的范围，且不是数组末尾的第一个位置，那么结果没有定义的。

- 允许指针指向救组的末尾是 ANSI C 新增加的，它使得我们可以像通常一样对数组元素建立循环。

运算符-作用的结果是两个运算分量的差值。可以从一个指针减去一个任意整类型的值，该运算的转换规则和条件与加法相同。

如果指向同一类型的两个指针相减，那么结果是一个有符号整类型数，表示所指向的对象之间的偏移量。相邻的对象之间的偏移量为 1。结果的类型依赖于具体的实现，但在标准头文件<stddef.h>中定义为 ptrdiff_t。只有当指针指向的对象属于同一数组时，差值才有意义。然而，如果 P 指向数组的最后一个成员，那么(P+1)-P 的值为 1。

7.8 移位运算符

移位运算符<<和>>遵循从左到右的结合规则。每个运算符的运算分量必须为整类型，并且遵循整提升原则。结果的类型是提升过的左运算分量的类型。如果右运算分量为负值，或者大于等于左运算分量的位数，那么结果没有定义。

移位表达式:

加法类表达式

移位表达式 << *加法类表达式*

移位表达式 >> *加法类表达式*

E1<<E2 的值为 E1(按位模式解释)左移 E2 个位。如果不发生溢出，此值等同于 E1 乘以 2^{E2}。E1>>E2 的值 E1 右移 E2 个位。如果 E1 为无符号数或为非负值，那么右移等同于 E1 除以 2^{E2}。其他情况结果由具体实现定义。

7.9 关系运算符

关系运算符遵循从左到右的结合规则，但这个规则没有多大作用。a<b<c 在语法分析时被解释为(a<b)<c，并且(a<b)的值为 0 或 1。

关系表达式:

移位表达式

关系表达式 < *移位表达式*

关系表达式 > *移位表达式*

关系表达式 <= *移位表达式*

关系表达式 >= *移位表达式*

当所指定的关系为假时，运算符<(小于)、>(大于)、<=(小于等于)和>=(大于等于)均返回 0；当关系为真时，它们均返回 1。结果的类型为 int 类型。如果运算分量为算术类型，那么要进行通常的类型转换。可以将指向同一类型的对象的指针进行比较(忽略任何限定符)，其结果依赖于所指对象在地址空间中的相对位置。指针比较只对相同对象的部分有定义：如果两个指针指向同一个简单对象，

那么它们的值相等；如果指针指向同一个结构的不同成员，那么指向后说明的成员的指针有较大的值；如果指针指向同一个联合的不同成员，那么它们有相同的值；如果指针指向一个数组的不同成员，那么它们的比较值等于对应的下标的比较值。如果指针 P 指向数组的最后一个成员，那么 P+1 比 P 大，尽管 P+1 已指向数组的界外。其他情况指针的比较没有定义。

- 这些规则允许指向同一个结构或联合的不同成员的指针进行比较，从而放宽了第 1 版所述的限制。这些规则还使得与指向正好超出数组末尾的指针的比较合法化。

7.10 相等类运算符

相等类运算符:

关系运算符

相等类运算符 == *关系运算符*

相等类运算符 != *关系运算符*

运算符==(等于)和!=(不等于)与关系运算符相似，但优先级不同。(任何时候只要 a<b 与 c<d 有相同的真值，那么 a<b == c<d 的值就为 1。)

相等类运算符与关系运算符有相同的规则，但这类运算符还允许做如下比较：指针可以与值为 0 的常量表达式或指向 void 的指针进行比较。参见 6.6 节。

7.11 按位与运算符

按位与表达式:

相等类表达式

按位与表达式 & *相等类表达式*

在进行按位与运算时要进行通常的算术转换，结果为运算分量的按位与。该运算符仅适用于整类型运算分量。

7.12 按位异或运算符

按位异或表达式：

按位与表达式

按位异或表达式 ^ *按位与表达式*

在进行按位异或运算时要进行通常的算术转换，结果为运算分量的按位异或。该运算符仅适用于整类型运算分量。

7.13 按位或运算符

按位或表达式：

按位异或表达式

按位或表达式 | *按位异或表达式*

在进行按位或运算时要进行通常的算术转换．结果为运算分量的按位或。该运算符仅适用于整类型运算分量。

7.14 逻辑与运算符

逻辑与表达式：

按位或表达式

逻辑与表达式 && *按位或表达式*

运算符&&遵循从左到右的结合规则。如果两个运算分量都不为 0，那么它返回 1，否则返回 0。与&不同，&&确保从左到右的求值次序：首先计算第一个运算分量，包括所有的副作用，如果为 0，那么整个表达式的值为 0。否则计算右运算分量，如果为 0，那么整个表达式的值为 0；否则为 1。

两个运算分量不需是同一类型的，但是每一个运算分量必须为算术类型或者是指针。结果为 int 类型。

7.15 逻辑或运算符

逻辑或表达式：

逻辑与表达式

逻辑或表达式 || *逻辑与表达式*

运算符||遵循从左到右的结合规则。如果有一个运算分量不为 0，那么它返回 1；否则返回 0。与|不同，||确保从左到右的求值次序：首先计算第一个运算分量，包括所有的副作用，如果不为 0，那么整个表达式的值为 1。否则计算右运算分量，如果不为 0，那么整个表达式的值为 1；否则为 0。

两个运算分量不需是同一类型的，但是每一个运算分量必须为算术类型或者是指针。结果为 int 类型。

7.16 条件运算符

条件表达式：

逻辑或表达式

逻辑或表达式 ? *表达式* : *条件表达式*

首先计算第一个表达式(包括所有的副作用)，如果该表达式的值不为 0，那么结果为第二个表达式的值，否则结果为第三个表达式的值。第二个和第三个运算分量中只有一个会被计算到。如果第二个和第三个运算分量为算术类型，那么要进行通常的算术转换以使它们有一个共同的类型，这个类型就是结果的类型。如果它们都是 void 类型，或是同一类型的结构或联合，或是指向同一类型的对象的指针，那么结果的类型为共同的类型。如果其中一个运算分量是指针，而另一个是常量 0，那么 0 被转换为指针类型，并且结果为指针类型。如果一个运算分量为指向 void 的指针，而另一个为普通指针，那么另一个指针被转换为指向 void 的指针，并且这是结果的类型。

在比较指针的类型时，指针所指对象的类型的任何类型限定符(见 8.2 节)将被忽略，但这些限定符都可被结果的类型继承。

7.17 赋值表达式

有几个赋值运算符，它们均从右到左结合。

赋值表达式:
  条件表达式
  一元表达式 赋值运算符 赋值表达式

赋值运算符: 任意一个
  = *= /= %= += -= <<= >>= &= ^= |=

所有这些运算符要求左运算分量为左值，并且此左值可以修改，它不可以是数组、不完全类型或函数。同时其类型不能有 const 限定符，如果它是结构或联合，那么它的任意一个成员或递归子成员不能有 const 限定符。赋值表达式的类型是其左运算分量的类型，值是赋值发生后存储在左运算分量中的值。

在用=的简单赋值中，用表达式的值替换左值所指向的对象的值。下面几个条件中必须有一个条件成立：或者两个运算分量均为算术类型，在此情况下右运算分量的类型通过赋值转换为左运算分量的类型；或者两个运算分量为同一类型的结构或联合；或者一个运算分量是指针，另一个运算分量是指向 void 的指针；或者左运算分量是指针．右运算分量是值为 0 的常量表达式；或者两个运算分量均为指向同一类型的函数或对象的指针，除了右运算分量可能没有 const 或 volatile 说明。

形式为 E1 *op*= E2 的表达式等价于 E1 = E1 *op* (E2)，唯一的区别是前者 E1 只求值一次。

7.18 逗号运算符

表达式：

赋值类表达式

表达式 ，*赋值类表达式*

由逗号分开的一对表达式的求值次序为从左到右，并且左表达式的值被丢弃掉。结果的类型和值就是右运算分量的类型和值。在开始计算右运算分量以前，计算左运算分量所带来的副作用将被完成。在逗号有特殊含义的上下文中，如在函数变元表(见 7.3.2 节)和初始化符表(8.7 节)中，所要求的语法单元是一个赋值表达式。这样逗号运算符仅出现在一个圆括号组中，例如，函数调用：

```
f(a, (t=3, t+2), c)
```

包含有三个变元，其中第二个变元的值为 5。

7.19 常量表达式

从语法上看，常量表达式是局限于运算符的某一个子集的表达式：

常量表达式:
条件表达式

一些上下文要求表达式的值为常量，例如：在 switch 语句中的 case 后面、作为数组的边界和位字段的长度、作为枚举常量的值、用在初始化符中以及在某些预处理表达式中。

除非作为 sizeof 的运算分量，否则常量表达式中可以不包含赋值、增一或减一运算符、函数调用或逗号运算符。如果要求常量表达式为整类型，那么它的运算分量必须由整数、枚举、字符和浮点常量组成。强制类型转换必须指定为整类型，任何浮点常量都被强制转换为整数。此规则必须将数组、间接指向、取地址和结构成员运算排除在外(但是，sizeof 可以带任何运算分量)。

初始化符中的常量表达式可以有更大的范围。运算分量可以是任意类型的常量，一元运算符&可以用于外部和静态对象以及以常量表达式为下标的外部和静态数组。一元运算符&也可以在出现无下标的数组或函数时被隐式地应用。初始化符计算的值必须或者为一个常量，或者为已说明的外部或静态对象的地址与一个常量的和或差。

允许在#if 后面的整类型常量的范围较小，不可以是 sizeof 表达式、枚举常量和强制转换。参见 12.5 节。

8 说明

说明用于指定每个标识符的含义，它们并不需要保留与每个标识符相关的存储空间。保留存储空间的说明称为定义。说明的形式为：

说明:

说明区分符 初始化说明符表$_{opt}$;

初始化说明符表中的说明符包含了被说明的标识符；说明区分符由一系列类型和存储类区分符组成。

说明区分符:

存储类区分符 说明区分符$_{opt}$

类型区分符 说明区分符$_{opt}$

类型限定符 说明区分符$_{opt}$

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 = *初始化符*

说明符将在稍后讨论(见 8.5 节)，它们包含了被说明的名字。一个说明必须包含至少一个说明符，或者其类型区分符必须说明一个结构标记、一个联合标记或枚举的成员。不允许空的说明。

8.1 存储类区分符

存储类区分符为：

存储类区分符:

auto

register

static

extern

typedef

关于存储类的含义已在 4.4 节讨论过。

区分符 auto 和 register 使得被说明的对象有自动存储类，它们仅可用在函数中。这种说明也起着定义的作用，并预留存储空间。带有 register 区分符的说明等价于带有 auto 区分符的说明，不同的是前者暗示了被说明的对象将被频繁地访问。只有很少的对象被真正放在寄存器中，并且只有特定类型才可以。所受的限制依赖于具体的实现。然而，如果一个对象被说明为 register，那么就不能对它应用一元运算符&，不论是显式地还是隐式地应用。

- 计算一个被说明为 register 而实际为 auto 的对象的地址是非法的，这是一个新的规定。

区分符 static 使得被说明的对象具有静态存储类，可以用在函数内或函数外。在函数内，该区分符使得存储空间被分配，起着定义的作用。对于在函数外的效果，参见 11.2 节。

用在函数内的 extern 说明用于指明被说明对象的存储空间在别处定义。对于在函数外的效果，见 11.2 节。

typedef 区分符没有预留存储空间，之所以称之为存储类区分符，只是为了语法描述上的方便。我们将在 8.9 节讨论它。

在一个说明中最多只能有一个存储类区分符，如果没有存储类区分符被指定，那么就使用如下规则：在函数内说明的对象被认为具有 auto 存储类；在函数内说明的函数被认为具有 extern 存储类；在函数外说明的对象与函数被认为具有带外部连接的静态存储类。参见 10 节至 11 节。

8.2 类型区分符

类型区分符定义如下：

类型区分符：

void

char

short

int

long

float

double

signed

unsigned

结构或联合区分符

枚举区分符

类型定义名字

在 long 和 short 这两个类型区分符中最多有一个可同时与 int 一起说明;在 int 缺省时含义也是一样的。long 可与 double 一起说明。signed 和 unsigned 这两个类型区分符中最多有一个可同时与 int、int 的 short 和 long 的变种或 char 一起指定。signed 和 unsigned 可以单独出现，这种情况下默认为 int。signed 区分符对强制 char 对象带符号位是非常有用的;对其他整类型也允许带 signed，但这是多余的。

除了上面这些情况，在一个说明中至多只能给出一个类型区分符。如果说明中没有类型区分符，则默认为 int。

类型也可以用限定符限定．以指定被说明对象的特殊性质。

类型限定符:

const

volatile

类型限定符可与任何类型区分符一起出现。const 对象可被初始化，但随后不能再被赋值。volatile 对象没有独立于实现的语义。

- const 和 volatile 性质是 ANSI 标准新增如的。const 的作用是声明可以放在只请存储器中的对象，并可能为优化提供机会。volatile 的作用是使实现屏蔽可能的优化。例如，对于具有内存映像输入/输出的机器，指向设备寄存器的指针可被说明为指向 volatile 的指针，目的是防止编译程序通过指针明显删除多余的引用。除了需要诊断改变 const 对象的明显企图，一个编译程序可能会忽略这些限定符。

8.3 结构和联合说明

结构是由不同类型的有名成员序列组成的对象。联合也是对象，在不同时刻，它含有许多不同类型成员中的任意一个。结构和联合区分符具有相同形式。

结构或联合区分符:

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

结构说明表是对结构或联合成员的说明序列：

结构说明表:

结构说明

结构说明表 结构说明

结构说明:

区分符限定符表 *结构说明符表*;

区分符限定符表:

类型区分符 *区分符限定符表*$_{opt}$

类型限定符 *区分符限定符表*$_{opt}$

结构说明符表:

结构说明符

结构说明符表 , *结构说明符*

通常，一个结构说明符就是对结构或联合成员的说明符。结构成员也可能包含指定的位数，这样的成员也叫做位字段，或称为字段，共长度通过跟在说明符后的冒号之后的常量表达式来指定。

结构说明符:

说明符

说明符$_{opt}$: *常量表达式*

一个形如

结构或联合标识符 { *结构说明表* }

的类型区分符说明了其中的标识符是由结构说明表指定的结构或联合的标记。我们可以在同一作用域或内层作用域内的后续说明中通过在不包含结构说明表的区分符中使用标记来表示同一类型：

结构或联合标识符

如果一个区分符中只有标记而无结构说明表并且没有说明标记，那么该区分符说明了一个不完整类型。具有不完整结构或联合类型的对象可被上下文引用，只要

该处不需要知道它们的大小。例如，在指定一个指针或用类型定义新建一个类型名字的说明(而不是定义)中，都可引用不完整类型，其余情况则不允许。在其后如果具有该标记的区分符再次出现并包含了结构说明表，那么该类型就成为完整类型。即使是在包含结构说明表的区分符中，在该结构说明表内所说明的结构或联合类型也是不完整的，一直到花括号}终止该区分符时，所说明的类型才成为完整类型。

结构中不能包含不完整类型的成员。因此，不能说明包含自身实例的结构或联合。然而，除了可以命名结构或联合类型外，标记还允许定义自引用结构；由于可以说明指向不完整类型的指针，结构和联合可包含指向自身实例的指针。

一个非常特殊的规则适用于如下形式的说明：

结构或联合标识符;

它用于说明结构或联合，但没有说明表和说明符。即使所说明的标识符是在外层作用域已说明过的结构或联合的标记(参见 11.1 节)，该说明仍使得该标识符成为在当前作用域内的一个新的不完整类型的结构或联合的标记。

- 这是 ANSI 中的一个新的比较难理解的规则。它旨在处理在内层作用域说明的相互递归调用结构，但这些结构的标记可能已在外层作用域中说明。

具有结构说明表而无标记的结构或联合区分符用于建立一个唯一的类型，它只能被它所在的说明直接引用。

成员和标记的名字不会相互冲突，也不会与普通变量冲突。一个成员名字不能在同一结构或联合中出现两次，但相同的成员名字可在不同的结构或联合中使用。

- 在本书的第 1 版中，结构或联合的成员名与其父辈无关联。然而在 ANSI 标准制定前，这种关联在编译程序中早已普遍。

一个结构或联合的非位字段成员可以具有任意对象类型。一个位字段成员(它无需说明符，因而可无名)具有类型 int、unsigned int 或 signed int，并且被解

释成用位表示其长度的整类型对象。int 类型位字段是否要当做有符号数则依赖于实现。结构的相邻位字段成员以依赖于实现的方式被一起放到依赖于实现的存储单元中去。如果在另一位字段之后的某一位字段无法全部存入已被前面位字段部分填充的存储单元中，那么可将它分成两部分存入相邻的存储单元，或者可以填充该单元。我们可以用宽度为 0 的无名位字段来强制做这种填充，从而下一位字段将从下一分配单元的边界开始存储。

- 在处理位字段方面，ANSI 标准比第 1 版更加依赖于实现。为了将位字段存储为无条件地依赖于实现，应当参照这一语言规则。具有位字段的结构可被方便地用来节省存储空间(代价是增加了指令空间和访问字段的时间)。同时，带位字段的结构也可被用来描述在位层次上的存储布局，不过这是一个不方便的方法。在第二种情况下，必须了解局部实现的规则。

结构成员的地址值技它们说明的顺序递增。一个非位字段结构成员根据其类型在地址边界上对齐，因而，在结构中可能会存在无名空穴。若指向一结构的指针被强制转换成指向该结构第一个成员的指针类型，那么结果将指示第一个成员。

联合可以被看成是结构，其所有成员起始偏移量都为 0，并且它的大小足以容纳它的任一成员。任一时刻它至多只能存储其中一个成员。如果指向某一联合的指针被强制转换成指向一个成员的指针类型，那么结果将指向那个成员。

结构说明的一个简单例子如下：

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
}
```

该结构包含一个具有 20 个字符的数组、一个整数以及两个指向类似结构的指针。一旦给出这样的说明，说明

```
struct tnode s, *sp;
```

就将声明 s 为所给定类型的结构，sp 为所结定类型结构的指针。有了这些说明，表达式

```
sp->count
```

就表示由 sp 所指向结构的 count 字段，而

```
s.left
```

就表示结构 s 的左子树指针；而

```
s.right->tword[0]
```

就表示 s 右子树中 tword 成员的第一个字符。

我们通常无法检查联合的某一成员，除非已用该成员给联合赋值。然而，有一个特殊的情况可以简化联合的使用：如果一个联合包含有共享一个公共初始序列的若干结构，并且该联合当前包含有这些结构中的某一个，那么引用这些结构中任一结构的公共初始部分是允许的。例如，下述这段程序是合法的：

```
union {
    struct {
        int type;
    } n;
    struct {
        int type;
        int intnode;
    } ni;
```

```
    struct {
        int type;
        float floatnode;
    } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
    ... sin(u.nf.floatnode) ...
```

8.4 枚举

枚举类型是这样一种特殊的类型，它的值包含在一个有名常量集合中，这些常量就叫做枚举符(枚举常量)。枚举区分符的形式借鉴了结构和联合区分符的形式。

枚举区分符：

enum *枚举标识符*$_{opt}$ { *枚举符表* }

enum *枚举标识符*

枚举符表：

枚举符

枚举符表 , *枚举符*

枚举符：

标识符

标识符 = *常量表达式*

枚举符表中的标识符被说明为 int 类型的常量，它们可以出现在需要常量的任何地方。如果没有带有=的枚举符出现，那么相应常量值从 0 开始，日当从左至右读取说明时枚举常量值加 l。带有=的枚举符对联系的标识符给出所指定的值，其后的标识符从所指定值开始继续递增。

在同一作用域内各个枚举符的名宁必须互不相同，也不能和普通变量名字相同，但其值不必是不同的。

枚举区分符中标识符的作用与结构区分符中结构标记的作用类似。它命名了一个特定的枚举类型。除了不存在不完整枚举类型外，枚举区分符在具有或缺少标记和枚举符表时的规则与结构或联合的那些规则相同。无枚举符表的枚举区分符标记必须指向作用域内的一个具有枚举符表的区分符。

- 相对于本书第 1 版，枚举类型是新概念，但作为 C 语言的一部分已有好些年了。

8.5 说明符

说明符的语法如下：

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:

* *类型限定符表*$_{opt}$

* *类型限定符表*$_{opt}$ *指针*

类型限定符表:

类型限定符

类型限定符表 类型限定符

说明符的结构与间接、函数及数组表达式类似，组合方式也相同。

8.6 说明符的含义

说明符表出现在类型和存储类区分符序列之后。每个说明符说明一个唯一的主标识符，该标识符是直接说明符产生式的第一个备选。存储类区分符可直接作用于该标识符，但其类型依赖于其说明符的形式。当说明符的标识符出现在具有与该说明符相同形式的表达式中时，该说明符就被看做是断言。

若只考虑说明区分符(参见 8.2 节)的类型部分及特定的说明符，那么一个说明具有形式“T D”，其中 T 是类型，D 是说明符。在不同形式的说明中，标识符的类型可用这个概念来归纳描述。

在说明 T D 中(其中 D 是不加任何修饰的标识符)，D 的类型是 T。

在说明 T D 中，若 D 具有

(D1)

的形式，则 D1 中标识符的类型与 D 中标识符的类型相同。圆括号不改变类型，但可改变复杂说明符的绑定。

8.6.1 指针说明符

在说明 T D 中，如果 D 具有如下形式：

* *类型限定符表*$_{opt}$ D1

且在说明 T D1 中的标识符的类型是“类型修饰符 T”，那么 D 的标识符的类型是“类型修饰符类型限定符表指向 T 的指针”。星号*后的限定符用于指针本身，而不是指针所指向的对象。

例如，考虑如下说明

```
int *ap[];
```

这里 ap[]起到 D1 的作用，说明“int ap[]”使得 ap 的类型为“整数数组类型”，类型限定符表为空，且类型修饰符为“的数组”。因此. 实际说明使 ap 具有“指向 int 的指针数组”类型。

作为另一个例子，说明

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

说明了一个整数 i 和一个指向整数的指针 pi。常量指针 cpi 的值不能修改，该指针总是指向同一位置，尽管它所指之处的值可以改变。整数 ci 是常量，也不能被修改(尽管可以初始化，如本例中所示)。pci 类型是“指向 const int 的指针”，pci 本身可以被改变而指向另一处，但它所指之处的值不能通过 pci 赋值来改变。

8.6.2 数组说明符

在说明 T D 中，如果 D 具有形式

D1 [*常量表达式*$_{opt}$]

且在说明 T D1 中标识符的类型是“类型修饰符 T”，那么 D 的标识符类型是“类型修饰符 T 的数组”。如果存在常量表达式，则该常量表达式必须为整类型且值大于 0。若数组缺少用于指定上界的常量表达式，那么该数组类型是不完整类型。

数组的成份类型可以是算术类型、指针类型、结构类型或联合类型，也可以是另一个数组(以生成多维数组)。数组成份的类型必须是完整类型，绝不能是不完整类型的数组或结构。这就意味着，对于多维数组，只有第一维可以缺省。一个不完整数组类型的对象的类型，可以通过对该类型的另一个完整说明(见 10.2 节)或通过对其初始化(见 8.7 节)来使它完整。例如，

```
float fa[17], *afp[17];
```

说明了一个浮点数组和一个指向浮点数的指针数组，而

```
static int x3d[3][5][7];
```

则说明了一个静态的三维整型数组，其大小为 3 X 5 X 7。具体而言，x3d 是一个由三个项组成的数组，每一个项都是由 5 个数组组成的数组，5 个数组中的每一个又都是由 7 个整数组成的数组。x3d、x3d[i]、x3d[i][j]与 x3d[i][j][k]都可以合理地出现在一个表达式中，前三者是数组类型，最后一个是整数类型。更明确地说，x3d[i][j]是一个有 7 个整数元素的数组，x3d[i]是含有 5 个由 7 个整数构成的数组的数组。

数组下标运算规定 E1[E2]与*(E1+E2)等同。因此，尽管看上去不对称，但下标运算是可交换的运算。根据作用于+和数组的转换规则(参见 6.6 节、7.1 节与 7.7 节)，若 E1 是数组且 E2 是整数，那么 E1[E2]代表 E1 的第 E2 个成员。

在本例中，x3d[i][j][k]等价于*(x3d[i][j] + k)。第一个子表达式 x3d[i][j]按 7.1 节所述转换成类型“指向整数数组的指针”，而根据 7.7 节中所述，加法运算涉及乘以整数大小的操作。它所遵循的规则是：数组按行存储(最后一维下标变动最快)，且说明中的第一维下标用于决定数组所需的存储区大小，但第一维下标在下标计算的其他方面不起作用。

8.6.3 函数说明符

在一个新方式的函数说明 T D 中，如果 D 具有形式

D1 (*参数类型表*)

并且在说明 T D1 中标识符的类型是“类型修饰符 T”，那么 D 的标识符的类型是“具有返回 T 的变元参数类型表的类型修饰符函数”。

参数的语法定义为：

参数类型表:
  参数表
  参数表 , ...

参数表:
  参数说明
  参数表 , *参数说明*

参数说明:
  说明区分符 说明符
  说明区分符 抽象说明符$_{opt}$

在这个新的说明中，参数表说明了参数的类型。作为一个持殊情况，无参数的新方式函数的说明符具有一个参数类型表，该表仅包含关键字 void。苦参数类型表以省略号“，...”结束，那么该函数接受的参数个数可比显式描述的参数个数多，参见 7.3.2 节。

若参数类型是数组或函数，则按照参数转换规则(见 10.1 节)将它们转换为指针。在参数的说明区分符中唯一允许的存储类区分符是 register，除非函数定义以函数说明符为首，否则该存储类说明符将被忽略。类似地，如果参数说明的说明符中包含标识符，且函数定义不以该函数说明符为首，那么该标识符超出了作用域。不涉及标识符的抽象说明符将在 8.8 节讨论。

在旧方式的函数说明 T D 中，如果 D 具有形式

D1(*标识符表*$_{opt}$)

并且在说明 T D1 中的标识符的类型是“类型修饰符 T”，那么 D 的标识符类型为“未指定返回 T 的变元的类型修饰符函数”。参数(若存在的话)具有形式

标识符表:
标识符
标识符表 , *标识符*

在旧方式说明符中，除非在函数定义的首部使用了说明符，否则标识符表必须缺省(参见 10.1 节)。说明不提供有关参数类型的信息。

例如，说明

```
int f(), *fpi(), (*pfi)();
```

说明了一个返回整数类型的函数 f、一个返回指向整数的指针的函数 fpi 以及一个指向返回整数的函数的指针 pfi。它们都未说明参数类型，因此都属旧方式的说明。

在新方式的说明

```
int strcpy(char *dest, const char *source), rand(void);
```

中，strcpy 是返回整数类型的函数，具有两个参数，第一个是字符指针，第二个是指向常量字符的指针。参数名字即是有效注解。第二个函数 rand 不带参数且返回类型 int。

- 到目前为止，带参数原型的函数说明符是由 ANSI 标准引入的最主要的语言变化。它们优于第 1 版中的“旧方式”说明符，因为它们提供了函数调用时的错误检测和参数强制转换，不过代价是在引入时带来了混乱和迷惑，而且还必须兼容这两种形式。为了兼容，不得不在语法上做一些手脚，即采用 void 作为无参数新方式函数的显式标记。
- 带有变长变元表的函数采用的省略号“, ...”也是新的标准，它和标准头文件<stdarg.h>中的宏共同形式化了一个机制，该机制在第 1 版中虽被禁止但可非正式地接受。
- 这些表示法来自 C++。

8.7 初始化

在说明一个对象时，对象的初始化说明符可为其指定一个初始值。初始化符紧随=之后，它或是一个表达式，或是一列嵌套在花括号中的初始化符。一列初始化符可以以逗号结束，这使得格式简洁优美。

初始化符:

赋值表达式

{ *初始化符表* }

{ *初始化符表* , }

初始化符表:

初始化符

初始化符表 , *初始化符*

静态对象或数组的初始化符中的所有表达式必须是如 7.19 节中所述的常量表达式。如果初始化符是用花括号括起来的初始化符表，那么 auto 或 register 对象或数组的初始化符中的表达式也同样必须是常量表达式。然而，若自动对象的初始化符是一个单一表达式，那么它不必是常量表达式，但必须符合对象赋值的类型要求。

- 第 1 版不支持自动结构、联合或数组的初始化。而 ANSI 标准是允许的，但只能通过常量结构，除非可用简单表达式表示初始化符。

一个未显式初始化的静态对象将被隐式初始化，它(或它的成员)被赋以常量 0。未显式初始化的自动对象的初始值是没有定义的。

指针或算术类型对象的初始化符是一个单一表达式，但可能括在花括号中。该表达式将赋值给相关对象。

结构的初始化符可以是具有相同类型的表达式，也可以是按其成员次序括在花括号中的初始化符表。无名的位字段成员被忽略，故不被初始化。若表中初始化符的数目比结构的成员数少，那么尾随的剩余结构成员将被初始化为 0。初始化符的数目不能比成员数多。

数组的初始化符是括在花括号中的数组成员的初始化符。若数组大小未知，那么初始化符的数目将决定数组的大小，从而使得数组类型变得完整。若数组大小固

定，则初始化符的数目不能超过数组成员的数目。若初始化符的数目比数组成员数目少，则尾随的剩余数组成员将被初始化为 0。

作为一个特殊情况，字符数组可用字符串字面值初始化。字符串的各个字符依次初始化数组中的相应成员。类似地，宽字符字面值(参见 2.6 节)可初始化 wchar_t 类型的数组。若数组大小未知，则数组大小将由字符串中字符数(包括结尾空字符)决定。若数组大小固定，则不计结尾空字符，字符串中字符数不能超过数组大小。

联合的初始化符可以是具有相同类型的表达式，也可以是括在花括号中的联合的第一成员的初始化符。

- 第 1 版不允许对联合初始化。“第一成员”规则显得有点笨拙，但没有新语法很难进行概括。除了允许联合至少以一个原始方式被显式初始化，这一 ANSI 规则还给出了非显式初始化的静态联合的精确语义。

聚集是一个结构或数组。若一个聚集包含聚集类型的成员，则初始化时将递归使用初始化规则。如下情况将在初始化中省略括号：若聚集的成员也是一个聚集，且该成员的初始化符以左花括号开头，那么后继部分用逗号隔开的初始化符表将初始化子聚集的成员。初始化符的数目不允许超过成员的数目。然而，如果子聚集的初始化符不以左花括号开头，那么表中只有足够元素被认为是子聚集的成员．任何剩余成员将用来初始化下一个子聚集所在的聚集的成员。

例如，

```
int x[] = { 1, 3, 5 };
```

说明并初始化 x 为一个只有三个成员的一维数组，闪为数组未指定大小且有三个初始化符。

```
float y[4][3] = {
    { 1, 3, 5 },
```

```
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

是一个完全用花括号隔开的初始化：1、3 和 5 这三个数初始化数组 y[0]的第一行，即 y[0][0]、y[0][1]和 y[0][2]。同样，另两行初始化 y[1]和 y[2]。因初始化符的数目不够，因此元素 y[3]被初始化为 0。确切地说，如下说明能得到相同的结果：

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

y 的初始化符以左花括号开始，但 y[0]的初始化符与其不同，因此 y[0]的初始化使用了表中三个元素。同理．y[1]使用了随后的三个，接着 y[2]用了最后三个。另外，

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

用于初始化 y 的第一列(将 y 当做一个二维数组)，其余的默认例始化为 0。

最后，

```
char msg[] = "Syntax error on line %s\n";
```

说明了一个字符数组，对其元素用了一个字符串字面值进行初始化，该数组大小包括终结空字符。

8.8 类型名字

在有些上下文中(例如，在需要显式指定强制类型转换时、在函数说明符中说明参数类型时以及在作为 sizeof 的变元时)．必须提供数据类型的名字。这一目的可以使用类型名字来达到。在语法上，类型名字就是该类型的某一对象的说明，只是省略了该对象的名字。

类型名:

区分符限定符表 *抽象说明符*$_{opt}$

抽象说明符:

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*$_{opt}$)

如果该结构是说明中的一个说明符，那么就有可能唯一确定标识符在抽象说明符中出现的位置。所指名的类型从而就与假想标识符的类型相同。例如，

```
int
int *
int *[3]
int (*)[]
```

```
int *()
int (*[])(void)
```

等 6 个说明分别命名了类型“整数”、“指向整数的指针”、“包含 3 个指向整数的指针的数组”、“指向未指定整数个数的数组的指针”、“未指定参数的函数，函数返回指向整数的指针”以及“大小未指定的数组，该数组的指针指向无参函数，每个参数返回一个整数”。

8.9 类型定义

如果一个说明的存储类区分符是 typedef，那么该说明并不说明对象，而是定义用于命名类型的标识符。这些标识符就称为类型定义名字。

类型定义名字:
  标识符

类型定义说明以通常的方式把一个类型指派给其说明符中的每个名字(参见 8.6 节)。此后，每个这样的类型定义名字在语法上就等同于相关类型的类型区分符关键字。

例如，在定义

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complex;
```

之后，结构

```
Blockno b;
extern Blockptr bp;
```

```
Complex z, *zp;
```

都是合法的说明。b 的类型是 long 类型，bp 的类型是“指向 long 类型的指针”，而 z 的类型是所指定的结构，zp 的类型是指向该结构的指针。

typedef 类型定义并不引入新类型，它只是可以用另一种方式说明的类型的同义词。在本例中，b 具有与其他任何 long 类型对象相同的类型。

typedef 类型定义名字可在内层作用域内重新说明，但必须给出一个类型区分符的非空集合。例如，

```
extern Blockno;
```

没有重新说明 Blockno，但

```
extern int Blockno;
```

重新说明了 Blockno。

8.10 类型等价

如果两个类型区分符表包含相同的类型区分符集合(把某些类型区分符的潜在等价关系考虑在内，例如，单个 long 就是 long int)，那么这两个类型区分符表就是等价的。具有不同标记的结构、联合和枚举是不相同的，不带标记的联合、结构或枚举指定各不相同的类型。

如果两个类型在展开了其中任何 typedef 类型并删除了所有函数参数标识符后，它们的抽象说明符(见 8.8 节)是相同的，直到类型区分符表的等价，那么就称这两个类型是相同的。数组大小和函数参数类型在这里是有效因素。

9 语句

如果不特别指明，语句都是顺序执行的。语句执行都有一定的结果，但没有值。语句可分为几种。

语句:
  带标号语句
  表达式语句
  复合语句
  选择语句
  循环语句
  跳转语句

9.1 带标号语句

语句可带有标号前缀。

带标号语句:
  标识符 : *语句*
  case *常量表达式* : *语句*
  default : *语句*

由标识符构成的标号用了说明该标识符. 标识符标号的唯一用途是作为 goto 语句的转向的目标。标识符的作用域是当前函数。因为标号有自己的名字空间，它们不会与其他的标识符混淆并且不能被重新说明。参见 11.1 节。

case 标号和 default 标号用在 switch 语句中(参见 9.4 节)。case 标号的常量表达式必须为整型。

标号本身不会改变程序控制流。

9.2 表达式语句

大部分语句为表达式语句，其形式如下：

表达式语句:
  表达式 $_{opt}$;

大多数表达式语句为赋值语句或函数调用语句。表达式的所有副作用在下一个语句执行前完成。如果没有表达式，则称做空语句。空语句通常用来提供一个空体给循环语句或用来放置标号。

9.3 复合语句

当需要把若干个语句作为一个语句来使用时，可以使用复合语句(也称“分程序”)。函数定义的体就是复合语句。

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*

如果在一分程序外说明的标识符又出现在分程序内的说明表中.那么外部说明在分程序内就被屏蔽了(参见 11.1 节)。在同一分程序内一个标识符只能说明一次。此规则适用于同一名字空间的标识符(参见 11 节)，不同名字空间的标识符被认为是不同的。

自动对象在每次进入分程序时按说明的顺序初始化。如果执行了一跳转语句进人分程序，则不进行初始化。静态对象仅在程序开始执行前初始化一次。

9.4 选择语句

选择语句有如下几种控制流形式：

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*

在两种形式的 if 语句中，表达式都必须为算术或指针类型。首先计算表达式的值包括所有的副作用，如果不等于 0 则执行第一个子语句。在第二种形式中，如果表达式为 0，则执行第二个子语句。通过将 else 与同一嵌套层中遇到的最近的未匹配 else 的 if 相连接可以解决 else 的歧义性。

switch 语句根据表达式的不同取值将控制转向相应的分支。关键字 switch 之后用圆括号括起来的表达式必须为整类型。由 switch 语句控制的子语句一般是复合语句。该子语句中的任一语句可带一个或多个 case 标号(参见 9.1 节)。控制表达式要进行整提升(参见 6.1 节)，case 常量被转换为整提升后的类型。与同一 switch 语句相关的两个 case 常量在转换后不可能有相同的值。也可能至多有

一个 default 标号与一个 switch 语句相关。switch 语句可以嵌套，一个 case 或 default 标号与包含它的最小的 switch 相关。

在 switch 语句执行时，首先计算表达式的值及其副作用，并将其值与每个 case 常量比较，如果某 case 常量与表达式的值相同，控制转向与 case 标号匹配的语句。如果没有 case 常量与表达式匹配，并且如果有 default 标号，控制转向有 default 标号的语句。如果没有 case 常量匹配，并且也没有 default 标号，则 switch 语句的所有子语句都不执行。

- 在本书第 1 版中，switch 语句的控制表达式与 case 常量都必须为 int 类型。

9.5 循环语句

循环语句用于指定程序段的循环执行。

循环语句:

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*

在 while 语句和 do 语句中，只要表达式的值不为 0，则其中的子语句一直重复执行。表达式必须为算术或指针类型。while 语句在语句执行前测试表达式并计算其副作用，而 do 语句在每次循环后测试。

在 for 语句中，第一个表达式被计算一次，以此对循环初始化。对该表达式的类型没有限制。第二个表达式必须为算术或指针类型，在每次开始循环前计算其值。如果该表达式的值等于 0，那么 for 语句终止执行。第三个表达式在每次循环后计算，以重新对循环进行初始化，它的类型也没有限制。所有表达式的副作用在计算其值后立即完成。如果在子语句中不包含 continue 语句，那么语句

for (*表达式1*; *表达式2*; *表达式3*) *语句*

等价于

```
表达式1;
while (表达式2) {
    语句
    表达式3;
}
```

for 语句的三个表达式中个任一表达式都可缺省。在第二个表达式缺省时等价于测试一个非 0 常量。

9.6 跳转语句

跳转语句用于无条件地转移控制。

跳转语句:

goto *标识符*;

continue;

break;

return *表达式*$_{opt}$;

在 goto 语句中，标识符必须为位于当前函数中的标号(参见 9.1 节)。控制转移到标号所指定的语句。

continue 语句只能出现在循环语句内，它将控制转向包含此语句的最内层循环部分。更精确地说，在下面任一语句内，

```
while (...) {      do {              for (...) {
```

```
  ...              ...              ...
contin: ;        contin: ;        contin: ;
}                } while (...);   }
```

如果 continue 语句不包含在更小的循环语句中，则 continue 语句与 goto contin 语句等价。

break 语句只能用在循环语句或 switch 语句中，用于终止包含这样语句的最内层循环语句的执行，并将控制转向到被终止语句的下一个语句。

return 语句用于将函数返回到调用者，当 return 语句后跟一表达式时，其值返回给函数调用者。像赋值一样，该表达式被转换为其所在函数返回的类型。

函数的自然结束等价于一个不带表达式的 return 语句。在两种情况下返回值都是没有定义的。

10 外部说明

提供给 C 编译程序处理的输入单元称做翻译单元，它由一外部说明序列组成，这些外部说明或者是说明，或者是函数定义。

翻译单元：

外部说明

翻译单元 外部说明

外部说明：

函数定义

说明

就像在分程序中说明的作用域为整个分程序一样，外部说明的作用域是其所在的翻译单元。外部说明除了只能出现在函数代码可能出现的位置外，其语法规则与其他说明一样。

10.1 函数定义

函数定义有如下形式：

函数定义：
说明区分符$_{opt}$ *说明符* *说明表*$_{opt}$ *复合语句*

在说明区分符中可以使用的存储类区分符只能是 extern 或 static，关于这两个存储类区分符之间的区别参见 11.2 节。

函数可返回一算术类型、结构、联合、指针或 void，但不能返回函数或数组。函数说明中的说明符必须显式指定所说明的标识符具有函数类型，也就是说，必须包含如下两种形式之一(参见 8.6.3 节)：

直接说明符 (*参数类型表*)
直接说明符 (*标识符表*$_{opt}$)

其中，直接说明符为一标识符或用圆括号括起来的标识符。特别地、不能通过 typedef 获得一个函数类型。

使用第一种形式的说明符的函数定义为新方式的函数定义，其参数及其类型都在参数类型表中说明，函数说明符后的说明表不能缺少。除非参数类型表中只有 void 类型(表示该函数没有参数)，参数类型表中的每个说明都必须包含一个标识符。如果参数类型表以“，...”结束，那么在调用该函数时所用的变元数就可多于参数数。在标准头文件<stdarg.h>中定义的、在 C 标准库中介绍的 va_arg 宏机制被用来表示额外的变元。变参函数必须至少有一个指名参数。

使用第二种形式的说明符的函数定义为旧方式的函数定义，标识符表给出了参数的名字，这些参数的类型由说明表指定。未做说明的参数默认为 int。说明表必须只说明在标识符表中指名的参数，不能进行初始化，register 是唯一可以使用的存储类区分符。

在这两种函数定义中，参数可理解为是在组成函数体的复合语句刚开始执行时说明的，因此在该复合语句中不能重新说明与参数同名的标识符(但可以像其他的标识符一样在该复合语句内的分程序中重新说明参数标识符)。如果某一参数说明为“某一类型 *type* 的数组”，那么该说明会被自动调整，使该参数为“指向类型 *type* 的指针”。类似地，如果一参数说明为“返回某一类型 *type* 的函数”，那么该说明会被调整使该参数为“指向返回类型 *type* 的函数的指针”。在调用函数时，必要时要对变元进行类型转换并赋值给参数，参见 7.3.2 节。

- 新方式函数定义是在 ANSI 标准中新引入的。关于提升的细节也有些细微的变化。第 1 版指定的 float 类型的参数说明被调整为 double 类型。当在函数内部生成一指向参数的指针时，区别就显而易见了。

下面是一个新方式函数定义的完整的例子：

```
int max(int a, int b, int c)
{
    int m;

    m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

这里 int 是说明区分符，max(int a, int b, int c)是函数说明符，{ ... }是给出函数代码的分程序。相应旧方式的定义为：

```
int max(a, b, c)
```

```
int a, b, c;
{
    /* ... */
}
```

这里 int max(a, b, c)是说明符，int a, b, c;是参数的说明表。

10.2 外部说明

外部说明用于指定对象、函数及其他标识符的特性。术语“外部”指它们位于函数外部，并且不直接与关键字 extern 连接。对外部说明的对象可以不指定存储类，也可指定为 extern 或 static。

对同一标识符的几个外部说明可存在于同一翻译单元中，只要它们类型和连接一致，并且对该标识符至多只有一个定义。

对一个对象或函数的两个说明只要遵循 8.10 节讨论的规则就认为是类型一致的。此外，如果两个说明的区别在于：一个类型为不完整结构、联合或枚举类型(参见 8.3 节)，而另一个是带同一标记的对应的完整类型，那么也认为这两个类型是一致的。此外，如果一个类型是不完整数组类型(参见 8.6.2 节)而另一个类型为完整数组类型，除此之外其他都相同，那么也认为这两个类型是一致的。最后，如果一个类型指定了一个老式函数，而另一个类型指定了带参数说明的在其他方面相同的新式函数，也认为它们是一致的。

如果对一个对象或函数的第一个外部说明包含 static 区分符，那么该标识符有内部连接，否则它有外部连接。关于连接将在 11.2 节讨论。

一个对象的外部说明若带初始化符则该说明就是定义。如果一个外部说明不带初始化符并且不含 extern 区分符，那么它就是一个暂时定义。如果一个对象的定义出现在翻译单元中，那么所有暂时定义都被认为仅仅是多余的说明；如果在该

翻译单元中不存在对该对象的定义，那么其暂时定义变为一个初始化符为 0 的单一的定义。

每个对象都必须正好有一个定义。对具有内部连接的对象，这个规则分别作用于每个翻译单元，这是因为内部连接的对象对每个翻译单元是独一无二的。对于具有外部连接的对象，这个规则作用于整个程序。

- 虽然单一定义规则在阐述上与本书第 1 版有所不同，但它在效果上与这里所述是一样的。有些实现通过将暂时定义的概念一般化而放宽了这个限制。在另一种阐述中，对一个程序的所有翻译单元中的外部连接对象的所有暂时定义被一起考虑，而不是在各个翻译单元中分别考虑，在 UNIX 系统中通常就采用的这种方法，并且被认为是该标准的一般扩展。如果一个定义出现在程序中的某个地方，邵么暂时定义仅被认为是说明，但如果没有定义出现，则所有暂时定义变为具有初始化符的定义。

11 作用域与连接

一个程序中所有单元不必同时编译。源文本可保存在包含翻译单元的若干个文件中，预编译过的程序段可以从库中装入。程序中函数间的通信可以通过调用和操纵外部数据来实现。

因此，我们就要考虑两种类型的作用域：第一种是标识符的词法作用域，它是体现标识符特性的程序文本区域；第二种是与外部连接的对象和函数相关的作用域，它决定在各个已编译的翻译单元内标识符之间的连接。

11.1 词法作用域

标识符可以在若干个名字空间中使用而互不影响。如果在不同的名字空间中使用同一标识符，那么即使是在同一作用域内，这个标识符也可用于不同的目的。名字空间包含如下几种：对象、函数、类型定义名字和枚举常量；标号；结构标记、联合标记和枚举标记；每个结构或联合的各自成员。

- 这些规则与本手册第 1 版所述有几点不同。标号以前没有自己的名字空间；结构和联合分别有各自的名字空间，在某些实现中枚举标记亦是如此；把不同种类的标记放在同一名字空间中是一种新的限制。与第 1 版的最大不同是每个结构和联合都为其成员创立不同的名字空间，因此同一名字可出现在若干个不同结构中。这一规则在近几年十分常用。

在一个外部说明中的对象或函数标识符的词法作用域从其说明结束开始直到所在翻译单元的结束。函数定义中参数酌作用域在定义函数的分程序开始处开始贯穿整个函数，函数说明中参数的作用域在说明符的末尾处结束。分程序头部中说明的标识符的作用域是其所在的整个分程序。标号的作用域是其所在函数。结构、联合、枚举标记或枚举常量的作用域从其出现在类型区分符中开始，到翻译单元结束(对外部说明)或分程序结束(对函数内的说明)。

如果一标识符显式地在分程序(包括组成函数的分程序)头部中说明，任何分程序外部说明的标识符将被覆盖直到分程序结束。

11.2 连接

在翻译单元中，具有内部连接的同一对象或函数标识符的所有说明均指同一实体，并且该对象或函数对这个翻译单元是唯一的。具有外部连接的同一对象或函数标识符的所有说明也指同一实体，并且该对象或函数在整个程序中共享。

如 10.2 节所述，如果使用了 static 区分符，那么对一标识符的最初的外部说明给出了标识符内部连接，否则，给出该标识符外部连接。如果在一分程序内对一个标识符的说明不包含 extern 区分符，那么该标识符无连接，并且对于函数是唯一的。如果这样的说明中包含 extern 区分符，并且在包含该分程序的作用域

中对该标识符的外部说明是活动的，那么该标识符与外部说明具有相同的连接，并表示同一对象或函数。但是，如果没有外部说明是可见的，那么其连接是外部的。

12 预处理

预处理程序用于执行宏替换、条件编译和引入指名的文件。以#开始的命令行(“#”前可以有空倍)就是预处理程序处理的对象。这些命令行的语法独立于语言的其他部分，它们可在任何地方出现，其作用可延续到所在翻译单元的末尾(与作用域无关)。行边界是有实际意义的，每一行都单独分析(关于如何将若干行连接起来，参见 12.2 节)。对预处理程序而言，单词就是任何语言的单词，或者像在#include 指令(参见 12.4 节)中用做文件名字的字符序列。此外，所有未做其他定义的字符都被认为是单词。但是，除空格和横向制表符之外的空白符的效果在预处理程序指令行中是没有定义的。

预处理本身是在逻辑上连续的几个阶段完成的，在某些特殊的实现中可以缩减。

1. 首先，把如 12.1 节所述的三字符序列替换为其等价字符，如果操作系统环境需要，还要在源文件的各行之间插入换行符。
2. 把指令行中位于换行符前的反斜杠符“\”删除掉，从而把各指令行连接起来(参见 12.2 节)。
3. 把程序分成用空白符隔开的单词，把注解替换为一个空白符。接着执行预处理指令．并进行宏扩展(参见 12.3 节至 12.10 节)。
4. 把字符常量和字符串字面值中的换码序列(参见 2.5.2 节与 2.6 节)替换为其等价字符，然后把相邻的字符串字面值连接起来。
5. 通过收集必要的程序和数据，并将外部函数和对象的引用与其定义相连接，翻译经过以上处理得到的结果，然后与其他程序和库连接起来。

12.1 三字符序列

C 源程序的字符集是 7 位 ASCII 码的子集，但它是 ISO 646-1983 不变代码集(Invariant Code Set)的超集。为了使程序能用这种缩减字符集表示，如下所示的所有三字符序列都要用相应的单个字符替换，这种替换在其他任何处理之前进行。

```
??=  #          ??(  [          ??<  {
??/  \          ??)  ]          ??>  }
??'  ^          ??!  |          ??-  ~
```

除此之外不进行其他替换。

- 三字符序列是 ANSI 标准新引入的。

12.2 行连接

通过把以反斜杠“\”结束的指令行末尾的反斜杠和其后的换行符删除掉，可以将若于指令行合并成一行。这种处理要在分离单词之前进行。

12.3 宏定义和扩展

形如

define *标识符 单词序列*

的控制行用于使预处理程序在此后将指定标识符的各个实例用给定的单词序列替换，单词序列前后的空白符都被丢掉。第二次用#define 指令定义同一标识符

是错误的，除非第二次定义中的单词序列与第一次相同(所有的空白分隔符被看做是相同的)。

形如

define *标识符* (*标识符表*) *单词序列*

的指令行是一个带有参数(由标识符表指定)的宏定义，其中第一个标识符与圆括号“(”之间没有空格。像第一种形式一样，单词序列前后的空白符都被丢弃掉。如果要对宏重定义，那么就要求其参数个数和拼写及其单词序列都必须与前面的定义相同。

形如

undef *标识符*

的控制行用于使指定标识符的预处现程序定义不被考虑。将#undef 应用于未知标识符(即未用#define 指令定义的标识符)不是错误。

当一个宏以第二种形式定义时，由宏标识符(后面可以跟一个空白符)及后随的用一对圆括号括起来的由逗号分隔的单词序列组成了一个宏调用。宏调用的变元是用逗号分隔的单词序列，用引号或嵌套的括号括起来的逗号不用于分隔变元。在处理过程中，变元不进行宏扩展。宏调用时变元的数目必须与其定义时参数的数目匹配。在变元被分离后．先将前导和结尾的空白符删除，然后在替换宏的单词序列时，用由各个变元产生的单词序列替换对应的未用引号括住的参数标识符。除非在替换序列中参数有前导#，或者前导或尾随有##，在插入前，要对宏调用的变元序列进行检查，在必要时进行扩展。

有两个特殊的操作符会影响替换过程。首先，如果在替换单词序列中参数有直接前导#，那么要用双引号"括住对应的参数，然后将#和参数标识符用被引号括住的变元替换。在字符串字面值或字符常量两边或内部的每个双引号"或反斜杠\前要插入一反斜杠\。

其次，如果两种类型的宏中无论哪一种宏的定义单词序列包含有一个##操作符，那么，在参数替换后要把##及其前后的空白符删除掉，以便将相邻单词连接起来形成一新的单词。如果如此所产生的单词是无效的，或者，如果结果依赖于##操作将的处理顺序，那么，其结果没有定义。同样，##也可能不出现在替换单词序列的开头或结尾。

对这两种类型的宏，都要重复扫描替换单词序列以找到更多的定义标识符。但是，一旦一个给定的标识符在给定扩展中被替换，当再次扫描再遇到此标识符时就不再对其进行替换，而是保持不变。

即使宏扩展的最终值以#开始，也不认为其是预处理指令。

- 有关处理宏扩展的细节在 ANSI 标准中比第 1 版描述得更详细，最重要的变化是加入了#和##操作符，这使得引用与连接成为可能。一些新规则，特别是与连接有关的规则是很奇怪的(参见下面的例子)。

例如，这种功能可在下面用做“显式常量”：

```
#define TABSIZE 100
int table[TABSIZE];
```

定义

```
#define ABSDIFF(a, b)  ((a)>(b) ? (a)-(b) : (b)-(a))
```

定义了一个用于返回其两个参数变元差的绝对值的宏。与函数不同的是，变元与返回值可以是任意算术类型甚至可以是指针。同样，变元可能有副作用，要计算两次，一次是进行测试，另一次产生值。

对定义

```
#define tempfile(dir)    #dir "%s"
```

宏调用 tempfile(/usr/tmp)将产生

```
"/usr/tmp" "%s"
```

接着被连接为一个单一字符串。在定义

```
#define cat(x, y)       x ## y
```

后，宏调用 cat(var, 123)产生 var123。但是，宏调用 cat(cat(1,2),3)没行定义：##的存在限制了外部调用的变元的扩展。因此将产生单词串

```
cat  (  1  ,  2  )3
```

并且)3 不是合法的单词，它是由第一个变元的最后一个单词与第二个变元的第一个单词连接而成。如果再进行第二层的宏定义：

```
#define xcat(x, y)      cat(x,y)
```

那么就会得到所要的效果：xcat(xcat(1, 2), 3)将产生 123，因为 xcat 自身的扩展不包含##操作符。

类似地，ABSDIFF(ABSDIFF(a,b),c)将产生所期望的完全扩展的结果。

12.4 文件包含

形如

```
# include <文件名>
```

的控制行用于将该行替换成由文件名所命名的文件的内容。文件名中不能出现大于字符>或换行符。如果文件名中包含字符"、'、\或/*，那么其行为没有定义。预处理程序将在依赖于实现的有关位置中查找所指名的文件。

类似地，形如

include "*文件名*"

的控制行首先从最初的源文件的目录开始搜索指名的文件(搜索过程依赖于实现)，如果没找到指名的文件，那么就像在第一种形式中那样处理，即在实现定义的有关位置上接着查找所指名的文件。在文件名中使用字符'、\或/*仍然是没有定义的，但可以使用字符>。

最后．形如

include *单词序列*

的、与上述两种情况都不匹配的指令通过像对普通文本一样扩展单词序列来解释。必须将其解释成具有<...>或"..."两种形式之一，然后再按上述方式进行相应处理。

#include 文件可以嵌套。

12.5 条件编译

对一个程序的某些部分可以进行条件编译，这时可按照如下所示的语法进行：

预处理程序条件:
if-行 *文本* *elif-部分* *else-部分*$_{opt}$ #endif

if-行:
if *常量表达式*

ifdef *标识符*

ifndef *标识符*

elif-部分:

elif-行 *文本*

elif-部分$_{opt}$

elif-行:

elif *常量表达式*

else-部分:

else-行 *文本*

else-行:

#else

其中，每个条件编译指令(if-行、elif-行、else-行及#endif)在程序中均单独占一行。预处理程序依次对#if 及后继的#elif 指令中的常量表达式进行计算，直到发现有一个指令的常量表达式有非0值，并删去值为0的指令行后面的文本。常量表达式不为0的#if 和#elif 指令之后的文本像其他程序代码一样进行编译。在这里，“文本”由任何不属于该条件编译指令结构的程序代码组成，它可以包含预处理指令，也可以为空。一旦预处理程序发现某个#if 或#elif 条件编译指令中的常量表达式的值不为0并选择紧随其后的程序代码供以后的编译阶段使用时，就删去后继的#elif 和#else 条件编译指令及相应的文本。如果#if 与后继的所有#elif 指令中的常量表达式的值都为0，并且该条件编译指令链中包含一条#else 指令，那么就选择在#else 指令之后的文本。除了对条件编译指令的嵌套进行检查之外，该条件编译指令的不活动分支(即条件为假的分支)所控制的文本都要跳过去。

在#if 及#elif 指令中的常量表达式中可以进行通常的宏替换。除此之外，任何形如

defined *标识符*

或

defined (*标识符*)

的表达式在进行宏扫描之前要进行替换，如果该标识符在预处理程序中已有定义，那么就用 1L 来替换它，否则，用 0L 来替换。在预处理程序进行宏扩展之后的任何标识符用 0L 来替换。最后，每个整数类型的常量都被预处理程序认为其后面跟有后缀 L，以便把所有的算术运算都当作是在长整数类型或无符号长整数类型的运算分量之间进行的运算。

进行上述处理之后的常量表达式(见 7.19 节)满足如下限制：它必须是整型的，并且其中不包含 sizeof 与强制转换运算符或枚举常量。

控制行

#ifdef *标识符*
#ifndef *标识符*

分别等价于：

if defined *标识符*
if ! defined *标识符*

- #elif 是 ANSI C 中新引入的条件编译指令. 尽管它已经在某些预处理程序中被实现。defined 预处理运算符也是新引入的。

12.6 行控制

为利于其他预处理程序生成C程序，形式为

```
# line 常量 "文件名"
# line 常量
```

之一的预处理指令，为了错误诊断的目的，使编译程序相信下一行源代码的行号被置为十进制整数常量，当前的输入文件由标识符命名。如果缺少带双引号的文件名部分，那么就不改变当前正被编译的源文件的名字。对行中的宏进行扩展后再进行解释。

12.7 错误信息生成

形如

```
# error 单词序列opt
```

的预处理程序行用于使预处理程序打印包含该单词序列的诊断信息。

12.8 编译指示

形如

```
# pragma 单词序列opt
```

的控制行(叫做编译指示)使得处理程序完成依赖于实现定义的动作。不能识别的编译指示被忽略掉。

12.9 空指令

形如

#

的预处理程序行无任何作用。

12.10 预定义名字

有些标识符是预定义的，它们被扩展后产生特定的信息。它们与预处理程序表达式操作符 defined 都不能用#undef 取消其定义或重新进行定义。

__LINE__	包含当前源文件的行号的十进制常量。
__FILE__	包含正被编译的源文件名字的字符串字面值。
__DATE__	包含源文件的编译日期的字符串字面佰，其形式为"Mmmm dd yyyy"。
__TIME__	包含编译时间的字符串字面值，其形式为"hh:mm:ss"。
__STDC__	整型常量 1。它表示该标识符只有在与标准一致的实现中被定义为 1。

- #error 与#pragma 是 ANSI 标准中新引入的。这些预定义的预处理程序宏也是新引入的，其中的一些宏已经在某些编译程序中实现。

13 语法

以下是对本手册前面部分所讨论的语法的一个简要概括。它们的内容完全相同，但所给的顺序不同。

本语法未定义终结符整形常量、字符常量、浮点常量、标识符、字符串和枚举常量。以打字机字体的形式表示的单词和符号是字面方式的终结符。本语法可以机

械地转换为自动语法分析程序生成器(parser-generator)可以接受的输入。除了增加语法标记以表明产生式中的可选项外，还有必要扩展“之一”结构，并(根据语法分析程序生成器的规则)复制每个带有 *opt* 符号的产生式，一次有该符号而一次没有。进一步更改即删除产生式

类型定义名字: *标识符*

使类型定义名字成为终结符。这个语法可被 YACC 语法分析程序生成器接受。但有一个冲突，是由 if-else 的结构歧义性产生的。

翻译单元:

外部说明

翻译单元 外部说明

外部说明:

函数定义

说明

函数定义:

说明区分符$_{opt}$ *说明符 说明表*$_{opt}$ *复合语句*

说明:

说明区分符 初始化说明符表$_{opt}$;

说明表:

说明

说明表 说明

说明区分符:

存储类区分符 说明区分符$_{opt}$

类型区分符 说明区分符$_{opt}$

类型限定符 说明区分符$_{opt}$

存储类区分符:

auto register static extern typedef

之一

类型区分符:

void char short int long float double signed

unsigned *结构或联合区分符 枚举区分符 类型定义名字*

之一

类型限定符:

const volatile

之一

结构或联合区分符:

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

之一

结构说明表:

结构说明

结构说明表 结构说明

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 = *初始化符*

结构说明:

区分符限定符表 *结构说明符表*;

区分符限定符表:

类型区分符 *区分符限定符表*$_{opt}$

类型限定符 *区分符限定符表*$_{opt}$

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

说明符$_{opt}$: *常量表达式*

枚举区分符:

enum *标识符*$_{opt}$ { *枚举符表* }

enum *标识符*

枚举符表:

枚举符

枚举符表 , *枚举符*

枚举符:

标识符

标识符 = *常量表达式*

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:

* *类型限定符表* $_{opt}$

* *类型限定符表* $_{opt}$ *指针*

类型限定符表:

类型限定符

类型限定符表 类型限定符

参数类型表:

参数表

参数表 , ...

参数表:

参数说明

参数表 , *参数说明*

参数说明：

说明区分符 说明符

说明区分符 抽象说明符$_{opt}$

标识符表：

标识符

标识符表 ， *标识符*

初始化符：

赋值表达式

{ *初始化符表* }

{ *初始化符表* ， }

初始化符表：

初始化符

初始化符表 ， *初始化符*

类型名字：

区分符限定符表 抽象说明符$_{opt}$

抽象说明符：

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*$_{opt}$)

类型定义名字:

标识符

语句:

带标号语句

表达式语句

复合语句

选择语句

循环语句

跳转语句

带标号语句:

标识符 : *语句*

case *常量表达式* : *语句*

default : *语句*

表达式语句:

表达式$_{opt}$;

复合语句:

{ *说明表*$_{opt}$ *语句表*$_{opt}$ }

语句表:

语句

语句表 *语句*

选择语句:

if (*表达式*) *语句*

if (*表达式*) *语句* else *语句*

switch (*表达式*) *语句*

循环语句:

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*

跳转语句:

goto *标识符*;

continue;

break;

return *表达式*$_{opt}$;

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

一元表达式 赋值运算符 赋值表达式

赋值运算符:

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

之一

条件表达式:

逻辑或表达式

逻辑或表达式 ? *表达式* : *条件表达式*

常量表达式：

条件表达式

逻辑或表达式：

逻辑与表达式

逻辑或表达式 || *逻辑与表达式*

逻辑与表达式：

按位或表达式

逻辑与表达式 && *按位或表达式*

按位或表达式:

按位异或表达式

按位或表达式 | *按位异或表达式*

按位异或表达式：

按位与表达式

按位异或表达式 ^ *按位与表达式*

按位与表达式：

相等类表达式

按位与表达式 & *相等类表达式*

相等类表达式：

关系表达式

相等类表达式 == *关系表达式*

相等类表达式 != *关系表达式*

关系表达式：

移位表达式

关系表达式 < *移位表达式*

关系表达式 > *移位表达式*

关系表达式 <= *移位表达式*

关系表达式 >= *移位表达式*

移位表达式：

加法类表达式

移位表达式 << *加法类表达式*

移位表达式 >> *加法类表达式*

加法类表达式：

乘法类表达式

加法类表达式 + *乘法类表达式*

加法类表达式 - *乘法类表达式*

乘法类表达式：

强制转换表达式

乘法类表达式 * *强制转换表达式*

乘法类表达式 / *强制转换表达式*

乘法类表达式 % *强制转换表达式*

强制转换表达式：

一元表达式

(*类型名字*) *强制转换表达式*

一元表达式：

后缀表达式

++一元表达式

--一元表达式

一元运算符 强制转换表达式

sizeof 一元表达式

sizeof (类型名字)

一元运算符:

& * + - ~ !

之一

后缀表达式:

初等表达式

后缀表达式[表达式]

后缀表达式(变元表达式表$_{opt}$)

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--

初等表达式:

标识符

常量

字符串

(表达式)

变元表达式表:

赋值表达式

变元表达式表 , 赋值表达式

常量:

整型常量

字符常量

浮点常量

枚举常量

以下是预处理程序的语法，它概述了控制行的结构，但不适合机械性语法分析。它包含符号文本(即通常的程序文本)、非条件预处理程序控制行或完全的预处理程序条件结构。

控制行:

define *标识符 单词序列*

define *标识符*(*标识符, ... , 标识符*) *单词序列*

undef *标识符*

include <*文件名*>

include "*文件名*"

include *单词序列*

line *常量* "*文件名*"

line *常量*

error *单词序列*$_{opt}$

pragma *单词序列*$_{opt}$

#

预处理条件指令

预处理条件指令:

if-行 文本 elif-部分 else-部分$_{opt}$ #endif

if-行:

if *常量表达式*

ifdef *标识符*

ifndef *标识符*

elif-部分:

elif-行 *文本*

elif-部分$_{opt}$

elif-行:

elif *常量表达式*

else-部分:

else-行 *文本*

else-行:

#else

索引: