UDC 519.68：800.92
L 74

中华人民共和国国家标准

GB/T 15272—94

程序设计语言 C

Programming languages—C

1994-12-07发布　　1995-08-01实施

国家技术监督局　发布

目　　次

中华人民共和国国家标准

GB/T 15272—94
ISO/IEC 9899—1990

程 序 设 计 语 言 C

Programming languages—C

本标准等同采用了国际标准ISO/IEC 9899—1990《程序设计语言C》。

0 引言

随着新设备和扩展字符集的引入，标准中可能会增加新的特征。在语言和库两章中的有关条文对实现者和程序员使用尽管本身是合法的，但可能与未来增加的内容相冲突的特征给出了告诫。

有一些特征属于*将逐渐废弃*的，这意味着在未来的标准版本中可能会撤消这些特征。本标准文本中仍然保留它们的原因是这些特征已使用得很广泛，但不鼓励在新的实现中使用这些与实现有关的特征，或在新的程序中使用这些语言特征(见6.9条)或库特征(见7.13条)。

本标准文本分为下列四个主要部分：

——引言和基本元素；

——翻译与执行C程序的环境的特性；

——语言的语法、约束与语义；

——库设施。

在有些条文中：给出了示例以说明所描述的构件的可能形式；加注以强调在相应条文或标准的其他地点所描述的规则的作用；涉及其他相关条文时给出了引用。附录部分总结了包含在标准中的信息。引言、示例、注、引用和附录均不属于标准的组成部分。

第6章语言是从"The C Reference Manual"中派生的。

第7章库是基于C用户协会1984年的标准(1984/usr/group standard)。

1 主题内容与适用范围

本标准规定了用程序设计语言C书写的程序的形式及其解释。

注：设计本标准的目的在于促进C程序在各个数据处理系统之间的可移植性。本标准的主要使用对象是实现者和程序员。与本标准相关的一个基本文件解释了制定本标准的技术委员会的许多决策。

本标准规定了：

——C程序的表示；

——C语言的语法和约束；

——解释C程序的语义规则；

——由C程序处理的输入数据的表示；

——由C程序产生的输出数据的表示；

——对遵从标准的C实现的限制和限定值。

本标准对以下内容未作规定：

——为数据处理系统使用而对C程序进行变换的机制；

——为数据处理系统使用而调用C程序的机制；

国家技术监督局1994-12-07批准　　1995-08-01实施

——为C程序使用而对输入数据进行变换的机制；
——在C程序产生输出数据后对其进行变换的机制；
——将超出任何特定数据处理系统或特殊处理机容量的C程序及其数据的长度与复杂性；
——对能够支持一个遵从标准的实现的全部最低要求。

2 引用标准

GB 1988 信息处理 信息交换用七位编码字符集
GB 5271 数据处理词汇[1]
GB 12406 表示货币和资金的代码
ANSI/IEEE 754 二进制浮点运算[2]

3 定义和约定

本标准中，动词“应”解释为对实现或程序应提出的要求。反之，动词“不应”解释为应禁止。

下列定义适用于本标准。在本标准中显式定义了的术语，不能被假定为隐含在其他地方定义的类似术语。在本标准中未定义的术语应按GB 5271解释。

3.1 对齐 alignment

指特定类型对象应放置在存储区边界的要求，该边界的地址是字节地址的特定倍数。

3.2 实参 argument

指函数调用表达式中由括号括起来并以逗号分隔的表中的表达式，或类似函数的宏调用中由括号括起来并以逗号分隔的表中的一系列预处理单词。也称“实在参数”。

3.3 (二进)位 bit

执行环境中的数据存储单位，它应大到足以容纳一个可能具有两种值之一的对象。不要求能表达一个对象中每个各别位的地址。

3.4 字节 byte

数据存储单位，足够大以容纳执行环境中基本字符集的任何一个成员。应能唯一表达一个对象中每个各别字节的地址。字节由一系列相邻接的二进位组成，字节中二进位的数目由实现定义。字节的最低有效二进位称为低位，最高有效二进位称为高位。

3.5 字符 character

一个字节内能容纳的二进位表示。源环境和执行环境的基本字符集中的每个成员的表示都应能被容纳在一个字节内。

3.6 约束 constraints

进一步解释语言元素评注的语法或语义限制。

3.7 诊断消息 diagnostic message

实现输出消息子集中的一类消息，是实现定义的。

3.8 提前引用的条文 forward reference

对标准中包含与当前条文相关信息的后续条文的提前引用。

3.9 实现 implementation

一个特定的软件集合，它运行在一个受到特殊控制任选的特定翻译环境，为一个特定执行环境实现

采用说明：

1] GB 5271 等同采用ISO 2382，在本标准第3章出现。

2] ANSI/IEEE 754在本标准第5章出现。

程序的翻译，并支持函数在该执行环境中的执行。

3.10 实现定义的行为 implementation-defined behavior

指对依赖于实现特性的、且是正确的程序构件和数据的行为。每个实现对这些行为均应用文档说明。

3.11 实现规定的限定值 implementation limits

实现对程序所规定的限定值。

3.12 地域特定的行为 locale-specific behavior

指依赖于地域、国别、文化和语言习惯的行为。每个实现对这些行为均应用文档说明

3.13 多字节字符 multibyte character

由一个或多个字节构成的序列，表示源环境或执行环境中扩展字符集的一个成员。扩展字符集是基本字符集的超集。

3.14 对象 object

执行环境中的数据存储区，其内容表示值。除位段以外，对象由一个或多个邻接的字节组成，对象中的字节数目、次序、编码或者显式规定或者由实现定义。当被引用时，对象可被解释为具有特殊类型（见6.2.2.1条）。

3.15 形参 parameter

一种对象，是声明为具有入口值的函数声明或函数定义的一部分，或是紧跟在类似函数的宏定义的宏名之后，由括号括起来并以逗号分隔的表中的标识符。也称“形式参数”。

3.16 未定义的行为 undefined behavior

指当使用了一个不可移植的或是错误的程序构件，或错误的数据，或无法确定值的对象时的行为，而标准并未对这些行为加以规定。所允许的对未定义的行为的处理包括：尽管可能出现不可预测的结果，也完全忽略该情况；在翻译或程序执行时按环境文档规定的特性处理（不保证出现诊断消息）；终止翻译或执行（保证出现诊断消息）。

若违反了在约束条文以外出现的“应”或“不应”要求，则该行为是未定义的。其他未定义的行为在本标准文本中用文字“未定义的行为”，或由省略任何显式的行为定义来指示。对这三种情况并不强调任何区别，它们都描述“没有定义的行为”。

3.17 未规定的行为 unspecified behavior

指标准未提出任何要求的且是正确的程序构件或数据的行为。

示例

a. 函数实参求值的顺序是一种未规定的行为。

b. 整数溢出时的行为是一种未定义的行为。

c. 有符号整数右移时高位如何传递是一种实现定义的行为。

d. 对除26个小写英文字母外的字符，islower函数是否返回真值是一种地域特定的行为。

提前引用的条文：逐位移位算符（6.3.7条），表达式（6.3条），函数调用（6.3.2.2条），函数islower（7.3.1.6条），本地化程序库前导文卷（7.4条）。

4 一致性

*一个严格遵从（标准）的程序*应当只使用本标准中规定了的那些语言和库特征，它不应产生依赖于未规定的、或未定义的、或实现所定义的行为的输出，也不应超出任何实现规定的最低限定值。

*遵从（标准的）实现*有两种形式：宿主型和独立型。一个*宿主型遵从实现*应当接受任何严格遵从的程序。一个*独立型遵从实现*应当接受任何严格遵从的程序，该程序中对在本标准第7章“库”中所规定了的特征的使用与标准前导文卷〈float.h〉、〈limits.h〉、〈stdarg.h〉和〈stddef.h〉一致。一个遵从实现可以有扩展（包括附加的库函数），只要这些扩展不改变严格遵从程序的任何行为。

*遵从程序*是指遵从实现可接受的程序。

注：① 这意味着一个遵从实现不得保留除本标准中显式规定保留的标识符以外的任何标识符。

② 预期严格遵从程序在各遵从实现之间有最大限度的可移植性。遵从程序可能会依赖于遵从实现的某些非可移植性特征。

一个实现应附有定义所有实现定义特征及其扩充的文档。

提前引用的条文：限定值前导文卷〈float.h〉和〈limits.h〉(7.1.5条)，变长实参库前导文卷〈stdarg.h〉(7.8条)，公用定义库前导文卷〈stddef.h〉(7.1.6条)。

5 环境

实现分别在两个数据处理环境中翻译C源文卷并执行C程序，在本标准中分别称它们为*翻译环境*和*执行环境*。它们的特性定义、并约束按照遵从实现的语法和语义规则所构造的遵从C程序的执行结果。

提前引用的条文：在第5章"环境"中，仅给出了许多可能的提前引用的条文中的一小部分。

5.1 概念化模型

5.1.1 翻译环境

5.1.1.1 程序结构

一个C程序并不一定要全都在同一时刻翻译。程序文本存放在本标准中称为*源文卷*的单位中。一个源文卷连同经预处理指示#include并入的所有前导文卷和源文卷，扣除由条件并入预处理指示所跳过的源程序行后，称为一个*翻译单位*。以前翻译过的翻译单位可单独保存或存在库中。一个程序的各个翻译单位之间通过例如对标识符具有外部链接的函数的调用，对标识符具有外部链接的对象的操作或对数据文卷的操作等手段进行通信。翻译单位可以先分别翻译，然后链接产生一个可执行的程序。

提前引用的条文：条件并入(6.8.1条)，标识符的链接(6.1.2.2条)，源文卷并入(6.8.2条)。

5.1.1.2 翻译阶段

翻译的语法规则之间的优先顺序由下列阶段规定：

阶段1. 若有必要，则将物理文卷字符映射到源字符集(对行尾指示符引入新行字符)。用相应的单字符内部表示替换三联符序列。

阶段2. 删除每个反斜线字符后紧跟一个新行字符的字符对，分割物理源程序行以构成逻辑源程序行。不为空的源文卷应以一个新行字符结束，该新行字符前不应有紧接的反斜线字符。

阶段3. 将源文卷分解为预处理单词和一系列的白空类符(包括注释)。源文卷不应在不完整的预处理单词或注释处结束。对每条注释都用一个空格字符替换。保留新行字符。对除新行外的不为空的一系列白空类符的处理是保留还是用一个空格字符替换由实现定义。

阶段4. 执行预处理指示，展开宏调用。#include预处理指示将导致递归地从阶段1到阶段4处理命名的前导文卷或源文卷。

阶段5. 将第一个源字符集成员以及字符常量和串字面值中的转义序列转换为执行字符集中的成员。

阶段6. 串接邻接的字符串字面值单词和邻接的宽串字面值单词。

阶段7. 此时分隔单词的白空类符不再有意义。每个预处理单词都转换为一个单词。对如此得到的单词进行语法分析和语义分析并加以翻译。

阶段8. 解决所有外部的对象引用和函数引用。链接有关的库成分以满足对未在当前翻译单位内定义的对象和函数的外部引用。将所有这类翻译程序的输出收集到一个程序映象中，该映象包含了在其执行环境中执行时所需的信息。

注：① 即使在实践中常常把几个阶段合并在一起，但实现必须表现为如同这些阶段分离出现一样。

② 如在6.1条中所描述的，将源文卷字符分解为预处理单词的过程是上下文相关的。作为例子，请参见在

#include预处理指示中对<的处理。

提前引用的条文：词法元素(6.1 条)，预处理指示(6.8 条)，三联符序列(5.2.1.1 条)。

5.1.1.3 诊断

一个遵从实现对每个违反任何语法规则或约束的翻译单位应至少以一种实现定义的方式产生一条诊断消息。在其他情况下，不需产生诊断消息。

注：目的是实现应标识每个违约的性质，并尽可能定位。当然，只要对一个合法的程序仍在进行正确的翻译，实现就可自由地产生任意数目的诊断消息。实现也可成功地翻译一个不合法的程序。

5.1.2 执行环境

共定义了两种执行环境：*独立环境* 与 *宿主环境*。在两种环境中，当由执行环境调用一个指定的 C 函数时，都发生*程序启动*。在程序启动前应对在静态存储区中的所有对象进行*初始化*（置为它们的初值）。这种初始化的方式和定时在其他地方都未规定。*程序终止* 把控制返回给执行环境。

提前引用的条文：初始化(6.5.7 条)。

5.1.2.1 独立环境

在独立环境，即 C 程序在其中的执行无需操作系统的任何支持的环境中，程序启动时调用的函数的名字和类型是实现定义的。除此以外再无保留的外部标识符。独立程序可用的任何库设施都是实现定义的。

在独立环境中，程序终止的效果也是实现定义的。

5.1.2.2 宿主环境

不一定需要提供宿主环境，但若提供，则应遵从下列规格说明。

5.1.2.2.1 程序启动

程序启动时调用的函数命名为 main。实现不必声明此函数的原型。它可被定义为不带形参：

```
int main(void)  {/*...*/}
```

或者带两个形参：

```
int main(int argc, char *argv[])  {/*...*/}
```

尽管在此使用了名字 argc 和 argv，但由于它们局部于声明它们的函数，因此可使用任何名字。

若定义了函数 main，则其形参应遵循下列约束：

——argc 的值不应为负值；

——argv[argc]应为一个空指针；

——若 argc 的值大于零，则数组成员 argv[0]至 argv[argc-1]应包含指向串的指针，由宿主环境在程序启动前把实现定义的值赋给这些串。这样做的目的是向程序提供由宿主环境的其他部分在程序启动前确定的信息。若宿主环境不能提供大小写字母并存的串，则实现应保证这些串以小写字母接收。

——若 argc 的值大于零，则由 argv[0]所指向的串表示*程序名*。若不能从宿主环境获得程序名，则 argv[0][0]应是空字符。若 argc 的值大于 1，则由 argv[1]至 argv[argc-1]所指向的串表示*程序参数*。

——形参 argc 和 argv 以及由 argv 数组所指向的串应是可由程序修改的，并保持从程序启动至程序终止期间最后一次存储的值。

5.1.2.2.2 程序执行

在宿主环境中，程序可使用第 7 章“库”中描述的所有函数、宏、类型定义和对象。

5.1.2.2.3 程序终止

从对函数 main 的初始调用中返回等价于用函数 main 的返回值作为实参调用函数 exit。若函数 main 执行一个不规定值的返回操作，则返回到宿主环境的终止状态是未定义的。

提前引用的条文：术语定义(7.1.1 条)，函数 exit(7.10.4.3 条)。

5.1.2.3 程序执行

本标准中的语义描述说明了一个抽象机的行为，在该抽象机中，不涉及优化问题。

访问易变型(volatile)对象、修改对象、修改文卷或调用执行上述任何操作的函数都是*副作用*，它们是执行环境的状态变化。对表达式求值可能出现副作用。在执行序列中称为*序点(表达式求值的顺序控制点)*的某些特定点，所有先前求值的副作用均应完成，并且后继求值所需的副作用不应在此之前发生。

在抽象机中，所有的表达式都按语义规定求值。若某个实际实现可以推断出尚不用某个表达式的值也不用产生必须的副作用，包括调用函数或访问易变型对象所引起的任何副作用，则该实现可不必对该表达式的任何部分求值。

当抽象机的处理由于收到一个信号而中断时，仅能依赖作为上一个序点的对象的值。在上一个序点和下一个序点之间可能被修改的对象不一定已接收到它们的正确值。

在进入自动存储期的对象所在的程序块时，对每个自动存储期的对象都产生一个与之相关的实例。这类对象在该程序块执行时以及在该程序块由于调用函数或收到一个信号而挂起时存在并保持其最后存储的值。

对一个遵从实现的最低要求是：

——当在序点时，易变型对象在前面的求值已完成且后继求值尚未发生这种意义上是稳定的。

——当在程序终止时，所有已写到文卷中的数据应与按抽象语义执行程序时所应该产生的结果相同。

——交互设备的动态输入输出应按 7.9.3 条所规定的发生。提出这些要求的目的是使无缓冲的或行缓冲的输出尽可能早出现，保证在程序等待输入以前已实际出现提示消息。

交互设备的组成是实现定义的。

抽象语义与实际语义之间更严格的对应关系可由每个实现定义。

示例

a. 实现可在抽象语义与实际语义之间定义一一对应关系：在每个序点，实际对象的值都与由抽象语义规定的对象的值一致。此时关键字 volatile 将是冗余的。

或者，实现也可在每个翻译单位内部进行各种各样的优化，使得仅当进行跨越翻译单位边界的函数调用时实际语义才与抽象语义一致。在这类实现中，对于主调函数和被调函数在不同翻译单位中的情况，当每次进入函数和由函数返回时，所有外部链接的对象的值和所有可通过指针访问的对象的值将与抽象语义一致。而且，在这类函数入口时，被调函数的形参值及所有可通过指针访问的对象的值将与抽象语义一致。在这类实现中，由函数 signal 激发的中断服务例行程序所引用的对象将需要 volatile 存储的显式规格说明，以及其他实现定义的限制。

b. 执行下列程序片断

```
char c1,c2;
/* ... */
c1=c1+c2;
```

时，“整型升格”要求抽象机先将每个变量的值升格为 int 的尺寸，再将两个 int 量相加，然后截断其和。只要能对两个 char 型量相加且不产生溢出异常，则实际执行只需产生相同的结果，可以省略升格。

c. 类似地，在程序片断

```
float f1,f2;
double d;
/* ... */
f1=f2 * d;
```

中，若实现可保证结果如同采用双精度运算执行(例如，用常量 2.0 替换 d，该常量的类型为 double)一样，则可用单精度运算执行乘法。或者，如果既不会超出值域范围也不会失去精度，那么仅涉及 int 或 float 的运算也可使用双精度运算执行。

d. 为了解释表达式组合的行为，在下列程序片断

```
int a, b;
/*…*/
a=a+32760+b+5;
```

中，由于加法算符的结合律与优先级，上述表达式语句的表现确实与语句

```
a=(((a+32760)+b)+5);
```

相同。即"(a+32760)"的结果与 b 相加，加得的结果再加上 5，得到赋予 a 的值。在溢出将产生异常、且 int 可表示的值的范围是[−32768，+32767]的机器上，实现不能将上述表达式重写为：

```
a=((a+b)+32765);
```

因为若 a 和 b 的值分别为−32754 和−15，则和 a+b 将产生一个异常，而原先的表达式则不会。上述表达式也不能重写为：

```
a=((a+32765)+b);
```

或

```
a=(a+(b+32765));
```

因为 a 和 b 的值可能分别为 4 和−8 或−17 和 12。然而在溢出不产生异常且溢出的结果可逆的机器上，实现可将该表达式语句重写成上述任何一种形式，因为它们全都会得到同样的结果。

e. 表达式的组合不完全决定它的求值。在下列程序片断

```
#include <stdio.h>
int sum;
char *p;
/*…*/
sum=sum * 10-'0'+(*p++=getchar());
```

中，该表达式语句如同写作：

```
sum=(((sum * 10)-'0')+((*(p++))=(getchar())));
```

一样组合。但 p 的实际增量可以在前一个序点和下一个序点(分号；)之间的任何时间出现，而对函数 getchar 的调用可在需要它的返回值之前的任何时刻出现。

提前引用的条文：复合语句或块(6.6.2 条)，表达式(6.3 条)，文卷(7.9.3 条)，序点(6.3 条，6.6 条)，函数 signal(7.7 条)，类型限定词(6.5.3 条)。

5.2 有关环境的考虑

5.2.1 字符集

应定义两个字符集及与它们相关的理序序列：用于书写源文卷的字符集和在执行环境中解释的字符集。执行字符集成员的值是实现定义的；任何在本条要求之外的附加成员均是地域特定的。

在字符常量或串字面值中，执行字符集中的成员应当用对应的源字符集成员或者用由一个反斜线字符\后紧跟一个或多个字符所组成的*转义序列*来表示。在基本执行字符集中应有一个其字节的所有位均置为 0 的，称为*空字符*的字符。空字符用来终止一个字符串字面值。

基本源字符集和基本执行字符集均应至少具有下列成员：

英文字母表中的 26 个大写字母

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

英文字母表中的 26 个小写字母

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

10 个十进制数字字符

0 1 2 3 4 5 6 7 8 9

下列 29 个图形字符

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

再加上空格字符以及表示横向制表、纵向制表和换页的控制字符。

在源字符集和基本执行字符集中，上述十进制数字字符表中 0 之后的每个字符的值均应比前一个字符的值大 1。在源文卷中，应有某种方式指示每一行正文的结束。本标准将这种行尾指示符按单个新行字符处理。在执行字符集中，应有表示告警、退格、回车和新行的控制字符。若在源文卷中遇到除出现在字符常量、串字面值、前导文卷名、注释或永不被转换为单词的预处理单词中的字符外的其他字符，则其行为是未定义的。

提前引用的条文：字符常量(6.1.3.4 条)，预处理指示(6.8 条)，串字面值(6.1.4 条)，注释(6.1.9 条)。

5.2.1.1 三联符序列

下列三个字符的序列称为*三联符序列*。它们在源文卷中的所有出现都用对应的单字符替换。

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

不存在其他的三联符序列。对不是开始上述三联符序列之一的？不作改变。

注：三联符序列使得可以输入未在 GB 1988 中定义的字符。GB 1988 是七位代码集的一个子集。

示例

下列源行

```
printf("Eh??? /n");
```

在替换三联符序列?? /后成为：

```
printf("Eh? \n");
```

5.2.1.2 多字节字符

源字符集中可包含多字节字符，用于表示扩展字符集的成员。执行字符集中也可包含多字节字符，但它们的编码不一定要与源字符集中的多字节字符的编码相同。下列条件对两个字符集均应成立：

——应包含在 5.2.1 条中所定义的单字节字符。

——任何附加成员的存在、含义和表示均是地域特定的。

——一个多字节字符可具有*依赖于状态的编码*。每个多字节字符序列以一个*初始转义状态*开始，当在序列中遇到特定的多字节字符时进入其他实现定义的*转义状态*。当处在初始转义状态时，所有单字节字符均保持它们通常的解释，且不改变转义状态。序列中后继字节的解释是当前转义状态的函数。

——所有位均为 0 的字节应独立于转义状态，一概解释为空字符。

——所有位均为 0 的字节不应作为多字节字符的第二个或后继的字节出现。

对源字符集，下列条件应成立：

——注释、串字面值、字符常量或前导文卷名均应以初始转义状态开始和结束。

——注释、串字面值、字符常量或前导文卷名均应由合法的多字节字符序列组成。

5.2.2 字符显示语义

*活动位置*是指在显示设备上由函数 fputc 输出的下一个字符应出现的位置。把一个可印刷的(如由

函数 isprint 所定义的)字符写到显示设备上的目的是在活动位置上显示该字符的图形表示，然后将活动位置推进到当前行的下一个位置。写的方向是地域特定的。若活动位置已处在一行的最后一个位置，则其行为是未规定的。

表示执行字符集中非图形字符的字母转义序列的作用是在显示设备上产生下列动作：

\a (*告警*)产生一个可听或可视的告警信息。不应改变活动位置。

\b (*退格*)将活动位置移至当前行的前一个位置。若活动位置已处于当前行的起始位置，则其行为是未规定的。

\f (*换页*)将活动位置移至下一个逻辑页面的起始位置。

\n (*新行*)将活动位置移至下一行的起始位置。

\r (*回车*)将活动位置移至当前行的起始位置。

\t (*横向制表*)将活动位置移至当前行的下一个横向表格位置。若活动位置已处于或越过所定义的最后一个横向表格位置，则其行为是未规定的。

\v (*纵向制表*)将活动位置移至当前行的下一个纵向表格起始位置。若活动位置已处于或越过所定义的最后一个纵向表格位置，则其行为是未规定的。

这些转义序列中的每一个均应产生一个实现定义的唯一值，该值应能存储在单个char 型对象中，它们在文本文卷中的外部表示不必与内部表示完全相同，且已超出本标准的范围。

提前引用的条文：函数 fputc(7.9.7.3 条)，函数 isprint(7.3.1.7 条)。

5.2.3 信号与中断

函数应当这样实现：它们在任何时刻均可被一个信号中断，或由一个信号处理程序调用，或二者兼可，并且在中断后不改变早些时候的、但仍活动着的调用者的控制流，也不改变函数的返回值或自动存储期的对象。所有这类对象均应按每次调用为基础保存在*函数映象*，即组成函数的可执行表示的指令之外。

不保证标准库中的函数能再入，并且它们可能修改静态存储期的对象。

5.2.4 环境限定值

翻译环境和执行环境二者均约束语言翻译程序和库的实现。下列条文总结了环境对遵从实现所施加的限定值。

5.2.4.1 翻译限定值

实现应至少翻译并执行一个程序，该程序应至少包含下列每一限定值的一个实例。

注：实现应尽可能避免施加固定的翻译限定值。

——复合语句、循环控制结构和选择控制结构的嵌套层次允许为 15 层。

——条件并入的嵌套层次允许为 8 层。

——可以有 12 个指针、数组和函数声明符(它们可以任何方式组合)修改声明中的算术、结构、联合或不完整类型。

——在一个完全的声明符中，加括号的声明符的嵌套层次允许 31 层。

——在一个完全的表达式中，加括号的表达式的嵌套层次允许 32 层。

——内部标识符或宏名中前 31 个字符有效。

——外部标识符中前 6 个字符有效。

——一个翻译单位内可有 511 个外部标识符。

——在一个块内声明的具有块作用域的标识符允许有 127 个。

——在一个翻译单位内允许同时定义 1024 个宏标识符。

——一个函数定义中允许有 31 个形参。

——一个函数调用中允许有 31 个实参。

——一个宏定义中允许有 31 个形参。

——一个宏调用中允许有 31 个实参。

——一个逻辑源行中允许有 509 个字符。

——(在串接后)一个字符串字面值或宽串字面值中允许有 509 个字符。

——一个对象允许占 32767 个字节(仅对宿主环境)。

——#include 的文卷允许嵌套 8 层。

——一个 switch 语句中允许有 257 个 case 标号(不包括嵌套的 switch 语句中的 case 标号)。

——单个结构或联合允许有 127 个成员。

——单个枚举中允许有 127 个枚举常量。

——单个结构声明表中允许有 15 层嵌套的结构或联合定义。

5.2.4.2 数值限定

遵从实现应用文档说明在本条中规定的所有限定值,这些限定值将在前导文卷〈limits.h〉和〈float.h〉中规定。

5.2.4.2.1 整型的尺寸〈limits.h〉

应使用在#if 预处理指示中所用的常量表达式替换下面所列出的值。而且除 CHAR_BIT 和 MB_LEN_MAX 外,应由表达式替换,替换用的表达式的类型与按整型升格将一个对应类型的对象转换成的表达式的类型相同。它们的实现定义的值应在量值(绝对值)上等于或大于所示的值,且符号与所示值的符号相同。

——不是位段(字节)的最小对象的位数:

CHAR_BIT 8

——signed char 类型对象的最小值:

SCHAR_MIN −127

——signed char 类型对象的最大值:

SCHAR_MAX +127

——unsigned char 类型对象的最大值:

UCHAR_MAX 255

——char 类型对象的最小值:

CHAR_MIN 见下面

——char 类型对象的最大值:

CHAR_MAX 见下面

——对任何所支持的地域环境,一个多字节字符的最大字节数:

MB_LEN_MAX 1

——short int 类型对象的最小值:

SHRT_MIN −32767

——short int 类型对象的最大值:

SHRT_MAX +32767

——unsigned short int 类型对象的最大值:

USHRT_MAX 65535

——int 类型对象的最小值:

INT_MIN −32767

——int 类型对象的最大值:

INT_MAX +32767

——unsigned int 类型对象的最大值:

UINT_MAX 65535

——long int 类型对象的最小值：

LONG_MIN　　−2147483647

——long int 类型对象的最大值：

LONG_MAX　　+2147483647

——unsigned long int 类型对象的最大值：

ULONG_MAX　　4294967295

若在表达式中使用时对 char 类型对象的值按有符号整数处理，则 CHAR_MIN 的值应与 SCHAR_MIN 的值相同，且 CHAR_MAX 的值应与 SCHAR_MAX 的值相同。否则，CHAR_MIN 的值应为 0，而 CHAR_MAX 的值应与 UCHAR_MAX 的值相同（见 6.1.2.5 条）。

5.2.4.2.2　浮点数的特性〈float.h〉

浮点数的特性用表示浮点数和值的模型描述，该模型提供实现所使用的浮点运算的有关信息。

注：使用浮点模型的目的在于使每个浮点特性的描述更为清晰，并不要求实现所使用的浮点运算完全一致。

定义每种浮点类型时使用了下列形参：

s　　符号（±1）

b　　指数部分的基数（大于 1 的整数）

e　　指数部分（最小值 e_{min} 和最大值 e_{max} 之间的一个整数）

p　　精度（有效数中的以 b 为基的数字位数）

f_k　　小于 b 的非负整数（有效数位）

一个规格化的浮点数 x（若 $x\neq 0$，$f_k>0$）用下列模型表示：

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, e_{min} \leqslant e \leqslant e_{max}$$

在<float.h>前导文卷的值中，FLT_RADIX 应是适合于在 #if 预处理指示中使用的常量表达式，所有其他的值则不必一定是常量表达式。除 FLT_RADIX 和 FLT_ROUNDS 以外的所有值对三种浮点类型均有分别的名字。除 FLT_ROUNDS 外，对其他所有值均给出了浮点模型表示。

浮点加的舍入模型由 FLT_ROUNDS 的值表征：

−1　　不可确定

0　　趋向零

1　　舍入为最接近的值

2　　趋向正无穷大

3　　趋向负无穷大

FLT_ROUNDS 的其他值表征实现定义的舍入行为。

下表中列出的值应该用实现定义的，量值（绝对值）等于或大于所示值且符号相同的表达式替换。

——指数部分的基数 b

FLT_RADIX　　2

——浮点有效数中的以 FLT_RADIX 为基的数字位数 p

FLT_MANT_DIG

DBL_MANT_DIG

LDBL_MANT_DIG

——十进制数字 q，使得一个有 q 个十进制数位的浮点数可舍入为有 p 个以 b 为基的数位的浮点数，且反变换后不会改变该 q 个十进制数字 $\lfloor (p-1)\times \log_{10} b \rfloor + \begin{cases} 1, \text{若 } b \text{ 是 10 的幂} \\ 0, \text{否则} \end{cases}$

FLT_DIG　　6

DBL_DIG 10

LDBL_DIG 10

——使得 FLT_RADIX 的该次幂－1 是一个规格化浮点数 e_{min} 的最小负整数

FLT_MIN_EXP

DBL_MIN_EXP

LDBL_MIN_EXP

——使得 10 的该次幂是规格化浮点数数值范围 $\lceil \log_{10} b^{e_{min}-1} \rceil$ 的最小负整数

FLT_MIN_10_EXP －37

DBL_MIN_10_EXP －37

LDBL_MIN_10_EXP －37

——使得 FLT_RADIX 的该次幂－1 是可表示的有穷规格化浮点数 e_{max} 的最大整数

FLT_MAX_EXP

DBL_MAX_EXP

LDBL_MAX_EXP

——使得 10 的该次幂是可表示的有穷规格化浮点数数值范围 $\lfloor \log_{10}((1-b^{-p}) \times b^{e_{max}}) \rfloor$ 的最大整数

FLT_MAX_10_EXP +37

DBL_MAX_10_EXP +37

LDBL_MAX_10_EXP +37

下表中列出的值应该用实现定义的、值等于或大于所示值的表达式替换：

——最大可表示的有穷浮点数 $(1-b^{-p}) \times b^{e_{max}}$

FLT_MAX 1E+37

DBL_MAX 1E+37

LDBL_MAX 1E+37

下表中列出的值应该用实现定义的、值等于或小于所示值的表达式替换：

——1 与在给定浮点类型中可表示的大于 1 的最小值之间的差 b^{1-p}

FLT_EPSILON 1E－5

DBL_EPSILON 1E－9

LDBL_EPSILON 1E－9

——规格化的最小正浮点数 $b^{e_{min}-1}$

FLT_MIN 1E－37

DBL_MIN 1E－37

LDBL_MIN 1E－37

示例

a. 下面描述一个满足本标准最小要求的人为浮点表示，以及在〈float.h〉前导文卷中适合于 float 类型的值：

$$x = s \times 16^e \times \sum_{k=1}^{6} f_k \times 16^{-k}, \qquad -31 \leqslant e \leqslant +32$$

FLT_RADIX	16
FLT_MANT_DIG	6
FLT_EPSILON	9.53674316E-07F
FLT_DIG	6
FLT_MIN_EXP	－31
FLT_MIN	2.93873588E－39F

FLT_MIN_10_EXP	−38
FLT_MAX_EXP	+32
FLT_MAX	3.40282347E+38F
FLT_MAX_10_EXP	+38

b. 下面描述的是也满足 ANSI/IEEE754 中单精度和双精度规格化浮点数要求的浮点表示，以及在〈float.h〉前导文卷中适合于 float 和 double 类型的值：

注：在 ANSI/IEEE 754 标准中的浮点模型从 b 的 0 次幂开始相加，因此指数的界值比这里所示的要小 1。

$$x_f = s \times 2^e \times \sum_{k=1}^{24} f_k \times 2^{-k}, \quad -125 \leqslant e \leqslant +128$$

$$x_d = s \times 2^e \times \sum_{k=1}^{53} f_k \times 2^{-k}, \quad -1\,021 \leqslant e \leqslant +1\,024$$

FLT_RADIX	2
FLT_MANT_DIG	24
FLT_EPSILON	1.19209290E-07F
FLT_DIG	6
FLT_MIN_EXP	−125
FLT_MIN	1.17549435E-38F
FLT_MIN_10_EXP	−37
FLT_MAX_EXP	+128
FLT_MAX	3.40282347E+38F
FLT_MAX_10_EXP	+38
DBL_MANT_DIG	53
DBL_EPSILON	2.2204460492503131E−16
DBL_DIG	15
DBL_MIN_EXP	−1021
DBL_MIN	2.2250738585072014E−308
DBL_MIN_10_EXP	−307
DBL_MAX_EXP	+1024
DBL_MAX	1.7976931348623157E+308
DBL_MAX_10_EXP	+308

提前引用的条文：条件并入(6.8.1 条)。

6 语言

在本标准所使用的语法记法中，语法类别(非终结符)用斜体表示，非终结符后的冒号(:)引出其定义。除在前面冠以词“下列之一”的场合，替代的定义均另起一行书写。任选项用前导词“任选的”指示，因此，

{任选的表达式}

指示在花括号中的一个任选的表达式。

6.1 词法元素

语法

单词:

关键字

标识符

常量
串字面值
算符
标点符号

预处理单词：

前导文卷名
标识符
预处理数
字符常量
串字面值
算符
标点符号
不在上述范围内的任一非白空类字符

约束

每个要转换为单词的预处理单词应具有关键字、标识符、常量、串字面值、算符、或者标点符号的词法形式。

语义

*单词*是在翻译阶段 7 和 8 中语言的最小词法元素。共有下列几类单词：*关键字*，*标识符*，*常量*，*串字面值*，*算符*和*标点符号*。*预处理单词*是在翻译阶段 3 至 6 中语言的最小的词法元素。共有下列几类预处理单词：*前导文卷名*，*标识符*，*预处理数*，*字符常量*，*串字面值*，*算符*，*标点符号*，以及在词法上不与其他预处理单词类别匹配的单个非白空类字符。若一个'或"字符与最后一个类别匹配，则行为是未定义的。预处理单词可被*白空类符*分隔，白空类符由注释（将在后面描述），或*白空类字符*（空格、横向制表、新行、纵向制表和换页），或二者构成。如 6.8 条中所述，在翻译阶段 4 中，在一定情况下，白空类符（或其缺席）的作用不仅是分隔预处理单词。白空类符可出现在一个预处理单词之中，但必须作为前导文卷名的一部分，或出现在字符常量或串字面值中的引号字符之间。

若直至某个给定字符为止的输入流字符序列已被分析为预处理单词，则下一个预处理单词是可构成预处理单词的最长的字符序列。

示例

a. 程序片断 1Ex 将被分析为一个预处理数单词（不是一个合法的浮点或整型常量单词），即使若将其分析为预处理单词 1 和 Ex 也可能产生一个合法的表达式（例如，当 Ex 是定义为+1 的宏时）。类似地，无论 E 是否是一个宏名，程序片断 1E1 都将被分析为一个预处理数，即一个合法的浮点常量单词。

b. 尽管分析为 x++ + ++y 可能产生一个正确的表达式，程序片断 x+++++y 也将被分析为违反对增量算符的一个约束条件的 x++ ++ + y。

提前引用的条文：字符常量（6.1.3.4 条），注释（6.1.9 条），表达式（6.3 条），浮点常量（6.1.3.1 条），前导文卷名（6.1.7 条），宏替换（6.8.3 条），后缀增量与减量算符（6.3.2.4 条），前缀增量与减量算符（6.3.3.1 条），预处理指示（6.8 条），预处理数（6.1.8 条），串字面值（6.1.4 条）。

6.1.1 关键字

语法

关键字：下列英文单词之一：

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

语义

上述英文单词(全部用小写字母书写)是在翻译阶段7和8中保留作为关键字用的,不应作其他用途。

6.1.2 标识符

语法

标识符:

非数字字符

标识符 *非数字字符*

标识符 *数字字符*

非数字字符:下列字符之一:

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

数字字符:下列字符之一:

0 1 2 3 4 5 6 7 8 9

描述

标识符是由非数字字符(包括下横线和大、小写字母)及数字构成的序列,其第一个字符应是非数字字符。

约束

在翻译阶段7和8中,组成标识符的字符序列不应与组成关键字的字符序列相同。

语义

标识符表记对象、函数或下列实体(将在后面描述)之一:结构、联合或枚举的标记或成员;自定义类型名;标号名;宏名;或宏形参。枚举成员称为*枚举常量*。由于在进入程序翻译的语义阶段之前,源文卷中出现的所有宏名都将被用构成其宏定义的预处理单词替换,故在此对宏名和宏形参不作更进一步的考虑。

对标识符的最大长度无特别的限定。

对实现的限制

实现应至少将*内部名*(无外部链接的宏名或标识符)的前31个字符作为有意义的字符处理,对相应的大、小写字母按不同字母对待。实现可进一步限定*外部名*(有外部链接的标识符)的有效字符数为6个,并对外部名中字母的大小写不加区别。这些对标识符的限制均由实现定义。

注:见6.9.1条"语言的发展趋向"。

任何有效字符不一样的标识符是不同的标识符。若两个标识符中仅非有效字符不同,则其行为是未定义的。

提前引用的条文:标识符的链接(6.1.2.2条),宏替换(6.8.3条)。

6.1.2.1 标识符的作用域

标识符仅在程序文本中称为其*作用域*的区域中是*可见的*(即可用的)。共有四类作用域:函数作用域、文卷作用域、块作用域和函数原型作用域。*函数原型*是一种对函数的声明,它声明了该函数的参数

类型。)

标号名是唯一具有*函数作用域*的一种标识符。它可被用在(在 goto 语句中)它所出现的函数中的任意位置,且由其出现的语法形式(后跟一个冒号:和一条语句)隐式地声明。在一个函数中的标号名应唯一。

其他各种标识符具有由其声明在声明符或类型区分符中所处位置所确定的作用域。若声明某标识符的声明符或类型区分符出现在块或形参表之外,则该标识符具有*文卷作用域*,其作用域在翻译单位的末尾终结。若声明某标识符的声明符或类型区分符出现在块内或函数定义的形参声明表之中,则该标识符具有*块作用域*,其作用域在关闭相关块的右花括号}处终结。若声明某标识符的声明符或类型区分符出现在函数原型,而不是函数定义的一部分的形参声明表之中,则该标识符具有*函数原型作用域*,其作用域在该函数声明符的末尾终结。如果在同一命名空间中存在任何词法上相同的外层标识符,则该外层标识符在当前作用域终结前是被屏蔽的,直到当前作用域终结后才再次成为可见的。

当且仅当两个标识符的作用域在同一位置终结时,它们才具有相同的作用域。

结构、联合和枚举标记的作用域在声明该标记的类型区分符中该标记出现后即开始。枚举常量的作用域在枚举符表中定义该常量的枚举符出现后即开始。所有其他标识符的作用域在其声明符结束后开始。

提前引用的条文:复合语句或块(6.6.2 条),声明(6.5 条),枚举区分符(6.5.2.2 条),函数调用(6.3.2.2 条),函数声明符(包括原型)(6.5.4.3 条),函数定义(6.7.1 条),goto 语句(6.6.6.1 条),带标号语句(6.6.1 条),标识符的名字空间(6.1.2.3 条),宏定义的作用域(6.8.3.5 条),源文卷并入(6.8.2 条),标记(6.5.2.3 条),类型区分符(6.5.2 条)。

6.1.2.2 标识符的链接

通过称为*链接* 的处理可使在不同作用域中声明或在同一作用域中声明多次的标识符引用同一对象或函数。共有三类链接:外部链接、内部链接和无链接。

在构成程序的一组翻译单位和库中,具有*外部链接* 的特定标识符的每个实例都表记同一对象或函数。在一个翻译单位中,具有*内部链接* 的标识符的每个实例也表记同一对象或函数。*无链接* 的标识符表记唯一的实体。

若某一对象或函数的标识符声明具有文卷作用域且含有存储类区分符 static,则该标识符具有内部链接。

注:仅当函数声明具有文卷作用域时,才能包含存储类区分符 static,见 6.5.1 条。

若某一对象或函数的标识符声明含有存储类区分符 extern,则该标识符与任何有可见声明的、具有文卷作用域的相同标识符具有同样的链接。若不存在指定文卷作用域的可见声明,则该标识符具有外部链接。

若函数标识符的声明中不含存储类区分符,则其链接按如同声明为具有存储类区分符 extern 一样来决定。若对象标识符具有文卷作用域且无存储类区分符,则其链接是内部的。

下列标识符无链接:不是声明为对象或函数的标识符;声明为函数形参的标识符;声明时无存储类区分符 extern 的对象的具有块作用域的标识符。

若在一个翻译单位中,同一标识符出现为既有内部链接又有外部链接,则其行为是未定义的。

提前引用的条文:复合语句或块(6.6.2 条),声明(6.5 条),表达式(6.3 条),外部定义(6.7 条)。

6.1.2.3 标识符的名字空间

若在一个翻译单位中的任一点有多于一个对特定标识符的声明是可见的,则需通过使用与语法有关的语境来消除对引用不同实体的歧义。因此,对不同类的标识符各有不同的*名字空间*,如下:

——*标号名*(由标号声明和使用的语法来消除歧义);

——结构、联合和枚举的*标记*(通过紧跟的关键字 struct、union 或 enum 中的任何一个来消除歧

义）；

注：尽管有三种可能，但对标记只取一种名字空间。

——结构或联合的*成员*；每个结构或联合对其成员有各自的名字空间（通过以.或->算符访问成员的表达式的类型来消除歧义）；

——所有其他标识符，称为*一般标识符*（在一般声明符中声明或声明为枚举常量）。

提前引用的条文：枚举区分符（6.5.2.2条），带标号语句（6.6.1条），结构和联合区分符（6.5.2.1条），结构和联合成员（6.3.2.3条），标记（6.5.2.3条）。

6.1.2.4 对象的存储期

对象具有*存储期*，它决定对象的生存期。共有两类存储期：静态存储期和自动存储期。

其标识符被声明为具有外部或内部链接，或声明中带有存储类区分符 static 的对象具有*静态存储期*。对这类对象，将保留其存储区，且仅在程序启动前，对其存储值初始化一次。在整个程序的执行过程中，该类对象始终存在且保存其最后一次存储的值。

注：对于易变型对象，其最后一次存储的值在程序中有可能不是显式的。

其标识符被声明为无链接，或声明中没有存储类区分符 static 的对象具有*自动存储期*。对于这类对象，当正常进入与其相关的块时，或当从块外部转向块中或所包含的块中的带标号语句时，总保证为它的一个新的实例保留存储区。若对存储在该类对象中的值规定了初始化要求，则每次正常进入时执行一次初始化，但对于经转向带标号语句进入块的情况不执行初始化。不论以何种方式终止该块的执行时，对这类对象的存储区不再保证一定保留。（进入一个被包含的块将挂起但不终止包含块的执行。调用一个函数也将挂起但不终止包含该调用的块的执行。）指向不再保证一定保留的自动存储期对象的指针所引用的值是不确定的。

提前引用的条文：复合语句或块（6.6.2条），函数调用（6.3.2.2条），初始化（6.5.7条）。

6.1.2.5 类型

存储在对象中的值或由函数返回的值的意义由访问它的表达式的*type* 决定。（声明为一个对象的标识符是这种表达式中最简单的一种，其类型在该标识符的声明中规定。）类型划分为*对象类型*（描述对象的类型），*函数类型*（描述函数的类型），和*不完整类型*（描述对象的类型，但还缺少决定其尺寸所需的信息）。

声明为 char 类型的对象应足够大以存储基本执行字符集中的任何成员。若要将5.2.1条中所列举的必需的源字符集成员存储在 char 类型对象中，则它们的值保证为正值。若要将其他量值存储在 char 类型对象中，其行为由实现定义；该值将或者作为有符号整数或者作为非负整数处理。

有符号整数类型 共有四类，分别由 signed char、short int、int 和 long int 指示。（有符号整数类型和其他类型还可以用几种附加的方式来指示，将在6.5.2条中描述。）

声明为 signed char 类型的对象与“普通”char 类型对象占据一样多的存储空间。“普通”int 类型对象具有由执行环境的体系结构所建议的自然的尺寸（足够大以容纳在前导文卷〈limits.h〉中所定义的在 INT_MIN 至 INT_MAX 范围间的任何值）。在上面所列出的有符号整数类型表中，每一类型的值的值域均是表中下一类型值的子值域。

对每种有符号整数类型，均有一种对应的但是不同的*无符号整数类型*，使用关键字 unsigned 来指示，与对应的有符号整数类型使用相同数量（包括符号信息在内）的存储空间，且具有相同的对齐要求。有符号整数类型的非负值值域是对应的无符号整数类型值的子值域，且每种类型中相同值的表示法均相同。涉及无符号操作数的计算决不会出现溢出，这是因为当计算得到的结果无法用无符号整数类型表示时，将用比该结果的无符号整数类型所能表示的最大值大1的值为模，对结果求模来缩小其值。

注：相同的表示法和对齐要求意味着隐含作为函数实参、函数返回值和联合成员的互换性。

浮点类型共有三种，分别由 float、double 和 long double 指示。float 类型的值集是 double 类型值集的子集，而 double 类型的值集是 long double 类型值集的子集。

char 类型，有符号和无符号整数类型，以及浮点类型统称为*基本类型*。即使实现将两种或多种基本类型的表示法定义成一样，它们也还是不同的类型。

char、signed char 和 unsigned char 三种类型统称为*字符类型*。

枚举 由一系列的命名整型常量组成，每个个别的枚举构成不同的*枚举类型*。

void 类型由一空值集组成，它是一种永不可能补充完整的不完整类型。

由对象、函数和不完整类型可构造出任意数目的*派生类型*，如下：

——*数组类型* 描述相邻接分配的对象的非空集合，该类对象具有特定的成员对象类型，称为*元素类型*。数组类型由其元素类型和数组中元素的数目表征。称数组类型由其元素类型派生，且若其元素类型为 T，则该数组类型有时被称为“T 数组”。从元素类型构造数组类型的过程称为“数组类型的派生”。

注：由于对象类型中不包括不完整类型，因此不可能构造一个不完整类型的数组。

——*结构类型* 描述顺序分配的成员对象的非空集合，每个成员对象具有任选的指定名，且它们的类型可能不同。

——*联合类型* 描述重叠的成员对象的非空集合，每个成员对象具有任选的指定名，且它们的类型可能不同。

——*函数类型* 描述具有指定返回值类型的函数。函数类型由其返回值类型以及形参的数目和类型表征。称函数类型由其返回值类型派生，且若其返回值类型为 T，则该函数有时被称为“返回 T 的函数”。从返回值类型构造函数类型的过程称为“函数类型的派生”。

——*指针类型* 可由函数类型、对象类型、或不完整的类型派生，派生指针类型的类型称为*引用类型*。指针类型描述一类对象，该类对象的值提供对该引用类型实体的引用。由引用类型 T 派生的指针类型有时称为“(指向)T 的指针”。从引用类型构造指针类型的过程称为“指针类型的派生”。

这些构造派生类型的方法可以递归地应用。

char 类型、有符号和无符号整数类型，以及枚举类型统称为*整型*。整型的表示法应该使用纯二进制计数系统来定义值。对浮点类型的表示法未作规定。

注：整数的位置表示法使用二进制数字 0 和 1，其中由相继的二进位表示的值相加，从 1 开始，分别乘以 2 的整数次幂，有时最高二进位除外。

整型和浮点类型统称为*算术类型*。算术类型和指针类型又统称为*标量类型*。数组和结构类型则统称为*聚集类型*。

注：注意聚集类型中不包括联合类型，因为联合类型的对象一次只能容纳一个成员。

未知尺寸的数组类型是不完整类型。对该类型的标识符，当在以后的声明中(连同内部或外部链接)指定了尺寸时，即成为完整的类型。未知内容(如 6.5.2.3 条中所述)的结构或联合类型是不完整类型。当以后在同一作用域中声明相同的结构或联合标记以及所定义的内容时，对所有该类型的声明，该类型均成为完整的。

数组、函数和指针类型统称为*派生的声明符类型*。所谓从类型 T 的*声明符类型的派生* 是指通过应用数组、函数或指针类型派生为 T 而从 T 构造派生的声明符类型的过程。

类型由其*类型类别* 表征，类型类别或者是某个派生类型的最外层派生(如在上述派生类型构造中注明的)，或者当该类型不由任何派生类型组成时即为该类型本身。

迄今为止所提到的类型均是*非限定类型*。每一非限定类型都有三种与该类型相应的*限定形式*：*const(常量)限定* 形式、*volatile(易变型)限定* 形式和兼有上述两种限定的形式。某个类型的限定与非限定形式属于同一类型类别，具有相同表示和对齐要求的不同的类型。派生类型不受派生它的类型的限定词(假如存在这样的限定词时)的限定。

注：有关限定的数组和函数类型，参见 6.5.3 条。

指向 void 类型的指针应与指向字符类型的指针具有相同的表示和对齐要求。类似地，指向相容类

型的限定或非限定形式的指针也应具有相同的表示和对齐要求。指向其他类型的指针不必具有相同的表示或对齐要求。

示例

a. 由"float *"指示的类型是"指向 float 的指针"。它的类型类别是指针,而不是浮点类型。该类型的 const 限定形式由"float * const"指示。而由"const float *"指示的类型不是限定类型——该类型是"指向 const 限定的 float 的指针",是一个指向限定类型的指针。

b. 指示为"struct tag (*[5]) (float)"的类型是"指向返回 struct tag 的函数的指针数组"。该类组的长度为 5,且该函数有一类型为 float 的单个形参。其类型类别是数组。

提前引用的条文:字符常量(6.1.3.4 条),相容类型和复合类型(6.1.2.6 条),声明(6.5 条),标记(6.5.2.3 条),类型限定词(6.5.3 条)。

6.1.2.6 相容类型和复合类型

若两种类型相同,则它们为*相容类型*。附加的确定两种类型是否相容的规则分别在 6.5.2 条类型区分符、6.5.3 条"类型限定词"和 6.5.4 条"声明符"中描述。而且,对于在分开的翻译单位中声明的两种结构、联合或枚举类型,假如它们的成员个数相同,成员名相同,并且成员类型相容,则它们是相容的;对于两种相容的结构,它们的成员顺序应相同;对于两种相容的结构或联合,它们的位段宽度应相同;对于两种相容的枚举类型,它们的成员应具有相同的值。

注:两种类型相容不一定需是完全同一的。

所有引用同一对象或函数的声明应具有相容类型,否则其行为是未定义的。

从两种相容的类型可构造出*复合类型*。复合类型是与该两种类型均相容的类型,且满足下列条件:

——若一种类型是已知尺寸的数组,则复合类型是该尺寸的数组。

——若仅有一种类型是带形参表的函数类型(函数原型),则复合类型是带该形参表的函数原型。

——若两种类型都是带形参表的函数类型,则复合的形参类型表中每个形参的类型是对应形参的相容类型。

这些规则可递归地应用于派生该两种相容类型的复合类型。

对于在同一作用域中作为对某标识符的另一个声明所声明的具有内部或外部链接的标识符,该标识符的类型成为复合类型。

示例

给定下列两个文卷作用域声明:

```
int f(int (*) (),double (*) [3] );
int f(int (*) (char *), double (*) []);
```

结果得到该函数的复合类型是:

```
int f(int (*) (char *),double (*) [3]);
```

提前引用的条文:声明符(6.5.4 条),枚举区分符(6.5.2.2 条),结构和联合区分符(6.5.2.1 条),类型定义(6.5.6 条),类型限定词(6.5.3 条),类型区分符(6.5.2 条)。

6.1.3 常量

语法

常量:

浮点常量
整数常量
枚举常量
字符常量

约束

常量的值应在其类型可表示的值域范围内。

语义

每一常量均有一种类型，由其形式和值确定，将在后面详细描述。

6.1.3.1 浮点常量

语法

浮点常量：

小数部分常量　任选的指数部分　任选的浮点后缀字符

数字字符序列　指数部分　任选的浮点后缀字符

小数部分常量：

任选的数字字符序列　数字字符序列

数字字符序列

指数部分：

e　任选的符号字符　数字字符序列

E　任选的符号字符　数字字符序列

符号字符：下列字符之一：

\+　　—

数字字符序列：

数字字符

数字字符序列　数字字符

浮点后缀字符：下列字符之一：

f　l　F　L

描述

浮点常量具有一个*有效数部分*，其后可能跟一个*指数部分*和一个说明该浮点常量的类型的后缀。有效数部分的成分可包含表示整数部分的数字序列，后跟一个小数点(.)，再跟一个表示小数部分的数字序列。指数部分的成分是字母 e 或 E 后跟由符号为任选的数字序列构成的指数。整数部分或小数部分两者应出现其一，小数点或指数部分两者应出现其一。

语义

有效数部分解释为十进制有理数；指数部分的数字序列解释为十进制整数。指数指示有效数部分应乘以的 10 的幂。若最终的值在该类型可表示的值的范围内，则结果或者是最接近的可表示的值，或者是比最接近的可表示的值稍大或稍小的、邻接最接近的可表示值的值，以由实现所定义的方式选择。

无后缀的浮点常量的类型是 double。若后缀是字母 f 或 F，其类型是 float。若后缀是字母 l 或 L，则其类型是 long double。

6.1.3.2 整数常量

语法

整数常量：

十进制常量　任选的整数后缀字符

八进制常量　任选的整数后缀字符

十六进制常量　任选的整数后缀字符

十进制常量：

非零数字字符

十进制常量　数字字符

八进制常量：

字符 0

八进制常量　八进制数字字符

十六进制常量:

字符 0x 十六进制数字字符
字符 0X 十六进制数字字符
十六进制常量 十六进制数字字符

非零数字字符:下列字符之一:

1 2 3 4 5 6 7 8 9

八进制数字字符:下列字符之一:

0 1 2 3 4 5 6 7

十六进制数字字符:下列字符之一:

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

整数后缀字符:

无符号后缀字符 任选的长整数后缀字符
长整数后缀字符 任选的无符号后缀字符

无符号后缀字符:下列字符之一:

u U

长整数后缀字符:下列字符之一:

l L

描述

整数常量以数字字符开始,但无小数点或指数部分。整数常量可有规定其基数的前缀,也可有指明其类型的后缀。

十进制常量以非零数字字符开始,由十进制数字字符序列组成。八进制常量由前缀0及任选地跟着的数字字符0至7的序列组成。十六进制常量由前缀0x 或0X,后跟十进制数字字符及字母 a(或 A)至 f(或 F)组成,字母 A 至 F 分别表示十进制值10至15。

语义

十进制常量的值以10为基计算;八进制常量的值以8为基计算;十六进制常量的值以16为基计算。词法上的第一个数字是最高有效位。

整数常量的类型是下列相应表中第一个能表示其值的类型:对于无后缀的十进制整数常量为:int, long int, unsigned long int;无后缀的八进制或十六进制整数常量为:int, unsigned int, long int, unsigned long int;以字母 u 或 U 为后缀的整数常量为:unsigned int, unsigned long int;以字母 l 或 L 为后缀的整数常量为:long int, unsigned long int;同时以字母 u 或 U 和字母 l 或 L 为后缀的整数常量为:unsigned long int。

6.1.3.3 枚举常量

语法

枚举常量:

标识符

语义

声明为枚举常量的标识符的类型是 int。

提前引用的条文:枚举区分符(6.5.2.2条)。

6.1.3.4 字符常量

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字符'、反斜线字符\、或新行字符外的任何字符

转义序列

转义序列:

简单转义序列

八进制转义序列

十六进制转义序列

简单转义序列: 下列字符序列之一:

\' \" \? \\

\a \b \f \n \r \t \v

八进制转义序列:

\ *八进制数字字符*

\ *八进制数字字符 八进制数字字符*

\ *八进制数字字符 八进制数字字符 八进制数字字符*

十六进制转义序列:

\x *十六进制数字字符*

十六进制转义序列 十六进制数字字符

描述

整型字符常量是由单引号括起来的一个或多个多字节字符组成的序列,如'x'或'ab'。宽字符常量除以字母L为前缀外,与整型字符常量相同。除后面将详细描述的几种例外情况外,该序列中的元素可是源字符集中的任何成员,它们以一种实现定义的方式映射到执行字符集上。

单引号'、双引号"、问号?、反斜线\、以及任意整型值可用下列转义序列表表示:

单引号 '	\'
双引号 "	\"
问号 ?	\?
反斜线 \	\\
八进制整数	*八进制数字字符*
十六进制整数	\x*十六进制数字字符*

双引号"和问号?可以它们本身或分别以转义序列\"和\?表示,但单引号'和反斜线\则应分别以转义序列\'和\\表示。

在八进制转义序列中跟在反斜线字符后的八进制数字字符是作为整型字符常量的单个字符或宽字符常量的单个字符等构造的一部分。这样构成的八进制整数的数值规定了所期望的字符或宽字符的值。

在十六进制转义序列中跟在反斜线字符和字母x后的十六进制数字字符是作为整型字符常量的单个字符或宽字符常量的单个字符构造的一部分。这样构成的十六进制整数的数值规定了所期望的字符或宽字符的值。

每个八进制转义序列或十六进制转义序列是可构成转义序列的字符序列中最长的序列。

此外,一些特定的非图形字符可用反斜线字符后跟一个小写字母,即\a、\b、\f、\n、\r、\t和\v所构成的转义序列表示。若遇到任何其他的转义序列,则其行为是未定义的(见6.9.2条"语言的发展趋向")。

注：这些字符的语义在5.2.2条中讨论。

约束

八进制或十六进制转义序列的值对于整型字符常量应在 unsigned char 类型的表示范围之内，而对于宽字符常量则应在对应于 wchar_t 的无符号类型的表示范围之内。

语义

整型字符常量的类型是 int。包含一个映射到基本执行字符集成员的单字符的整型字符常量的值是将所映射的字符的表示解释为整数时的数值。包含多于一个字符，或包含一个不在基本执行字符集中表示的字符或转义序列的整型字符常量的值是实现定义的。若整型字符常量包含单个字符或转义序列，则它的值是当将一个其值是该单字符或转义序列的值的 char 类型的对象转换为 int 类型时所得到的值。

宽字符常量的类型是 wchar_t，wchar_t 是在前导文卷〈stddef.h〉中定义的一种整类型。包含一个映射到扩展执行字符集成员的单个多字节字符的宽字符常量的值是对应于该多字节字符的具有实现定义的当前地域环境的*宽字符*（代码），宽字符由函数 mbtowc 定义。包含多于一个多字节字符，或包含一个不在扩展执行字符集中表示的多字节字符或转义序列的宽字符常量的值是实现定义的。

示例

a. 构件'\0'通常用于表示空字符。

b. 考虑用补码表示整数，用8个二进位表示 char 类型对象的实现。若某种实现中 char 类型的值的范围与 signed char 的一样，则整型字符常量'\xFF'的值为－1；若 char 类型的值的范围与 unsigned char 的一样，则字符常量'\xFF'的值为＋255。

c. 即使在用8个二进位表示 char 类型对象的情况下，构件'\x123'也是规定只包含一个字符的整型字符常量。（该整型字符常量的值是实现定义的，且违反了上述约束。）要规定一个包含值为0x12和'3'的两个字符的整型字符常量，可使用构件'\0223'，因为十六进制转义序列仅能以非十六进制字符结束。（该两字符整型字符常量的值是实现定义的。）

d. 即使在 wchar_t 类型对象是用12个或更多个二进位表示的情况下，构件 L'\1234'也是规定实现定义的、由值0123和'4'组合后所得到的值。

提前引用的条文：字符型和整型（6.2.1.1条），公用定义库前导文卷〈stddef.h〉（7.1.6条），函数 mbtowc（7.10.7.2条）。

6.1.4 串字面值

语法

串字面值：

"任选的串字符序列"

L*"任选的串字符序列"*

串字符序列：

串字符

串字符序列 *串字符*

串字符：

源字符集中除双引号字符"、反斜线字符\、或新行字符外的任何字符*转义序列*

描述

字符串字面值是由双引号括起来的零个或多个多字节字符组成的序列，如"xyz"。宽串字面值除以字母 L 为前缀外，其他与串字面值相同。

除单引号字符可以其本身或转义序列\'表示，双引号字符应以转义序列\"表示外，对整型字符常量或宽字符常量中序列元素的描述也适用于字符串字面值或宽串字面值中序列的每个元素。

语义

在翻译阶段6，把由任何邻接的字符串字面值单词，或任何邻接的宽串字面值单词所规定的多字节

字符序列串接成一单个多字节字符序列。若一个字符串字面值单词与一个宽串字面值单词邻接，则其行为是未定义的。

在翻译阶段7，在由一个或多个串字面值或字面值得到的每个多字节字符序列的末尾添加一个值为0的字节或代码。然后用该多字节字符序列初始化静态存储期的且长度恰好足够包含该多字节字符序列的数组。对字符串字面值，该数组的元素类型为 char，用该多字节字符序列中的各个字节初始化数组的各个元素。对宽串字面值，该数组的元素类型为 wchar_t，且是用对应于该多字节字符序列的宽字符序列对数组进行初始化。

注：由于可通过\0转义序列将一个空字符嵌入到字符串字面值中，所以字符串字面值不一定需是一个串（见7.1.1条）。

对两种形式的等同的若干串字面值不必加以区别。若程序意图修改两种形式中任一种形式的串字面值，则其行为是未定义的。

示例

由于转义序列是恰在邻接的串字面值被串接前转换为单个执行字符集成员的，所以一对邻接的字符串字面值

"\x12" "3"

产生包含两个值分别为\x12和'3'的字符的单个串字面值。

提前引用的条文：公用定义库前导文卷<stddef.h>(7.1.6条)。

6.1.5 算符

语法

算符： 下列字符(或字符序列)之一：

```
[    ]    (    )    .    ->
++   --   &    *    +    -    ~    !    sizeof
/    %    <<   >>   <    >    <=   >=   ==   ^    |    &&   ||
?    :
=    *=   /=   %=   +=   -=   <<=  >>=  &=   ^=   |=
,    #    ##
```

约束

算符[]，()，和?:应成对出现，可以用表达式分隔。算符#和##应只出现在定义宏的预处理指示中。

语义

算符说明要执行的运算(*求值*)，该运算产生一个值，或一个指示符，或一个副作用，或它们的组合。*操作数*是算符所作用的实体。

提前引用的条文：表达式(6.3条)，宏替换(6.8.3条)。

6.1.6 标点符号

语法

标点符号：下列字符之一：

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

约束

标点符号[]，()，和{}在翻译阶段4之后应成对出现，可能被表达式、声明或语句分隔。标点符号#应只出现在预处理指示中。

语义

标点符号是具有独立的语法和语义含义的符号，但并不指定一个执行后获得值的运算。依赖于上下文，同一符号也可表示一个算符或某个算符的一部分。

提前引用的条文：表达式(6.3条)，声明(6.5条)，预处理指示(6.8条)，语句(6.6条)。

6.1.7 前导文卷名

语法

前导文卷名：

<前导文卷字符序列>

"引号内字符序列"

前导文卷字符序列：

前导文卷字符

前导文卷字符序列 前导文卷字符

前导文卷字符：

源字符集中除新行字符和右尖括号字符>外的任何字符

引号内字符序列：

引号内字符

引号内字符序列 引号内字符

引号内字符：

源字符集中除新行字符和双引号字符"外的任何字符

约束

前导文卷名预处理单词只应出现在#include 预处理指示中。

语义

如6.8.2条中所述，两种形式的前导文卷名序列以实现定义的方式映射到前导文卷或外部源文卷名。

若字符'、\、"或/*出现在定界符<和>之间的序列中，其行为是未定义的。类似地，若字符'、\、或/*出现在定界符"之间的序列中，则其行为也是未定义的。

注：这样，类似转义序列的字符序列导致未定义的行为。

示例

下列字符序列：

```
0x3<1/a.h>1e2
#include <1/a.h>
#define const.member@$
```

构成下列预处理单词序列(每个各别的预处理单词左边由{，右边由}定界)：

```
{0x3}{<}{1}{/}{a}{.}{h}{>}{1e2}
{#}{include}{<1/a.h>}
{#}{define}{const}{.}{member}{@}{$}
```

提前引用的条文：源文卷并入(6.8.2条)。

6.1.8 预处理数

语法

预处理数：

数字字符

.数字字符

预处理数 数字字符

预处理数 非数字字符

预处理数 e符号字符

预处理数 E符号字符

预处理数 .

描述

预处理数以一个数字字符开始，前面可任选地加一个小数点(.)，且后面可跟字母、下横线、数字字符、小数点，以及e+、e-、E+或E-字符序列。

预处理数单词词法上包含所有浮点和整数常量单词。

语义

预处理数无类型或值。在它作为翻译阶段7的一部分成功地转换为浮点常量单词或整数常量单词后，将获得类型和值。

6.1.9 注释

除出现在字符常量、串字面值或注释中外，字符对/*引入一注释。对注释内容的检查仅为了辨识多字节字符以及发现终止注释的字符对*/。

注：因此注释不会嵌套。

6.2 转换

某些算符自动将操作数的值从一种类型转换为另一种类型。本条说明这种*隐式转换*所要求的结果，以及强制(一种*显式转换*)的结果。6.2.1.5条中的表摘要说明了大部分一般算符所执行的转换，在6.3条讨论每一种算符时，还将根据需要加以补充。

将一个操作数的值转换为相容类型不改变该值或其表示。

提前引用的条文：强制(转换)算符(6.3.4条)。

6.2.1 算术操作数

6.2.1.1 字符型与整型

在表达式中可使用int或unsigned int的场合均可使用char、short int类型，int型位段，或它们的有符号或无符号变种，或枚举类型的对象。若int可以表示原始类型的所有值，则原始类型的值转换为int型；否则就转换为unsigned int型。这称为*整型升格*。整型升格不改变所有其他算术类型的值。

注：整型升格仅作为通常的算术转换的一部分而应用于某些实参表达式，一元+、-和~算符的操作数，移位算符的两个操作数，参见相应条目的说明。

整型升格维持包括符号在内的值。如前所述，是否将“普通”char型作为有符号数处理是实现定义的。

提前引用的条文：枚举区分符(6.5.2.2条)，结构与联合区分符(6.5.2.1条)。

6.2.1.2 有符号整数与无符号整数

当一种整型数的值转换为另一种整型数值时，若该值可被新类型表示，则该值不变。

当一个有符号整数转换为相同或更大尺寸的无符号整数时，若有符号整数不为负值，则该值不变。否则，若无符号整数的尺寸较大，则该有符号整数首先升格为与该无符号整数相应的有符号整数，然后通过将其值加上比该无符号整数所能表示的最大值大1的数而转换为无符号整数。

注：对于补码表示法，若无符号整数的尺寸较大，则除了在高位填充符号位之外，其他二进位模式并不改变。

当一个整数值降格为较小尺寸的无符号整数时，其结果是该值除以比该较小尺寸类型可以表示的最大无符号数大1的数后所得到的非负余数。当一个整数值降格为较小尺寸的有符号整数，或将无符号整数转换为相应的有符号整数时，若不能表示结果值，则结果将是实现定义的。

6.2.1.3 浮点型与整型

浮点型的值转换为整型时，小数部分将被忽略。若整数部分的值不能用该整类型表示，则行为是未定义的。

注：当浮点型值转换为无符号类型时，不一定需要执行整型值转换为无符号类型时所需的求余数运算，因此，可移植的浮点值的范围是(-1，无符号类型的最大值加1)。

整型值转换为浮点型时，若要转换的值在浮点型能表示的范围内而又不能确切表示时，结果将是最

接近的或者稍大或者稍小的值，以实现定义的方式选择。

6.2.1.4 浮点类型

当 float 升格为 double 或 long double，或 double 升格为 long double 时，值不变。

当 double 降格为 float，或者 long double 降格为 double 或 float 时，若要转换的值超出了可表示的范围，则行为是未定义的。若要转换的值在能表示的范围内而又不能确切表示时，结果将是最接近的或者稍大或者稍小的值，以实现定义的方式选择。

6.2.1.5 一般算术转换

许多期望算术型操作数的二元算符均以类似的方式引发转换和产生结果类型。其目的是产生一个公用类型，该类型也是结果的类型。这称为*一般算术转换*，其规则如下：

首先，若有一个操作数的类型是 long double，则另一个操作数将转换为 long double。

否则，若有一个操作数的类型是 double，则另一个操作数也将转换为 double。

否则，若有一个操作数的类型是 float，则另一个操作数也将转换为 float。

否则，对两个操作数都执行整型升格，然后应用下列规则：

若有一个操作数的类型是 unsigned long int，则另一个操作数也转换为 unsigned long int。

否则，若一个操作数的类型是 long int，另一个操作数的类型是 unsigned int，如果 long int 可以表示 unsigned int 的所有值，则将 unsigned int 类型的操作数转换为 long int；如果 lont int 不能表示 unsigned int 的所有值，则两个操作数都转换为 unsigned long int。

否则，若有一个操作数的类型是 long int，则另一个操作数也转换为 long int。

否则，若有一个操作数的类型是 unsigned int，则另一个操作数也转换为 unsigned int。

否则，两个操作数的类型都为 int。

表示浮点操作数和浮点表达式的结果的精度和范围可能比原类型所要求的精度高、范围大，所以不再改变结果的类型。

注：强制(转换)和赋值算符仍然必须执行所指定的转换，如6.2.1.3条和6.2.1.4条中所述。

6.2.2 其他操作数

6.2.2.1 左值与函数指示符

*左值*是一个指示对象的表达式，其类型是一种对象类型或除 void 外的不完整类型，当称某对象具有一种特定的类型时，该类型由用以指示该对象的左值来规定。所谓*可修改的左值*指的是满足下列条件的左值：不是数组类型、也不是不完整类型及 const 限定的类型，且若该左值是结构或联合时，其任何成员(包括递归地包含的所有结构或联合的任何成员)均不是 const 限定的类型。

注：名称"左值"最早出自赋值表达式 E1＝E2，其中左操作数 E1必须是一个(可修改的)左值。或许将左值看作是表示一个对象的"定位符值"更好。别处有时被称为"右值"的在本标准中描述为"表达式的值"。

左值的一个明显的例子是对象的标识符。作为进一步的例子，若 E 是一个一元表达式，且该表达式是指向一个对象的指针，则 * E 是指示由 E 所指向的对象的左值。

除作为 sizeof 算符、一元 & 算符、＋＋算符、－－算符的操作数，或者．算符或赋值算符的左操作数时外，不是数组类型的左值总是被转换为存储在所指示的对象中的值(于是不再成为左值)。若左值是限定类型，则该值的类型是该左值类型的非限定形式；否则该值的类型就取左值的类型。若左值是不完整类型且不是数组类型，则行为是未定义的。

除当作为 sizeof 算符或一元 & 算符的操作数，或是用于初始化字符类型数组的字符串字面值，或用于初始化其元素类型与 wchar_t 相容的数组的宽字符串字面值时外，对具有"某*type* 数组"类型的左值，将转换为一个表达式，该表达式的类型为"指向该*type* 的指针"，而该指针指向该数组的第一个元素，且不再为左值。

*函数指示符*是具有函数类型的表达式。除当作为 sizeof 算符或一元 & 算符的操作数时外，类型为"返回某*type* 的函数"的函数指示符将转换为一个表达式，该表达式的类型为"指向返回该*type* 的函数

的指针”。

注：由于不会出现这种转换，所以 sizeof 算符的操作数仍保持为函数指示符，并违反6.3.3.4条中的约束。

提前引用的条文：地址与间接算符(6.3.3.2条)，赋值算符(6.3.16条)，公用定义库前导文卷<stddef.h>(7.1.6条)，初始化(6.5.7条)，后缀增量与减量算符(6.3.2.4条)，前缀增量与减量算符(6.3.3.1条)，sizeof 算符(6.3.3.4条)，结构与联合成员(6.3.2.3条)。

6.2.2.2 void

不应以任何方式使用*void 表达式*(即类型为 void 的表达式)的实际上并不存在的值，也不应对这类表达式应用除转换到 void 类型之外的任何隐式或显式转换。若在需要 void 表达式的语境中出现了其他类型的表达式，则将忽略该表达式的值或指示符。对 void 表达式求值仅是为了获得副作用。

6.2.2.3 指针

指向 void 的指针可转换为指向任何不完整类型或对象类型的指针，反之亦然。若将指向任何不完整类型或对象类型的指针转换为指向 void 的指针后，再作反向转换，结果与原始的指针比较时应相等。

对于任何限定词 q，指向非 q 限定类型的指针可转换为指向该类型的 q 限定形式的指针；存储在原始的指针和转换后的指针中的值在比较时应相等。

值为0的整型常量表达式，或强制(转换)为 void * 类型的此类表达式，称为*空指针常量*。当将一个空指针常量赋予一个指针或与指针作比较时，将把该常量转换为指向该类型的指针。这样的指针，称为*空指针*，空指针在与指向任何对象或函数的指针作比较时保证不会相等。

注：作为空指针常量，宏 NULL 在<stddef.h>中定义，见7.1.6条。

两个空指针，即使是可能通过不同的强制(转换)序列转换为指针类型的，在比较时也应相等。

提前引用的条文：强制(转换)算符(6.3.4条)，相等类算符(6.3.9条)，简单赋值(6.3.16.1条)。

6.3 表达式

*表达式*是由算符和操作数组成的序列，它规定了对一个值的计算，或指示一个对象或函数，或产生一种副作用，或上述几种作用的组合。

在两个相继的序点之间，最多可通过对表达式求值将一个对象的存储值修改一次，而且，访问先前存储的值应仅是为了确定当时所要存储的值。

注：此段可供用于判定未定义的语句表达式，例如：

语句 i=++i+1;是未定义的，而语句 i=i+1;是允许的。

除已由语法所指示的，或后面对于函数调用算符()、&&、||、?:和逗号算符另作的规定外，对于子表达式的求值顺序以及副作用发生的顺序均未作规定。

注：语法规定了表达式求值过程中算符的优先级，它与本条中主要子条目排列的次序相同，最高优先级的算符条目先出现。因而，例如允许作为二元+算符(6.3.6条)的操作数的表达式应是在6.3.1条至6.3.6条中所定义的那些表达式。但下列情况例外：作为一元算符(6.3.3条)的操作数的强制(转换)表达式，以及包含在下列每对算符之间的操作数：组合圆括号( )(6.3.1条)、下标方括号[ ](6.3.2.1条)、函数调用括号( )(6.3.2.2条)和条件算符?:(6.3.15条)。

在每一主要子条目中的算符具有相同的优先级。在每一子条目中，由所讨论的表达式的语法指示是左结合还是右结合。

某些算符(一元算符~，二元算符≪、≫、&、^和|，它们统称为*位算符*)的操作数类型应为整型。这些算符返回的值取决于整型的内部表示，因而对于有符号类型，将受实现定义的影响。

若在表达式求值的过程中出现了某种*异常*，即若结果在数学上无定义或不在其数据类型可表示的范围内，则行为是未定义的。

仅能通过具有下列类型之一的左值才能访问对象的存储值：

——该对象所声明的类型；

——该对象所声明类型的限定形式；

——对应于该对象所声明类型的有符号或无符号类型；

——对应于该对象所声明类型的限定形式的有符号或无符号类型；

——聚集或联合类型，其成员（包括嵌套的子聚集成员或嵌套包含的联合成员）的类型中包含前面所提到的类型之一；或

——字符类型。

注：上表的作用是说明对一个对象是否可起别名的情况。

6.3.1 初等表达式

语法

初等表达式：

标识符

常量

串字面值

(表达式)

语义

只要标识符已被声明为指示一个对象（此时它是一个左值），或函数（此时它是一个函数指示符），则标识符就是一个初等表达式。

一个常量是初等表达式，其类型取决于它的形式和值，已在6.1.3条中详细描述。

一个串字面值是初等表达式，它是一个左值，其类型已在6.1.4条中详细描述。

加括号的表达式也是初等表达式。其类型和值与未加括号的表达式完全相同。若未加括号的表达式是一个左值、函数指示符或 void 表达式，则加括号的表达式也相应是一个左值、函数指示符或 void 表达式。

提前引用的条文：声明（6.5条）。

6.3.2 后缀算符

语法

后缀表达式：

初等表达式

后缀表达式 *[表达式]*

后缀表达式 *(任选的实参表达式表)*

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 *——*

实参表达式表：

赋值表达式

实参表达式表 *,赋值表达式*

6.3.2.1 下标数组

约束

其中一个表达式的类型应是“指向对象 *type* 的指针”，而另一个表达式的类型应为整型，结果的类型取“*type*”。

语义

在其后跟有一个在方括号内的表达式的后缀表达式是数组对象元素的加下标的指示。下标算符[ ]的定义是：E1[E2]与(*(E1+(E2)))完全相同。由于应用于二元+算符的转换规则，若 E1是一个数组对象（或等价地，是指向数组对象第一个元素的指针），E2是一个整数，则 E1[E2]指示 E1的第 E2个元素

(从0开始计数)。

连续的下标算符指示多维数组对象的一个元素。若E是一个n维数组(n≥2),其维数是i×j×…×k,则E(不用作左值)将转换为指向各维是j×…×k的(n−1)维数组的指针。若对此指针显式地、或作为下标应用的结果隐式地应用一元*算符,则结果是所指向的(n−1)维数组,且若结果不用作左值,则该结果也转换为一个指针。由此可见,数组是按行优先的顺序存储的,即最后一个下标变化最快。

示例

考虑由声明:

int x[3][5];

所定义的数组对象,其中x是一个3×5的int数组;更确切地说,x是有三个元素对象的数组,每个元素对象又是五个int的数组。在表达式x[i](等价于(*(x+(i))))中,x首先转换为指向第一个五个int的数组,然后按照x的类型调整i,这在概念上是将i乘以该指针所指向的对象的尺寸,在此即是五个int对象的数组的尺寸。将乘得的结果与x相加,再应用间接以得到一个五个int的数组。当用在表达式x[i][j]中时,上述结果又转换为指向第一个int的指针,于是x[i][j]得到一个int。

提前引用的条文:加减类算符(6.3.6条),地址与间接算符(6.3.3.2条),数组声明符(6.5.4.2条)。

6.3.2.2 函数调用

约束

表记被调函数的表达式的类型应是指向一个函数的指针,该函数返回void类型,或者返回除数组类型外的对象类型。

注:表记被调函数的表达式通常是对是函数指示符的标识符进行转换后的结果。

若表记被调函数的表达式的类型包含原型,则实参的数目应与形参的数目一致。每个实参的类型应使得它的值可被赋予某类对象,该类对象的类型是与实参所对应的形参类型的非限定形式。

语义

其后跟有括号( ),括号中包含可能为空的、以逗号分隔的表达式表的后缀表达式是一个函数调用。该后缀表达式表记被调函数,而括号内的表达式表规定了函数的实参。

若在函数调用中加括号的表达式表之前的表达式仅由一个标识符组成,且若对该标识符无可见的声明,则该标识符是隐式声明的,效果如同在包含该函数调用的最内部的块中出现声明

extern int 标识符( );

时一样。

注:即该标识符标识一个块作用域的,且被声明为具有外部链接,类型为无形参信息、并返回int的函数。如果事实上该函数并非定义为"返回int的函数"类型,则行为是未定义的。

实参可以是任何对象类型的表达式。在准备调用函数之前,先对实参求值,并将实参值赋给每个相对应的形参。函数调用表达式的值在6.6.6.4条中说明。

注:函数可以改变其形参的值,但这些改变不应影响实参值。另一方面,可以传递指向一个对象的指针,而函数可以改变该指针所指向的对象的值。声明为数组类型或函数类型的形参将转换为指针类型的形参,如6.7.1条所述。

若表记被调函数的表达式的类型不包含原型,则先对每个实参都执行整型升格,且把类型为float的实参升格为double。这称为*默认的实参升格*。若实参数目与形参数目不一致,则行为是未定义的。若函数定义时的类型不包含原型,而升格后的实参类型与升格后的形参类型不相容,则行为也是未定义的。若函数定义时的类型包含原型,而升格后的实参类型与升格后的形参类型不相容,或者原型以省略号(,…)结尾,则行为也是未定义的。

若表记被调函数的表达式的类型包含原型,则实参将如同通过赋值一样隐式转换为相应形参的类型。函数原型声明符中的省略号记法将使得在最后一个所声明的形参后停止对实参类型的转换,对尾部遗留的实参将执行默认的实参升格。若函数定义时的类型与由表记被调函数的表达式所指向的表达式的类型不相容,则行为也是未定义的。

除此以外，再无其他隐式执行的转换。特别是，将不把函数调用表达式中实参的数目和类型与不包含函数原型声明符的函数定义中形参的数目和类型作比较。

对函数指示符、实参及实参中子表达式的求值顺序未作规定，但在实际调用前有一个序点。

应允许函数递归调用，直接递归或通过任何其他函数调用链的间接递归均允许。

示例

在函数调用(*pf[f1()](f2(),f3()+f4())中，可按任意次序调用函数f1、f2、f3和f4。在进入由pf[f1()]所指向的函数前，所有副作用均应完成。

提前引用的条文：函数声明符(包括原型)(6.5.4.3条)，函数定义(6.7.1条)，return语句(6.6.6.4条)，简单赋值(6.3.16.1条)。

6.3.2.3 结构与联合成员

约束

.算符的第一操作数应为限定或非限定的结构或联合类型，第二操作数应命名该类型的一个成员。

->算符的第一操作数类型应为“指向限定或非限定结构的指针”或者“指向限定或非限定联合的指针”，第二操作数应命名所指向类型的一个成员。

语义

跟有一个圆点.和一个标识符的后缀表达式指示结构或联合对象的一个成员。其值是所命名成员的值，且若第一个表达式是左值，则该值也是一个左值。若第一个表达式为一种限定类型，则结果的类型取与所指示的成员类型相同限定的形式。

跟有一个向右箭头->和一个标识符的后缀表达式也指示结构或联合对象的一个成员。其值是所命名的由第一个表达式所指向的对象成员的值，且是一个左值。若第一个表达式是指向某种限定类型的指针，则结果的类型取与所指示的成员类型相同限定的形式。

注：若&E是一个有效的指针表达式，其中&是“地址”算符，它产生一个指向其操作数的指针，则表达式(&E)->MOS的作用与E.MOS相同。

有一点例外，若在已将某值存储在同一联合对象的某成员中后，再访问该联合对象另一成员的值，则行为是实现定义的。为了简化联合的使用，提供了一种特别的保证：若某联合包含几个共享一个公用的初始序列的结构，且该联合对象当前包含这些结构之一，则允许检查这些结构中任何一个的公用初始部分。当两个结构的相对于一个或多个初始成员序列的相应成员的类型相容(且对于位段，宽度也相同)时，则称它们共享*公共的初始序列*。

注：对于不滥用双关类型的独立的程序，标量类型的“字节顺序”是不可见的，但当需与外部强加的存储区布局一致时，则必须考虑。(所谓使用双关类型指的是，例如，对某联合的一个成员赋值，然后通过访问另一个成员来检查存储区，而该另一个成员是尺寸适当的字符型数组。)

示例

a. 若f是一个返回结构或联合的函数，x是该结构或联合的一个成员，则f().x是一个有效的后缀表达式，但不是左值。

b. 下面所列是一个合法的程序片断：

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

```
struct{
        int         type;
        double      doublenode;
        }nf;
}u;
/*...*/
u.nf.type=1;
u.nf.doublenode=3.14;
/*...*/
if (u.n.alltypes==1)
            /*...*/ sin(u.nf.doublenode) /*...*/
```

提前引用的条文：地址与间接算符(6.3.3.2条)，结构与联合区分符(6.5.2.1条)。

6.3.2.4 后缀增量与减量算符

约束

后缀增量或减量算符的操作数应为限定或非限定的标量类型，且应是可修改的左值。

语义

后缀＋＋算符的结果是其操作数的值。在获得结果后，操作数的值增量，即操作数的值加上适当类型的值1。有关约束、类型、转换和对指针运算的影响等信息参见对加减类算符和复合赋值的讨论。更新其操作数存储值的副作用应在前一个和下一个序点之间发生。

后缀－－算符除操作数的值减量，即减去适当类型的值1外，与后缀＋＋算符类似。

提前引用的条文：加减类算符(6.3.6条)，复合赋值(6.3.16.2条)。

6.3.3 一元算符

语法

一元表达式：

后缀表达式

＋＋一元表达式

－－一元表达式

一元算符　强制(转换)表达式

sizeof 一元表达式

sizeof(类型名)

一元算符： 下列字符之一：

& * ＋ － ~ !

6.3.3.1 前缀增量与减量算符

约束

前缀增量或减量算符的操作数应为限定或非限定的标量类型，且应是可修改的左值。

语义

前缀＋＋算符操作数的值先增量。结果是其操作数增量后的新值。表达式＋＋E 等价于(E＋＝1)。有关约束、类型、转换和对指针运算的影响等信息参见对加减类算符和复合赋值的讨论。

前缀－－算符除操作数的值减量外，与前缀＋＋算符类似。

提前引用的条文：加减类算符(6.3.6条)，复合赋值(6.3.16.2条)。

6.3.3.2 地址与间接算符

约束

一元 & 算符的操作数应是一个函数指示符或者表示一个对象的左值，该对象不是位段，且声明时

无 register 存储类区分符。

一元 * 算符的操作数应是指针类型。

语义

一元 &(地址)算符的结果是指向由其操作数所指示的对象或函数的指针。若其操作数的类型是某 *type* ,则结果的类型是"指向该 *type* 的指针"。

一元 * 算符表记间接。若其操作数指向一个函数,则结果是一个函数指示符;若其操作数指向一个对象,则结果是一个指示该对象的左值。若其操作数的类型是"指向某 *type* 的指针",则结果的类型就是该 *type* "。若已将一个无效值赋予了该指针,则一元 * 算符的行为是未定义的。

注:下列途述总是为真:若 E 是一个函数指示符或左值,且是一元 & 算符的有效操作数,则 * &E 等价于 E 的左值。若 * p 是一个左值,T 是一个对象指针类型名,则强制(转换)表达式 * (T)P 是一个左值,其类型与 T 所指向的类型相容。

由一元 * 算符解除引用的无效指针值有以下几种:空指针、与所指向的对象类型没有适当对齐的地址、以及与该对象相关联的块已终止执行后的自动存储期对象的地址。

提前引用的条文:存储类区分符(6.5.1条),结构与联合区分符(6.5.2.1条)。

6.3.3.3 一元算术算符

约束

一元+算符或一元-算符的操作数应为算术类型;一元~算符的操作数应为整型;一元!算符的操作数应为标量类型。

语义

一元+算符的结果是其操作数的值。对其操作数执行整型升格,结果取升格后的类型。

一元-算符的结果是其操作数取负。对其操作数执行整型升格,结果取升格后的类型。

一元~算符的结果是其操作数按位变反,即当且仅当转换后的操作数的相应位未置位时,结果的相应位才置位。对其操作数执行整型升格,结果取升格后的类型。表达式~E 在 E 升格为 unsigned long 类型时等价于(ULONG_MAX-E);在 E 升格为 unsigned int 类型时等价于(UINT_MAX-E)。(常量 ULONG_MAX 和 UINT_MAX 在前导文卷<limits.h>中定义。)

逻辑非!算符的结果,当其操作数的值不等于0时为0;否则为1。结果的类型取 int。表达式!E 等价于(0==E)。

提前引用的条文:限定值前导文卷<float.h>和<limits.h>(7.1.5条)。

6.3.3.4 sizeof 算符

约束

不应将 sizeof 算符应用于函数类型或不完整类型的表达式,加括号的函数类型名或不完整类型名,或指示位段对象的左值。

语义

sizeof 算符得出其操作数的(按字节计的)尺寸,其操作数可以是一个表达式,或加括号的类型名。尺寸由其操作数类型确定,并不对其操作数求值。结果是一个整数常量。

当 sizeof 算符应用于 char、unsigned char 或 signed char(或它们的限定形式)类型的操作数时,结果是1;当应用于数组类型的操作数时,结果是该数组的总字节数;当应用于结构或联合类型的操作数时,结果是这类对象的总字节数,包括内部和结尾的充填字节在内。

注:当应用于声明为数组或函数类型的形参时,sizeof 算符产生通过如6.2.2.1条所述的转换而得到的指针的尺寸,见6.7.1条。

结果值是实现定义的,结果的类型(一种无符号类型)是在前导文卷<stddef.h>中所定义的 size_t。

示例

a. sizeof 算符主要用于与诸如存储分配程序和输入输出系统等例行程序的通信。存储分配函数可能接收某个需分配的对象的按字节计的尺寸，并返回指向 void 的指针。例如：

```
extern void *alloc(size_t);
double *dp=alloc(sizeof *dp);
```

函数 alloc 的实现应保证其返回值已对齐以适合于转换为指向 double 的指针。

b. sizeof 算符的另一个用途是计算数组中元素的数目：

```
sizeof array/ sizeof array[0]
```

提前引用的条文：公用定义库前导文卷<stddef.h>(7.1.6条)，声明(6.5条)，结构与联合区分符(6.5.2.1条)，类型名(6.5.5条)。

6.3.4 强制(转换)算符

语法

强制(转换)表达式：

一元表达式

(类型名)强制(转换)表达式

约束

除非类型名规定了 void 类型，否则类型名应规定限定或非限定的标量类型，操作数也应为标量类型。

语义

在表达式之前冠以一个加括号的类型名将使表达式的值被转换为所命名的类型。这种构造称为*强制(转换)*。未规定转换类型的强制不影响表达式的类型或值。

注：强制并不产生左值。因而，强制为限定类型与强制为该类型的非限定形式作用相同。

涉及指针的转换(除由6.3.16.1条的约束所允许的外)应通过显式的强制(转换)来规定，实现定义的以及未定义的有关指针转换的内容如下：

指针可转换为整型。所要求的整型尺寸及结果是实现定义的。若所提供的空间不够，则行为是未定义的。

任何整数均可转换为指针，结果是实现定义的。

注：将指针转换为整型或将整型转换为指针的映射函数总是力图与执行环境的编址结构一致。

指向某个对象或不完整类型的指针可转换为指向不同对象类型或不同不完整类型的指针。若转换后得到的指针对于所指向的类型对齐不当，则结果不一定有效。然而，有一点是得到保证的，即指向给定对齐要求的对象的指针可转换为指向对齐要求相同或更松一些的对象的指针，并可再作反向转换，反向转换的结果与原始的指针比较时应相等。(字符型对象的对齐要求最松。)

指向某种类型函数的指针可转换为另一种类型函数的指针，并再反向转换，结果与原始的指针比较时应相等。若转换后的指针被用于调用一个类型与该被调函数不相容的函数，则行为是未定义的。

提前引用的条文：相等类算符(6.3.9条)，函数声明符(包括原型)(6.5.4.3条)，简单赋值(6.3.16.1条)，类型名(6.5.5条)。

6.3.5 乘除类算符

语法

乘除类表达式：

强制(转换)表达式

*乘除类表达式 * 强制(转换)表达式*

乘除类表达式/强制(转换)表达式

乘除类表达式%强制(转换)表达式

约束

每个操作数均应为算术类型。%算符的操作数应为整型。

语义

对操作数执行一般算术转换。

二元 * 算符的结果是其操作数的积。

/算符的结果是其第一操作数除以第二操作数所得的商；而%算符的结果是余数。在上述两种运算中，若第二操作数的值为零，则行为是未定义的。

当整数相除且不能整除时，若两个操作数均为正，则/算符的结果是小于其代数商的最大整数，且%算符的结果为正。若两个操作数均为负，则/算符的结果是小于等于其代数商的最大整数还是大于等于其代数商的最小整数由实现定义，%算符结果的符号也由实现定义。若商 a/b 是可以表示的，则表达式 (a/b) * b+a%b 应等于 a。

6.3.6 加减类算符

语法

加减类表达式：

乘除类表达式

加减类表达式+乘除类表达式

加减类表达式-乘除类表达式

约束

对于加，应是两个操作数均为算术类型，或者一个操作数是指向某对象类型的指针，而另一个操作数是整型。(增量等价于加1。)

对于减，下列之一应成立：

——两个操作数均为算术类型；

——两个操作数均为指向相容对象类型的限定或非限定形式的指针；或

——左操作数是指向一个对象类型的指针，右操作数是整型。(减量等价于减1。)

语义

若两个操作数均为算术类型，则对它们执行一般算术转换。

二元+算符的结果是其操作数之和。

二元-算符的结果是从第一操作数中减去第二操作数所得的差。

对这些算符，指向非数组对象的指针的行为与指向长度为1、以该对象类型为其元素类型的数组中第一个元素的指针相同。

当指针加上或减去一个整型表达式时，结果的类型取指针操作数的类型。若该指针操作数指向某个数组对象的元素，且该数组足够大，则结果指向一个与原始元素有一定偏移的元素，该元素与原始的元素的下标之差等于该整型表达式。换言之，若表达式 p 指向某数组对象的第 i 个元素，则表达式(p)+N(或等价地，N+(p))和(p)-N(其中 N 的值为 n)分别指向该数组对象的第 i+n 和 i-n 个元素，只要这样的元素存在。而且，若表达式 p 指向某数组对象的最后一个元素，则表达式(p)+1指向刚超越该数组对象最后一个元素的位置；若表达式 Q 指向刚超越某数组对象最后一个元素的位置，则表达式(Q)-1指向该数组对象的最后一个元素。若指针操作数和结果都指向同一数组对象的元素，或者刚超越最后一个元素的位置，则求值过程中不应产生溢出，否则行为是未定义的。除指针操作数和结果均指向同一数组对象的元素，或者指针操作数指向刚超越某数组对象最后一个元素的位置，而结果指向同一数组对象中的元素的情况外，若将结果用作一元 * 算符的操作数时，行为是未定义的。

当两个指向同一数组对象的指针相减时，结果是两个数组元素下标之差。结果的尺寸由实现定义，结果的类型(一种有符号类型)是在前导文卷<stddef.h>中所定义的 ptrdiff_t。如同其他任何算术溢出一样，若结果不能被容纳在所提供的空间中，则行为是未定义的。换言之，若表达式 P 和 Q 分别指向某数组对象的第 i 和第 j 个元素，只要值 i-j 能容纳在一个 ptrdiff_t 类型的对象中，表达式(P)-(Q)的

值就等于i—j。而且，若表达式P指向某数组对象的一个元素，或刚超越某数组对象最后一个元素的位置，且表达式Q指向同一数组对象的最后一个元素，则表达式((Q)+1)-(P)与((Q)-(P))+1及-((P)-((Q)+1))的值相同。且若表达式P指向刚超越该数组对象最后一个元素的位置，即使表达式(Q)+1并不指向该数组对象的一个元素，上述表达式的值也为零。除两个指针均指向同一数组对象的元素，或刚超越最后一个元素的位置的情况外，行为是未定义的。

注：另一种实现指针运算的方法是首先将指针转换为字符指针；以这种模式，则要与转换后的指针相加或从转换后的指针中减去的整型表达式首先乘以原先所指向的对象的尺寸，结果的指针再转换回原先的类型。对于指针相减，作为结果的两个字符指针的差也将类似地除以原先所指向的对象的尺寸。

当以这种方式看待时，实现仅需在该类对象的末尾后提供一个额外的字节(在程序中该字节可与另一个对象重叠)，以便满足"超越最后一个元素"的需求。

提前引用的条文：公用定义库前导文卷<stddef.h>(7.1.6条)。

6.3.7 逐位移位算符

语法

移位类表达式：

加减类表达式

移位类表达式≪加减类表达式

移位类表达式≫加减类表达式

约束

每个操作数均应为整型。

语义

对每个操作数均执行整型升格。结果的类型取升格后的左操作数的类型。若右操作数的值为负或大于等于升格后左操作数的二进位宽度，则行为是未定义的。

E1≪E2的结果是E1左移E2个二进位，空出的二进位用零充填。若E1为无符号类型，则结果的值是先将E1乘以2的E2次幂，然后按E1的类型是否是unsigned long，分别对ULONG_MAX+1或UINT_MAX+1求模缩小所得的值。(常量ULONG_MAX和UINT_MAX在前导文卷<limits.h>中定义)。

E1≫E2的结果是E1右移E2个二进位。若E1为无符号类型或E1为有符号类型并且值不为负，则结果的值是E1的整数部分除以2的E2次幂所得的商；若E1为有符号类型并且值为负，则结果的值是实现定义的。

6.3.8 关系类算符

语法

关系表达式：

移位类表达式

关系表达式<　移位类表达式

关系表达式>　移位类表达式

关系表达式<=移位类表达式

关系表达式>=移位类表达式

约束

下列条件之一应成立：

——两个操作数均为算术类型；

——两个操作数均为指向相容对象类型的限定或非限定形式的指针；或

——两个操作数均为指向相容的不完整类型的限定或非限定形式的指针。

语义

若两个操作数均为算术类型，则执行一般算术转换。

对于这类算符，指向非数组对象的指针的行为如同指向长度为1、以该对象类型为其元素类型的数组中第一个元素的指针一样。

将两个指针作比较时，结果取决于它们所指向的对象在地址空间的相对位置。若所指向的对象是同一聚集对象的成员，则在对指针作比较时，后声明的结构成员的指针高于在该结构中先声明的成员的指针；下标值较大的数组元素的指针高于同一数组中下标值较小的元素的指针；指向同一联合对象的成员的指针均相等。若所指向的对象不是同一聚集或联合对象的成员，则结果是未定义的，但下述情况例外：如果表达式P指向某数组对象的一个元素，表达式Q指向该数组对象中的最后一个元素，则表达式Q+1高于表达式P，尽管Q+1并不指向该数组对象的任何元素。

若两个对象或不完整类型的指针均指向同一对象，或均指向刚超越同一数组对象最后一个元素的位置，则它们比较时相等；反过来，若两个对象或不完整类型的指针比较时相等，则它们都指向同一对象，或都指向刚超越同一数组对象最后一个元素的位置。

注：如果前面已经出现无效的指针运算，例如访问数组时越界，导致了未定义的行为，则后继比较的效果也是未定义的。

算符<（小于）、>（大于）、<=（小于等于）和>=（大于等于）中的每一个在所规定的关系为真时都应得1，否则都应得0。结果的类型是int。

注：表达式a<b<c不按一般数学意义解释。正如其语法所规定的，它意味着(a<b)<c；换言之，即"若a<b则用1与c比较，否则用0与c比较"。

6.3.9 相等类算符

语法

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

约束

下列条件之一应成立：

——两个操作数均为算术类型；

——两个操作数均为指向相容对象类型的限定或非限定形式的指针；

——一个操作数是指向某对象或不完整类型指针，另一个操作数是指向Void的限定或非限定形式的指针；或

——一个操作数是指针，另一个操作数是空指针常量。

语义

==（等于）和!=（不等于）算符除优先级较低外，其余均与关系算符类似。当它们的操作数类型与值能适用于关系算符时，则可应用6.3.8条中所详细描述的语义。

注：由于优先级的关系，只要a<b和c<d的真值相同，则a<b==c<d为1。

若两个指向对象或不完整类型的指针均为空指针，则它们比较时相等。若两个指向对象或不完整类型的指针比较时相等，则它们或者都是空指针，或者都指向同一对象，或者都指向刚超越同一数组对象最后一个元素的位置。若两个指向函数类型的指针均为空指针或都指向同一函数，则它们比较时相等。若两个指向函数类型的指针比较时相等，则它们或者都是空指针，或者都指向同一函数。若其中一个运算数是指向对象或不完整类型的指针，而另一个操作数是void的限定或非限定形式的指针，则前者将转换为后者的类型。

6.3.10 按位与算符

语法

按位与表达式：

相等类表达式

按位与表达式 & *相等类表达式*

约束

每个操作数均应为整型。

语义

对操作数执行一般算术转换。

二元 & 算符的结果是其操作数的按位与，即当且仅当转换后的操作数中相应二进位均置位时，结果中相应二进位才置位。

6.3.11 按位加算符

语法

按位加表达式：

按位与表达式

按位加表达式 ^ *按位与表达式*

约束

每个操作数均应为整型。

语义

对操作数执行一般算术转换。

^ 算符的结果是其操作数的按位加，即当且仅当转换后的操作数的相应二进位中只有一个置位时，结果中相应二进位才置位。

6.3.12 按位或算符

语法

按位或表达式：

按位加表达式

按位或表达式 | *按位加表达式*

约束

每个操作数均应为整型。

语义

对操作数执行一般算术转换。

| 算符的结果是其操作数的按位或（即，当且仅当转换后的操作数的相应二进位中至少有一个置位时，结果中相应二进位才置位）。

6.3.13 逻辑与算符

语法

逻辑与表达式：

按位或表达式

逻辑与表达式 && *按位或表达式*

约束

每个操作数均应为标量类型。

语义

&& 算符在其两个操作数与0比较结果均不相等时得1，否则得0。结果的类型是 int。

与二元按位 & 算符不同，&& 算符保证按自左至右的顺序求值。在第一操作数求值后有一个序点。若第一操作数与0比较结果相等，则不再对第二操作数求值。

6.3.14 逻辑或算符

语法

逻辑或表达式:

逻辑与表达式

逻辑或表达式||逻辑与表达式

约束

每个操作数均应为标量类型。

语义

||算符在其操作数中有一个与0比较结果不相等时得1,否则得0。结果的类型是int。

与二元按位|算符不同,||算符保证按自左至右的顺序求值。在第一操作数求值后有一个序点。若第一操作数与0比较结果不相等,则不再对第二操作数求值。

6.3.15 条件算符

语法

条件表达式:

逻辑或表达式

逻辑或表达式?表达式:条件表达式

约束

第一操作数应为标量类型。

对第二和第三操作数,下列条件之一应成立:

——二者均为算术类型;

——二者均为相容的结构或联合类型;

——二者均为void类型;

——二者均为指向相容类型的限定或非限定形式的指针;

——其中之一是指针,另一个是空指针常量;或

——其中之一是指向一个对象或不完整类型的指针,另一个是指向void的限定或非限定形式的指针。

语义

首先对第一操作数求值。在其求值后有一个序点。仅当第一操作数与0比较结果不相等时才对第二操作数求值;而仅当第一操作数与0比较结果相等时才对第三操作数求值。结果将取第二或第三操作数的值(即求了值的操作数)。

注:条件表达式不产生左值。

若第二和第三操作数均为算术类型,则执行一般算术转换,以使它们都有公共的类型,结果也取该公共类型。若第二和第三操作数均为结构或联合类型,则结果就取该结构或联合类型。若第二和第三操作数均为void类型,则结果也为void类型。

若第二和第三操作数均为指针,或其中之一是空指针常量而另一个是指针,则结果的类型是指针,该指针指向的类型被两个操作数所指向类型的所有限定词所限定。而且若两个操作数都是指向相容类型、或相容类型的不同限定形式的指针,则结果取其复合类型;若一个操作数是空指针常量,则结果取另一操作数的类型;否则,若有一个操作数是void的指针或void的某种限定形式的指针,则另一操作数将转换为void的指针类型,而且结果也是指向void的指针类型。

示例

当第二和第三操作数是指针类型时,所得到的公共类型由两个互相独立的阶段决定。例如,合适的限定词并不取决于该两个指针是否具有相容类型。

给定声明:

```
const void *c_vp;
void *vp;
```

```
const int *c_ip;
volatile int *v_ip;
int *ip;
const char *c_cp;
```

下表中的第三列是条件表达式所得的公共类型，表中第一、第二列是第二和第三操作数(次序任意)：

c_vp	c_ip	const void *
v_ip	0	volatile int *
c_ip	v_ip	const volatile int *
vp	c_cp	const void *
ip	c_ip	const int *
vp	ip	void *

6.3.16 赋值算符

语法

赋值表达式：

条件表达式

一元表达式 赋值算符 赋值表达式

*赋值算符：*下列字符序列之一：

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

约束

所有赋值算符的左操作数均应为可修改的左值。

语义

赋值算符在其左操作数所指示的对象中存储一个值。赋值表达式在赋值完成后具有其左操作数的值，但不是左值。除其左操作数为限定类型的情况外，赋值表达式的类型取其左操作数的类型。对左操作数为限定类型的情况，赋值表达式的类型是其左操作数类型的非限定形式。应在前一个和下一个序点之间发生副作用，以更新其左操作数的存储值。

对操作数的求值顺序未加规定。

6.3.16.1 简单赋值

约束

下列条件之一应成立：

注：对于类型限定词的这些约束看起来不对称，是由于(在6.2.2.1条中所规定的)转换将左值改变为“表达式的值”，这种改变将从表达式的类型类别中移去任何类型限定词。

——左操作数为限定或非限定的算术类型，右操作数为算术类型；

——左操作数的类型为与右操作数类型相容的结构或联合类型的限定或非限定形式；

——两个操作数都是指向相容类型的限定或非限定形式的指针，且左操作数所指向的类型具有右操作数所指向的类型的全部限定词；

——一个操作数是指向某对象或不完整类型的指针，而另一个是指向 void 的限定或非限定形式的指针，且左操作数所指向的类型具有右操作数所指向的类型的全部限定词；或

——左操作数是一个指针，右操作数是一个空指针常量。

语义

对于*简单赋值*(=)，右操作数的值转换为赋值表达式的值，并用它替换存储在由左操作数所指示的对象中的值。

若存储在某个对象中的值通过另一个对象来访问，且后者与前者在存储区中以某种方式重叠，则重

叠应是完全的，且该两个对象应为相容类型的限定或非限定形式；否则行为是未定义的。

示例

在程序片断：

```
int f(void);
char c;
/* ... */
/* ... */((c=f())==-1)/* ... */
```

中，当由函数返回的int值存储在char中时可能会先被截断，然后在进行比较之前再转换回int的宽度。若实现时"普通"char类型与unsigned char类型的值范围相同(且char比int窄)，则转换的结果不能为负，因而，所比较的操作数决不会相等。所以，若要达到完全的可移植性，则变量c应声明为int类型。

6.3.16.2 复合赋值

约束

对于+=和-=算符，或者其左操作数应是一个指向对象类型的指针，而右操作数是整型；或者其左操作数是一个限定或非限定的算术类型，而右操作数是算术类型。

对于其他算符，每个操作数均应为与对应的二元算符所允许的类型一致的算术类型。

语义

形式为E1算符=E2的*复合赋值*与简单赋值表达式E1=E1算符(E2)的不同之处仅在于对左值E1只求值一次。

6.3.17 逗号算符

语法

表达式：

赋值表达式

表达式，*赋值表达式*

语义

逗号算符的左操作数象void表达式一样求值；在求值后有一个序点。然后对右操作数求值；结果取右操作数的类型和值。

注：逗号算符不产生左值。

示例

正如语法所指示的，在逗号作为标点符号(在函数的实参表以及初值符表中)的语境中，不能出现本条中所描述的逗号算符。另一方面，在这类语境中，逗号算符可用在加括号的表达式或条件算符的第二个表达式中。在函数调用：

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

中，函数f具有三个实参，其中第二个实参的值是5。

提前引用的条文：初始化(6.5.7条)。

6.4 常量表达式

语法

常量表达式：

条件表达式

描述

*常量表达式*可在翻译时而不是在运行时求值，因而可用在任何可出现常量的场合。

约束

除非是包含在sizeof算符的操作数中的情况，否则常量表达式中不应包含赋值、增量、减量、函数调用或逗号算符。

注：对 sizeof 算符的操作数不求值(6.3.3.4条)，因而可以使用6.3条中的任何算符。

每个常量表达式都应求值为一个常量，该常量的值在对常量表达式的类型而言是可以表示的值的范围内。

语义

在几种语境中需要求值为常量的表达式。若在翻译环境中对浮点表达式求值，则其值的算术精度和值域应至少与如果在执行环境中对该表达式求值时的精度和值域一样大。

注：必须使用整型常量表达式来规定结构的位段成员的尺寸、枚举常量的值、数组的尺寸、或 case 常量的值。对用在条件并入预处理指示中的整型常量表达式的进一步的限制在6.8.1条中讨论。

整型常量表达式 的类型应为整型，且应只使用下列类型的操作数：整数常量、枚举常量、字符常量、sizeof 表达式、以及作为强制(转换)的立即操作数的浮点表达式。除作为 sizeof 算符的操作数的一部分时外，整型常量表达式中的强制算符应只将算术类型转换为整型。

对初值符中的常量表达式允许更多的自由。这类常量表达式应求值为下列之一：

——算术常量表达式；

——空指针常量；

——地址常量；或

——某对象类型的地址常量加上或减去一个整型常量表达式。

算术型常量表达式 的类型应为算术类型，且应只使用下列类型的操作数：整数常量、浮点常量、枚举常量、字符常量、以及 sizeof 表达式。除作为 sizeof 算符的操作数的一部分时外，算术型常量表达式中的强制(转换)算符应只将算术类型转换为算术类型。

地址常量表达式 是指向左值的指针，该左值指示某静态存储期对象；或者是指向某函数指示符的指针。它应通过使用一元 & 算符显式地创建，或由使用一个数组或函数类型的表达式而隐式地创建。在创建地址常量时，可以使用数组下标[ ]、成员访问. 和 −> 算符，地址 & 和间接 * 一元算符，以及强制(转换)为指针，但不应通过使用这些算符来访问某对象的值。

实现可接受其他形式的常量表达式。

常量表达式求值的语义规则与非常量表达式的相同。

注：因此，在初始化语句

```
static int i=2 ||1/0;
```

中，表达式是一个有效的整型常量表达式，其值为1。

提前引用的条文：初始化(6.5.7条)。

6.5 声明

语法

声明：

声明区分符　任选的初值声明符表；

声明区分符：

存储类区分符　任选的声明区分符

类型区分符　任选的声明区分符

类型限定词　任选的声明区分符

初值声明符表：

初值声明符

初值声明符表，初值声明符

初值声明符：

声明符

声明符＝初值符

约束

一个声明应至少申明一个声明符、标记、或枚举的成员。

若某标识符无链接，则除在6.5.2.3条中说明的标记外，在同一作用域和同一名字空间内，对该标识符(在声明符或类型区分符中)的声明不应超过一个。

在同一作用域中，涉及同一对象或函数的所有声明均应规定相容的类型。

语义

声明 规定一组标识符的解释和属性。而同时还导致为由标识符所指名的对象或函数保留存储区的声明则是*定义*。

注：函数定义的语法不同，将在6.7.1条中描述。

声明区分符由一系列区分符和限定词组成，这些区分符和限定词指示链接、存储期、及声明符所表记的实体的类型部分。初值声明符表是由逗号分隔的声明符序列，其中每个声明符均可有附加的类型信息或初值符，或二者兼有。声明符包含所要声明的任何标识符。

若某对象的标识符声明为无链接，则在其声明符结尾时，或当有初值符时在初值声明符结尾时，其类型应是完整的，

提前引用的条文：声明符(6.5.4条)，枚举区分符(6.5.2.2条)，初始化(6.5.7条)，标记(6.5.2.3条)。

6.5.1 存储类区分符

语法

存储类区分符：

typedef

extern

static

auto

register

约束

在一个声明的声明区分符中最多只能给出一个存储类区分符。

注：见“语言的发展趋向”(6.9.3条)。

语义

称 typedef 区分符为“存储类区分符”仅是为了语法上方便；typedef 区分符将在6.5.6条中讨论。各种链接和存储期的含义已在6.1.2.2条和6.1.2.4条中描述。

对某对象标识符的声明中有存储类区分符 register 时，暗示对该对象的访问应尽可能地快。这种暗示的有效程度是实现定义的。

注：实现可将任何 register 声明简单地按 auto 声明处理。然而，无论是否实际使用了可编址的存储器，声明时有存储类区分符 register 的对象的任何部分的地址可能无法计算，不论是显式地(使用在6.3.2条中所讨论的 & 算符)，还是隐式地(通过如在6.2.2.1条中所讨论的将数组名转换为指针)。因此，对声明时有存储类区分符 register 的数组能使用的唯一算符是 sizeof。

具有块作用域的函数的标识符的声明中不应有除 extern 外的任何显式存储类区分符。

提前引用的条文：类型定义(6.5.6条)。

6.5.2 类型区分符

语法

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
自定义类型名

约束

每个类型区分符表都应为下列集合之一(当在一行中有多于一个集合时,用逗号分隔);类型区分符可以任意次序出现,并可与其他声明区分符混杂。

——void
——char
——signed char
——unsigned char
——short, signed short, short int,或 signed short int
——unsigned short,或 unsigned short int
——int, signed, signed int,或没有类型区分符
——unsigned,或 unsigned int
——long, signed long, long int,或 signed long int
——unsigned long,或 unsigned long int
——float
——double
——long double
——结构或联合区分符
——枚举区分符
——自定义类型名

语义

结构、联合和枚举的区分符在6.5.2.1条至6.5.2.3条中讨论。自定义类型名的声明在6.5.6条中讨论。其他类型的特征在6.1.2.5条中讨论。

除位段以外,上述逗号分隔的集合中的每一个都指示同一类型。类型 signed int(或 signed)有可能与 int(或没有类型区分符)不同。

提前引用的条文:枚举区分符(6.5.2.2条),结构与联合区分符(6.5.2.1条),标记(6.5.2.3条),类型定义(6.5.6条)。

6.5.2.1 结构与联合区分符

语法

结构或联合区分符:
标识结构或联合的关键字 任选的标识符{结构声明表}
标识结构或联合的关键字 标识符
标识结构或联合的关键字:
struct

union

结构声明表：

结构声明

结构声明表　结构声明

结构声明：

区分符或限定词表　结构声明符表；

区分符或限定词表：

类型区分符　任选的区分符或限定词表

类型限定词　任选的区分符或限定词表

结构声明符表：

结构声明符

结构声明符表，结构声明符

结构声明符

声明符

任选的声明符：常量表达式

约束

结构或联合中不应包含不完整类型或函数类型的成员，因此，它不应包含其自身的实例，但可包含指向其自身实例的指针。

规定位段宽度的表达式应为整型常量表达式，其值应不为负值，且不超过与之类型相容的普通对象中的二进位个数。若其值为零，则其声明中应无声明符。

语义

如在6.1.2.5条中所讨论的，结构是一种由一系列命名成员所组成的类型，各成员的存储区按顺序分配；联合也是一种由一系列命名成员所组成的类型，但其成员的存储区重叠。

结构区分符与联合区分符的形式相同。

结构或联合区分符中出现的结构声明表声明了在一个翻译单位中的一种新类型。结构声明表是一系列对结构或联合成员的声明。若结构声明表中不包含命名成员，则行为是未定义的。直至终止结构声明表的}之前，该新类型是不完整的。

结构或联合的成员可以是任何对象类型。此外，成员可被声明为由指定数量的二进位组成（若有符号位，则包括在内），这种成员称为*位段*，其宽度说明前有一个冒号。

注：一元&(地址)算符不可应用于位段对象，因此不存在位段对象的指针或位段对象数组。

位段的类型应为int、unsigned int或signed int之一的限定或非限定形式，是否将“普通”int位段的高位二进位作为符号对待由实现定义。位段被解释为由指定数量的二进位所组成的整型。

实现可分配任何足够大的可编址的存储单元以存放位段。若存放某个位段后还余足够的空间，则可将结构中紧跟该位段的另一位段放在同一单元的相邻二进位中。若余下的空间不够，则是将该另一位段放到下一单元，还是与相邻的二进位重叠由实现定义。在一个单元中位段的分配次序（是从高位至低位或是从低位至高位）也由实现定义。对可编址的存储单元的对齐未作规定。

无声明符、仅有一个冒号和一个宽度说明的位段声明指示一个未命名的位段。作为未命名位段的一种特殊情况，宽度为零的位段结构成员指示在已存放了前面已有的位段的存储单元中将不再存放其他位段。

注：未命名的位段结构成员供存储充填用，以便与外部强制的存储布局要求相一致。

每一非位段的结构对象或联合对象的成员均以某种与其类型相适当的，实现定义的方式对齐。

在一个结构对象中，非位段成员以及位段所处的单元的地址值按它们被声明的次序增加。结构对象的指针，经适当转换后，可指向其初始成员（或当该成员是位段时，指向该位段所处的单元），反之亦然。

因此，在结构对象中可能有未命名的充填，以便必要时获得适当的对齐，但充填不会出现在结构的始端。

联合的尺寸应足够容纳其最大的成员。任一时刻，在联合对象内最多只能存储其一个成员的值。联合对象的指针，经适当转换后，指向其每个成员，或当某成员是位段时，指向该位段所处的单元。反之亦然。

若结构或联合是某数组的元素时，在结构或联合的末尾仍然可以有未命名的充填，以便必要时获得适当的对齐。

提前引用的条文：标记(6.5.2.3条)。

6.5.2.2 枚举区分符

语法

枚举区分符：

enum *任选的标识符{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：

枚举常量

枚举常量=常量表达式

约束

定义枚举常量值的表达式应为整型常量表达式，其值应为 int 型可表示的值。

语义

枚举符表中的标识符被声明为类型为 int 型的常量，它们可出现在任何允许 int 的场合。带有=的枚举符其枚举常量定义为枚举表达式的值。若第一个枚举符中无=，则其枚举常量的值为零。每个后继的无=的枚举符定义其枚举常量为前一个枚举常量值加1所得到的常量表达式的值。(使用带=的枚举符可产生其值与同一枚举中其他值重复的枚举常量。)枚举中的枚举符也称为其成员。

注：因而，在同一作用域中声明的枚举常量的标识符都应互不相同，并与普通声明符中所声明的其他标识符也不相同。

每个枚举的类型均应与一种整类型相容；具体类型的选择是实现定义的。

示例

```
enum hue { chartreuse, burgundy, claret=20, winedark };
/* ... */
enum hue col, *cp;
/* ... */
col=claret;
cp=&col;
/* ... */
/* ... */ (*cp !=burgundy) /* ... */
```

使 hue 成为一个枚举的标记，然后声明 col 为一个该类型的对象，cp 为指向一个该类型对象的指针。所枚举的值包含在集合{0,1,20,21}中。

提前引用的条文：标记(6.5.2.3条)。

6.5.2.3 标记

语义

形式为：

标识结构或联合的关键字　标识符〈*结构声明符表*〉或

enum*标识符*〈*枚举符表*〉

的类型区分符声明该标识符为该结构、联合或表中所规定的枚举的*标记*。该表定义了*结构内容*、*联合内容*或*枚举内容*。若此标记的声明是可见的，则后继的使用省略加花括号的表的此标记就声明了规定明的结构、联合或枚举类型。在同一作用域中的后继声明中应省略加括号的表。

若形式为：

标识结构或联合的关键字　标识符

的类型区分符出现在定义内容的声明之前，则该结构或联合是一个不完整类型[1]。它声明了一个规定某种类型的标记，仅当不需要所指定类型对象的尺寸时才可使用该标记[2]。要使该类型完整，则在同一作用域中应有另一个有关该标记的声明定义其内容(但不应是在某个所包含的块中声明，因为那样将声明一种仅在该块内有效的新类型)。形式为：

标识结构或联合的关键字　标识符；

的声明规定了一种结构或联合类型，并声明了一个标记，二者都是仅在出现该声明的作用域中是可见的。它规定了一种与在包含它的作用域中使用相同标记的任何类型有区别的新类型。

注：1) 对 enum 不存在类似的构造，且也是不必要的，因为在枚举类型与任何其他类型的声明之间不可能有相互依赖关系。

2) 例如，当一个自定义类型名被声明为一种结构或联合的区分符时，或在声明一个指向一个结构或联合的指针或声明一个返回结构或联合的函数时，并不需要所规定类型对象的尺寸(见6.1.2.5条中不完整类型)。在调用或定义上述函数之前，规格说明应完整。

形式为：

标识结构或联合的关键字　〈结构声明符表〉

或

enum*〈枚举符表〉*

的类型区分符规定一种新的结构、联合或枚举类型，在该翻译单位中，这些类型仅能被包含该类型的声明所引用。

注：当然，当声明是自定义类型名时，后继的声明可使用该自定义类型名来声明具有所规定的结构、联合或枚举类型的对象。

示例

a. 这种机制允许声明自引用的结构。

```
struct tnode {
    int count;
    struct tnode *left, *right;
};
```

规定一种结构，该结构包含一个整数和两个指向相同(结构)类型对象的指针。一旦给出了此声明，则声明：

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

声明了 s 为一个具有给定类型的对象，sp 是一个指向给定类型对象的指针。在给出这些声明的情况下，表达式 sp->left 指的是由 sp 所指向的对象的左 struct tnode 指针；表达式 s.right->count 指示在 s 的右 struct tnode 所指对象的 count 成员。

下列替代的形式使用了 typedef 机制：

```
typedef struct tnode TNODE;
struct tnode {
    int count;
```

```
    TNODE *left, *right;
};
TNODE s, *sp;
```

b. 为说明如何通过先声明标记以规定一对互相引用的结构，下例中的声明：

```
struct s1 {struct s2, *s2p;/* ... */};/* D1 */
struct s2 {struct s1, *s1p;/* ... */};/* D2 */
```

规定了一对结构，它们都包含指向对方的指针。然而请注意：若 s2是早已在包含此声明的作用域中所声明的标记，则声明 D1所引用的将是该标记，而不是在 D2中声明的标记 s2。为消除这种与语境的密切联系，可在 D1之前插入声明：

```
struct s2;
```

它在内层作用域范围内声明一个新的标记 s2；然后声明 D2使该新类型的规格说明完整。

提前引用的条文：类型定义(6.5.6条)。

6.5.3 类型限定词

语法

类型限定词：

const

volatile

约束

无论是直接出现还是通过一个或多个 typedef 出现，同一类型限定词不应在同一区分符表或限定词表中出现多次。

语义

与限定类型相关联的性质仅对是左值的表达式才有意义。

注：实现可将非 volatile 类型的 const 对象放置在只读存储区中。而且，如果从不使用它，则实现可不必为这样的对象分配存储区。

若试图通过使用非 const 限定类型的左值去修改定义为 const 限定类型的对象，则其行为是未定义的。

若试图通过使用非 volatile 限定类型的左值引用定义为 volatile 限定类型的对象，其行为也是未定义的。

注：这也适用于其行为如由限定类型定义的那些对象，即使实际上从未将它们定义为程序中的对象(例如存储映射的输入输出地址)。

volatile 限定类型的对象可用实现所未知的方式修改，或具有其他未知的副作用。所以引用这类对象的任何表达式应严格按抽象机的规则求值，如5.1.2.3条中所述。而且，在每一序点，除被前述的未知因素修改了之外，在对象中上次存储的值应与抽象机所规定的一致。由什么构成对易变限定类型的对象的访问是实现定义的。

注：volatile 声明可用于描述相应于存储映射的输入输出端口的对象，或由异步中断函数访问的对象。除对表达式求值的规则所允许的以外，实现不应对声明为 volatile 的对象的动作进行任何“优化”或重新排序。

若数组类型的规格说明中包含任何类型限定词，则它是对其元素类型的限定，而不是对数组类型的限定。若函数类型的规格说明中包含任何类型限定词，则其行为是未定义的。

注：这两种情况都只有通过使用 typedef 才会出现。

为使两个限定类型相容，则二者都应为相容类型的完全相同的限定形式；在区分符表或限定词表中的类型限定词的次序并不影响所规定的类型。

示例

a. 声明为：

```
extern const volatile int real_time_clock;
```

的对象可由硬件修改，但不能被赋值、增量或减量。

b. 下列声明与表达式说明了类型限定词修改聚集类型时的行为：

```
const struct s {int mem;} cs={ 1 };
struct s ncs;/*对象 ncs 是可修改的 */
typedef int A[2][3];
const A a={{4,5,6},{7,8,9}};/*const int 数组的数组 */
int *pi;
const int *pci;
ncs=cs;          /*有效 */
cs=ncs;          /*违反对=的可修改的左值约束 */
pi=&ncs.mem;     /*有效 */
pi=&cs.mem;      /*违反对=的类型约束 */
pci=&cs.mem;     /*有效 */
pi=a[0];         /*无效:a[0]的类型为"const int *" */
```

6.5.4 声明符

语法

声明符：

任选的指针 直接声明符

直接声明符：

标识符

(声明符)

直接声明符 [任选的常量表达式]

直接声明符 (形参类型表)

直接声明符 (任选的标识符表)

指针：

** 任选的类型限定词表*

** 任选的类型限定词表 指针*

类型限定词表：

类型限定词

类型限定词表 类型限定词

形参类型表：

形参表

形参表,…

形参表：

形参声明

形参表,形参声明

形参声明：

声明区分符 声明符

声明区分符 任选的抽象声明符

标识符表：

标识符

标识符表,标识符

语义

每个声明符声明一个标识符，并表明当某个表达式中出现与该声明符形式相同的操作数时，则该操作数指示一个函数或对象，它们具有由声明区分符所指出的作用域、存储期及类型。

在下面的子条目中，考虑声明：

T D1

其中T包含规定一个类型*T*（例如int）的声明区分符，D1是一个声明符，它包含一个标识符*ident*。在各种形式的声明符中对标识符*ident*的类型区分符均用此记法归纳描述。

若在声明"T D1"中，D1的形式为：

标识符

则对*ident*所规定的类型是*T*。

若在声明"T D1"中，D1的形式为：

(D)

则*ident*具有由声明"T D"所规定的类型。因此，在括号内的声明符与不加括号的声明符完全相同，但复杂声明符的结合可用括号来改变。

对实现的限制

实现应允许类型的规格说明中至少有（或者直接地，或者通过一个或多个typedef）12个指针、数组、和修饰算术的函数声明符（以任意有效的组合）、一个结构、一个联合或一个不完整类型。

提前引用的条文：类型定义（6.5.6条）。

6.5.4.1 指针声明符

语义

若在声明"T D1"中，D1的形式为：

* 任选的类型限定词表 D

且在声明"T D"中对*ident*所规定的类型是"*派生的声明符类型表 T*"，则对*ident*所规定的类型是"*派生的声明符类型表 类型限定词表 T*的指针"。对表中每一限定词，*ident*都是一个这样限定的指针。

若要两个指针类型相容，则二者的限定应完全相同，且应都为相容类型的指针。

示例

下列两个声明示范说明了"指向常量值的变量指针"与"指向变量值的常量指针"之间的差别：

```
const int *ptr_to_constant;
int *const constant_ptr;
```

由ptr_to_constant所指向的对象的内容不应通过该指针而被修改，但ptr_to_constant本身可改为指向另一个对象。类似地，由constant_ptr所指向的int的内容可被修改，但constant_ptr本身应总是指向同一位置。

常量指针constant_ptr的声明可通过加入一个对类型"指向int的指针"的定义而更清楚：

```
typedef int *int_ptr;
const int_ptr constant_ptr;
```

上述声明说明constant_ptr是一个类型为"指向int的const限定的指针"的对象。

6.5.4.2 数组声明符

约束

由[和]定界的表达式（该表达式规定数组的尺寸）应为值大于零的整型常量表达式。

语义

若在声明"T D1"中，D1的形式为：

D[*任选的常量表达式*]

且在声明"T D"中对*ident*所规定的类型是"*派生的声明符类型表 T*"，则对*ident*所规定的类型是

"*派生的声明符类型表 T* 的数组"。若未给出尺寸，则该数组的类型是不完整类型。

注：当几个"array of"规格说明邻接一起时，则是声明一个多维数组。

若要两个数组类型相容，则二者应有相容的元素类型，且若对二者都给出了尺寸说明符，则它们应具有相同的值。

示例

a. float fa[11], *afp[17]

声明了一个 float(浮点)数的数组和一个指向 float 数的指针的数组。

b. 注意声明：

```
extern int *x;
```

和

```
extern int y[];
```

之间的区别。第一个声明 x 为指向 int 的指针；而第二个声明 y 是一个尺寸未指定的 int 数组(不完整类型)，它所用到的存储区将在其他地方定义。

提前引用的条文：函数定义(6.7.1条)，初始化(6.5.7条)。

6.5.4.3 函数声明符(包括原型)

约束

函数声明符不应指定返回类型为函数类型或数组类型。

能在形参声明中出现的唯一存储类区分符是 register。

在不是函数定义的组成部分的函数声明符中的标识符表应为空。

语义

若在声明"T D1"中，D1的形式为：

D(*形参类型表*)

或

D(*任选的标识符表*)

且在声明"T D"中对 *ident* 所规定的类型是"*派生的声明符类型表 T*"，则对 *ident* 所规定的类型是"*派生的声明符类型表* 返回 *T* 的函数"。

形参类型表说明了该函数的形参类型，并可声明该函数形参的标识符。若该表以省略号(,...)终结，则对逗号后的形参的数目及类型不提供任何信息。表中唯一的表项为 void 时的特殊情况说明该函数无形参。

注：在前导文卷<stdarg.h>(7.8条)中定义的宏可用于访问对应于省略号的实参。

在形参声明中，在括号中的单个自定义类型名是作为一个抽象声明符，它说明一个单形参的函数，而不应看作是围绕某声明符的标识符的冗余的括号。

若形参声明的声明区分符中存在存储类区分符，则将被忽略，除非所声明的形参是某函数定义的参数类型表中的一个成员。

标识符表仅声明该函数的形参标识符。属于某函数定义的一部分的函数声明符中标识符表为空说明该函数无形参。而不属于某函数定义的一部分的函数声明符中标识符表为空则说明对形参的数目及类型未提供任何信息。

注：见"语言的发展趋向"(6.9.4条)。

若要两个函数类型相容，则二者都应说明相容的返回类型。而且，若二者都有形参类型表，则其中形参的数目以及是否使用省略号作终止符都应一致，且对应的形参应为相容类型。若其中一个类型有参数类型表，而另一个类型由不属于某函数定义的一部分的函数声明符说明，且该函数声明符中包含空的标识符表，则前者的形参表不应使用省略号作终止符，且每一形参的类型都应与应用默认的实参升格后所得的类型相容。若其中一个类型有形参类型表，而另一个类型由包含一个可能为空的标识符表的函数定

义说明，则二者的形参数目应一致，且每一原型形参的类型都应与对对应的标识符类型应用默认的实参升格后所得的类型相容。(对每个声明为函数类型或数组类型的形参，作这些比较时将使用把参数类型转换为指针类型后的结果，如6.7.1条所述。对每个声明为限定类型的形参，作这些比较时将使用所声明的形参类型的非限定形式。)

注：若两个函数的类型都是"旧风格"，则不对形参类型作比较。

示例

a. 声明：

```
int f(void), *fip( ),(*pfi)( );
```

声明了一个无形参、返回 int 的函数 f；一个无形参规格说明、返回一个指向 int 的指针的函数 fip；和一个指向无形参规格说明、返回 int 的函数的指针 pfi。对后二者作一下比较将是特别有用的。*fip 的结合是 *(fip( ))，因此，该声明揭示了(表达式也要求同样的构造)先调用函数 fip，然后通过指针结果使用间接得到一个 int。在声明符(*pfi)( )中，额外的括号是必要的，以便指示通过指向函数的指针间接而得到一个函数指示符，然后用该指示符调用该函数，其返回类型是 int。

若声明出现在任何函数之外，则该标识符具有文卷作用域以及外部链接。若声明出现在函数内部，则函数 f 和 fip 具有块作用域，内部或者外部链接(对这些标识符而言，取决于哪些文卷作用域声明是可见的)。指针 pfi 的标识符具有块作用域且无链接。

b. 声明：

```
int(*apfi[3])(int *x,int *y);
```

声明了一个数组 apfi，它包含三个指向返回 int 的函数的指针。每个函数有两个形参，它们都是指向 int 的指针。声明标识符 x 和 y 仅是为了描述目的，在 apfi 声明的末尾就不再有实际意义。

c. 声明：

```
int (*fpfi(int (*)(long),int))(int,…);
```

声明了一个函数 fpfi，它返回一个指针，该指针指向一个返回 int 的函数。函数 fpfi 有两个形参：一个是指针，它指向一个返回 int 的函数(该函数有一个形参，其类型为 long)，另一个是 int。由 fpfi 返回的指针指向一个函数，该函数有一个 int 类型的形参，并还可接受零个或多个任何类型的形参。

提前引用的条文：函数定义(6.7.1条)，类型名(6.5.5条)。

6.5.5 类型名

语法

类型名：

区分符或限定词表　任选的抽象声明符

抽象声明符：

指针

任选的指针　直接抽象声明符

直接抽象声明符：

(抽象声明符)

任选的直接抽象声明符　[任选的常量表达式]

任选的直接抽象声明符　(任选的形参类型表)

语义

在某些语境中希望说明某种类型，这可通过使用类型名来实现。类型名在语法上是一种对该类型的函数或对象的省略标识符的声明。

注：正如由语法所指示的，类型名中的空括号被解释为"无形参规格说明的函数"，而不是所省略的标识符两边的冗余的括号。

示例

以下构造：

a. int

b. int *

c. int *[3]

d. int (*)[3]

e. int *( )

f. int(*)(void)

g. int(*const[ ])(unsigned int,…)

分别命名了下列类型：a. int，b. 指向 int 的指针，c. 三个指向 int 的指针的数组，d. 指向三个 int 数组的指针，e. 无形参规格说明、返回指向 int 的指针的函数，f. 指向无形参、返回 int 的函数的指针，以及 g. 一个未指定元素数目的数组，其元素是指向函数的常量指针，每个函数都有一个形参的类型是 unsigned int 的形参，而对其他形参的数目未加指定，每个函数都返回 int。

6.5.6 类型定义

语法

自定义类型名：

标识符

语义

在存储类区分符为 typedef 的声明中，每个声明符将一个标识符定义为自定义类型名，该自定义类型名按6.5.4条中所述的方式说明对该标识符所规定的类型。typedef 声明并不引入一个新类型，仅对所指明的类型给出一个同义词。即，在下列声明中：

```
typedef T type_ident;
type_ident D;
```

type_ident 定义为一个自定义类型名，其类型由 T 的声明区分符指定(已知是 *T*)，且 D 中的标识符的类型为"*派生的声明符类型表 T* "，其中*派生的声明符类型表* 由 D 的声明符说明。自定义类型名与在一般声明符中所声明的其他标识符共享同一个名字空间。若该标识符在内层作用域被重新声明，或者在同一作用域或内层作用域被声明为一个结构或联合的成员，则在内层声明中不应省略类型区分符。

示例

a. 在声明：

```
typedef int MILES, KLICKSP( );
typedef struct {double re, im;}complex;
```

之后，下列构造：

```
MILES distance;
extern KLICKSP *metricp;
complex x;
complex z, *zp;
```

都是有效的声明。distance 的类型是 int，metricp 的类型是"指向无形参规格说明、返回 int 的函数的指针"，x 和 z 的类型是所规定的结构，zp 是一个指向这种结构的指针。对象 distance 的类型与任何其他 int 对象相容。

b. 在声明：

```
typedef struct s1{int x;}t1, *tp1;
typedef struct s2{int x;}t2, *tp2;
```

之后，类型 t1 与由 tp1所指向的类型相容。类型 t1也与类型 struct s1相容，但与类型 struct s2、t2、由 tp2 所指向的类型，以及 int 不相容。

c. 下列晦涩的构造：

```
typedef signed int t;
typedef int plain;
struct tag{
    unsigned t:4;
    const t:5;
    plain r:5;
};
```

声明一个类型为 signed int 的自定义类型名 t，一个类型为 int 的自定义类型名 plain，和一个有三个位段成员的结构，一个命名为 t，它的值在范围[0,15]内，一个未命名的、const 限定的位段，若可对它访问则其值至少在范围[−15,+15]内，一个命名为 r，它的值在范围[0,31]内或至少在范围[−15,+15]内（范围的选择由实现定义）。前两个位段声明的差别在于：unsigned 是一个类型区分符（它强制 t 成为一个结构成员的名），而 const 是一个类型限定词（它修饰 t，而 t 作为自定义类型名仍然是可见的）。

若上述声明后在内层作用域中跟有：

```
t f(t(t));
long t;
```

则声明了一个函数 f，其类型为“返回 signed int 的函数，它有一个未命名的形参，该形参的类型是指向一个返回 signed int 的函数的指针，所指向的函数有一未命名的、类型为 signed int 的形参”，以及一个类型为 long 的标识符 t。

d. 另一方面，自定义类型名可用于改善代码的可读性。所有下列三个对函数 signal 的声明确实都说明相同的类型，而第一个未使用任何自定义类型名。

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

提前引用的条文：函数 signal(7.7.1.1条)。

6.5.7 初始化

语法

初值符：

赋值表达式

{初值符表}

{初值符表,}

初值符表：

初值符

初值符表,初值符

约束

在初值符表中的初值符的数目不应超过需初始化的对象数目。

需初始化的实体类型应是对象类型或未知尺寸的数组。

对具有静态存储期的对象的初值符中的表达式，或对聚集或联合类型对象的初值符中的表达式均应为常量表达式。

若某标识符的声明具有块作用域，且该标识符具有外部或内部链接，则该声明不应有对该标识符的初值符。

语义

初值符规定存储在对象中的初始值。

在初始化过程中将忽略所有未命名的结构或联合成员。

若未显式初始化一个自动存储期的对象，则其值不确定。若未显式初始化一个静态存储期的对象，则将隐式地对它初始化，其效果如同对每个算术类型的成员都赋于零、对每个指针类型的成员都赋于空指针常量一样。

注：与基本文件不同，任何自动存储期对象均可被初始化。

对标量的初值符应为任选地括在花括号内的单个表达式。该对象的初始值是该表达式的值，并将该标量的类型取作声明类型的非限定形式，应用与简单赋值相同的类型约束和转换。

用于联合对象的用花括号括起来的初值符初始化在联合类型的声明表中出现的第一个成员。

用于自动存储期的结构或联合类型的初值符应或者为一个将在下面描述的初值描述表，或者为单个具有相容结构或联合类型的表达式。在后者的情况下，对象的初始值就是该表达式的值。

本条的其余部分将针对聚集或联合类型对象的初值符。

字符型数组可用任选地括在花括号内的字符串字面值初始化。该字符串字面值中相继的字符初始化该数组(若数组还有空间，或者未规定尺寸时，包括终止字符串字面值的空字符在内)的元素。

元素类型与 wchar_t 相容的数组可用任选地括在花括号内的宽串字面值初始化。该宽串字面值中相继的代码初始化该数组(若数组还有空间，或者未规定尺寸时，包括终止宽串字面值的零值代码在内)的元素。

否则，对聚集类型对象的初值符应为一个括在花括号内的、用于该聚集成员的初值符表，表中的初值符按下标或成员次序递增的顺序书写；对联合类型对象的初值符应为一个括在花括号内的、用于该联合的第一个成员的初值符。

若聚集自身包含聚集或联合类型的成员，或者联合的第一个成员是聚集或联合，则上述规则将递归地应用于子聚集或所包含的联合。若子聚集或所包含的联合的初值符以左花括号开始，则用该左花括号和与之匹配的右花括号括起来的初值符初始化该子聚集成员或所包含的联合的第一个成员。否则，将仅考虑使用表中足够的初值符初始化该子聚集成员或所包含的联合的第一个成员；表中任何余下的初值符将用于初始化包含聚集(当前的该子聚集或所包含的联合是该聚集的组成部分)中的下一个成员。

若括在花括号内的表中的初值符少于要初始化的聚集的成员数目，则该聚集中剩余的成员将象静态存储期的对象一样被隐式地初始化。

初始化未知尺寸的数组时，该数组的尺寸由为其提供的初值符的数目确定。在其初值符表的末尾，该数组不再是不完整类型。

示例

a. 声明：

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

定义并初始化 x 为一个具有三个元素的一维数组对象，因为未规定尺寸而有三个初值符。

b. 声明：

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

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

是带完全加花括号的初始化说明的定义：1、3和5初始化 y 的第一行(数组对象 y[0])，即 y[0][0]、y[0][1]和 y[0][2]。同样地，初始化说明中下两行初始化 y[1]和 y[2]。由于初值符结束得较早，所以 y[3]被初始化为零。通过

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

可获得完全相同的效果。对 y[0]的初值符不以左花括号开始，所以使用表中三个表项。同样地，对于 y[1]和 y[2]，分别用表中下三个相继的表项。

c. 声明：

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

按所说明的初始化 z 的第一列，并将其他元素初始化为零。

d. 声明：

```
struct {int a[3],b;}w[ ]={{1},2};
```

是一个定义，且它所带有的初始化说明中所加花括号不一致。它定义了一个有两个结构元素的数组：w[0].a[0]是1，w[1].a[0]是2，所有其他元素都是零。

e. 声明：

```
short q[4][3][2]={
      {1},
      {2,3},
      {4,5,6}
};
```

包含不完全的，然而所加花括号是一致的初始化说明。它定义了一个三维数组对象：q[0][0][0]是1，q[1][0][0]是2，q[1][0][1]是3；4、5和6分别初始化 q[2][0][0]、q[2][0][1]和 q[2][1][0]；所有其他元素是0。对 q[0][0]的初值符不以左花括号开始，所以可在当前表中使用直至六个表项。但当前表中只有一个表项，所以其余五个元素的值初始化为0。同样地，对 q[1][0]和 q[2][0]的初值符也不以左花括号开始，所以也可在当前表中使用直至六个表项，以分别初始化它们的二维子聚集。假设在上述任何表中有多于六个表项，则将发布一个诊断消息。同样的初始化结果可通过：

```
short q[4][3][2]={
    1,0,0,0,0,0,
    2,3,0,0,0,0,
    4,5,6
};
```

得到，或者通过：

```
short q[4][3][2]={
    {
        {1},
    },
    {
        {2,3},
    },
    {
        {4,5},
        {6},
    }
```

```
};
```

得到，这是一种使用完全的花括号的形式。

注意，一般情况下，使用完全的花括号的形式或最简花括号的形式最不容易引起混淆。

f. 一种使数组类型完整化的初始化形式涉及自定义类型名。给定声明：

```
typedef int A[ ];
```

根据不完整类型的规则，声明：

```
A a={1,2},b={3,4,5};
```

等价于

```
int a[ ]={1,2},b[ ]={3,4,5};
```

g. 声明：

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

定义了"普通"char 数组对象 s 和 t，它们的元素用字符串字面值初始化。此声明与

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

完全相同。上述数组的内容是可修改的。另一方面，声明：

```
char *p="abc";
```

定义 p 的类型为"指向 char 的指针"，它被初始化为指向一个对象，该对象的类型为"char 数组"，长度为 4，其元素用一字符串字面值初始化。若试图使用 p 去修改该数组的内容，则行为是未定义的。

提前引用的条文：公用定义库前导文卷<stddef.h>(7.1.6条)。

6.6 语句

语法

语句：

带标号语句
复合语句
表达式语句
选择语句
循环语句
跳转语句

语义

语句 规定所要执行的动作。除非特别规定外，语句均按顺序执行。

完全表达式 是这样的一种表达式，它不是另一表达式的组成部分。下面所列出的每一种都是完全表达式：初值符；表达式语句中的表达式；选择(if 或 switch)语句中的控制表达式；while 或 do 语句中的控制表达式；for 语句中三个(任选的)表达式中的每一个；return 语句中的(任选的)表达式。完全表达式的末尾是一个序点。

提前引用的条文：表达式语句与空语句(6.6.3条)，选择语句(6.6.4条)，循环语句(6.6.5条)，return 语句(6.6.6.4条)。

6.6.1 带标号语句

语法

带标号语句：

标识符：*语句*
case *常量表达式*：*语句*
default：*语句*

约束

标号 case 或 default 仅能出现在 switch 语句中。对这样的标号的进一步的约束在 switch 语句的条文中讨论。

语义

任何语句前都可加一个将某一标识符声明为标号名的前缀。标号本身不改变控制流，控制流将不受妨碍地越过标号。

提前引用的条文：goto 语句(6.6.6.1条)，switch 语句(6.6.4.2条)。

6.6.2 复合语句或块

语法

复合语句：

{任选的声明表 任选的语句表}

声明表：

声明

声明表 声明

语句表：

语句

语句表 语句

语义

复合语句(也称为*块*)允许将一系列语句组合成一个语法单位，该语法单位可以有其自己的一组声明和初始化说明(如在6.1.2.4条中所讨论的)。将对自动存储期对象的初值符求值，并按在翻译单位中它们的声明符所出现的次序将值存入该对象。

6.6.3 表达式语句与空语句

语法

表达式语句：

任选的表达式；

语义

表达式语句中的表达式将按 void 表达式求值以获得其副作用。

注：例如赋值以及具有副作用的函数调用等。

空语句 仅由一个分号组成，不执行任何操作。

示例

a. 若仅为其副作用而将一个函数调用作为表达式语句求值，可通过强制(转换)将该表达式转换为 void 表达式而显式地表明不需要表达式的值：

```
int p(int);
/* … */
(void)p(0);
```

b. 在程序片断：

```
char *s;
/* … */
while (*s++ != '\0')
        ;
```

中，使用了一个空语句为循环语句提供空的循环体。

c. 在复合语句的闭}前也可使用空语句来插入一个标号：

```
while (loop1){
    /* … */
```

```
        while (loop2){
            /* … */
            if (want_out)
                goto end_loop1;
            /* … */
        }
        /* … */
    end_loop1:;
    }
```

提前引用的条文：循环语句(6.6.5条)。

6.6.4 选择语句

语法

选择语句：

if(表达式)语句

if(表达式)语句 else 语句

switch(表达式)语句

语义

依据控制表达式的值，选择语句在一系列语句中作选择。

6.6.4.1 if 语句

约束

if 语句的控制表达式应为标量类型。

语义

对两种形式的 if 语句，若表达式与零比较结果不相等，则都执行第一个子语句。对 else 形式的语句，若表达式与零比较结果相等，则执行第二个子语句。若第一个子语句是经标号到达的，则不执行第二个子语句。

else 与在同一块(但不是嵌套的内层块)中的、词面上紧先于它的无 else 的 if 相关联。

6.6.4.2 switch 语句

约束

switch 语句的控制表达式应为整型。每一个 case 标号的表达式应为整型常量表达式。不允许同一个 switch 语句中有两个 case 常量表达式在转换后具有相同的值。一个 switch 语句中最多可有一个 default 标号。(任何嵌套的内层 switch 语句中可有一个 default 标号或 case 常量表达式的值与嵌套的外层 switch 语句中的 case 常量表达式的值相同。)

语义

switch 语句使得控制转向、转入或越过*开关体* 内的语句，这取决于控制表达式的值，取决于是否存在 default 标号，以及取决于开关体的、或开关体内的 case 标号的值。case 或 default 标号仅在最紧密的嵌套外层 switch 语句中是可访问的。

对控制表达式将执行整型升格。每一 case 标号中的常量表达式都转换为控制表达式升格后的类型。若某个转换后的值与升格后的控制表达式的值匹配，则控制转向所匹配的 case 标号后的语句。否则，若存在 default 标号，则控制转向该带标号语句。若无转换后的 case 常量表达式与控制表达式匹配，且不存在 default 标号，则开关体内的任何部分均不执行。

实现规定的限定值

如前面已讨论的(见5.2.4.1条)，实现可限定 switch 语句中 case 值的数目。

示例

在下列特意编制的程序片断中：

```
switch (expr)
{
    int i=4;
    f(i);
case 0:
    i=17;/*失败后进入 default代码*/
default:
    printf("%d\n",i);
}
```

标识符为 i 的对象存在且(在该块中)具有自动存储期，但始终不会被初始化。因而，若控制表达式为非零值，则对函数 printf 的调用将访问一个不确定的值。类似地，对函数 f 的调用也永远不会到达。

6.6.5 循环语句

语法

循环语句：

while(*表达式*)*语句*

do*语句* while(*表达式*)；

for(*任选的表达式*；*任选的表达式*；*任选的表达式*)*语句*

约束

循环语句的控制表达式应为标量类型。

语义

循环语句使得称为*循环体*的语句重复执行直至控制表达式与零比较结果相等。

6.6.5.1 while 语句

在每次执行循环体前对控制表达式求值。

6.6.5.2 do 语句

在每次执行循环体后对控制表达式求值。

6.6.5.3 for 语句

除在循环体内的 continue 语句的行为外，语句

for(*表达式1*；*表达式2*；*表达式3*)*语句*

与语句序列

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

是等价的。

注：这样，*表达式1* 指定了对循环的初始化；*表达式2*，即控制表达式，规定在每次循环前求值一次，以便循环的执行得以继续，直至该表达式与零比较结果相等；*表达式3* 规定在每次循环后执行的一个操作(例如增量)。

表达式1 和 *表达式3* 都可省略。每个都按 void 表达式求值。省略的 *表达式2* 可用非零常量替换。

提前引用的条文：continue 语句(6.6.6.2条)。

6.6.6 跳转语句

语法

跳转语句：

goto*标识符*；
continue；
break；
return*任选的表达式*；

语义

跳转语句导致(控制)无条件地转向另一处。

6.6.6.1 goto 语句

约束

goto 语句中的标识符应命名一个标号，该标号位于包含函数中的某处。

语义

goto 语句导致控制无条件地转向包含函数中以所命名的标号为前缀的语句。

示例

有时转向复杂语句组的中间将会带来很多方便。下面的框架给出对一个问题的一种可能的方法，该方法是基于下列三个假设：

a. 通用初始化代码仅访问对当前函数是可见的对象；

b. 通用初始化代码过于庞大，重复它的代价太大；

c. 决定下一个操作的代码必须在循环的起始处。(例如，以允许 continue 语句到达它。)

```
/*...*/
goto first_time;
for(;;){
    /*决定下一个操作*/
    /*...*/
    if(需要重新初始化){
        /*只用于重新初始化的代码*/
        /*...*/
    first_time:
        /*通用初始化代码*/
        /*...*/
        continue;
    }
    /*处理其他操作*/
    /*...*/
}
```

6.6.6.2 continue 语句

约束

continue 语句应仅在循环体内或作为循环体出现。

语义

continue 语句导致控制转向包含它的最小循环语句的继续循环部分，即转向循环体的末尾。更确切地说，在下列每个语句中：

```
while(/*...*/){        do{              for(/*...*/){
/*...*/               /*...*/           /*...*/
continue;              continue;        continue;
/*...*/               /*...*/           /*...*/
```

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

除上面所示的 continue 语句是在一个被包含的循环语句中(此时它被解释为在该语句之内)的情况外，它等价于 goto contin;

注：跟在 contin:标号后的是一个空语句。

6.6.6.3 break 语句

约束

break 语句应仅在开关体或循环体内出现，或者作为开关体或循环体出现。

语义

break 语句终止包含它的最小 switch 语句或循环语句的执行。

6.6.6.4 return 语句

约束

带表达式的 return 语句不应出现在其返回类型为 void 的函数中。

语义

return 语句终止当前函数的执行并将控制返回其调用者。一个函数可有任何数目的 return 语句，可以带也可不带表达式。

若执行带表达式的 return 语句，该表达式的值将作为函数调用表达式的值返回给调用者。若该表达式的类型与它所在的函数的类型不同，则将对它进行转换，如同将它赋于一个该类型的对象一样。

若执行不带表达式的 return 语句，且调用者要使用该函数调用的值，则行为是未定义的。控制到达终止函数的}等价于执行一个不带表达式的 return 语句。

6.7 外部定义

语法

翻译单位:

外部声明

翻译单位 外部声明

外部声明:

函数定义

声明

约束

存储类区分符 auto 和 register 不应出现在外部声明的声明区分符中。

在一个翻译单位中，对声明为有内部链接的每一标识符不应有多于一个的外部定义。而且，若声明为有内部链接的标识符用在表达式中(不是作为 sizeof 算符的操作数的组成部分)，则在该翻译单位中某处对该标识符应确实有一个外部定义。

语义

如在5.1.1.1条中所讨论的，经预处理后的程序文本单位是一个翻译单位，它由一系列外部声明所组成。将这些描述为"外部的"是因为它们出现在任何函数之外，因而具有文卷作用域。如6.5条所述，同时导致保留由标识符所命名的对象或函数所用的存储区的声明是一个定义。

外部定义 是一个外部声明，该声明也是一个函数或对象的定义。若声明为有外部链接的某个标识符被用在表达式中(不是作为 sizeof 算符的操作数的组成部分)，则在整个程序中的某一处，对该标识符应确实有一个外部定义。

注：因此，若声明为有外部链接的标识符不用在表达式中，则不必有对它的外部定义。

6.7.1 函数定义

语法

函数定义：

任选的声明区分符表　声明符　任选的声明表　复合语句

约束

在函数定义中声明的标识符(它是该函数的名)应为函数类型，该类型由函数定义的声明符部分所说明。

注：其意图是使函数定义中的类型类别不能从 typedef 中继承：

```
typedef int F(void);            /* 类型 F 是"无实参,返回 int 的函数" */
F f,g;                          /* f 和 g 都具有与 F 相容的类型 */
F f{/*...*/}                    /* 出错:语法约束错 */
F g( ){/*...*/}                 /* 出错:声明了 g 返回一个函数 */
int f(void){/*...*/}            /* 正确:f 的类型与 F 相容 */
int g( ){/*...*/}               /* 正确:g 的类型与 F 相容 */
F *e(void){/*...*/}             /* e 返回指向一个函数的指针 */
F *((e))(void){/*...*/}         /* 同上:括号不相关 */
int (*fp)(void);                /* fp 指向一个类型为 F 的函数 */
F *Fp;                          /* Fp 指向一个类型为 F 的函数 */
```

函数的返回类型应是 void 或一种非数组的对象类型。

在声明区分符中若有存储类区分符，则应为 extern 或 static。

若声明符包含一个形参类型表，则每个形参的声明中应包含一个标识符(形参表由单个 void 类型的形参组成的特殊情况例外，在该情况下，不应有标识符)。后面不应跟任何声明表。

若声明符包含一个标识符表，则该声明表中每一个声明应至少有一个声明符，且那些声明符应仅声明该标识符表中的标识符。声明为自定义类型名的标识符不应被重新声明为形参。该声明表中的声明应不包含除 register 外的存储类区分符，也不含初始化说明。

语义

函数定义中的声明符指定要定义的函数名及其形参的标识符。若该声明符包含形参类型表，则该表也说明所有形参的类型；这样的声明符也作为函数原型，用于在同一翻译单位中较后调用该函数。若该声明符包含标识符表，则形参的类型可在后面的声明表中声明。未声明的形参类型均为 int。

注：见"语言的发展趋向"(6.9.5条)。

若在定义接受实参数目可变的函数时未带以省略号记法结尾的形参类型表，则行为是未定义的。

进入函数时，每一实参表达式的值都将转换为与其对应的形参的类型，如同给形参赋值一样，作为实参的数组表达式和函数指示符在调用前转换为指针。凡是"*type* 的数组"的形参声明将调整为"指向 *type* 的指针"，凡是"返回 *type* 的函数"的形参声明将调整为"指向返回 *type* 的函数的指针"，如6.2.2.1条中所述。结果得到的形参类型应为对象类型。

每一形参均具有自动存储期。其标识符是一个左值。对形参的存储布局未作规定。

注：形参实际上是在构成函数体的复合语句的起始部分声明的，所以不可以在函数体内重新声明，但可在所包含的块中重新定义。

示例

a. 在下列程序段中：

```
extern int max(int a, int b)
{
    return a>b?a:b;
}
```

extern 是存储类区分符，int 是类型区分符(由于是默认值，每一个都可省略)；max(int a, int b)是函数声明符；{return a>b?a:b;}是函数体。下列类似的定义对形参声明使用了标识符表形式：

```
extern int max(a,b)
int a,b;
{
    return a>b?a:b;
}
```

其中 int a,b;是形参的声明表,由于是默认值,它们可被省略。上述两种定义的差别在于:第一种形式作用为函数原型,它强制转换对该函数的后继调用时的实参,而第二种形式下不一定转换。

b. 要将一个函数传递给另一个,可写作:

```
int f(void);
/*...*/
g(f);
```

注意 f 必须在调用函数中显式声明,因为它在表达式 g(f)中出现时后面未跟(。然后 g 的定义可写为:

```
g(int (*funcp)(void))
{
    /*...*/(*funcp)()/*或 funcp()...*/
}
```

或等价地:

```
g(int func (void))
{
    /*...*/func()/*或(*func)()...*/
}
```

6.7.2 外部对象定义

语义

若对某对象标识符的声明具有文卷作用域和初值符,则该声明是对该标识符的一个外部定义。

某对象标识符的声明具有文卷作用域,但无初值符,无存储类区分符或有存储类区分符 static,则构成一个*临时性定义*。若一个翻译单位中包含一个或多个某标识符的临时性定义,且该翻译单位中不包含该标识符的外部定义,则其行为与如同该翻译单位中包含一个对该标识符的文卷作用域声明,该标识符的类型为该翻译单位结束时的复合类型,初值符等于零时的效果完全一样。

若对某对象标识符的声明是一个临时性定义并具有内部链接,则所声明的类型不应是一个不完整类型。

示例

```
int i1=1;           /*定义,外部链接*/
static int i2=2;    /*定义,内部链接*/
extern int i3=3;    /*定义,外部链接*/
int i4;             /*临时性定义,外部链接*/
static int i5;      /*临时性定义,内部链接*
int i1;             /*合法的临时性定义,指的是前面所定义的*/
int i2;             /*按6.1.2.2条判定为未定义的,链接不一致*/
int i3;             /*合法的临时性定义,指的是前面所定义的*/
int i4;             /*合法的临时性定义,指的是前面所定义的*/
int i5;             /*按6.1.2.2条判定为未定义的,链接不一致*/
extern int i1;      /*指的是前面所定义的,其链接为外部的*/
extern int i2;      /*指的是前面所定义的,其链接为内部的*/
```

```
extern int i3;          /*指的是前面所定义的,其链接为外部的*/
extern int i4;          /*指的是前面所定义的,其链接为外部的*/
extern int i5;          /*指的是前面所定义的,其链接为内部的*/
```

6.8 预处理指示

语法

预处理文卷:

任选的程序组

程序组:

程序组成分

程序组 程序组成分

程序组成分:

任选的 pp 单词 新行

条件节

控制行

条件节:

条件组 任选的多重嵌套条件组 任选的否则组 条件终行

条件组:

#if *常量表达式 新行 任选的程序组*

#ifdef *标识符 新行 任选的程序组*

#ifndef *标识符 新行 任选的程序组*

多重嵌套条件组:

嵌套条件组

多重嵌套条件组 嵌套条件组

嵌套条件组:

#elif *常量表达式 新行 任选的程序组*

否则组:

#else *新行 任选的程序组*

条件终行:

#endif *新行*

控制行:

#include *pp 单词 新行*

#define *标识符 替换表 新行*

#define *标识符 左括号 任选的标识符表)替换表 新行*

#undef *标识符 新行*

#line *pp 单词 新行*

#error *任选的 pp 单词 新行*

#pragma *任选的 pp 单词 新行*

新行

左括号:

不带前导白空类符的左圆括号字符

替换表:

任选的 pp 单词

pp 单词:

预处理单词

pp 单词　预处理单词

新行：

新行字符

描述

预处理指示由以＃预处理单词开始的一系列预处理单词组成，＃预处理单词或者是源文卷中的第一个字符(可任选地跟在不包含新行字符的白空类符之后)，或者是跟在至少包含一个新行字符的白空类符之后，并以下一个新行字符结束的符号。

注：因此，通常将预处理指示称为“行”。这些“行”无其他语法意义，因为除在预处理过程中特定的情况下，所有白空类符都是等价的(例如，参见6.8.3.2条中创建＃字符串字面值算符)。

约束

在一个预处理指示中，在引出预处理指示的＃预处理单词后，至终止预处理指示的新行字符前，两个预处理单词之间所仅能出现的白空类字符是空格和横向制表，包括在翻译阶段3已替换了注释或其它白空类字符后的空格。

语义

实现可有条件地处理或跳过源文卷中的一段，并入其他源文卷，或替换宏。这些能力称为*预处理*，因为从概念上说，它们出现在对所得到的翻译单位进行翻译之前。

除非另加说明，否则对一个预处理指示内的预处理单词不进行宏展开。

6.8.1　条件并入

约束

控制条件并入的表达式应为整型常量表达式，但不应包含强制(转换)。标识符(包括那些在词法上与关键字相同的)按下面所描述的解释；且它可包含形式为：

defined *标识符*

或

defined (*标识符*)

的一元算符表达式。若该标识符当前已定义为宏名(即若它是预定义的，或若它已成为某个＃define 预处理指示的主题，且并无主题标识符相同的＃undef 指示插入)时求值为1，否则求值为零。

注：由于控制常量表达式是在翻译阶段4求值，所有标识符要么是宏名，要么不是宏名——简单地说，没有关键字、枚举常量，等等。

在所有宏替换都出现后剩余的每一预处理单词都应具有单词的词法形式。

语义

形式为：

＃if *常量表达式　新行　任选的程序组*

＃elif *常量表达式　新行　任选的程序组*

的预处理指示测试控制常量表达式是否求值为非零值。

在求值前，先替换将成为控制常量表达式的预处理单词表中的宏调用(那些由 defined 一元算符修改的宏名除外)，就象在正常的正文中一样。若单词 defined 是作为此替换处理的结果产生的，或者在宏替换前一元算符 defined 的使用不与上述两种形式之一匹配，则行为是未定义的。在所有由于宏展开和 defined 一元算符所引起的替换都完成后，将所有余留的标识符都用预处理数零替换，然后将每个预处理单词都转换为一个单词。所得到的单词构成该控制常量表达式，对该表达式的求值按照6.4条中的规则，使用至少是5.2.4.2条中所规定的范围运算，而对 int 和 unsigned int 则按如同它们分别与 long 和 unsigned long 具有同样的表示形式来对待。这包括解释字符常量，其中可能涉及将转义序列转换为执行字符集的成员。这些字符常量的数值是否与当相同的标识符出现在不在＃if 或＃elif 指示中的表达式

中时所获得的值匹配，是实现定义的。同样，单字符字符常量是否可取负值也是实现定义的。

注：因此不保证对下列#if 指示和 if 语句中的常量表达式的求值在这两种语境中能得到相同的值。

```
#if'z'-'a'==25
if('z'-'a'==25)
```

形式为：

#ifdef *标识符* *新行* *任选的程序组*

#ifndef *标识符* *新行* *任选的程序组*

的预处理指示测试该标识符当前是否定义为宏名。它们的条件分别等价于#if defined*标识符* 和#if !defined*标识符* 。

每一指示的条件按次序检查。若它求值为假(零)，则跳过它所控制的程序组；指示的处理仅通过确定该指示的名进行，以便跟踪嵌套条件的层次；指示中其余的预处理单词均被省略，如同该程序组中的其他预处理单词一样。只有控制条件求值为真(非零)的第一个程序组会被处理。若条件中没有一个求值为真，且存在#else 指示，则处理由该#else 指示所控制的程序组；若无#else 指示，则直至#endif 的所有程序组都将被跳过。

注：如语法所指示的，预处理单词不应在结尾的新行字符前跟随#else 或#endif 指示。然而，注释可出现在源文卷中任何位置，包括在预处理指示中。

提前引用的条文：宏替换(6.8.3条)，源文卷并入(6.8.2条)。

6.8.2 源文卷并入

约束

#include 指示应标识一个可被实现处理的前导文卷或源文卷。

语义

形式为：

include <*前导文卷字符序列*>*新行*

的预处理指示在一系列实现定义的位置搜索唯一地由<和>定界符之间指定的序列所标识的前导文卷，并导致用该前导文卷的全部内容替换该指示。位置如何规定或前导文卷如何标识，均是实现定义的。

形式为：

#include "*引号内字符序列*"*新行*

的预处理指示导致用由"定界符之间指定的序列所标识的源文卷的全部内容替换该指示。对该指名的源文卷的搜索是按实现定义的方式。若不支持这种搜索，或搜索失败，则该指示将按如同写作：

#include <*前导文卷字符序列*>*新行*

那样重新处理，后者所包含的序列(包括有能的)字符)与原先的指示中的完全相同。

形式为：

#include *pp 单词* *新行*

的预处理指示(它不与前述两种形式之一匹配)是允许的。对此指示中 include 之后的预处理单词将象通常正文一样处理。(当前定义为宏名的每一标识符由其预处理单词替换表替换。)所有替换完成后得到的指示应与前述两种形式之一匹配。将在预处理单词对<和>之间或一对"字符之间的预处理单词序列组合为单个前导文卷名预处理单词的方法是实现定义的。

注：注意邻接的串字面值并不串接为一单个串字面值(见5.1.1.2条中翻译阶段)，因而，导致两个串字面值的展开是无效的指示。

在定界的序列与外部源文卷名间应有一个实现定义的映射。实现应为由一个或多个字母(如在5.2.1条中定义的)后跟一个圆点(.)和一单个字母所组成的序列提供唯一的映射。实现可忽略大小写字母的差别并限制在圆点之前只映射六个有效字符。

#include 指示可出现在由于另一文卷中的一个#include 指示而被读入的源文卷中，直至到达实

现定义的嵌套层次限定值(见5.2.4.1条)。

示例

a. 最常见的对#include 预处理指示的使用如下:

```
#include <stdio.h>
#include "myprog.h"
```

b. 下例示范说明了一个宏替换的#include 指示:

```
#if VERSION==1
    #define INCFILE "vers1.h"
#elif VERSION==2
    #define INCFILE "vers2.h"
                /*等等*/
#else
    #define INCFILE "versN.h"
#endif
/*...*/
#include INCFILE
```

提前引用的条文:宏替换(6.8.3条)。

6.8.3 宏替换

约束

当且仅当两个替换表中的预处理单词的数目、次序、拼写、以及白空类符分隔都相同时,该两个替换表是等同的。替换表中所有的白空类符分隔都认为是等同的。

当前定义为不使用左括号的宏(一种*类似对象*的宏)的标识符可用另一个#define 预处理指示重新定义,只要第二个定义是类似对象的宏定义并且两个替换表等同。

当前定义为使用左括号的宏(一种*类似函数*的宏)的标识符可用另一个#define 预处理指示重新定义,只要第二个定义是类似函数的宏定义,且形参的数目和拼写相同,以及两个替换表等同。

在类似函数的宏调用中的实参数目应与宏定义中的形参数目一致,且应存在一个终止该调用的预处理单词)。

在类似函数的宏调用中间的形参标识符应是在其作用域内被唯一地声明的。

语义

紧跟 define 的标识符称为*宏名*。宏名有一个名字空间。对任何一种形式的宏,在预处理单词的替换表之前或之后的任何白空类字符均不认为是替换表的组成部分。

若某个其后跟一个标识符的#预处理单词,出现在词法上可以开始一个预处理指示的位置,则该标识符不从属于宏替换。

形式为:

#define*标识符　替换表　新行*

的预处理指示定义一个类似对象的宏,它使得后继的每一个该宏名的实例都被构成该指示余下的部分的预处理单词替换表所替换。然后重新扫描该替换表以搜索是否还有宏名,如下面所述。

注:因为在宏替换的时刻,所有字符常量和串字面值都是预处理单词,而不是可能包含类似于标识符的子序列的序列(见5.1.1.2条翻译阶段),所以决不对它们进行扫描,去寻找宏名或形参。

形式为:

#define*标识符　左括号　任选的标识符表)替换表　新行*

的预处理指示定义一个带形参的类似函数的宏,语法上类似于函数调用。形参由任选的标识符表说明,它们的作用域从标识符表中对它们的声明扩展到直至终止该#define 预处理指示的新行字符为止。后

继的每一个该宏名的实例，后跟一个作为下一个预处理单词的(，引出将以定义中的替换所替换的预处理单词序列，即对该宏的一次调用。预处理单词的替换序列以与该((左圆括号)相匹配的)(右圆括号)预处理单词终止，并且要跳过中间交替插入的成对的左、右括号预处理单词。在构成类似函数的宏调用的预处理单词序列中，新行被认为是一种普通的白空类字符。

由最外层的相匹配的括号所定界的预处理单词序列构成该类似函数的宏的实参表，该表中各个实参由逗号预处理单词分隔，但在相匹配的内层括号之间的逗号预处理单词不分隔实参。若在实参代换前任何实参的组成中无预处理单词，则行为是未定义的。若在实参中存在否则可作为预处理指示的预处理单词序列，则行为也是未定义的。

6.8.3.1 实参代换

在辨识出类似函数的宏调用的实参之后，则进行实参代换。除前面有#或##预处理单词，或后面跟有##预处理单词(见下面)的情况外，替换表中的形参将在对应的实参中所包含的所有宏都展开后由对应的实参替换。在替换之前，对每一实参的预处理单词都进行完全的宏替换，如同由它们构成该翻译单位的其余部分一样；不再有任何预处理单词可用。

6.8.3.2 #算符

约束

每个用于类似函数的宏的替换表中的#预处理单词后应跟有一个形参作为该替换表中的下一个预处理单词。

语义

若在替换表中，有一个形参紧先于一个#预处理单词，则(该形参和#预处理单词)两者由一单字符串字面值预处理单词替换，该字符串字面值预处理单词中包含用于相应实参的预处理单词的拼写。在实参的预处理单词之间出现的每一白空类符在该字符串字面值中都变成单个空格字符。在构成该实参的预处理单词序列之前和之后的白空类符都删除。否则，在该字符串字面值中保持实参中的每一预处理单词的原始拼写，下述产生串字面值和字符常量拼写的特殊处理除外：在字符常量和串字面值中的每个"字符(包括定界用的"字符)和\字符前将插入一个\字符。若该替换得到一个不是有效的字符串字面值，则行为是未定义的。对#和##算符的求值次序未作规定。

6.8.3.3 ##算符

约束

在两种形式的宏定义中，##预处理单词均不应出现在替换表的始端和末尾。

语义

若在替换表中，有一个##预处理单词紧先于或紧后于某形参，则该形参由对应的实参的预处理单词序列替换。

对类似对象和类似函数的宏调用二者，在重新检查替换表寻找更多的宏名以作替换之前，在该替换表中(不是实参中)的每一个##预处理单词的实例都被删除，并将它们前后的预处理单词串接。若结果不是一个有效的预处理单词，则行为是未定义的。结果所得到的单词可为进一步的宏替换所用。对##算符的求值次序未作规定。

6.8.3.4 重新扫描和进一步替换

在代换替换表中所有的形参后，对结果所得到的预处理单词连同源文卷中余下的预处理单词一起进行重新扫描，寻找更多的宏名以作替换。

若在这次扫描替换表(不包括源文卷中余下的预处理单词)的过程中发现了要替换的宏名，则不替换它，而且若任何嵌套的替换遇到所要替换的宏名，也不作替换。这些非替换的宏名预处理单词对于进一步的替换不再可用，即使以后在特定的语境中重新检查它们时也是如此，否则在这些语境中该宏名预处理单词将早已被替换。

结果所得的完全宏替换后的预处理单词序列，即使它们象预处理指示，也不作为预处理指示处理。

6.8.3.5 宏定义的作用域

宏定义持续(与块结构无关)直至遇到对应的#undef 指示为止,或在未遇到任何#undef 指示时直至翻译单位的末尾为止。

形式为:

#undef *标识符* *新行*

的预处理指示使得所指定的标识符不再定义为宏名。若所指定的标识符当前未定义为宏名,则忽略此指示。

示例

a. 这种设施的最简单的使用是用于定义"明显的常量",如在:

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

b. 下面所列的定义了一个类似函数的宏,它的值是其实参中的最大值。它的优点是:可用于任何相容的实参类型;生成直接插入的代码,避免了函数调用的开销。但它的缺点是:对它两个实参中的某一个两次求值(包括副作用);并且当多次调用时,会比函数生成的代码多。也不能取它的地址,因为它没有地址。

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

其中的括号保证实参和所得的表达式均被恰当地定界。

c. 为阐明重写定义和重新检查的规则,序列

```
#define   x       3
#define   f(a)    f(x * (a))
#undef    x
#define   x       2
#define   g       f
#define   z       z[0]
#define   h       g(~
#define   m(a)    a(w)
#define   w       0.1
#define   t(a)    a

f(y+1)+f(f(z))%t(t(g)(0)+t)(1);
g(x+(3,4)-w|h(5)&m
    (f)^ m(m);
```

将得出:

```
f(2 * (y+1))+f(2 * (f(2 * (z[0]))))%f(2 * (0))+t(1);
f(2 * (2+(3,4)-0,1))|f(2 * ( ~5))&f(2 * (0,1))^ m(0.1);
```

d. 为说明创建字符串字面值和串接单词的规则,序列:

```
#define str(s)          #s
#define xstr(s)         str(s)
#define debug(s,t)      printf("x"#s"=%d,x"#t"=%s",\
                          x##s,x##t)
#define INCFILE(n)      vers##n/*引自前面 #include的示例 */
#define glue(a,b)       a##b
#define xglue(a,b)      glue(a,b)
```

```
#define HIGHLOW     "hello"
#define LOW         LOW",world"

debug(1,2);
fputs(str(strncmp("abc\0d","abc",'\4')/*这将消失*/
    ==0)str(:@\n),s);
#include xstr(INCFILE(2),h)
glue(HIGH,LOW);
xglue(HIGH,LOW);
```

将得出：

```
printf("x""1""=%d,x""2""=%s",x1,x2);
fputs("strncmp(\"abc\\0d\",\"abc\",'\\4')==0"":@\n,s);
    #include "vers2.h"(在宏替换后,访问文卷前)
"hello";
"hello"",world"
```

或者，在串接字符串字面值后：

```
printf("x1=%d,x2=%s",x1,x2);
fputs("strncmp(\"abc\\0d",\"abc\",'\\4')==0:@\n,s);
#include"vers2.h"(在宏替换后,访问文卷前)
"hello";
"hello,world"
```

在宏定义中围绕#和##单词的空格是任选的。

e. 最后，为示范重定义规则，下列序列是有效的：

```
#define OBJ_LIKE            (1-1)
#define OBJ_LIKE            /*白空类符*/(1-1)/*其他*/
#define FTN_LIKE(a)         (a)
#define FTN_LIKE(a)         (   /*注意白空类符*/   \
                              a/*此行的其他内容
                                */)
```

但下列重定义是无效的：

```
#define OBJ_LIKE            (0)         /*单词序列不同*/
#define OBJ_LIKE            (1-1)       /*白空类符不同*/
#define FTN_LIKE(b)         (a)         /*形参使用不同*/
#define FTN_LIKE(b)         (b)         /*形参拼写不同*/
```

6.8.4 行控制

约束

若给出#line指示中的串字面值，则它应是一个字符串字面值。

语义

当前源行的*行号*比在翻译阶段1(5.1.1.2条)当将源文卷处理到当前单词时所读入或引入的新行字符个数大1。

形式为：

```
#line  数字字符序列  新行
```

的预处理指示使得实现表现为如同此指示以后的源行序列以其行号为该数字字符序列(解释为十进制

整数)所规定的数的源行开始。该数字字符序列所规定的数不应为零,也不应超过32767。

形式为:

#line *数字字符序列* *"任选的串字符序列"* *新行*

的预处理指示类似第一种形式设置行号,并将假设的源文卷名改为该字符串字面值的内容。

形式为:

#line *pp单词* *新行*

的预处理指示(它与前述两种形式都不匹配)是允许的。指示中line后的预处理单词按在正常正文中一样处理,当前被定义为宏名的每一标识符都用其预处理单词的替换表替换。在全部替换后,所得到的指示应与前述两种形式之一匹配,然后再对它进行适当处理。

6.8.5 出错处理指示

语义

形式为:

#error *任选的pp单词* *新行*

的预处理指示使实现产生一个诊断消息,该信息中包括所指定的预处理单词序列。

6.8.6 编译指示

语义

形式为:

#pragma *任选的pp单词* *新行*

的预处理指示使实现按一种实现定义的方式动作。实现所不能识别的任何编译指示都将被忽略。

6.8.7 空指示

语义

形式为:

新行

的预处理指示不起作用。

6.8.8 预定义的宏名

实现应定义下列宏名:

__LINE__ 当前源行的行号(十进制常量)。

__FILE__ 假设的源文卷名(字符串字面值)。

__DATE__ 翻译源文卷的日期(形式为"Mmm dd yyyy"的字符串字面值,其中月份名与asctime函数所产生的相同,当其值小于10时dd的第一个字符是空格字符)。若翻译时的日期不可用,则应提供一个实现定义的有效日期。

__TIME__ 翻译源文卷的时间(形式为"hh:mm:ss"的字符串字面值,如函数asctime所产生的一样)。若翻译时的时间不可用,则应提供一个实现定义的有效时间。

__STDC__ 十进制常量1,用以指示是一个符合标准的实现。

预定义的宏的值(除__LINE__和__FILE__外),在整个翻译单位中保持不变。

这些宏名中的任何一个,以及标识符defined,都不应是#define或#undef预处理指示的主题。所有预定义的宏名都应以一个下横线开始,后跟一个大写字母或第二个下横线。

提前引用的条文:函数asctime(7.12.3.1条)。

6.9 语言的发展趋向

6.9.1 外部名

对外部名的有效字符数限制为31个字符,以及只允许在大小写中取一种的限制是将逐渐废弃的特征,这是对现有的实现的一种让步。

6.9.2 字符转义序列

以小写字母作为转义序列将被保留用于未来的标准化。在扩展时可用其他字符。

6.9.3 存储类区分符

将存储类区分符放置在声明中声明区分符的始端以外的地方是一种将逐渐废弃的特征。

6.9.4 函数声明

使用带空括号的函数声明符(非原型格式形参类型声明符)是一种将逐渐废弃的特征。

6.9.5 函数定义

使用带分离的形参标识符和声明表的函数定义(非原型格式形参类型和标识符声明符)是一种将逐渐废弃的特征。

6.9.6 数组形参

使用在分离的左值中声明为数组类型的两个形参(在调整为指针类型之前)指示同一对象是一种将逐渐废弃的特征。

7 库

7.1 引言

7.1.1 术语定义

串 是字符的一个相继序列,它由第一个空字符结尾,串中包括该空字符。指向串的"指针"是指向其初始(最低地址的)字符的指针。串的"长度"是在该空字符之前的字符个数,且串的"值"是它所包含的字符值按序排列的序列。

字母 是执行字符集中的可印刷字符,它们对应于源字符集中的52个必需的小写和大写字母,已在5.2.1条中列出。

十进制小数点字符 是由一些函数所用的字符,它用于在浮点数与字符序列之间进行转换时表记这样的字符序列中小数部分的开始。它在正文和示例中用一个圆点表示,但可用函数 setlocale 改变其表示。

注:使用十进制小数点字符的函数有:localeconv、fprintf、fscanf、printf、scanf、sprintf、sscanf、vfprintf、vprintf、vsprintf、atof 以及 strtod。

提前引用的条文:字符处理(7.3条),函数 setlocale(7.4.1.1条)。

7.1.2 标准前导文卷

每个库函数都在一个*前导文卷* 中声明,前导文卷的内容通过 #include 预处理指示成为可用。前导文卷声明一组相关的函数,并定义为方便对这些函数的使用所需的必要的类型和附加的宏。

注:前导文卷不一定是一个源文卷,在前导文卷名中由<和>定界的序列也不一定是有效的源文卷名。

标准前导文卷有:

<assert.h>	<locale.h>	<stddef.h>
<ctype.h>	<math.h>	<stdio.h>
<errno.h>	<setjmp.h>	<stdlib.h>
<float.h>	<signal.h>	<string.h>
<limits.h>	<stdarg.h>	<time.h>

若某个名字与上述由<和>定界的序列之一相同的文卷,不是作为实现的一部分提供,而是被放置在对并入的源文卷所规定的标准位置上,则行为是未定义的。

前导文卷可按任意次序并入,每个前导文卷在给定的范围内可被并入多次,其效果与只并入一次并无区别,但并入<assert.h>的情况除外,并入<assert.h>的效果取决于 NDEBUG 的定义。若使用某个前导文卷,则它应在任何外部声明或外部定义之外并入,且它的第一次并入应在首次引用任何它所定义的函数或对象之前,或者首次引用它所定义的类型或宏之前。然而,若在多个前导文卷中声明或定义了某标识符,则第二个以及后继的前导文卷可在第一次引用该标识符之后并入。并入前,程序中不应有

任何宏名在词法上与当前定义的关键字等同。

提前引用的条文：诊断程序库前导文卷(7.2条)。

7.1.3 保留的标识符

每个前导文卷都声明或定义在与其相关的条文中所列出的所有标识符，并任选地声明或定义在与其相关的“库的发展趋向”条文中所列出的标识符，以及总是保留的、或者用于任何用途、或者用作文卷作用域标识符的标识符。

——以下横线和一个大写字母或另一个下横线开始的标识符总是保留用于任何用途。

——以下横线开始的标识符总是保留用作在普通标识符空间和标记名空间具有文卷作用域的标识符。

——在以下任何条文(包括“库的发展趋向”条)中所列出的宏名，若任何与其相关的前导文卷被并入时，保留用于任何用途。

——在以下任何条文(包括“库的发展趋向”条)中具有外部链接的所有标识符，总是保留用作具有外部链接的标识符。

注：保留的具有外部链接的标识符表包括 errno、setjmp 和 va_end。

——在以下任何条文(包括“库的发展趋向”条)中所列出具有文卷作用域的标识符，若任何与其相关的前导文卷被并入时，保留用作在同一名字空间中具有文卷作用域的标识符。

再无其他标识符是保留的。若程序声明或定义了与该语境中保留的标识符名字相同的标识符(除7.1.7条所允许的外)，则行为是未定义的。

注：由于一旦发现宏名即作替换，且替换时不受作用域和名字空间约束，因而，若并入任何相关的前导文卷时，不得定义任何与保留的标识符名相匹配的宏名。

7.1.4 出错处理程序库前导文卷<errno.h>

前导文卷<errno.h>定义了几个宏，都与报告出错条件有关。

这些宏是：

EDOM

ERANGE

它们展开为值不相同的非零值整型常量表达式，适合于用在#if 预处理指示中；以及

errno

它展开为类型是 int 的可修改的左值，其值由几个库函数置为正的出错号。对 errno 是一个宏还是一个具有外部链接的标识符未作规定。若抑制宏定义以便访问一个实际的对象，或程序定义了一个名为 errno的标识符，则行为是未定义的。

注：宏 errno 不必是某对象的标识符。它可展开为由函数调用所得到的可修改的左值(例如 *errno())。

在程序起始时，errno 的值是零，但它决不会被任何库函数置为零。不论是否出错，errno 的值都可被库函数调用置为非零值，只要对 errno 的使用未在本标准中该函数的描述中注明。

注：因此，使用 errno 作差错检查的程序应在库函数调用前将它置为零，然后在下一个库函数调用前检查它。当然，库函数可在入口时先保存 errno 的值，再把它置为零，只要当即将返回前 errno 的值仍为零时，能恢复其原先的值。

实现也可规定附加的宏定义，以 E 和一个数字字符或 E 和一个大写字母开始。

注：见“库的发展趋向”(7.13.1条)。

7.1.5 限定值前导文卷<float.h>和<limits.h>

前导文卷<float.h>和<limits.h>定义了几个展开为各种限定值和形参的宏。

这些宏的意义和对它们的值的约束已在5.2.4.2条中列出。

7.1.6 公用定义库前导文卷<stddef.h>

下列类型和宏在标准前导文卷<stddef.h>中定义。其中一些也在其他前导文卷中定义，在相关的条文中已加注明。

所定义的类型是：

ptrdiff_t

它是两个指针相减所得结果的有符号整型；

size_t

它是 sizeof 算符结果的无符号整型；以及

wchar_t

它是一个整型，其值的范围可表示在所支持的所有地域环境中规定的最大扩展字符集中所有成员的各别代码；空字符的代码应为零值，在2.2.1条中定义的基本字符集中每一个成员应有一个代码，该代码的值等于该基本字符集成员用作一个整型字符常量中唯一的字符时该常量的值。

所定义的宏是：

NULL

它展开为一个实现定义的空指针常量；以及

offsetor(*type*,*成员指示符*)

它展开为一个类型为 size_t 的整型常量表达式，其值是从某结构(由 *type* 指示)的始端开始，到该结构成员(由*成员指示符*指示)为止的按字节计的偏移量。*成员指示符*应为如此：若给定：

static *type* t;

则表达式 &(t.*成员指示符*)求值为一个地址常量。(若所指定的成员是一个位段，则行为是未定义的。)

提前引用的条文：本地化程序库前导文卷(7.4条)。

7.1.7 库函数的使用

除非在库函数后的详细描述中另加说明，否则下列每一陈述均适用。若函数的某个实参值为无效值(例如超出该函数定义域的值，或超出程序的地址空间的指针，或空指针)，则行为是未定义的。若某函数实参被描述为是一个数组，则实际传递给该函数的指针的值应使得所有地址计算和对象访问都有效(若该指针并未指向该数组的第一个元素，也是有效的)。在一个前导文卷中声明的任何函数都可被另外实现为在该前导文卷中定义的一个宏，所以当并入某库函数的前导文卷时，不应显式声明该库函数。任何函数的宏定义都可通过将该函数名括在括号内而被局部地抑制，因为那时该名后不再跟有指示展开宏函数名的左括号了。出于相同的语法上的理由，即使库函数被定义为一个宏，取它的地址也是允许的[1]。使用 #undef 移去任何宏定义也将保证所引用的是一个实际的函数。任何以宏实现的库函数调用应展开为这样的代码：对其每个实参确实求值一次，必要时用完全的括号保护，所以，使用任意的表达式作为实参通常是安全的。同样，在下面的条文中描述的那些类似于函数的宏，均可在任何可以调用返回类型与之相容的函数的地方用表达式来调用[2]。所列出的所有展开为整型常量表达式的类似于对象的宏，均应附加地适合于用在 #if 预处理指示中。

注：1) 这意味着即使实现已为某库函数提供了一个宏，它也必须对该库函数再提供一个实际的函数。

2) 由于以下横线开始的外部标识符和某些宏名是保留的，实现可为这些名提供特殊的语义。

例如，标识符 __BUILTIN_abs 可被用于规定为函数 abs 生成直接插入代码。因此，对于代码生成器能接受这类说明的编译程序，在某个适当的前导文卷中应说明：

#define abs(x) __BUILTIN_abs(x)

以这种方式，则对于希望能保证给定的库函数(例如 abs)是一个真正的函数的用户，可以书写：

#undef abs

而不管实现的前导文卷是否为 abs 提供了一种宏实现，还是一个内部函数。出现在任何宏定义之前且被宏定义所掩盖的该函数的原型，也就显露出来了。

只要可以无需引用在前导文卷中定义的任何类型而声明某库函数，则也允许不并入与之相关的前导文卷，而显式或隐式地声明该函数。若未显式地或通过并入与之相关的前导文卷而声明一个接受实参数目可变的函数，则行为是未定义的。

示例

可以几种方式使用函数 atoi：

——通过与其相关的前导文卷(可能产生一个宏展开)

```
#include <stdlib.h>
const char *str;
/* ... */
i=atoi(str);
```

——通过与其相关的前导文卷(保证产生一个真正的函数引用)

```
#include <stdlib.h>
#undef atoi
const char *str;
/* ... */
i=atoi(str);
```

或

```
#include <stdlib.h>
const char *str;
/* ... */
i=(atoi)(str);
```

——通过显式声明

```
extern int atoi(const char *);
const char *str;
/* ... */
i=atoi(str);
```

——通过隐式声明

```
const char *str;
/* ... */
i=atoi(str);
```

7.2 诊断程序库前导文卷<assert.h>

前导文卷<assert.h>定义了宏 assert 并引用另一个宏：

NDEBUG

它不是由<assert.h>所定义的。若在源文卷中并入<assert.h>的位置，NDEBUG 是定义为一个宏名，则宏 assert 将简单地定义为：

```
#define assert(ignore)((void)0)
```

宏 assert 应被真正实现为一个宏，而不是一个实际的函数。若该宏定义被抑制以便访问一个实际的函数，则行为是未定义的。

7.2.1 程序的诊断

7.2.1.1 宏 assert

调用序列规格说明

```
#include <assert.h>
void assert(int 表达式);
```

描述

宏 assert 在程序中加入诊断消息。当它执行时，若*表达式*为假(即与零比较结果相等)，则宏 assert 以实现定义的格式，在标准出错信息文卷中写入有关失败的特定调用的信息(包括实参的正文、源文卷名、以及源行号——后者分别是预处理宏__FILE__和__LINE__的相对值)，然后调用函数 abort。

注：所写出的消息的形式可能为：

断言失败：*表达式*，文卷 *xyz*，行 *nnn*

返回值

宏 assert 不返回值。

提前引用的条文：函数 abort(7.10.4.1条)。

7.3 字符处理程序库前导文卷<ctype.h>

前导文卷<ctype.h>声明了几个对测试和映射字符有用的函数。在所有情况下，实参都是一个 int，其值应可作为 unsigned char 表示，或者应等于宏 EOF 的值。若实参为任何其他值，则行为是未定义的。

注：见"库的发展趋向"(7.13.2条)。

这些函数的行为受当前地域环境的影响。仅当不在"C"地域环境时才具有实现定义的面貌的那些函数将在下面注明。

术语可*印刷字符* 指的是实现定义的字符集中的成员，每一个都在显示设备上占据一个印刷位置；术语*控制字符* 指的是实现定义的字符集中的不是可印刷字符的成员。

注：在使用七位编码字符集的实现中，可印刷字符是值从0x20(空格)到0x7E(上横线)的那些字符；控制字符是值从0(NUL)到0x1F(US)的那些字符，以及字符0x7F(DEL)。

提前引用的条文：EOF(7.9.1条)，本地化程序库前导文卷(7.4条)。

7.3.1 字符测试函数

本条中的函数当且仅当实参 c 的值与函数所描述的一致时才返回非零(真)值。

7.3.1.1 函数 isalnum

调用序列规格说明

```
#include <ctype.h>
int isalnum(int c);
```

描述

函数 isalnum 测试使函数 isalpha 或函数 isdigit 为真的任何字符。

7.3.1.2 函数 isalpha

调用序列规格说明

```
#include <ctype.h>
int isalpha(int c);
```

描述

函数 isalpha 测试使函数 isupper 或函数 islower 为真的任何字符，或者是实现定义的字符集中的一个成员，且使函数 iscntrl、isdigit、ispunct、或 isspace 均不为真的任何字符。当在"C"地域环境时，仅当对某字符函数 isupper 或 islower 为真时，函数 isalpha 才返回真值。

7.3.1.3 函数 iscntrl

调用序列规格说明

```
#include <ctype.h>
int iscntrl(int c);
```

描述

函数 iscntrl 测试字符是否是控制字符。

7.3.1.4 函数 isdigit

调用序列规格说明

```
#include <ctype.h>
int isdigit(int c);
```

描述

函数 isdigit 测试字符是否是如5.2.1条所定义的十进制数字字符。

7.3.1.5 函数 isgraph

调用序列规格说明

```
#include <ctype.h>
int isgraph(int c);
```

描述

函数 isgraph 测试任何除空格(' ')外的可印刷字符。

7.3.1.6 函数 islower

调用序列规格说明

```
#include <ctype.h>
int islower(int c);
```

描述

函数 islower 测试任何是小写字母的字符，或者实现定义的字符集中的一个成员，对该字符，函数 iscntrl、isdigit、ispunct、或 isspace 均不为真。当在"C"地域环境时，函数 islower 仅对定义为如5.2.1条所定义的小写字母的字符才返回真值。

7.3.1.7 函数 isprint

调用序列规格说明

```
#include <ctype.h>
int isprint(int c);
```

描述

函数 isprint 测试包括空格(' ')在内的任何可印刷字符。

7.3.1.8 函数 ispunct

调用序列规格说明

```
#include <ctype.h>
int ispunct(int c);
```

描述

函数 ispunct 测试既不是空格(' ')，也不是使函数 isalnum 为真的任何可印刷字符。

7.3.1.9 函数 isspace

调用序列规格说明

```
#include <ctype.h>
int isspace(int c);
```

描述

函数 isspace 测试任何是标准白空类字符的字符，或者是实现定义的字符集中的一个成员，且使函数 isalnum 为假的任何字符。标准白空类字符如下：空格(' ')、换页('\f')、新行('\n')、回车('\r')、横向制表('\t')和纵向制表('\v')。当在"C"地域环境时，函数 isspace 仅对标准白空类字符返回真值。

7.3.1.10 函数 isupper

调用序列规格说明

```
#include <ctype.h>
int isupper(int c);
```

描述

函数 isupper 测试任何是大写字母的字符，或者实现定义的字符集中的一个成员，对该字符，函数 iscntrl、isdigit、ispunct 或 isspace 均不为真。当在"C"地域环境时，函数 isupper 仅对定义为如5.2.1条所

定义的大写字母的字符才返回真值。

7.3.1.11 函数 isxdigit

调用序列规格说明

```
#include <ctype.h>
int isxdigit(int c);
```

描述

函数 isxdigit 测试是如6.1.3.2条中所定义的十六进制数字字符的任何字符。

7.3.2 大小写字符映射函数

7.3.2.1 函数 tolower

调用序列规格说明

```
#include <ctype.h>
int tolower(int c);
```

描述

函数 tolower 将大写字母转换为对应的小写字母。

返回值

若实参是一个使函数 isupper 为真的字符，且存在一个对应的使函数 islower 为真的字符，则函数 tolower 返回该对应的字符，否则返回未作改变的实参。

7.3.2.2 函数 toupper

调用序列规格说明

```
#include <ctype.h>
int toupper(int c);
```

描述

函数 toupper 将小写字母转换为对应的大写字母。

返回值

若实参是一个使函数 islower 为真的字符，且存在一个对应的使函数 isupper 为真的字符，则函数 toupper 返回该对应的字符，否则返回未作改变的实参。

7.4 本地化程序库前导文卷<locale.h>

前导文卷<locale.h>中声明了两个函数和一种类型，并定义了几个宏。所声明的类型是：

struct lconv

其中包含与数值格式化有关的成员。该结构应至少以任意次序包含下列成员。这些成员的语义和它们的值的正常范围在7.4.2.1条中解释。在地域环境为"C"时，这些成员应具有在注释中所描述的值。

```
char *decimal_point;          /* "." */
char *thousands_sep;          /* "" */
char *grouping;               /* "" */
char *int_curr_symbol;        /* "" */
char *currency_symbol;        /* "" */
char *mon_decimal_point;      /* "" */
char *mon_thousands_sep;      /* "" */
char *mon_grouping;           /* "" */
char *positive_sign;          /* "" */
char *negative_sign;          /* "" */
char int_frac_digits;         /* CHAR_MAX */
char frac_digits;             /* CHAR_MAX */
```

```
char p_cs_precedes;          /* CHAR_MAX */
char p_sep_by_space;         /* CHAR_MAX */
char n_cs_precedes;          /* CHAR_MAX */
char n_sep_by_space;         /* CHAR_MAX */
char p_sign_posn;            /* CHAR_MAX */
char n_sign_posn;            /* CHAR_MAX */
```

所定义的宏是 NULL(已在7.1.6条中描述);和

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME

这些宏将展开为具有各别的值、适用于作为函数 setlocale 的第一个实参的常量表达式。实现也可规定以字符 LC_和一个大写字母开头的附加的宏定义(见7.13.3条"库的发展趋向")。

7.4.1 地域环境控制

7.4.1.1 函数 setlocale

调用序列规格说明

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

描述

函数 setlocale 选择由 category 和 locale 实参所规定的程序地域环境中合适的部分。可以使用 setlocale 函数改变或询问程序的整个当前地域环境或其中的一部分。category 的 LC_ALL 值命名程序的整个地境,而其他值只命名程序地域环境的一部分。LC_COLLATE 影响函数 strcoll 和 strxfrm 的行为。LC_CTYPE 影响字符处理函数和多字节函数的行为。LC_MONETARY 影响由函数 localeconv 返回的金融格式信息。LC_NUMERIC 影响格式化输入输出函数和串转换函数的十进制小数点字符,以及由函数 localeconv 返回的非金融格式信息。LC_TIME 影响函数 strftime 的行为。

注:在7.3条所描述的函数中仅函数 isdigit 和 isxdigit 不受当前地域环境的影响。

locale 的值为"C"规定 C 翻译的最小环境。locale 的值为""规定实现定义的本地环境。其他实现定义的串可作为第二个实参传递给函数 setlocale。

在程序启动时,执行等价于

setlocale(LC_ALL,"C");的操作。

实现应表现为如同无任何库函数调用函数 setlocale 一样。

返回值

若为 locale 给出了指向串的指针且选择得到批准,则函数 setlocale 返回一个指针,该指针指向用作新地域环境的与 category 相关的串,若不能批准给定的选择,则函数 setlocale 返回一个空指针且程序的地域环境不变。

对 locale 给出一个空指针将使得函数 setlocale 返回一个指向与 category 相关的表示程序当前地域环境的串的指针,程序的地域环境不变。

注:由于当 category 的值为 LC_ALL 时地域环境的异构性,实现必须安排将对各个类的编码放在一个串中。

由函数 setlocale 返回的指向串的指针将使得后继的以该串值及与之相关的类别调用函数 setlocale 时恢复该部分程序的地域环境。该指针所指向的串不应由程序修改,但可用后继的对函数 setlocale 的调用重写。

提前引用的条文：格式化输入输出函数(7.9.6条)，多字节字符函数(7.10.7条)，多字节串函数(7.10.8条)，串转换函数(7.10.1条)，函数 strcoll(7.11.4.3条)，函数 strftime(7.12.3.5条)，函数 strxfrm(7.11.4.5条)。

7.4.2 询问数值格式约定

7.4.2.1 函数 localeconv

调用序列规格说明

```
#include <locale.h>
struct lconv *localeconv(void);
```

描述

函数 localeconv 按当前地域环境的规则将类型为 struc lconv 的对象的分量置为适合于格式化(金融的或其他的)数值表示的值。

类型为 char * 的结构成员是指向串的指针，它们中(除 decimal_point 外)的任何一个均可指向""，以指示当前地域环境中的值不可用或长度为零。类型为 char 的成员是非负数，它们中的任何一个均可为 CHAR_MAX，指示在当前地域环境中无值可用。该结构包括下列成员：

char *decimal_point

用于格式化非金融量的十进制小数点字符。

char *thousands_sep

用于格式化非金融量的十进制小数点字符之前分隔数位组的字符。

char *grouping

一个串，其元素指示在格式化非金融量中每个数位组的尺寸。

char *int_curr_symbol

适用于当前地域环境的国际货币符号。前三个字符包含与在 GB 12406中所规定的符号一致的字母型国际货币符号。紧先于空字符的第四个字符是用于分隔国际货币符号与金融量的字符。

char *currency_symbol

适用于当前地域环境的本地货币符号。

char *mon_decimal_point

用于格式化金融量的十进制小数点字符。

char *mon_ thousands_sep

在格式化金融量的十进制小数点字符之前数字位组的分隔符。

char *mon_grouping

一个串，其元素指示在格式化金融量中每个数字位组的尺寸。

char *positive_sign

指示一个非负值格式化金融量的串。

char *negative_sign

指示一个负值格式化金融量的串。

char int_frac_digits

以国际格式化金融量形式显示的小数部分(十进制小数点之后的)位数。

char frac_digits

以格式化金融量形式显示的小数部分(十进制小数点之后的)位数。

char p_cs_precedes

根据 currency_symbol 在非负值格式化金融量之前或之后，分别置为1或0。

char p_sep_by_space

根据 currency_symbol 与非负值格式化金融量之间是否以一个空格分隔，分别置为1或0。

char n_cs_precedes

根据 currency_symbol 在负值格式化金融量之前或之后，分别置为1或0。

char n_sep_by_space

根据 currency_symbol 与负值格式化金融量之间是否以一个空格分隔，分别置为1或0。

char p_sign_posn

置为一个值，该值指示非负值格式化金融量中 positive_sign 的位置。

char n_sign_posn

置为一个值，该值指示负值格式化金融量中 negative_sign 的位置。

grouping 和 mon_grouping 的元素按如下解释：

CHAR_MAX	不再进一步分组。
0	对余下的数位重复使用上一个元素。
其他	该整数值是组成当前组的位数。检验下一个元素以确定在当前组之前的下一个位组的尺寸。

p_sign_posn 和 n_sign_posn 的值按如下解释：

0	量值和 currency_symbol 由括号括起来。
1	符号串在量值和 currency_symbol 之前。
2	符号串在量值和 currency_symbol 之后。
3	符号串紧先于 currency_symbol。
4	符号串紧后于 currency_symbol。

返回值

函数 localeconv 返回一个指向所充填的对象的指针。由返回值所指向的结构不应被程序修改，但可由后继的对函数 localeconv 的调用所重写。此外，以类别 LC_ALL，LC_MONETARY 或 LC_NUMERIC 调用函数 setlocale 也可重写该结构的内容。

示例

下表说明了适合于四个国家用于格式化金融量的规则：

国别	正值格式	负值格式	国际格式
意大利	L. 1. 234	−L. 1. 234	ITL. 1. 234
荷　兰	F 1,234. 56	F−1,234. 56	NLG 1,234. 56
挪　威	kr1. 234,56	kr1. 234,56−	NOK 1. 234,56
瑞　士	SFrs. 1,234. 56	SFrs. 1,234. 56C	CHF 1,234. 56

对上述四个国家，由函数 localeconv 返回的结构中有关金融量的成员值是：

	意大利	荷　兰	挪　威	瑞　士
int_curr_symbol	"ITL. "	"NLG"	"NOK"	"CHF
currency_symbol	"L. "	"F"	"kr"	"SFrs. "
mon_decimal_point	""	","	","	"."
mon_thousands_sep	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"−"	"−"	"−"	"C"
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1

p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2

7.5 数学程序库前导文卷<math.h>

前导文卷<math.h>声明了几个数学函数，并定义了一个宏。这些函数取双精度的实参，并返回双精度的结果值。整型算术函数和转换函数将在后面讨论。

注：见"库的发展趋向"(7.13.4条)。

所定义的宏是：

HUGE_VAL

它展开为正的 double 表达式，且不一定需要能表示为 float。

注：在支持无穷大的实现中，HUGE_VAL 可以为正无穷大。

提前引用的条文：整型算术函数(7.10.6条)，函数 atof(7.10.1.1条)，函数 strtod(7.10.1.4条)。

7.5.1 出错条件的处理

对于其输入实参的所有可表示的值，这些函数中每一个的行为都是定义了的。每个函数都应如同单个操作一样执行，而不产生任何内部可见的异常。

对所有函数，若一个输入实参超出了该数学函数的定义域，则出现一个*定义域错*。每一函数的描述中列出了必需给出的定义域错；实现可定义附加的定义域错，只要这样的错误与该函数的数学定义一致。出现定义域错时，函数返回一个实现定义的值；宏 EDOM 的值存储在 errno 中。

注：在支持无穷大的实现中，这允许若该函数的定义域中不包括无穷大时，以无穷大作实参成为一种定义域错。

类似地，若函数的结果不能作为 double 值表示，则出现*值域错*。若结果溢出(结果的量值太大以致不能在所规定类型的对象中表示)，函数返回宏 HUGE_VAL 的值，符号与该函数的正确值相同(对 tan 函数例外)；宏 ERANGE 的值存储在 errno 中。若结果溢出(结果的量值太小以致不能在所规定类型的对象中表示)，函数返回0；整型表达式 errno 中是否存入宏 ERANGE 的值由实现定义。

7.5.2 三角函数

7.5.2.1 函数 acos

调用序列规格说明

```
#include <math.h>
double acos(double x);
```

描述

函数 acos 计算 x 的反余弦的主值。若实参值不在范围[−1，+1]之间则出现定义域错。

返回值

函数 acos 返回范围在[0，π]弧度内的反余弦。

7.5.2.2 函数 asin

调用序列规格说明

```
#include <math.h>
double asin(double x);
```

描述

函数 asin 计算 x 的反正弦的主值。若实参值不在范围[−1，+1]之间则出现定义域错。

返回值

函数 asin 返回范围在[−π/2，+π/2]弧度内的反正弦。

7.5.2.3 函数 atan

调用序列规格说明

#include <math.h>

double atan(double x);

描述

函数 atan 计算 x 的反正切的主值。

返回值

函数 atan 返回范围在[$-\pi/2$,$+\pi/2$]弧度内的反正切。

7.5.2.4 函数 atan2

调用序列规格说明

#include <math.h>

double atan2(double y, double x);

描述

函数 atan2计算 y/x 的反正切的主值,用两个实参的符号决定返回值所在的象限。若两个实参均为0,则可能出现定义域错。

返回值

函数 atan2返回 y/x 的反正切,范围在[$-\pi$,$+\pi$]弧度内。

7.5.2.5 函数 cos

调用序列规格说明

#include <math.h>

double cos(double x);

描述

函数 cos 计算 x 的余弦。

返回值

函数 cos 返回以弧度为度量单位的余弦值。

7.5.2.6 函数 sin

调用序列规格说明

#include <math.h>

double sin(double x);

描述

函数 sin 计算 x 的正弦。

返回值

函数 sin 返回以弧度为度量单位的正弦值。

7.5.2.7 函数 tan

调用序列规格说明

#include <math.h>

double tan(double x);

描述

函数 tan 计算 x 的正切。

返回值

函数 tan 返回以弧度为度量单位的正切值。

7.5.3 双曲函数

7.5.3.1 函数 cosh

调用序列规格说明

#include <math.h>

double cosh(double x);

描述

函数 cosh 计算 x 的双曲余弦。若 x 的量值太大时则出现值域错。

返回值

函数 cosh 返回双曲余弦值。

7.5.3.2 函数 sinh

调用序列规格说明

#include <math.h>

double sinh(double x);

描述

函数 sinh 计算 x 的双曲正弦。若 x 的量值太大时则出现值域错。

返回值

函数 sinh 返回双曲正弦值。

7.5.3.3 函数 tanh

调用序列规格说明

#include <math.h>

double tanh(double x);

描述

函数 tanh 计算 x 的双曲正切。

返回值

函数 tanh 返回双曲正切值。

7.5.4 指数和对数函数

7.5.4.1 函数 exp

调用序列规格说明

#include <math.h>

double exp(double x);

描述

函数 exp 计算 x 的指数函数。若 x 的量值太大时则出现值域错。

返回值

函数 exp 返回指数值。

7.5.4.2 函数 frexp

调用序列规格说明

#include <math.h>

double frexp(double value, int *exp);

描述

函数 frexp 将一个浮点数分为一个规格化的小数部分和一个2的整数次幂。并将该整数存储在 exp 所指向的 int 对象中。

返回值

函数 frexp 返回值 x，使得 x 为量值在区间[1/2,1)内、或是0的 double，且 value 等于 x 乘以2的 *exp 次幂。若 value 为0，则结果的两部分都为0。

7.5.4.3 函数 ldexp

调用序列规格说明

#include <math.h>

double ldexp(double x, int exp);

描述

函数 ldexp 将一个浮点数乘以一个2的整数次幂。可能会出现值域错。

返回值

函数 ldexp 返回值 x 乘以2的 exp 次幂。

7.5.4.4 函数 log

调用序列规格说明

#include <math.h>

double log(double x);

描述

函数 log 计算 x 的自然对数。若实参为负则出现定义域错。若实参为0则可能出现值域错。

返回值

函数 log 返回自然对数。

7.5.4.5 函数 log 10

调用序列规格说明

#include <math.h>

double log 10(double x);

描述

函数 log 10计算 x 的以10为底的对数。若实参为负则出现定义域错。若实参为0则可能出现值域错。

返回值

函数 log10返回以10为底的对数。

7.5.4.6 函数 modf

调用序列规格说明

#include <math.h>

double modf(double value, double *iptr);

描述

函数 modf 将实参 value 分为整数和小数两部分，每一部分的符号均与实参的符号相同。它将整数部分按 double 存储在由 iptr 所指向的对象中。

返回值

函数 modf 返回 value 的有符号的小数部分。

7.5.5 幂函数

7.5.5.1 函数 pow

调用序列规格说明

#include <math.h>

double pow(double x, double y);

描述

函数 pow 计算 x 的 y 次幂。若 x 为负而 y 不是整数值时则出现定义域错。若当 x 为0且 y 小于等于0而使结果不能被表示时，也出现定义域错。也可能出现值域错。

返回值

函数 pow 返回 x 的 y 次幂的值。

7.5.5.2 函数 sort

调用序列规格说明

#include <math.h>

double sort(double x);

描述

函数 sqrt 计算 x 的非负平方根。若实参为负则出现定义域错。

返回值

函数 sqrt 返回 x 的平方根值。

7.5.6 最近整数、绝对值和余数函数

7.5.6.1 函数 ceil

调用序列规格说明

#include <math.h>

double ceil(double x);

描述

函数 ceil 计算不小于 x 的最小整数。

返回值

函数 ceil 返回不小于 x 的最小整数，以双精度浮点表示。

7.5.6.2 函数 fabs

调用序列规格说明

#include <math.h>

double fabs(double x);

描述

函数 fabs 计算浮点数 x 的绝对值。

返回值

函数 fabs 返回 x 的绝对值。

7.5.6.3 函数 floor

调用序列规格说明

#include <math.h>

double floor(double x);

描述

函数 floor 计算不大于 x 的最大整数。

返回值

函数 floor 返回不大于 x 的最大整数，以双精度浮点表示。

7.5.6.4 函数 fmod

调用序列规格说明

#include <math.h>

double fmod(double x,double y);

描述

函数 fmod 计算 x/y 的浮点余数。

返回值

函数 fmod 返回值 $x-i*y$（i 是一个整数），使得若 y 不为0时，结果的符号与 x 的符号相同，结果的量值小于 y 的量值。若 y 为0，是出现定义域错还是函数 fmod 返回零值由实现定义。

7.6 非局部跳转库前导文卷<setjmp.h>

前导文卷<setjmp.h>定义了一个宏 setjmp，并声明了一个函数和一种类型，用于越过正常的函数调用并返回主流程。

注：这些函数对处理在程序的低级函数中所遇到的不寻常情况是有用的。

所声明的类型是：

jmp_buf

它是一个数组类型，适用于存放恢复调用环境所需的信息。

对 setjmp 是一个宏还是一个声明为具有外部链接的标识符未作规定。若抑制宏定义以便访问一个实际的函数，或程序定义了一个名为 setjmp 的外部标识符，则行为是未定义的。

7.6.1 保存调用环境

7.6.1.1 宏 setjmp

调用序列规格说明

#include <setjmp.h>

int setjmp(jmp_buf env);

描述

宏 setjmp 将它的调用环境保存在实参 jmp_buf 中，供以后由函数 longjmp 使用。

返回值

当从直接调用返回时，宏 setjmp 返回值0，而当从调用 longjmp 函数返回时，宏 setjmp 返回一个非零值。

环境约束

对宏 setjmp 的调用应仅出现在下列语境中：

——选择语句或循环语句的整个控制表达式中；

——关系算符或相等类算符的一个操作数中，条件是另一个操作数是整型常量表达式，且结果所得的表达式是选择语句或循环语句的整个控制表达式；

——当结果所得的表达式是选择语句或循环语句的整个控制表达式时一元!算符的操作数中；或

——表达式语句中的整个表达式中(可能强制转换为 void)。

7.6.2 恢复调用环境

7.6.2.1 函数 longjmp

调用序列规格说明

#include <setjmp.h>

void longjmp(jmp_bufenv,int val);

描述

函数 longjmp 使用实参 jmp_buf，恢复由在程序的同一次激活中由最近一次调用宏 setjmp 所保存的环境。若先前没有对宏 setjmp 的调用，或者包含对宏 setjmp 调用的函数在间歇期间已经终止执行，则行为是未定义的。

注：例如，经执行 return 语句，或由于另一个 longjmp 调用已引起跳转到在一系列嵌套调用中较早的函数中的 longjmp 调用。

所有可访问的对象都具有调用函数 longjmp 时的值，但有一个例外：对于不为 volatile 限定类型的自动存储期对象，当该对象局部于包含相应的对宏 setjmp 调用的函数，且已在对 setjmp 的调用和 longjmp 调用之间被改变了时，该对象的值不可确定。

由于函数 longjmp 越过正常的函数调用和返回机制，所以函数 longjmp 应在中断、信号以及与它们相关的任何函数中正确地执行。然而，若函数 longjmp 是从嵌套的信号处理程序(即从作为在处理另一个信号期间激发的信号的结果而激活的函数)中调用的，则函数 longjmp 的行为是未定义的。

返回值

在 longjmp 完成以后，程序继续执行，如同对应的对宏 setjmp 的调用刚返回由 val 所指定的值一样。函数 longjmp 不能使宏 setjmp 返回值0；若 val 为0，则宏 setjmp 返回值1。

7.7 信号处理程序库前导文卷<signal.h>

前导文卷<signal.h>声明了一种类型、两个函数和定义了一些宏，用于处理各种各样的*信号*（在程序执行期间可报告的情况）。

所定义的类型是：

sig_atomic_t

它是一个对象的整体类型，即使在出现异步中断时，也可作为一个原子实体访问。

所定义的宏是：

SIG_DFL

SIG_ERR

SIG_IGN

它们展开为具有各别值的常量表达式，类型与函数 signal 的第二个实参以及返回值相容，且它们的值与任何可声明的函数的地址比较结果都不相等；以及下列宏，它们中的每一个都展开为一个正的整型常量表达式，这些表达式的值是对应于所指定的条件的信号的号码。

SIGABRT	非正常终止，例如由函数 abort 所引发的
SIGFPE	一个错误的算术运算，例如除以零或引起结果溢出的运算
SIGILL	检测出一个无效的函数映象，例如一条非法指令
SIGINT	收到一个交互的“注意”信号
SIGSEGV	无效的存储区访问
SIGTERM	送给程序的终止请求

除作为显式调用函数 raise 的结果外，实现不必产生这些信号中的任何一个。实现也可指定附加的信号和指向不可声明的函数的指针，以及分别以 SIG 和一个大写字母及 SIG_ 和一个大写字母开始的宏。完全的信号集合、它们的语义以及对它们的默认处理是实现定义的；所有信号的号码都应是正值。

注：见“库的发展趋向”(7.13.5条)，信号号码的名称分别反映下列术语：异常终止、浮点异常、非法指令、中断、段越界和终止。

7.7.1 规定信号处理

7.7.1.1 函数 signal

调用序列规格说明

```
#include<signal.h>
void(*signal (int sig, void (*func)(int)))(int);
```

描述

函数 signal 选择三种方式之一，其中接收信号号码 sig 都是为了随后的处理。若 func 的值是 SIG_DFL，则进行对该信号的默认处理。若 func 的值是 SIG_IGN，则忽略该信号。否则 func 应指向一个当该信号出现时所调用的函数。这种函数称为*信号处理程序*。

当某信号出现时，若 func 指向一个函数，则首先执行等价于 signal(sig, SIG_DFL)；的动作，或执行一种实现定义的对该信号的阻塞。(若 sig 的值是 SIGILL，则 SIG_DFL 的其余部分是否发生是实现定义的。)然后执行等价于(*func)(sig)；的动作。函数 func 可由执行一条 return 语句或由调用函数 abort、exit 或 longjmp 终止。若 func 执行 return 语句且 sig 的值是 SIGFPE，或者任何其他实现定义的对应于一种计算异常的值，则行为是未定义的。否则，程序将在被中断处恢复执行。

若出现的信号不是调用函数 abort 或 raise 的结果，且若该信号的处理程序以对应于引起调用该处理程序的信号的号码为第一实参调用在标准库中除函数 signal 本身以外的任何函数，或者除对类型为 volatile sig_atomic_t 的静态存储期变量赋值外，引用任何静态存储期对象，则行为是未定义的，而且，若这种对函数 signal 的调用导致返回 SIG_ERR，则 errno 的值是不确定的。

注：若由异步信号处理程序产生任何信号，则函数 signal 的行为是未定义的。

在程序启动时，将对某些以实现定义的方式选择的信号执行等价于：

signal (sig,SIG_IGN)；

的动作，对所有其他实现定义的信号执行等价于：

signal(sig, SIG_DFL)；

的动作。

实现应表现为如同无任何库函数调用函数 signal 一样。

返回值

若能批准请求，函数 signal 返回对指定信号 sig 最近一次调用 signal 时 func 的值。否则，返回 SIG_ERR 的值，并将一个正值存入 errno 中。

提前引用的条文：函数 abort(7.10.4.1条)，函数 exit(7.10.4.3条)。

7.7.2 发送信号

7.7.2.1 函数 raise

调用序列规格说明

#include <signal.h>

int raise(int sig)；

描述

函数 raise 向正在执行的程序发送信号 sig。

返回值

若成功，函数 raise 返回0，否则返回非零值。

7.8 变长实参库前导文卷<stdarg.h>

前导文卷<stdarg.h>声明了一种类型并定义了三个宏，用于访问在翻译时尚不清楚实参个数和类型的实参表。

可使用数目和类型都可变的实参调用函数。如6.7.1条中所述，其形参表包含一个或多个形参，其中最右边的形参在访问机制中起特殊作用，在本条中描述时记为*parmN* 。

所声明的类型是：

va_list

它是一种适合于存放由宏 va_start、va_arg 和 va_end 所需信息的类型。若希望访问变化的实参，被调用的函数应声明一个类型为 va_list 的对象(在本条中用 ap 表示)。对象 ap 可作为实参传递给另一个函数；若该函数以实参 ap 调用宏 va_arg，则 ap 的值在调用函数中将成为不确定的，并应在任何对 ap 的进一步引用前传递给宏 va_end。

7.8.1 访问变长实参表的宏

在本条中所描述的宏 va_start 和 va_arg 应确实实现为宏，而不应实际上是函数。对 va_end 是宏还是一个具有外部链接的标识符未作规定。若抑制宏定义以便访问一个实际的函数，或者程序中定义了一个名为 va_end 的外部标识符，则行为是未定义的。若希望访问变化的实参，则应在接收数目可变的实参的函数中调用宏 va_start 和 va_end。

7.8.1.1 宏 va_start

调用序列规格说明

#include <stdarg.h>

void va_start(va_list ap,*parmN*)；

描述

宏 va_start 应在访问任何未命名的实参之前调用。

宏 va_start 初始化 ap 以便随后由 va_arg 和 va_end 使用它。

形参*parmN* 是在函数定义中变长形参表中最右边形参(在,... 之前的一个)的标识符。若形参

parmN 被声明为 register 存储类、类型为函数或数组，或类型与应用默认的实参升格后所得的结果类型不相容，则宏 va_start 的行为是未定义的。

返回值

宏 va_start 不返回值。

7.8.1.2 宏 va_arg

调用序列规格说明

```
#include <stdarg.h>
type va_arg(va_list ap, type );
```

描述

宏 va_arg 展开为一个表达式，该表达式的类型和值与调用中下一个实参的类型和值相同。形参 ap 应与由 va_start 所初始化的 va_list ap 相同。每次调用 va_arg 都修改 ap 以便依次返回相继的实参值。形参 *type* 是所指定的类型名，使得可简单地通过在 *type* 前加一个前缀 * 而获得指向所指定类型对象的指针类型。若实际上并不存在下一个实参，或 *type* 与实际的下一个实参的类型(按默认的实参升格规则升格后)不相容，则宏 va_arg 的行为是未定义的。

返回值

在调用宏 va_start 后首次调用 va_arg 时，将返回由 *paramN* 所规定的实参之后的那个实参的值。后继的调用相继返回余下的实参。

7.8.1.3 宏 va_end

调用序列规格说明

```
#include <stdarg.h>
void va_end(va_list ap);
```

描述

宏 va_end 加速从某个函数的正常返回，该函数的变长实参表由初始化 va_list ap 的 va_start 的展开式所引用。宏 va_end 可以修改 ap 使它成为不再可用(无需中间插入的 va_start)。若无对应的对宏 va_start 的调用，或者在返回前未调用宏 va_end，则行为是未定义的。

返回值

宏 va_end 不返回值。

示例

函数 f1先收集实参表并存入一个数组，这些实参是指向串的指针(但不超过 MAXARGS 个实参)，然后将该数组作为单个实参传递给函数 f2。指针的个数由函数 f1的第一个实参指定。

```
#include <stdarg.h>
#define MAXARGS 31

void f1(int n_ptrs,...);
{
    va_list ap;
    char *array [MAXARGS];
    int ptr_no=0;

    if(n_ptrs>MAXARGS)
        n_ptrs=MAXARGS;
    va_start(ap,n_ptrs);
        while (ptr_no<n_ptrs)
```

```
            array[ptr_no++]=va_arg(ap,char *);
        va_end(ap);
        f2(n_ptrs,array);
    }
```

每次调用 f1时该函数的定义或形如 void f1(int,...);的声明都应是可见的

```
    void f1(int,...);
```

7.9 输入输出程序库前导文卷<stdio.h>

7.9.1 引言

前导文卷<stdio.h>声明了用于完成输入输出的三种类型、一些宏和许多函数。

所声明的类型是：

size_t(已在7.1.6条中描述)；

FILE

它是一种对象类型，能记录控制一个流所需的全部信息，包括其文卷位置指示符、指向与其相关的缓冲区的一个指针(若存在这种缓冲区)、一个记录是否出现了读/写出错的*出错指示符*、以及一个记录是否已到达文卷结尾的*文卷尾指示符*；以及

fpos_t

它是一种对象类型，能记录唯一地指定在文卷中的每一位置而所需的信息。

所定义的宏是：

NULL(已在7.1.6条中描述)；

_IOFBF

_IOLBF

_IONBF

它们展开为不同值的整型常量表达式，适合于用作函数 setvbuf 的第三个实参；

BUFSIZ

它展开为一个整型常量表达式，该表达式的值是由函数 setbuf 所使用的缓冲区的尺寸；

EOF

它展开为一个负的整型常量表达式，由某些函数返回以指示*文卷尾*，即在流中不再有输入信息；

FOPEN_MAX

它展开为一个整型常量表达式，该表达式的值是实现保证能同时打开的最少文卷数；

FILENAME_MAX

它展开为一个整型常量表达式，该表达式的值是一个 char 数组所需的尺寸，该数组足够大以存放实现所保证能打开的最长的文卷名串；

注：若实现对文卷名串的长度不加实际限制，则 FILENAME_MAX 的值应代之而为对意图存放文卷名串的数组所推荐的尺寸。当然，文卷名串的内容还受其他系统所特有的约束；所以不能期望*所有*可能的长度为 FILENAME_MAX 的串都能被成功地打开。

L_tmpnam

它展开为一个整型常量表达式，该表达式的值是一个 char 数组所需的尺寸，该数组足够大以存放由函数 tmpnam 所生成的临时性文卷名；

SEEK_CUR

SEEK_END

SEEK_SET

它们展开为不同值的整型常量表达式，适合于用作函数 fseek 的第三个实参；

TMP_MAX

它展开为一个整型常量表达式，该表达式的值是由函数 tmpnam 所生成的唯一文卷名的最少数量；

stderr

stdin

stdout

它们是"指向 FILE 的指针"类型的表达式，分别指向与标准出错信息流、标准输入流和标准输出流相关的 FILE 对象。

提前引用的条文：文卷(7.9.3条)，函数 fseek(7.9.9.2条)，流(7.9.2条)，函数 tmpnam(7.9.4.4条)。

7.9.2 流

不论是发往或来自诸如终端和磁带驱动器等物理设备的输入和输出，或是发往或来自结构存储设备所支持的文卷的输入和输出，都映射到逻辑数据*流*。流的性质比与它们对应的各种输入和输出都更一致。共支持两种形式的映射：*文本流* 和*二进制流*。

注：实现也可不必区分文本流和二进制流。在这类实现中，在文本流中无需新行字符，对一行的长度也无限制。

文本流是组成*行*的字符的有序序列，每一行由零个或多个字符再加上结尾用的新行字符组成。最后一行是否需要一个结尾用的新行字符由实现定义。为符合在宿主环境中表示正文的不同习惯，可在输入和输出中增加、修改或删除字符。因此，流中的字符与在外部表示中的字符间不一定存在一一对应关系。仅当在下列情况下，从文本流中读入的数据与早些时候写出到该流中的数据才必须比较结果相等：该数据仅由可印刷字符和控制字符横向制表以及新行字符组成；不存在紧后于空格字符的新行字符；且最后一个字符是新行字符。写出时紧先于新行字符的空格字符在读入时是否出现由实现定义。

二进制流是一个可透明地记录内部数据的字符的有序序列。从二进制流中读入的数据应与早些时候写出到该流中的数据比较结果相等。然而，在这类流的末尾可添加其数量由实现定义的空字符。

对环境的限制

实现应支持一行中连同结尾的新行字符在内至少包含254个字符的文本文卷。宏 BUFSIZ 的值至少应为256。

7.9.3 文卷

打开 文卷操作使一个流与一个外部文卷(可以是物理设备)相关联，打开操作可能涉及*创建* 一个新文卷。创建一个已存在的文卷将导致必要时丢弃该文卷中原有的内容。若文卷可支持定位请求(例如与终端相对立的磁盘文卷)，则与该流相关联的*文卷位置指示符* 将定位于该文卷的始端(字符号0)，除非该文卷是打开为添加模式的。对添加模式，文卷位置指示符初始时定位于文卷的始端还是末尾是由实现所定义的。文卷位置指示符由后继的读、写和定位请求维护，以便于有序地通过文卷。所有发生的输入都如同字符是由相继调用函数 fgetc 所读入的一样；所有发生的输出都如同字符是由相继调用 fputc 函数所写出的一样。

注：在基本文件中，文卷位置指示符被描述为*文卷指针*。本标准中不使用该术语，以避免与指向一个 FILE 类型对象的指针相混淆。

除7.9.5.3条中所定义的外，不截断二进制文卷。在文本流中写是否会导致相关联的文卷在超出该点处被截断则由实现定义。

当流是*非缓冲* 的时，可预期字符将尽可能快地从源中或在目的地出现。否则，可以先积累字符，再将它们按块传至宿主环境或从宿主环境中传入。当流是*完全缓冲* 的时，则当缓冲区被填满时字符才按块传至宿主环境或从宿主环境中传入。当流是*行缓冲* 的时，则当遇到一个新行字符时，就将字符按块传至宿主环境或从宿主环境中传入。而且，也可以在缓冲区被填满时，当对一个非缓冲的流请求输入时，或当对一个需要从宿主环境传输字符的行缓冲的流请求输入时，将字符按块传输至宿主环境。对这些特征的支持是实现定义的，且可受函数 setbuf 和 setvbuf 的影响。

通过*关闭* 文卷操作可将一个文卷与控制它的流解除关联。在与该文卷解除关联前，将对输出流进

行刷新(将任何尚未写出的缓冲区内容都传输到宿主环境)。在相关联的文卷(包括标准文本流)被关闭后,指向 FILE 对象的指针值是不确定的。长度为零的文卷(输出流并未对它写出任何字符)是否能实际存在也由实现定义。

文卷打开后可由同一个或另一个程序的执行重新打开,并重新声明或(若可重定位至文卷的始端时)修改其内容。若由函数 main 返回到其初始调用者,或者函数 exit 被调用,则在程序终止前,所有打开着的文卷都将被关闭(因此所有输出流都被刷新)。其他终止程序的途径,例如调用函数 abort,不一定能恰当地关闭所有文卷。

用于控制一个流的 FILE 对象的地址可能是有重要意义的;一个 FILE 对象的副本不一定能代替原始对象。

在程序启动时,有三个文本流是预先定义好的且无需显式打开——*标准输入流*(用于读入常规输入),*标准输出流*(用于写出常规输出),和*标准出错信息流*(用于写出诊断消息)。当被打开时,标准出错信息流不是完全缓冲的;而当且仅当可以确定并非与交互设备相关时,标准输入流和标准输出流才是完全缓冲的。

打开其余(非临时性)文卷的函数需要一个*文卷名*,文卷名是一个串。有效文卷名的构成规则由实现定义。同一文卷能否同时被打开多次也由实现定义。

对环境的限制

宏 FOPEN_MAX 的值应至少为8,包括三个标准文本流在内。

提前引用的条文:函数 exit(7.10.4.3条),函数 fgetc(7.9.7.1条),函数 fopen(7.9.5.3条),函数 fputc(7.9.7.3条),函数 setbuf(7.9.5.5条),函数 setvbuf(7.9.5.6条)。

7.9.4 文卷操作

7.9.4.1 函数 remove

调用序列规格说明

```
#include <stdio.h>
int remove(const char *filename);
```

描述

函数 remove 使得名字是由 filename 所指向的串的文卷不再能通过该名来访问。以后试图用该名打开该文卷时将失败,除非是用该名重新创建一个文卷。若该文卷是打开的,则函数 remove 的行为由实现定义。

返回值

若操作成功,函数 remove 返回零,否则返回非零值。

7.9.4.2 函数 rename

调用序列规格说明

```
#include <stdio.h>
int rename(const char *old,const char *new);
```

描述

函数 rename 使得名字是由 old 所指向的串的文卷从此改名为由 new 所指向的串所给定的名字。命名为 old 的文卷不再能通过该名访问。若在调用函数 rename 之前,已存在由 new 所指向的串所命名的文卷,则行为是实现定义的。

返回值

若操作成功,函数 rename 返回零,否则返回非零值。在操作失败的情况下,若先前的文卷存在,则它仍使用原先的名字。

注:实现会导致函数 rename 失败的理由有:该文卷已打开或有必要先复写其内容以完成换名。

7.9.4.3 函数 tmpfile

调用序列规格说明

#include <stdio.h>

FILE *tmpfile(void);

描述

函数 tmpfile 创建一个临时性二进制文卷，该文卷在关闭或程序终止时将被自动删除。若程序非正常终止，是否删除一个打开的临时性文卷则由实现定义。该文卷以“wb+”模式打开以作更新。

返回值

函数 tmpfile 返回一个指向它所创建的文卷的流的指针。若不能创建该文卷，则函数 tmpfile 返回一个空指针。

提前引用的条文：函数 fopen(7.9.5.3条)。

7.9.4.4 函数 tmpnam

调用序列规格说明

#include <stdio.h>

char *tmpnam(char *s);

描述

函数 tmpnam 生成一个串，该串是一个有效的文卷名，且不同于任何已存在的文卷名。

注：仅在该名与用对于实现是传统的命名规则所生成的文卷名不冲突的意义上，用由函数 tmpnam 所生成的串创建的文卷才是临时性的。仍有必要在用完之后，以及在程序终止前用函数 remove 删除这种文卷。

每次调用函数 tmpnam 时，它生成一个不同的串，至多 TMP_MAX 次。若调用超过 TMP_MAX 次，则行为由实现定义。

实现应表现为如同无任何库函数调用函数 tmpnam 一样。

返回值

若实参是一个空指针，则函数 tmpnam 将其结果留在一个内部的静态对象中，并返回一个指向该对象的指针。后继调用函数 tmpnam 时可能修改该对象。若实参不是空指针，则假定它是指向一个至少 L tmpnam 个 char 的数组；函数 tmpnam 将结果写入该数组，并将实参作为返回值。

对环境的限制

宏 TMP_MAX 的值至少应为25。

7.9.5 文卷访问函数

7.9.5.1 函数 fclose

调用序列规格说明

#include <stdio.h>

int fclose (FILE *stream);

描述

函数 fclose 导致由 stream 所指向的流被刷新并且关闭相关联的文卷。对于该流，任何尚未写出的缓冲数据都提交给宿主环境以便写入该文卷；任何尚未读入的缓冲数据都被丢弃。然后该流与该文卷解除关联。若相关联的缓冲区是自动分配的，则回收该缓冲区。

返回值

若成功地关闭了该流，函数 fclose 返回零，若检测出任何出错则返回 EOF。

7.9.5.2 函数 fflush

调用序列规格说明

#include <stdio.h>

int fflush(FILE *stream);

描述

若 stream 指向一个输出流或其中最近的操作不是输入的更新流，则函数 fflush 导致所有对于该流尚未写出的数据都提交给宿主环境以便写入该文卷；否则函数 fflush 的行为是未定义的。

若 stream 是一个空指针，则函数 fflush 对所有最近的操作不是输入的输出流或更新流执行刷新动作。

返回值

若出现写错误，函数 fflush 返回 EOF，否则返回零。

7.9.5.3 函数 fopen

调用序列规格说明

#include <stdio.h>

FILE * fopen(const char * filename, const char * mode);

描述

函数 fopen 打开一个文卷，该文卷的名是由 filename 所指向的串，并使一个流与之关联。

实参 mode 指向一个由以下序列之一开始的串：

r	打开文本文卷用于读
w	截断为零长度或创建一个文本文卷用于写
a	添加；打开或创建一个文本文卷用于在文卷尾处开始写
rb	打开二进制文卷用于读
wb	截断为零长度或创建一个二进制文卷用于写
ab	添加；打开或创建一个二进制文卷用于在文卷尾处开始写
r+	打开文本文卷用于更新(读和写)
w+	截断为零长度或创建一个文本文卷用于更新
a+	添加；打开或创建一个文本文卷用于更新，在文卷尾处开始写
r+b 或 rb+	打开二进制文卷用于更新(读和写)
w+b 或 wb+	截断为零长度或创建一个二进制文卷用于更新
a+b 或 ab+	添加；打开或创建一个二进制文卷用于更新，在文卷尾处开始写

注：在这些序列后可有附加的字符。

若文卷不存在或不能读时，则用读模式打开文卷(在实参 mode 中第一个字符是'r')将失败。

用添加模式打开文卷(在实参 mode 中第一个字符是'a')使得所有后继对该文卷的写都强制为从当时的文卷尾处开始，不论是否有中间插入的函数 fseek 调用。在某些实现中用添加模式打开二进制文卷(在上述 mode 实参表中，'b'作为第二或第三个字符)时，由于充填了空字符，可能使该流的文卷位置指示符的初始定位超过最后所写出的数据。

当用更新模式打开文卷(在上述 mode 实参表中，'+'作为第二或第三个字符)时，输入与输出都可在相关联的流上执行。然而，输出之后可能不能在无中间插入的函数 fflush 调用，或者对文卷定位函数(fseek、fsetpos、或 rewind)的调用的情况下直接跟有输入；输入之后也可能不能在无中间插入的对文卷定位函数的调用的情况下直接跟有输出，除非输入操作遇到文卷尾。在某些实现中，用更新模式打开或创建一个文本文卷可能会代之以打开或创建一个二进制文卷。

当打开时，当且仅当能确定不是指交互式设备时，流才是完全缓冲的。对该流的出错指示符和文卷终指示符都清除。

返回值

函数 fopen 返回一个指向控制该流的对象的指针。若打开操作失败，函数 fopen 返回一个空指针。

提前引用的条文：文卷定位函数(7.9.9条)。

7.9.5.4 函数 freopen

调用序列规格说明

#include <stdio.h>

FILE * freopen(const char * filename,const char * mode, FILE * stream);

描述

函数 freopen 打开一个文卷，该文卷的名字是由 filename 所指向的串，并使由 stream 所指向的流与该文卷相关联。实参 mode 的使用与在函数 fopen 中的使用相同。

注：函数 freopen 的基本用途是改变与标准文本流 stderr、stdin 或 stdout 相关联的文卷，因为那些标识符不一定是能赋予函数 fopen 的返回值的可修改的左值。

函数 freopen 首先试图关闭与所指定的流相关联的任何文卷。若不能成功地关闭该文卷也不管。对该流的出错指示符和文卷尾指示符都清除。

返回值

若打开操作失败，则函数 freopen 返回一个空指针，否则函数 freopen 返回 stream 的值。

7.9.5.5 函数 setbuf

调用序列规格说明

#include <stdio.h>

void setbuf(FILE * stream, char * buf);

描述

除不返回值外，函数 setbuf 等价于对 mode 用值_IOFBF、对 size 用值 BUFSIZ、或(当 buf 为空指针时)对 mode 用值_IONBF 调用函数 setvbuf。

返回值

函数 setbuf 不返回值。

提前引用的条文：函数 setvbuf(7.9.5.6条)。

7.9.5.6 函数 setvbuf

调用序列规格说明

#include <stdio.h>

int setvbuf(FILE * stream, char * buf,int mode, size_t size);

描述

仅在由 stream 所指向的流已与一个打开的文卷相关联，并在该流上执行任何其他操作之前，才可调用函数 setvbuf。实参 mode 的值决定对 stream 的输入输出应如何缓冲，如下：_IOFBF 完全缓冲；_IOLBF 行缓冲；_IONBF 无缓冲。若 buf 不是一个空指针，可使用它所指向的数组代替由函数 setvbuf 所分配的缓冲区。实参 size 指定该数组的尺寸。在任何时候，该数组的内容都是不确定的。

注：缓冲区的生存期必须至少与所打开的文卷一样长，所以应在自动存储期的缓冲区在从块出口时被回收之前关闭该流。

返回值

成功时函数 setvbuf 返回零，若对 mode 给出了一个无效值或不能批准请求时则返回非零值。

7.9.6 格式化输入输出函数

7.9.6.1 函数 fprintf

调用序列规格说明

#include <stdio.h>

int fprintf(FILE * stream, const char * format,…);

描述

函数 fprintf 在由 format 所指向的串的控制下将输出写到由 stream 所指向的流，由 format 所指向的串规定了后继的实参在输出时应如何转换。如与格式相对应的实参不够，则函数 fprintf 的行为是未定义的。若格式已用完而还有实参，则将会对多余的实参求值，但是忽略这些实参。函数 fprintf 在到达

格式串的末尾时返回。

格式应是一个多字节字符序列，以初始转义状态开始和结束。格式由零个或多个下列指示组成：一般的多字节字符(非%)，对它们将不作改变而复写到输出流中；以及转换说明符，每个转换说明符都导致取零个或多个后继的实参。转换说明符由%字符引导。在%字符之后，将依次出现下列内容：

——修改转换说明符意义的零个或多个*旗标*(可以任意次序出现)。

——一个任选的最小*域宽*。若转换后的值的字符数比域宽少，则将在域宽的左边(或若给出了左对齐旗标时则在右边，左对齐旗标将在后面描述)充填默认的空格字符。域宽的形式是星号 *(将在后面描述)，或者十进制整数。

注：注意0是作为旗标用的，而不是作为域宽的开始。

——一个任选的*精度*，它给出了：对 d、i、o、u、x 和 x 转换所应出现的最少数字字符数；对 e、E 和 f 转换在十进制小数点字符后所应出现的数字字符数；对 g 或 G 转换的最大有效数字字符数；或在 S 转换时从一个串中写出的最大字符数。精度的形式为一个圆点(.)后跟一个星号 *(将在后面描述)或一个任选的十进制整数；若只给出了圆点，则精度取零。若精度与任何其他转换说明符同时出现，则函数 fprintf 的行为是未定义的。

——一个任选的字母 h，它说明在它之后的 d、i、o、u、x 或 X 转换说明符适用于一个 short int 或 unsigned short int 实参(该实参应已按整型升格规则升格，且其值应在打印前转换为 short int 或 unsigned short int)；或任选的字母 h，它说明在它之后的 n 转换说明符适用于一个指向 short int 的指针实参；或任选的字母 l，它说明在它之后的 d、i、o、u、x 或 X 转换说明符适用于一个 long int 或 unsigned long int 实参；或任选的字母 l，它说明在它之后的 n 转换说明符适用于一个指向 long int 的指针实参；或任选的字母 L，它说明在它之后的 e、E、f、g、或 G 转换说明符适用于一个 long double 实参。若任何其他转换说明符中出现 h、l 或 L，则函数 fprintf 的行为是未定义的。

——一个字符，说明要应用的转换类型。

如上面所述，域宽或精度，或二者，都可用星号指示。在此情况下，由一个 int 实参提供域宽或精度。说明域宽或精度，或二者的实参应以该次序出现在要转换的任何实参之前。负的域宽作为一个一旗标后跟一个正的域宽对待。负的精度将按省略精度对待。

旗标字符及它们的意义如下：

－　转换后的结果在域内是左对齐的。(若不给出此旗标，则将为右对齐。)

＋　有符号数转换后的结果将总是以正号或负号开始。(或不给出此旗标，则仅在转换负值时才以符号开始。

空格　若有符号数转换的第一个字符不是符号，或有符号数转换的结果无字符，则结果将加一个前缀间隔字符。若旗标*空格*和＋同时出现，则*空格*旗标将被忽略。

#　结果将被转换为一种“替代形式”。对于 o 转换，它增加精度以强制结果的第一位数字是零。对 x(或 X)转换，非零结果将有0x(或0X)为前缀。对 e、E、f、g 和 G 转换，结果将总是包含一个十进制小数点字符，即使后面没有数字也是如此。(通常，仅在后面有数字时这些转换的结果中才出现十进制小数点字符。)对 g 和 G 转换，结果中将不去掉结尾的0。对其他转换，行为是未定义的。

0　对 d、i、o、u、x、X、e、E、f、g 和 G 转换，(跟随任何符号或基数指示的)前导零将用于充填域宽而不充填空格字符。若0和－旗标同时出现，则0旗标将被忽略。对 d、i、o、u、x 和 X 转换，若规定了精度，则0旗标也将被忽略。对其他转换，函数 fprintf 的行为是未定义的。

转换说明符及它们的意义如下：

d,i　将 int 实参转换为形式为[－]dddd 的有符号十进制数。精度规定应出现的最少数字字符数，若要转换的值可用较少数位表示，则使用前导零将它扩展。默认的精度是1。用零精度转换一个零值的结果是无字符。

o,u,x,X　　将 unsigned int 实参转换为形式为*dddd* 的无符号的八进制(o),无符号的十进制(u),或无符号的十六进制(x 或X);字母 abcdef 用于 x 转换,字母 ABCDEF 用于 X 转换。精度规定应出现的最少数字字符数;若要转换的值可用较少数位表示,则使用前导零将它扩展。默认的精度是1。用零精度转换一个零值的结果是无字符。

f　　将 double 实参转换为形式为[－]ddd. ddd 的十进制记法,其中在十进制小数点后的数字字符数等于精度规格说明。若不出现精度,则按精度为6对待;若精度为零且未规定＃旗标,则不出现十进制小数点字符。若出现十进制小数点字符,则在它之前至少应出现一位数字。值舍入为恰当位数的数字。

e,E　　将 double 实参转换为[－]d. ddde±dd 的形式,其中在十进制小数点前有一位数字(若实参不为零则它也不为零),在十进制小数点后的数字字符数等于精度。若不出现精度,则按精度为6对待;若精度为零且未规定＃旗标,则不出现十进制小数点字符。值舍入为恰当位数的数字。E 转换说明符将产生以 E 代替 e 标志指数部分的数。指数部分总是至少包含两个数字。若值为零,则指数部分也为零。

g,G　　将 double 实参转换为 f 或 e 格式(或在 G 转换说明符的情况下转换为 E 格式),精度规定有效数的位数。若精度为零,则取为1。所用的格式取决于所要转换的值;格式 e(或 E)仅用在这类转换结果的指数部分小于－4或大于等于精度时。结果的小数部分中结尾的零将被移去;仅当其后跟有数字时才会出现十进制小数点字符。

c　　将 int 实参转换为 unsigned char,并写出所得到的字符。

s　　实参应为指向字符类型数组的指针。将数组中的字符写出,直至但不包括结尾用的空字符为止;若规定了精度,则所写出的字符数不超过精度所规定的。若未规定精度或精度大于数组的尺寸,则数组中应包含一个空字符。

注:对多字节字符未规定特殊的支持。

p　　实参应为指向 void 的指针。该指针的值将以实现定义的方式转换为一系列可印刷字符。

n　　实参应为指向一个整型对象的指针,在该对象中将*写入* 由本次函数 fprintf 调用迄今为止已写到输出流中的字符数。对实参不作转换。

%　　写出一个%字符。对实参不作转换。完整的转换规格说明应为%%。

若转换规格说明无效,则函数 fprintf 的行为是未定义的。

注:见(7.13.6条)"库的发展趋向"。

若任何实参是一个联合或聚集类型,或指向联合或聚集类型(除使用%s 转换的字符类型的数组,或使用%p 转换的指针外),则函数 fprintf 行为是未定义的。

决不会出现不存在的或者较小的域宽引起截断一个域的情况;若转换后的结果超过域宽,则将域扩展到可容纳转换后的结果。

返回值

函数 fprintf 返回所传输的字符数,或当出现输出错误时返回一个负值。

对环境的限制

由单个转换所能产生的最大字符数至少应为509。

示例

打印形式为"Sunday, July 3,10:02"的日期和时间,后跟 π 至小数点后五位:

```
#include <math.h>
#include <stdio.h>
/* ... */
char *weekday, *month;  /* 指向串的指针 */
```

```
int day, hour, min;
fprintf (stdout, "%s,%s  %d,%.2d:%.2d\n",
        weekday, month, day, hour,min);
fprintf(stdout, "pi=%.5f\n",4 * atan(1.0));
```

7.9.6.2 函数 fscanf

调用序列规格说明

```
#include <stdio.h>
int fscanf)(FILE * stream, const char * format,...);
```

描述

函数 fscanf 在由 format 所指向的串的控制下从由 stream 所指向的流中读输入，由 format 所指向的串规定了可接受的输入序列以及如何转换输入以便赋值，使用后继的实参作为指向接收转换后的输入的对象的指针。如与格式相对应的实参不够，则函数 fscanf 的行为是未定义的。若格式已用完而还有实参，则将会对多余的实参求值，但是忽略这些实参。

格式应是一个多字节字符序列，以初始转义状态开始和结束。格式由零个或多个下列指示组成：一个或多个白空类字符；一个一般的多字节字符(既不是%也不是白空类符)；或一个转换说明符。每个转换说明符都由%字符引导。在%字符之后，将依次出现下列内容：

——一个任选的抑制赋值字符 *。

——一个任选的非零十进制整数，它规定了最大域宽。

——一个任选的字母 h、l 或 L，指示接收输入的对象的尺寸。若相应的实参是指向 short int 而不是 int 的指针，则转换说明符 d、i 和 n 之前应有字母 h，或若相应的实参是指向 long int 的指针，则转换说明符 d、i 和 n 之前应有字母 l。类似地，若相应的实参是指向 unsigned short int 而不是 unsigned int 的指针，则转换说明符 o、u 和 x 之前应有字母 h，或若相应的实参是指向 unsigned long int 的指针，则转换说明符 o、u 和 x 之前应有字母 l。最后，若相应的实参是指向 double 而不是 float 的指针，则转换说明符 e、f 和 g 之前应有字母 h，或若相应的实参是指向 long double 的指针，则转换说明符 e、f 和 g 之前应有字母 L。若 h、l 或 L 与任何其他转换说明符同时出现，则函数 fscanf 的行为是未定义的。

——一个字符，说明要应用的转换类型。有效的转换说明符在下面描述。

函数 fscanf 依次执行每个格式指示。若有一个指示失败(将在下面详细描述)，则函数 fscanf 返回。失败可描述为输入失败(由于无可用的输入字符)，或匹配失败(由于不恰当的输入)。

由白空类字符组成的指示的执行是读输入直至第一个非白空字符(该白空类字符不读)，或无更多的字符可读为止。

是一般多字节字符的指示的执行是读流中后面的字符。若有一个字符与组成该指示的字符不同，则该指示失败，且所不同的那个字符及其后面的字符都不读。

是转换说明符的指示定义一组匹配输入的序列，对每一说明符的匹配序列都将在下面描述。转换说明符以下列步骤执行：

除非规格说明中包括[、c 或 n 说明符，否则输入中的白空类字符(如由函数 isspace 所规定的)将被跳过。

注：对于指定的位段宽度，将不计这些白空类字符。

除非规格说明中包括一个 n 说明符，否则从流中读入一个输入项。输入项定义为输入字符的最长匹配序列，除非它超出所规定的域宽。在超出域宽的情况下，输入项是该序列中长度为域宽的初始序列。在输入项后的第一个字符(若还有字符)不读入。若输入项的长度为零，则指示的执行失败；这种情况是匹配失败，除非是由出错阻止了从流中的输入，那时是输入失败。

除在%说明符的情况下，输入项(或在%n 指示的情况下，输入字符计数)将转换为适合于该转换说明符的类型。若输入项不是一个匹配序列，则该指示的执行失败；这种情况是匹配失败。除非已由 * 指明

了抑制赋值，转换的结果将放到由跟在 format 实参之后的、尚未接收转换结果的第一个实参所指示的对象中。若该对象的类型不合适，或所提供的空间不能表示转换后的结果，则函数 fscanf 的行为是未定义的。

下列转换说明符是有效的：

d　匹配一个任选的有符号十进制整数，其格式与函数 strtol 的 base 实参值为10时所预期的主体序列相同。相应的实参应为指向整数的指针。

i　匹配一个任选的有符号整数，其格式与函数 strtol 的 base 实参值为0时所预期的主体序列相同。相应的实参应为指向整数的指针。

o　匹配一个任选的有符号八进制整数，其格式与函数 strtoul 的 base 实参值为8时所预期的主体序列相同。相应的实参应为指向无符号整数的指针。

u　匹配一个任选的有符号十进制整数，其格式与函数 strtoul 的 base 实参值为10时所预期的主体序列相同。相应的实参应为指向无符号整数的指针。

x　匹配一个任选的有符号十六进制整数，其格式与函数 strtoul 的 base 实参值为16时所预期的主体序列相同。相应的实参应为指向无符号整数的指针。

e、f、g　匹配一个任选的有符号浮点数。其格式与函数 strtod 所预期的主体序列相同。相应的实参应为指向浮点数的指针。

s　匹配一个非白空类字符序列。相应的实参应为指向一个数组的初始字符位置的指针，该数组足够大以接收该序列以及一个结尾用的空字符，空字符将是自动加入的。

注：对多字节字符未规定特殊的支持。

[　从一个预期的字符集合(*扫描集*)中匹配一个非空的字符序列。相应的实参应为指向一个数组的初始字符位置的指针，该数组足够大以接收该序列以及一个自动加入的、结尾用的空字符。转换说明符包括 format 串中[之后直至与之匹配的右方括号(])前、包括该右方括号在内的所有字符。在方括号之间的字符(*扫描表*)构成扫描集，除非在左方括号后的字符是一个声调符号(^)，在该情况下扫描集包括不出现在扫描表中声调符号与右方括号之间的所有字符。若转换说明符以[ ]或[^]开始，则该右方括号字符是在扫描表中，且下一个右方括号字符才是终止规格说明的(与左方括号)相匹配的右方括号；否则第一个右方括号即是终止规格说明的那一个。若一个－字符是在扫描表中且不是第一个字符，也不是在第一个字符是^ 时的第二个字符，也不是最后一个字符，则其行为是实现定义的。

注：对多字节字符未规定特殊的支持。

c　匹配一个字符序列，字符数由域宽规定(若指示中无域宽则为1)。相应的实参应为指向一个数组的第一个字符位置的指针，该数组足够大以接收该序列。不加入空字符。

注：对多字节字符未规定特殊的支持。

p　匹配一个实现定义的序列集合，该序列集合应与由函数 fprintf 的%p 转换可产生的序列集合相同。相应的实参应为一个指向 void 的指针的指针。对输入项的解释由实现定义。若输入项是一个在同一次程序执行中早些时候转换后的值，则结果所得的指针与该值比较结果应相等；否则%p 转换的行为是未定义的。

n　不消耗输入。相应的实参应为指向一个整数的指针，在该整数对象中将写入由本次 fscanf 函数调用迄今为止已从输入流中读入的字符数。执行%n 指示不增加函数 fscanf 执行完成时所返回的赋值计数。

%　匹配一单个的%；不作转换或赋值。完全的转换说明符应为%%。

若转换说明符无效，则函数 fscanf 的行为是未定义的。

注：见(7.13.6条)“库的发展趋向”。

转换说明符 E、G 和 X 也是有效的，其行为分别与 e、g 和 x 相同。

若在输入过程中遇到文卷尾，则终止转换。若在读入任何与当前指示相匹配的字符（不是前导白空类符，那是允许的）之前遇到文卷尾，则以输入失败终止当前指示的执行；否则，除非当前指示的执行是以匹配失败终止的，否则后面的任何指示的执行也以输入失败终止。

若因某输入字符与说明符相抵触而终止转换，则不读该字符，将它留在输入流中。除非是与某个指示匹配，否则结尾的白空类符（包括新行字符）均留在流中，不读入。除通过%n 指示外，字面值匹配和抑制赋值的成功与否是不能直接确定的。

返回值

若在任何转换前出现输入失败，则函数 fscanf 返回宏 EOF 的值。否则，fscanf 函数返回所赋值的输入项数，在早期匹配失败的情况下，该返回值可小于所实际提供的项数，甚至等于零。

示例

a. 调用：

```
#include <stdio.h>
/* ... */
int n, i;float x;char name[50];
n=fscanf(stdin, "%d%f%s",&i,&x,name);
```

对于输入行：

```
25 54.32E-1 thompson
```

将对*n* 赋于值3，对*i* 赋于值25，对*x* 赋于值5.432，且*name* 中将包含 thompson\0。

b. 调用：

```
#include <stdio.h>
/* ... */
int n, i;float x;char name[50];
fscanf(stdin, "%2d%f%*d%[0123456789]",&i,&x,name);
```

对于输入行：

```
56789 0123 56a72
```

对*i* 赋于值56，对*x* 赋于值789.0，将跳过0123，且*name* 中将包含56\0。输入流中下一个读入的字符将是 a。

c. 要重复地从 stdin 中读入一个量值、一个度量单位和一个项目名称，程序可写作：

```
#include <stdio.h>
/* ... */
int count; float quant; char units[21],item[21];
while (! feof(stdin) &&! ferror (stdin)){
    count =fscanf (stdin,"%f%20s of %20s",
        &quant, units, item);
    fscanf (stdin,"%*[^ \n]");
}
```

若 stdin 流中包含下列行：

```
2 quarts of oil
-12.8degrees Celsius
lots of luck
10.0LBS     of
dirt
100ergs of energy
```

上例的执行可与执行下列赋值语句类比：

```
quant=2;strcpy(units,"quarts");strcpy(item,"oil");
count=3;
quant=-12.8;strcpy(units,"degrees");
count=2;/*"C"未能与"0"匹配*/
count=0;/*"/"未能与"%f"匹配*/
quant=10.0;strcpy(units,"LBS");strcpy(item,"dirt");
count=3;
count=0;/*"100e"未能与"%f"匹配*/
count=EOF;
```

提前引用的条文：函数 strtod(7.10.1.4条)，函数 strtol(7.10.1.5条)，函数 strtoul(7.10.1.6条)。

7.9.6.3 函数 printf

调用序列规格说明

```
#include <stdio.h>
int printf(const char *format,...);
```

描述

函数 printf 等价于使用函数 printf 的所有实参、并在这些实参前插入一个 stdout 实参的 fprintf 函数。

返回值

函数 printf 返回所传输的字符数，或当输出出错时返回一个负值。

7.9.6.4 函数 scanf

调用序列规格说明

```
#include <stdio.h>
int scanf(const char *format,...);
```

描述

函数 scanf 等价于使用函数 scanf 的所有实参、并在这些实参前插入一个实参 stdin 的函数 fscanf。

返回值

若在任何转换前出现输入失败，则函数 scanf 返回宏 EOF 的值。否则，函数 scanf 返回所赋值的输入项数，在早期匹配失败的情况下，该返回值可能小于所实际提供的项数，甚至等于零。

7.9.6.5 函数 sprintf

调用序列规格说明

```
#include <stdio.h>
int sprintf(char *s,const char *format,...);
```

描述

函数 sprintf 除实参 s 指定了一个用于写入所产生的输出的数组，而不是写入某个流中外，等价于函数 fprintf。在写出的字符末尾写一个空字符；该空字符不计入所返回的累计数中。若出现在重叠的对象之间的复写，则函数 sprintf 的行为是未定义的。

返回值

函数 sprintf 返回写入该数组的字符数，不包括结尾用的空字符。

7.9.6.6 函数 sscanf

调用序列规格说明

```
#include <stdio.h>
int sscanf(const char *s, const char *format,...);
```

描述

函数 sscanf 除实参 s 指定一个从中获取输入的串，而不是从某个流中读入外，等价于函数 fscanf。到达该串的末尾等价于函数 fscanf 遇到文卷尾的情况。若出现在重叠的对象之间的复写，则函数 sscanf 的行为是未定义的。

返回值

若在任何转换前出现输入失败，则函数 sscanf 返回宏 EOF 的值。否则，函数 sscanf 返回所赋值的输入项数，在早期匹配失败的情况下，该返回值可能小于所实际提供的项数，甚至等于零。

7.9.6.7 函数 vfprintf

调用序列规格说明

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, va_list arg);
```

描述

函数 vfprintf 等价于函数 fprintf，但用 arg 代替了变长实参表，arg 应已由宏 va_start 及可能的后继 va_arg 调用初始化。函数 vfprintf 不调用宏 va_end。

注：由于函数 vfprintf、vsprintf 和 vprintf 调用宏 va_arg，在返回后 arg 的值是不确定的。

返回值

函数 vfprintf 返回所传输的字符数，或当输出出错时返回一个负值。

示例

下例说明了在一个通用出错报告例行程序中函数 vfprintf 的使用：

```
#include <stdarg.h>
#include <stdio.h>

void error (char *function_name, char *format,...)
{
    va_list args;
    va_start(args, format);
    /*打印出导致出错的函数名*/
    fprintf(stderr, "ERROR in %s:",function_name);
    /*打印出其余信息*/
    vfprintf(stderr, format, args);
    va_end(args);
}
```

7.9.6.8 函数 vprintf

调用序列规格说明

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

描述

函数 vprintf 等价于函数 printf，但用 arg 代替了变长实参表，arg 应已由宏 va_start 及可能的后继 va_arg 调用初始化。函数 vprintf 不调用宏 va_end。

注：由于函数 vfprintf、vsprintf 和 vprintf 调用宏 va_arg，在返回后 arg 的值是不确定的。

返回值

函数 vprintf 返回所传输的字符数，或当输出出错时返回一个负值。

7.9.6.9 函数 vsprintf

调用序列规格说明

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *s, const char *format,va_list arg);
```

描述

函数 vsprintf 等价于函数 sprintf，但用 arg 代替了变长实参表，arg 应已由宏 va_start 及可能的后继 va_arg 调用初始化。函数 vsprintf 不调用宏 va_end。若在重叠的对象间发生复写，则函数 vsprintf 的行为是未定义的。

注：由于函数 vfprintf、vsprintf 和 vprintf 调用宏 va_arg，在返回后 arg 的值是不确定的。

返回值

函数 vsprintf 返回写入该数组的字符数，不包括结尾用的空字符。

7.9.7 字符输入输出函数

7.9.7.1 函数 fgetc

调用序列规格说明

```
#include <stdio.h>
int fgetc(FILE *stream);
```

描述

函数 fgetc 从由 stream 所指向的输入流中获得下一个字符(若还有字符)作为转换为 int 的 unsigned char，并当该文卷位置指示符有定义时推进与该流相关联的文卷位置指示符。

返回值

函数 fgetc 返回由 stream 所指向的输入流中的下一个字符。若该流已处于文卷尾，则置位该流的文卷尾指示符且函数 fgetc 返回 EOF。若出现了读错误，则置位该流的出错指示符且函数 fgetc 返回 EOF。

注：文卷尾和读出错可通过使用函数 feof 和 ferror 来区别。

7.9.7.2 函数 fgets

调用序列规格说明

```
#include <stdio.h>
char *fgets(char *s,int n,FILE *stream);
```

描述

函数 fgets 从由 stream 所指向的输入流中读字符至由 s 所指向的数组中，最多读入的字符数比由 n 所规定的少一个。在新行字符(它将被保留)后或文卷尾后无附加的字符被读入。在最后一个字符读入数组后紧接着写入一个空字符。

返回值

若成功，函数 fgets 返回 s。若遇到文卷尾且未读入字符到数组中，则该数组的内容保持不变，并返回一个空指针。若在操作过程中出现读错误，该数组的内容是不确定的，且返回一个空指针。

7.9.7.3 函数 fputc

调用序列规格说明

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

描述

函数 fputc 将由 c 所表示的字符(转换为 unsigned char)写到由 stream 所指向的输出流中由与该流相关联的文卷位置指示符(若该文卷位置指示符有定义)所指示的位置，并恰当地推进该文卷位置指示

符。若该文卷不能支持定位请求，或该流是以添加模式打开的，则将该字符添加到该输出流的末尾。

返回值

函数 fputc 返回所写出的字符。若出现了写错误，则置位该流的出错指示符且函数 fputc 返回 EOF。

7.9.7.4 函数 fputs

调用序列规格说明

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

描述

函数 fputs 将由 s 所指向的串写到由 stream 所指向的输出流中。终止串的空字符不写出。

返回值

若出现了写错误，则函数 fputs 返回 EOF；否则返回一个非负值。

7.9.7.5 函数 getc

调用序列规格说明

```
#include <stdio.h>
int getc(FILE *stream);
```

描述

函数 getc 除是以宏实现的外，等价于函数 fgetc。函数 getc 可能对 stream 求值多于一次，所以其实参决不应是有副作用的表达式。

返回值

函数 getc 返回由 stream 所指向的输入流中的下一个字符。若该流已处于文卷尾，则置位该流的文卷尾指示符且函数 getc 返回 EOF。若出现了读错误，则置位该流的出错指示符且函数 getc 返回 EOF。

7.9.7.6 函数 getchar

调用序列规格说明

```
#include <stdio.h>
int getchar (void);
```

描述

函数 getchar 等价于实参为 stdin 的函数 getc。

返回值

函数 getchar 返回由 stdin 所指向的输入流中的下一个字符。若该流已处于文卷尾，则置位该流的文卷尾指示符且函数 getchar 返回 EOF。若出现了读错误，则该流的出错指示符被置位且函数 getchar 返回 EOF。

7.9.7.7 函数 gets

调用序列规格说明

```
#include <stdio.h>
char *gets (char *s);
```

描述

函数 gets 从由 stdin 所指向的流中读字符至由 s 所指向的数组中，直至遇到文卷尾或读入一个新行字符为止。任何新行字符都被丢弃，且在最后一个字符读入数组后紧接着写入一个空字符。

返回值

若成功，函数 gets 返回 s。若遇到文卷尾且未读入字符到数组中，则该数组的内容保持不变，并返回一个空指针。若在操作过程中出现读错误，该数组的内容是不确定的，且返回一个空指针。

7.9.7.8 函数 putc

调用序列规格说明

```
#include <stdio.h>
int putc(int c,FILE *stream);
```

描述

函数 putc 除是以宏实现的外，等价于函数 fputc。函数 putc 可能对 stream 求值多于一次，所以其实参决不应是有副作用的表达式。

返回值

函数 putc 返回所写出的字符。若出现了写错误，则置位该流的出错指示符且函数 putc 返回 EOF。

7.9.7.9 函数 putchar

调用序列规格说明

```
#include <stdio.h>
int putchar (int c);
```

描述

函数 putchar 等价于以 stdout 为第二实参的函数 putc。

返回值

函数 putchar 返回所写出的字符。若出现了写错误，则置位该流的出错指示符且函数 putchar 返回 EOF。

7.9.7.10 函数 puts

调用序列规格说明

```
#include <stdio.h>
int puts(const char *s);
```

描述

函数 puts 将由 s 所指向的串写到由 stdout 所指向的输出流中，并将一个新行字符添加到输出中。终止串的空字符不写出。

返回值

若出现了写错误，则函数 puts 返回 EOF；否则返回一个非负值。

7.9.7.11 函数 ungetc

调用序列规格说明

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

描述

函数 ungetc 将由 c 所表示的字符(转换为 unsigned char)退回由 stream 所指向的输入流中。所退回的字符可由随后对该流的读操作以与退回相反的次序返回。成功的中间插入的(对由 stream 所指向的流)文卷定位函数(fseek、fsetpos、或 rewind)调用将丢弃在该流中的任何所退回的字符。对应于该流的外部存储不变。

保证可退回一个字符。若对同一个流连续调用函数 ungetc 的次数太多，而无中间插入的对该流的读操作或文卷定位操作，则 ungetc 操作可能失败。

若 c 的值等于宏 EOF 的值，则 ungetc 操作失败且输入流不变。

对函数 ungetc 的成功的调用将清除该流的文卷尾指示符。在读入或丢弃所有被退回的字符后，该流的文卷位置指示符的值应与退回字符前相同。对文本流，其文卷位置指示符的值在成功地调用函数 ungetc 之后，直至读入或丢弃所有被退回的字符之前是未规定的。对二进制流，其文卷位置指示符的值在每次成功地调用函数 ungetc 后减1；若文卷位置指示符的值在调用前为零，则在调用后是不确定的。

返回值

函数 ungetc 返回转换后所退回的字符；或当操作失败时返回 EOF。

提前引用的条文：文卷定位函数(7.9.9条)。

7.9.8 直接输入输出函数

7.9.8.1 函数 fread

调用序列规格说明

#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

描述

函数 fread 从由 stream 所指向的流中读直至 nmemb 个元素到由 ptr 所指向的数组中，元素的尺寸由 size 规定。当该流的文卷位置指示符有定义时该文卷位置指示符推进成功地读入的字符数。若出错，则该流的文卷位置指示符的结果值是不确定的。若只读入某元素的一部分，则其值也是不确定的。

返回值

函数 fread 返回成功地读入的元素数，若出现了读错误或遇到文卷尾，则该数可能小于 nmemb。若 size 或 nmemb 为零，则函数 fread 返回零，且该数组的内容和流的状态都保持不变。

7.9.8.2 函数 fwrite

调用序列规格说明

#include <stdio.h>

size_t fwrite (const void *ptr, size_t size, size_t nmemb, FILE *stream);

描述

函数 fwrite 从由 ptr 所指向的数组中写直至 nmemb 个元素到由 stream 所指向的流中，元素的尺寸由 size 规定。当该流的文卷位置指示符有定义时该文卷位置指示符推进成功地写出的字符数。若出现错误，则该流的文卷位置指示符的结果值是不确定的。

返回值

函数 fwrite 返回成功地写出的元素数，仅当出现写错误时，则该数才可能小于 nmemb。

7.9.9 文卷定位函数

7.9.9.1 函数 fgetpos

调用序列规格说明

#include <stdio.h>

int fgetpos(FILE *stream, fpos_t *pos);

描述

函数 fgetpos 将由 stream 所指向的流的文卷位置指示符的当前值存入由 pos 所指向的对象中。所存储的值中包含未说明的信息，该信息可由函数 fsetpos 用来将该流重新定位在调用函数 fgetpos 时所处的位置。

返回值

若成功，函数 fgetpos 返回零；失败时函数 fgetpos 返回非零值，并在 errno 中存入一个实现定义的正值。

提前引用的条文：函数 fsetpos(7.9.9.3条)。

7.9.9.2 函数 fseek

调用序列规格说明

#include <stdio.h>

int fseek(FILE *stream, long int offset, int whence);

描述

函数 fseek 设置由 stream 所指向的流的文卷位置指示符。

对于二进制流，用从文卷始端起的字符数来度量的新位置，可在由 whence 所指定的位置加上 off-

set 得到。所指定的位置，在 whence 是 SEEK_SET 时是文卷的始端；在 whence 是 SEEK_CUR 时是文卷位置指示符的当前值；在 whence 是 SEEK_END 时是文卷的末尾。对于二进制流，不一定需要有实际意义地支持 whence 值是 SEEK_END 的 fseek 函数调用。。

对于文本流，offset 应或者是零，或者是对同一个流早些时候调用函数 ftell 的返回值，且 whence 应是 SEEK_SET。

成功地调用函数 fseek 将清除该流的文卷尾指示符，并消除函数 ungetc 对同一流的任何影响。在 fseek 函数调用后，在更新流上的下一个操作应或者是输入或者是输出。

返回值

仅当请求不能被满足时函数 fseek 才返回非零值。

提前引用的条文：函数 ftell(7.9.9.4条)。

7.9.9.3 函数 fsetpos

调用序列规格说明

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

描述

函数 fsetpos 将由 stream 所指向的流的文卷位置指示符设置为由 pos 所指向的对象的值，该值是对同一个流早些时候调用函数 fgetpos 时所获得的。

成功地调用函数 fsetpos 将清除该流的文卷尾指示符，并消除函数 ungetc 对同一流的任何影响。在 fsetpos 函数调用后，在更新流上的下一个操作应或者是输入或者是输出。

返回值

若成功，函数 fsetpos 返回零；失败时函数 fsetpos 返回非零值，并在 errno 中存入一个实现定义的正值。

7.9.9.4 函数 ftell

调用序列规格说明

```
#include <stdio.h>
long int ftell(FILE *stream);
```

描述

函数 ftell 获得由 stream 所指向的流的文卷位置指示符的当前值。对于二进制流，该值是从文卷始端起的字符数。对于文本流，其文卷位置指示符包含未说明的信息，该信息可由函数 fseek 用来返回该流的文卷位置指示符，使之恢复到调用函数 ftell 时的位置；在这两种返回值之间的差别不一定反映实际写或读的字符数。

返回值

若成功，函数 ftell 返回该流的文卷位置指示符的当前值。失败时函数 ftell 返回 −1L，并在 errno 中存入一个实现定义的正值。

7.9.9.5 函数 rewind

调用序列规格说明

```
#include <stdio.h>
void rewind (FILE *stream);
```

描述

函数 rewind 将由 stream 所指向的流的文卷位置指示符置为文卷的始端。除同时还清除该流的出错指示符外，它等价于：

```
(void)fseek(stream,OL,SEEK_SET);
```

返回值

函数 rewind 不返回值。

7.9.10 出错处理函数

7.9.10.1 函数 clearerr

调用序列规格说明

```
#include <stdio.h>
void clearerr(FILE *stream);
```

描述

函数 clearerr 清除由 stream 所指向的流的文卷尾指示符和出错指示符。

返回值

函数 clearerr 不返回值。

7.9.10.2 函数 feof

调用序列规格说明

```
#include <stdio.h>
int feof(FILE *stream);
```

描述

函数 feof 测试由 stream 所指向的流的文卷尾指示符。

返回值

当且仅当由 stream 所指向的流的文卷尾指示符置位时函数 feof 才返回非零值。

7.9.10.3 函数 ferror

调用序列规格说明

```
#include <stdio.h>
int ferror(FILE *stream);
```

描述

函数 ferror 测试由 stream 所指向的流的出错指示符。

返回值

当且仅当由 stream 所指向的流的出错指示符置位时函数 ferror 才返回非零值。

7.9.10.4 函数 perror

调用序列规格说明

```
#include <stdio.h>
void perror(const char *s);
```

描述

函数 perror 将整型表达式 errno 中的出错号映射到一条出错消息。它将一系列字符按如下格式写到标准出错流中:(假设 s 不是一个空指针,且由 s 所指向的字符不是空字符。)首先是由 s 所指向的串,后跟一个冒号(:)和一个空格;然后是恰当的出错信息串,后跟一个新行字符。出错信息串的内容与以 errno 为实参时调用函数 strerror 所返回的内容相同,是实现定义的。

返回值

函数 perror 不返回值。

提前引用的条文:函数 strerror(7.11.6.2条)。

7.10 通用实用程序库前导文卷<stdlib.h>

前导文卷<stdlib.h>中声明了四种类型和一些通用实用函数,并定义了几个宏。

注:见(7.13.7条)"库的发展趋向"。

所声明的类型是:

size_t 和 wchar_t(二者都已在7.1.6条中描述),

div_t

它是一种结构类型，是函数 div 所返回值的类型，以及

ldiv_t

它也是一种结构类型，是函数 ldiv 所返回值的类型。

所定义的宏是：

NULL(已在7.1.6条中描述)；

EXIT_FAILURE

和

EXIT_SUCCESS

这两个宏将展开为整型表达式，可被用作函数 exit 的实参，以分别向宿主环境返回终止状态是不成功或成功；

RAND_MAX

它展开为一个整型常量表达式，该表达式的值是函数 rand 返回值中的最大值；以及

MB_CUR_MAX

它展开为一个正的整型表达式，该表达式的值是由当前地域环境(类型类别 LC_TYPE)所规定的扩展字符集中一个多字节字符的最大字节数，且该值决不大于 MB_LEN_MAX。

7.10.1 串转换函数

在出错情况下，函数 atof、atoi 和 atol 不一定影响整型表达式 errno 的值。若结果值不能表示，则串转换函数行为是未定义的。

7.10.1.1 函数 atof

调用序列规格说明

```
#include <stdlib.h>
double atof(const char *nptr);
```

描述

函数 atof 将 nptr 所指向的串中的起始部分转换为 double 表示。除出错时的行为外，它等价于：

```
strtod(nptr,(char **)NULL);
```

返回值

函数 atof 返回转换后的值。

提前引用的条文：函数 strtod(7.10.1.4条)。

7.10.1.2 函数 atoi

调用序列规格说明

```
#include <stdlib.h>
int atoi(const char *nptr);
```

描述

函数 atoi 将 nptr 所指向的串中的起始部分转换为 int 表示。除出错时的行为外，它等价于：

```
(int)strtol(nptr,(char **)NULL,10);
```

返回值

函数 atoi 返回转换后的值。

提前引用的条文：函数 strtol(7.10.1.5条)。

7.10.1.3 函数 atol

调用序列规格说明

```
#include <stdlib.h>
long int atol(const char *nptr);
```

描述

函数 atol 将 nptr 所指向的串中的起始部分转换为 long int 表示。除出错时的行为外，它等价于：

```
strtol(nptr, (char * * )NULL,10);
```

返回值

函数 atol 返回转换后的值。

提前引用的条文：函数 strtol(7.10.1.5条)。

7.10.1.4 函数 strtod

调用序列规格说明

```
#include <stdlib.h>
double strtod(const char *nptr, char * *endptr);
```

描述

函数 strtod 将 nptr 所指向的串中的起始部分转换为 double 表示。首先，它把输入串分解为三部分：一个初始部分，是可能为空的一系列白空类字符(如由函数 isspace 所规定的)；一个类似浮点常量的主体序列；以及由一个或多个不可识别的字符，包括终止输入串的空字符的终串。然后，尝试将主体序列转换为浮点数，并返回转换后的结果。

所期望的主体序列的形式是：先是一个任选的正号或负号；然后是一个非空的数字序列，可任选地包含一个十进制小数点；最后面是任选的、如6.1.3.1条所定义的指数部分，但无浮点后缀。主体序列定义为输入串中以第一个非白空类字符开始，与期望形式相符的最长的初始子序列。若输入串为空，或所包含的全部是白空类字符，或者第一个非白空类字符不是数符、数字或十进制小数点时，则主体序列中不包含任何字符。

若主体序列具有所期望的形式，由第一个数字或十进制小数点开始(无论哪个在前)的字符序列，除该十进制小数点是用来代替一个圆点外，将按6.1.3.1条的规则解释为一个浮点数。若既不出现指数部分也不出现十进制小数点字符，则假设串中最后一个字符后跟有一个十进制小数点。若主体序列以一个负号开始，则将转换后所得的结果值变负。只要 endptr 不是一个空指针，在由 endptr 所指向的对象中将存入一个指向终串的指针。

在除"C"以外的地域环境中，可以接受附加的实现定义的主体形式。

若主体序列为空，或不具有所期望的形式，则不执行转换；只要 endptr 不是一个空指针，则将 nptr 的值存入由 endptr 所指向的对象中。

返回值

若有转换结果，则函数 strtod 返回转换后的值。若不能执行转换，则返回零。若正确值超出可表示值的范围，则(按照值的符号)返回正 HUGE_VAL 或负 HUGE_VAL，并在 errno 中存入宏 ERANGE 的值。若正确值会导致下溢，则返回零，并也在 errno 中存入宏 ERANGE 的值。

7.10.1.5 函数 strtol

调用序列规格说明

```
#include <stdlib.h>
long int strtol(const char *nptr, char * *endptr,int base);
```

描述

函数 strtol 将 nptr 所指向的串中的起始部分转换为 long int 表示。首先，它把输入串分解为三部分：一个初始部分，是可能为空的一系列白空类字符(如函数 isspace 所规定的)；一个类似于以由 base 值所确定的基数所表示的整数的主体序列；以及由一个或多个不可识别的字符，包括终止输入串的空字符的终串。然后，尝试将主体序列转换为整数，并返回转换后的结果。

若 base 值为零，则所期望的主体序列的形式是如6.1.3.2条所定义的整数常量的形式，前面可任选地有一个正号或负号，但不包含整数后缀。若 base 值在2至36之间，则所期望的主体序列的形式是表示

一个以由 base 所规定的值为基的整数的字母或数字字符序列，前面可任选地有一个正号或负号，但不包含整数后缀。字母 a(或 A)至 z(或 Z)对应于值10至35；仅允许对应值小于 base 值的字母。若 base 值为16，则字符0x 或0x 可任选地出现在字母和数字字符序列之前，(若有数符则)在数符之后。

主体序列定义为输入串中以第一个非白空类字符开始，与期望形式相符的最长的初始子序列。若输入串为空，或所包含的全部是白空类字符，或者第一个非白空类字符不是数符或所允许的字母或数字字符时，则主体序列中不包含任何字符。

若主体序列具有所期望的形式且 base 值为零，则由第一个数字字符开始的字符序列将按6.1.3.2条的规则解释为一个整数常量。若主体序列具有所期望的形式且 base 值是在2与36之间，则将 base 值用作转换的基数，对每个字母赋予如上所述的对应值。若主体序列以一个负号开始，则将转换后所得的结果值变负。只要 endptr 不是一个空指针，在由 endptr 所指向的对象中将存入一个指向终串的指针。

在除"C"以外的地域环境中，可以接受附加的实现定义的主体序列形式。

若主体序列为空，或不具有所期望的形式，则不执行转换；只要 endptr 不是一个空指针，则将 nptr 的值存入由 endptr 所指向的对象中。

返回值

若有转换结果，则函数 strtol 返回转换后的值。若不能执行转换，则返回零。若正确值超出可表示值的范围，则(按照值的符号)返回 LONG_MAX 或 LONG_MIN，并在 errno 中存入宏 ERANGE 的值。

7.10.1.6 函数 strtoul

调用序列规格说明

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

描述

函数 strtoul 将 nptr 所指向的串中的起始部分转换为 unsigned long int 表示。首先，它把输入串分解为三部分：一个初始部分，是可能为空的一系列白空类字符(如由函数 isspace 所规定的)；一个类似于以由 base 值所确定的基数所表示的无符号整数的主体序列；以及由一个或多个不可识别的字符，包括终止输入串的空字符的终串。然后尝试将主体序列转换为无符号整数，并返回转换后的结果。

若 base 值为零，则所期望的主体序列的形式是如6.1.3.2条所定义的整数常量的形式，前面可任选地有一个正号或负号，但不包含整数后缀。若 base 值在2至36之间，则所期望的主体序列的形式是表示一个以由 base 所规定的值为基的整数的字母或数字字符序列，前面可任选地有一个正号或负号，但不包含整数后缀。字母 a(或 A)至 z(或 Z)对应于值10至35；仅允许对应值小于 base 值的字母。若 base 值为16，则字符0x 或0X 可任选地出现在字母和数字字符序列之前，若有数符则在数符之后。

主体序列定义为输入串中以第一个非白空类字符开始，与期望形式相符的最长的初始子序列。若输入串为空，或所包含的全部是白空类字符，或者第一个非白空类字符不是数符或所允许的字母或数字字符时，则主体序列中不包含任何字符。

若主体序列具有所期望的形式且 base 值为零，则由第一个数字字符开始的字符序列将按6.1.3.2条的规则解释为一个整数常量。若主体序列具有所期望的形式且 base 值是在2与36之间，则将 base 值用作转换的基数，对每个字母赋予如上所述的对应值。若主体序列以一个负号开始，则将转换后所得的结果值变负。只要 endptr 不是一个空指针，在由 endptr 所指向的对象中将存入一个指向终串的指针。

在除"C"以外的地域环境中，可以接受附加的实现定义的主体序列形式。

若主体序列为空，或不具有所期望的形式，则不执行转换；只要 endptr 不是一个空指针，则将 nptr 的值存入由 endptr 所指向的对象中。

返回值

若有转换结果，则函数 strtoul 返回转换后的值。若不能执行转换，则返回零。若正确值超出可表值的范围，则返回 ULONG_MAX。并在 errno 中存入宏 ERANGE 的值。

7.10.2 伪随机序列生成函数

7.10.2.1 函数 rand

调用序列规格说明

```
#include <stdlib.h>
int rand(void);
```

描述

函数 rand 计算范围在0至 RAND_MAX 之间的伪随机整数序列。

实现应表现为如同无任何库函数调用函数 rand 一样。

返回值

函数 rand 返回一个伪随机整数。

对环境的限制

宏 RAND_MAX 的值应至少为32767。

7.10.2.2 函数 srand

调用序列规格说明

```
#include <stdlib.h>
void srand(unsigned int seed);
```

描述

函数 srand 用其实参作为一个由后继的 rand 函数调用所返回的新的伪随机数序列的种。若随后用相同的种值调用函数 srand，则该伪随机数序列将重复。若在未调用函数 srand 之前调用函数 rand，则将产生与先以种值为1调用函数 srand 时相同的伪随机数序列。

实现应表现为如同无任何库函数调用函数 srand 一样。

返回值

函数 srand 不返回值。

示例

下列函数定义了 rand 和 srand 的一种可移植的实现。

```
static unsigned long int next=1;
int rand(void)  /*假设 RAND_MAX 为 32767*/
{
    next=next *1103515245+12345;
    return (unsigned int)(next/65536)%32768;
}
void srand(unsigned int seed)
{
    next=seed;
}
```

7.10.3 存储管理函数

对相继调用函数 calloc、malloc 和 realloc 所分配的存储区的次序和相邻接性未作规定。分配成功时所返回的指针已作适当的对齐调整，以便可将它赋予一个指向任何对象类型的指针，然后用来访问在所分配的空间内的这类对象或这类对象的数组直至该空间显式释放或重新分配为止。这类分配中的每一个都产生指向一个对象的指针，该对象与任何其他对象都不相交。所返回的指针指向所分配空间的起点（最低字节地址）。若不能分配所要求的空间，则返回一个空指针。若请求的空间为零，则行为由实现定义；所返回的指针应或者是一个空指针，或者是一个唯一的指针。引用已释放的空间的指针值是不确定的。

7.10.3.1 函数 calloc

调用序列规格说明

#include <stdlib.h>

void *calloc(size_t nmemb,size_t size);

描述

函数 calloc 为一个有 nmemb 个对象的数组分配空间，每个对象的尺寸都是 size。所分配的空间中所有二进位均初始化为零。

注：注意这不一定需与浮点零或空指针常量的表示相同。

返回值

函数 calloc 或者返回一个空指针，或者返回一个指向所分配的空间的指针。

7.10.3.2 函数 free

调用序列规格说明

#include <stdlib.h>

void free(void *ptr);

描述

函数 free 导致释放 ptr 所指向的空间，即使得该空间可用于下一步的分配。若 ptr 是一个空指针，则不发生任何动作。否则，若实参与早些时候由函数 calloc、malloc 或 realloc 所返回的指针不匹配，或者该空间早已通过调用函数 free 或 realloc 释放，则函数 free 的行为是未定义的。

返回值

函数 free 不返回值。

7.10.3.3 函数 malloc

调用序列规格说明

#include <stdlib.h>

void *malloc(size_t size);

描述

函数 malloc 为一个对象分配空间，该对象的尺寸由 size 规定，且值是不确定的。

返回值

函数 malloc 或者返回一个空指针，或者返回一个指向所分配的空间的指针。

7.10.3.4 函数 realloc

调用序列规格说明

#include <stdlib.h>

void *realloc(void *ptr, size_t size);

描述

函数 realloc 将 ptr 所指向的对象的尺寸改变为 size 所规定的尺寸。该对象的内容在直至新老尺寸中的小者部分应不变。若新尺寸较大，则该对象中新分配部分的值是不确定的。若 ptr 是一个空指针，则函数 realloc 的行为类似对该指定尺寸的函数 malloc。否则，若 ptr 与早些时候由函数 calloc、malloc 或 realloc 所返回的指针不匹配，或者该空间早已通过调用函数 free 或 realloc 释放，则函数 realloc 的行为是未定义的。若不能分配空间，则由 ptr 所指向的对象不变。若 size 为零且 ptr 不是一个空指针，则释放 ptr 所指向的对象。

返回值

函数 realloc 或者返回一个空指针，或者返回一个可能已移动了的已分配的空间的指针。

7.10.4 与环境通信

7.10.4.1 函数 abort

调用序列规格说明

#include <stdlib.h>

void abort(void);

描述

除非是信号 SIGABRT 正被截获且信号处理程序不返回的情况，否则函数 abort 将导致程序的非正常终止。是否刷新打开着的输出流，或关闭打开着的流，或删除临时性文卷，都是实现定义的。将通过调用 raise(SIGABRT)向宿主环境返回一个*不成功终止* 状态，其形式由实现定义。

返回值

函数 abort 不可能返回到调用它的程序。

7.10.4.2 函数 atexit

调用序列规格说明

#include <stdlib.h>

int atexit(void (*func)(void));

描述

函数 atexit 登记 func 所指向的函数，以便在程序正常终止时无实参地调用该函数。

对实现的限制

实现应支持至少登记32个函数。

返回值

若登记成功，则函数 atexit 返回零，失败时返回非零值。

提前引用的条文：函数 exit(7.10.4.3条)。

7.10.4.3 函数 exit

调用序列规格说明

#include <stdlib.h>

void exit(int status);

描述

函数 exit 导致程序正常终止。若一个程序执行的 exit 函数调用多于一次，则函数 exit 的行为是未定义的。

首先，调用由函数 atexit 所登记的所有函数，调用的顺序与它们登记的顺序相反。

注：调用每个函数的次数与其登记的次数一样多。

其次，刷新所有尚有未写出的缓冲数据的打开着的输出流，关闭所有打开着的流，删除所有由函数 tmpfile 创建的(临时性)文卷。

最后，控制返回到宿主环境。若 status 的值为零或 EXIT_SUCCESS，则返回一个形式由实现定义的*成功终止* 状态。若 status 的值为 EXIT_FAILURE，则返回一个形式由实现定义的*不成功终止* 状态。否则，所返回的状态是实现定义的。

返回值

函数 exit 不可能返回到调用它的程序。

7.10.4.4 函数 getenv

调用序列规格说明

#include <stdlib.h>

char *getenv(const char *name);

描述

函数 getenv 在一个由宿主环境提供的*环境表* 中搜索一个与 name 所指向的串相匹配的串。环境名集合以及改变环境表的方法都是实现定义的。

实现应表现为如同无任何库函数调用函数 getenv 一样。

返回值

函数 getenv 返回一个指向与所匹配的表成员相关联的串的指针。所指向的串不应被程序修改，但可由后继的 getenv 函数调用复写。若未找到所指定的 name，则返回一个空指针。

7.10.4.5 函数 system

调用序列规格说明

```
#include <stdlib.h>
int system(const char *string);
```

描述

函数 system 将由 string 所指向的串传递给宿主环境，以便由*命令处理程序*按一个实现定义的方式执行。对 string 可使用一个空指针以询问命令处理程序是否存在。

返回值

若实参是一个空指针，仅当命令处理程序可用时函数 system 才返回非零值。若实参不是空指针，则函数 system 返回一个实现定义的值。

7.10.5 查找与排序实用程序

7.10.5.1 函数 bsearch

调用序列规格说明

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base, size_t nmemb,
        size_t size,int (*compar)(const void *,const void *));
```

描述

函数 bsearch 在一个有 nmemb 个对象的数组中查找与 key 所指向的对象匹配的一个元素，base 指向该数组的第一个元素。数组中每个元素的尺寸由 size 规定。

使用两个实参调用由 compar 所指向的比较函数。第一个实参指向 key 对象，第二个实参指向数组中的一个元素。根据 key 对象小于、匹配或大于该数组元素，该比较函数分别返回小于零、等于零或大于零的整数。数组的组成应为：所有比较时小于 key 对象的元素、所有比较时等于 key 对象的元素以及所有比较时大于 key 对象的元素，并以该次序排列。

注：实际上，整个数组已按比较函数排序。

返回值

函数 bsearch 返回一个指向该数组中匹配元素的指针，或当找不到匹配的元素时返回一个空指针。若有两个元素比较结果相等，对选哪一个作为所匹配的元素未作规定。

7.10.5.2 函数 qsort

调用序列规格说明

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb,size_t size,
        int (*compar)(const void *,const void *));
```

描述

函数 qsort 对一个有 nmemb 个对象的数组进行排序，base 指向该数组的第一个元素。每个对象的尺寸由 size 规定。

数组的内容按由 compar 所指向的比较函数以降序排列。调用比较函数时的两个实参指向要作比较的两个对象。按照第一个实参小于、匹配或大于第二个实参，该比较函数分别返回小于零、等于零或大于零的整数。

若有两个元素比较结果相等，对它们在排序后的数组中的次序未作规定。

返回值

函数 qsort 不返回值。

7.10.6 整型算术函数

7.10.6.1 函数 abs

调用序列规格说明

```
#include <stdlib.h>
int abs(int j);
```

描述

函数 abs 计算一个整数 j 的绝对值。若结果不能被表示，则函数 abs 的行为是未定义的。

注：在补码表示中，最大负数的绝对值是不能表示的。

返回值

函数 abs 返回该绝对值。

7.10.6.2 函数 div

调用序列规格说明

```
#include <stdlib.h>
div_t div(int numer,int denom);
```

描述

函数 div 计算被除数 numer 除以除数 denom 所得的商和余数。若不能整除，取结果的商的绝对值等于最接近于代数商绝对值的两个整数中较小的那一个。若结果不能被表示，则函数 div 的行为是未定义的；否则，quot * denom+rem 应等于 numer。

返回值

函数 div 返回一个由商和余数组成的类型为 div_t 的结构。该结构应以任意次序包含下列成员：

```
int quot;  /*商*/
int rem;   /*余数*/
```

7.10.6.3 函数 labs

调用序列规格说明

```
#include <stdlib.h>
long int labs(long int j);
```

描述

除实参及结果的类型都是 long int 外，函数 labs 类似于函数 abs。

7.10.6.4 函数 ldiv

调用序列规格说明

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

描述

除实参及返回的结构(其类型为 ldiv_t)成员的类型都是 long int 外，函数 ldiv 类似于函数 div。

7.10.7 多字节字符函数

多字节字符函数的行为受当前地域环境的 LC_CTYPE 类别的影响。对于依赖于状态的编码，由一个字符指针实参 s 为空指针的调用将每个函数置于初始转义状态。后继的以 s 实参为非空指针的调用必要时将使得函数的内部状态发生改变。以 s 实参为空指针的调用使这些函数在编码依赖于状态时返回一个非零值，否则返回零。改变 LC_CTYPE 类别将导致这些函数的转义状态变得不可确定。

注：若实现使用了特殊字节改变转义状态，则这些字节不产生分离的宽字符代码，而是与一个邻接的多字节字符组合在一起。

7.10.7.1 函数 mblen

调用序列规格说明

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

描述

若 s 不是空指针，函数 mblen 确定 s 所指向的多字节字符中所包含的字节数。除函数 mbtowc 的转义状态不受影响外，它等价于：

```
mbtowc((wchar_t *)0,s,n);
```

实现应表现为如同无任何库函数调用函数 mblen 一样。

返回值

若 s 是空指针，根据多字节字符的编码依赖或不依赖于状态，函数 mblen 分别返回一个非零值或零值。若 s 不是空指针，函数 mblen 或者当 s 指向空字符时返回零，或者若下 n 个或少于 n 个字符构成一个有效的多字节字符时返回该多字节字符中所包含的字节数，或者当它们不能构成一个有效的多字节字符时返回－1。

提前引用的条文：函数 mbtowc(7.10.7.2条)。

7.10.7.2 函数 mbtowc

调用序列规格说明

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s,size_t n);
```

描述

若 s 不是空指针，则函数 mbtowc 先确定构成 s 所指向的多字节字符的字节数，然后确定对应于该多字节字符的 wchar_t 类型值的代码(对应于空字符的代码值是0)。若该多字节字符有效，且 pwc 不是空指针，则函数 mbtowc 将该代码存入由 pwc 所指向的对象中。最多可检验由 s 所指向的数组中的 n 个字节。

实现应表现为如同无任何库函数调用函数 mbtowc 一样。

返回值

若 s 是空指针，根据多字节字符的编码依赖或不依赖于状态，函数 mbtowc 分别返回一个非零值或零值。若 s 不是空指针，函数 mbtowc 或者当 s 指向空字符时返回零，或者若下 n 个或少于 n 个字符构成一个有效的多字节字符时返回构成该多字节字符的字节数，或者当它们不能构成一个有效的多字节字符时返回－1。

在任何情况下，返回值都不应大于 n 或宏 MB_CUR_MAX 的值。

7.10.7.3 函数 wctomb

调用序列规格说明

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

描述

函数 wctomb 确定表示对应于值是 wchar 的代码的多字节字符所需的字节数(包括任何改变转义状态的字节在内)。若 s 不是空指针，它将该多字节字符的表示存入由 s 所指向的数组对象中，最多只能存 MB_CUR_MAX 个字符。若 wchar 的值是0，则函数 wctomb 留在初始转义状态。

实现应表现为如同无任何库函数调用函数 wctomb 一样。

返回值

若 s 是空指针，根据多字节字符的编码依赖或不依赖于状态，函数 wctomb 分别返回一个非零值或零值。若 s 不是空指针，则若 wchar 的值不对应于一个有效的多字节字符时，函数 wctomb 返回－1，否

则返回组成对应于 wchar 的值的多字节字符的字节数。

在任何情况下，返回值都不应大于宏 MB_CUR_MAX 的值。

7.10.8 多字节串函数

多字节串函数的行为受当前地域环境的 LC_CTYPE 类别的影响。

7.10.8.1 函数 mbstowcs

调用序列规格说明

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

描述

函数 mbstowcs 将 s 所指向的数组中，以初始转义状态开始的一系列多字节字符转换为一系列对应的代码，并将不超过 n 个代码存入由 pwcs 所指向的数组中。对跟在空字符（空字符将转换为一个值为0的代码）后面的多字节字符不作检验和转换。除函数 mbtowc 的转义状态不受影响外，对每个多字节字符如同调用函数 mbtowc 一样转换。

在由 pwcs 所指向的数组中修改的元素不超过 n 个。若出现两个重叠的对象间的复写，则其行为是未定义的。

返回值

若遇到一个无效的多字节字符，函数 mbstowcs 返回(size_t)－1，否则，函数 mbstowcs 返回修改过的数组元素数目，不包括可能出现的任何用以终止串的0代码。

7.10.8.2 函数 wcstombs

调用序列规格说明

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs,size_t n);
```

描述

函数 wcstombs 将 pwcs 所指向的数组中的一系列对应于多字节字符的代码转换为由初始转义状态开始的一系列多字节字符，并将它们逐个存入由 s 所指向的数组中，直到多字节字符将超出总共 n 个字节的限制或存入一个空字符时停止。除函数 wctomb 的转义状态不受影响外，对每个代码如调用 wctomb 函数一样转换。

在由 s 所指向的数组中修改的元素不超过 n 个。若出现两个重叠的对象间的复写，则函数 wcstombs 的行为是未定义的。

返回值

若遇到一个不对应于有效多字节字符的代码时，函数 wcstombs 返回(size_t)－1，否则函数 wcstombs 返回修改过的字节数，不包括可能出现的用以终止串的空字符。

注：若返回值是 n，则数组将不以空或0结束。

7.11 串处理程序库前导文卷<string.h>

7.11.1 串函数的约定

前导文卷<string.h>声明了一种类型与一些函数，并定义了一个宏；这些对操作字符型数组以及按字符型数组对待的其他对象是有用的。所声明的类型是 size_t，所定义的宏是 NULL（二者都已在7.1.6条中描述）。可使用各种方法来确定数组的长度，但在所有情况下，char * 或 void * 实参都指向数组中第一个（最低地址的）元素。若访问数组时超出一个对象的末尾，则函数的行为是未定义的。

注：见(7.13.8条)“库的发展趋向”。

7.11.2 复写类函数

7.11.2.1 函数 memcpy

调用序列规格说明

```
#include <string.h>
void *memcpy(void *s1,const void *s2,size_t n);
```

描述

函数 memcpy 从 s2所指向的对象中复写 n 个字符到由 s1所指向的对象中。若在两个重叠的对象之间发生复写，则函数 memcpy 的行为是未定义的。

返回值

函数 memcpy 返回 s1的值。

7.11.2.2 函数 memmove

调用序列规格说明

```
#include <string.h>
void *memmove(void *s1,const void *s2,size_t n);
```

描述

函数 memmove 从 s2所指向的对象中复写 n 个字符到 s1所指向的对象中。复写的过程如同先从 s2所指向的对象中复写 n 个字符到一个 n 个字符的临时性数组中，该数组与 s1和 s2所指向的对象都不重叠，然后再把该 n 个字符从临时性数组中复写到 s1所指向的对象中一样。

返回值

函数 memmove 返回 s1的值。

7.11.2.3 函数 strcpy

调用序列规格说明

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

描述

函数 strcpy 将 s2所指向的串(包括结尾的空字符)复写到 s1所指向的数组中。若在两个重叠的对象之间发生复写，则函数 strcpy 的行为是未定义的。

返回值

函数 strcpy 返回 s1的值。

7.11.2.4 函数 strncpy

调用序列规格说明

```
#include <string.h>
char *strncpy(char *s1, const char *s2,size_t n);
```

描述

函数 strncpy 从 s2所指向的数组中复写不超过 n 个字符(跟在空字符后的字符不复写)到 s1所指向的数组中。若在两个重叠的对象之间发生复写，则函数 strncpy 的行为是未定义的。

注：这样，若在 s2所指向的数组中的前 n 个字符中无空字符，则结果将不以空字符结尾。

若 s2所指向的数组是一个少于 n 个字符的串，则在 s1所指向的数组的副本中将添加空字符，直至总共写完 n 个字符为止。

返回值

函数 strncpy 返回 s1的值。

7.11.3 串接函数

7.11.3.1 函数 strcat

调用序列规格说明

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

描述

函数 strcat 在 s1所指向的串的末尾添加一个 s2所指向的串(包括结尾的空字符在内)的副本,s2所指向的串中第一个字符覆盖 s1所指向的串末尾的空字符。若在两个重叠的对象之间发生复写,则函数 strcat 的行为是未定义的。

返回值

函数 strcat 返回 s1的值。

7.11.3.2 函数 strncat

调用序列规格说明

```
#include <string.h>
char *strncat(char *s1,const char *s2,size_t n);
```

描述

函数 strncat 将 s2所指向的数组中不超过 n 个字符(空字符及其后的字符不添加)添加到 s1所指向的串的末尾。s2所指向的串中第一个字符复盖 s1所指向的串末尾的空字符。在结果中总是添加一个空字符。若在两个重叠的对象之间发生复写,则函数 strncat 的行为是未定义的。

注:这样,在 s1所指向的数组中的最大字符数是 strlen(s1)+n+1。

返回值

函数 strncat 返回 s1的值。

提前引用的条文:函数 strlen(7.11.6.3条)。

7.11.4 比较函数

由比较函数 memcmp、strcmp 和 strncmp 返回的非零值的符号,是根据在所比较的对象中的第一对不同的字符值(二者都解释为 unsigned char)的差的符号确定的。

7.11.4.1 函数 memcmp

调用序列规格说明

```
#include <string.h>
int memcmp(const void *s1,const void *s2,size_t n);
```

描述

函数 memcmp 比较 s1所指向的对象和 s2所指向的对象中的前 n 个字符。

注:在结构对象中因对齐目的而充填的"空穴"内容是不确定的。比所分配的空间短的串,以及联合,也都会在比较时带来问题。

返回值

按照由 s1所指向的对象是大于、等于或小于由 s2所指向的对象,函数 memcmp 分别返回一个大于、等于或小于零的整数。

7.11.4.2 函数 strcmp

调用序列规格说明

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

描述

函数 strcmp 将 s1所指向的串与 s2所指向的串作比较。

返回值

按照 s1所指向的串是大于、等于或小于 s2所指向的串,函数 strcmp 分别返回一个大于、等于或小于零的整数。

7.11.4.3 函数 strcoll

调用序列规格说明

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

描述

函数 strcoll 将 s1所指向的串与 s2所指向的串作比较。二者都解释为适合于当前地域环境的 LC_COLLATE 类别。

返回值

按照当二者都解释为适合于当前地域环境的 LC_COLLATE 类别时，s1所指向的串是大于、等于或小于 s2所指向的串，函数 strcoll 分别返回一个大于、等于或小于零的整数。

7.11.4.4 函数 strncmp

调用序列规格说明

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

描述

函数 strncmp 将由 s1所指向的数组中不超过 n 个字符(跟在空字符后的字符不比较)与由 s2所指向的数组作比较。

返回值

按照由 s1所指向的、可能以空字符结尾的数组是大于、等于或小于由 s2所指向的、可能以空字符结尾的数组，函数 strncmp 分别返回一个大于、等于或小于零的整数。

7.11.4.5 函数 strxfrm

调用序列规格说明

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

描述

函数 strxfrm 变换 s2所指向的串，并把结果所得的串置于 s1所指向的数组中。所作的变换产生的效果是：若对两个变换后的串应用函数 strcmp，函数 strcmp 分别返回一个大于、等于或小于零的整数，这与对两个原始串应用函数 strcoll 产生的结果对应，在由 s1所指向的结果数组中放置的字符数，包括结尾的空字符在内，不超过 n 个。若 n 为零，允许 s1为空指针。若在两个重叠的对象之间发生复写，则函数 strxfrm 的行为是未定义的。

返回值

函数 strxfrm 返回变换后的串(不包括结尾的空字符)的长度。若所返回的值是 n 或比 n 大，则 s1所指向的数组的内容是不确定的。

示例

下列表达式的值是放置对 s 所指向的串进行变换后所得串的数组的尺寸：

```
1+strxfrm(NULL,s,0);
```

7.11.5 查找函数

7.11.5.1 函数 memchr

调用序列规格说明

```
#include <string.h>
void *memchr(const void *s,int c, size_t n);
```

描述

函数 memchr 确定在 s 所指向的对象的前 n 个字符(每个都解释为 unsigned char)中第一次出现 c(转换为 unsigned char 类型)的位置。

返回值

函数 memchr 返回一个指向所定位的字符的指针，或当该字符不出现在该对象中时返回一个空指针。

7.11.5.2 函数 strchr

调用序列规格说明

```
#include <string.h>
char *strchr(const char *s, int c);
```

描述

函数 strchr 确定在 s 所指向的串中第一次出现 c(转换为 char 类型)的位置。结尾的空字符被认为是串的一部分。

返回值

函数 strchr 返回一个指向所定位的字符的指针，或当该字符不出现在该串中时返回一个空指针。

7.11.5.3 函数 strcspn

调用序列规格说明

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

描述

函数 strcspn 计算在 s1所指向的串中最大的初始段长度，该初始段完全由不在 s2所指向的串中的字符组成。

返回值

函数 strcspn 返回该段的长度。

7.11.5.4 函数 strpbrk

调用序列规格说明

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

描述

函数 strpbrk 确定在 s1所指向的串中第一次出现 s2所指向的串中任何一个字符的位置。

返回值

函数 strpbrk 返回一个指向所定位的字符的指针，或当 s2中的字符都不出现在 s1中时返回一个空指针。

7.11.5.5 函数 strrchr

调用序列规格说明

```
#include <string.h>
char *strrchr(const char *s, int c);
```

描述

函数 strrchr 确定在 s 所指向的串中最后一次出现 c(转换为 char 类型)的位置。结尾的空字符被认为是该串的一部分。

返回值

函数 strrchr 返回一个指向所定位的字符的指针，或当 c 不出现在该串中时返回一个空指针。

7.11.5.6 函数 strspn

调用序列规格说明

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

描述

函数 strspn 计算在 s1所指向的串中最大的初始段长度，该初始段完全由在 s2所指向的串中的字符组成。

返回值

函数 strspn 返回该初始段的长度。

7.11.5.7 函数 strstr

调用序列规格说明

```
#include <string.h>
char *strstr(const char *s1,const char *s2);
```

描述

函数 strstr 确定在 s1所指向的串中第一次出现 s2所指向的串中的字符序列(不包括结尾的空字符)的位置。

返回值

函数 strstr 返回一个指向所定位的串的指针，或当在 s1中未发现该串时返回一个空指针。若 s2指向一个长度为零的串，则函数 strstr 返回 s1。

7.11.5.8 函数 strtok

调用序列规格说明

```
#include <string.h>
char *strtok(char *s1,const char *s2);
```

描述

一系列的 strtok 函数调用将 s1所指向的串分解为一系列的单词，每个单词都用 s2所指向的串中的一个字符定界。在调用序列中，第一次调用应使用 s1作为第一实参，而随后的调用以一个空指针作为第一实参。s2所指向的分隔符串每次调用时可以不同。

在调用序列中的第一次调用在 s1所指向的串中寻找第一个不包含在 s2所指向的分隔符串中的字符。若找不到这样的字符，则在 s1所指向的串中无单词，且函数 strtok 返回一个空指针。若找到这样的字符，则它是第一个单词的开始。

然后函数 strtok 从该位置开始寻找一个包含在当前分隔符串中的字符。若找不到这样的字符，则单词扩展到 s1所指向的串的末尾，并且随后再查找一个单词时将返回一个空指针。若找到这样的字符，则用一个空字符覆盖它，该空字符终止当前的单词。函数 strtok 保存一个指向下一个字符的指针，以便下一次查找单词可从该位置开始。

每个后继的调用，以一个空指针作为第一实参值，将从所保存的指针所指位置开始查找，其行为如上所述。

实现应表现为如同无任何库函数调用函数 strtok 一样。

返回值

函数 strtok 返回指向一个单词的第一个字符的指针，或当单词不存在时返回一个空指针。

示例

```
#include <string.h>
static char str[ ]="?a???b,,,#c";
char *t;

t=strtok(str,"?");          /* t指向单词"a" */
t=strtok(NULL,",");         /* t指向单词"??b" */
t=strtok(NULL,"#,");        /* t指向单词"c" */
t=strtok(NULL,"?");         /* t是一个空指针 */
```

7.11.6 其他函数

7.11.6.1 函数 memset

调用序列规格说明

#include <string.h>

void *memset(void *s,int c, size_t n);

描述

函数 memset 将 s 所指向的对象中前 n 个字符都写为 c 的值(转换为 unsigned char 类型)。

返回值

函数 memset 返回 s 的值。

7.11.6.2 函数 strerror

调用序列规格说明

#include <string.h>

char *strerror(int errnum);

描述

函数 strerror 将 errnum 中的出错号映射为一条出错消息。

实现应表现为如同无任何库函数调用函数 strerror 一样。

返回值

函数 strerror 返回指向一个串的指针,该串的内容是实现定义的。所指向的数组不应被程序修改,但可被后继的 strerror 函数调用覆盖。

7.11.6.3 函数 strlen

调用序列规格说明

#include <string.h>

size_t strlen(const char *s);

描述

函数 strlen 计算 s 所指向的串的长度。

返回值

函数 strlen 返回在终止该串的空字符之前的字符数。

7.12 日期与时间函数库前导文卷<time.h>

7.12.1 时间的分量

前导文卷<time.h>定义了两个宏,并声明了四种类型和一些函数,用于对时间进行操作。许多函数处理*日历时间*,它表示当前日期(按照格里历)和时间。某些函数处理*当地时间*,它是以某个特定时区表达的日历时间;以及*夏令时*,它在确定当地时间的算法上有一点改变。当地时区和夏令时是实现定义的。

所定义的宏是:

NULL(已在7.1.6条中描述);及

CLOCKS_PER_SEC

它是每一秒内包含的由函数 clock 所返回的处理器时间单位数。

所声明的类型是:

size_t(已在7.1.6条中描述);

clock_t 和

time_t

它们是能表示时间的算术类型;以及

struct tm

它容纳被称为*分段时间*的日历时间的分量。该结构应以任意次序至少包含下列成员。这些成员的语义以及它们正常的范围在注释中表达。

注：tm_sec 的范围为[0,61]允许多达两个闰秒。

```
int tm_sec;     /*一分钟内的秒数——[0,61]*/
int tm_min;     /*一小时内地分数——[0,59]*/
int tm_hour;    /*午夜后的小时数——[0,23]*/
int tm_mday;    /*一月内的日——[1,31]*/
int tm_mon;     /*一月份起计的月份——[0,11]*/
int tm_year;    /*1900年起的年份*/
int tm_wday;    /*星期日起计的日——[0,6]*/
int tm_yday;    /*一月一日起计的天数——[0,365]*/
int tm_isdst;   /*夏时制标志*/
```

若正在使用夏令时则 tm_isdst 为正，若不在使用夏令时则 tm_isdst 为零，若无有关夏令时的信息则 tm_isdst 为负。

7.12.2 时间操作函数

7.12.2.1 函数 clock

调用序列规格说明

```
#include <time.h>
clock_t clock(void);
```

描述

函数 clock 确定所使用的处理器时间。

返回值

函数 clock 返回实现所能给出的对程序所用的处理器时间的最佳近似值，该时间从只与程序调用相关的一个实现定义的纪元起点开始计算。要确定以秒为单位的时间，则函数 clock 的返回值应除以宏 CLOCKS_PER_SEC 的值。若所用的处理器时间不可用或其值不能被表示，则函数 clock 返回(clock_t)-1。

注：为度量程序所消耗的时间，应在程序开始时调用函数 clock，将该返回值从后继调用函数 clock 的返回值中减去。

7.12.2.2 函数 difftime

调用序列规格说明

```
#include <time.h>
double difftime(time_t time1,time_t time0);
```

描述

函数 difftime 计算两个日历时间的差：time1-time0。

返回值

函数 difftime 返回以秒为单位的时间差，类型为 double。

7.12.2.3 函数 mktime

调用序列规格说明

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

描述

函数 mktime 将按当地时间表达的、在 timeptr 所指向的结构中的分段表示的时间转换为日历时间值。该日历时间值与函数 time 的返回值的编码相同。结构中 tm_wday 和 tm_yday 分量的原始值将被忽

略，且其余分量的值并不限于在上文中所指定的范围内。成功地完成转换时，结构中 tm_wday 和 tm_yday 分量将被置为恰当的值，其余分量均被设置以表示所指定的日历时间，但它们的值将强制在上文所述的范围内；在确定 tm_mon 和 tm_year 之前，不置 tm_mday 的最终值。

注：因此，tm_isdst 的值为正、或零，将使得函数 mktime 最初分别假定所指定的时间是、或不是夏令时，tm_isdst 的值为负将使得函数 mktime 试图去确定所指定的时间是否是夏令时。

返回值

函数 mktime 返回按类型为 time_t 的值编码的所指定的日历时间。若不能表示该日历时间，则函数 mktime 返回(time_t)－1。

示例

计算2001年7月4日是星期几的程序如下：

```
#include <stdio.h>
#include <time.h>
static const char *const wday[ ]={
        "Sunday","Moday","Tuesday","Wednesday",
        "Thursday","Friday","Saturday","-unknown-"
};
struct tm time_str;
/*…*/

time_str.tm_year        =2001-1900;
time_str.tm_mon         =7-1;
time_str.tm_mday        =4;
time_str.tm_hour        =0;
time_str.tm_min         =0;
time_str.tm_sec         =1;
time_str.tm_isdst       =-1;
if(mktime(&time_str)    ==-1)
    time_str.tm_wday    =7;
printf("%s\n",wday[time_str.tm_wday]);
```

7.12.2.4 函数 time

调用序列规格说明

```
#include <time.h>
time_t time(time_t *timer);
```

描述

函数 time 确定当前的日历时间。值的编码是未规定的。

返回值

函数 time 返回实现所能给出的对当前日历时间的最佳近似值。若当前日历时间不可用，则返回值(time_t)－1。若 timer 不是一个空指针，则所返回的值也赋予 timer 所指向的对象。

7.12.3 时间转换函数

除函数 strftime 以外，这些函数的返回值在两个静态对象之一中：一个是分段表示的时间结构，另一个是 char 数组。这些函数中任何一个的执行可能覆盖任何其他函数返回在这两个对象中的信息。实现应表现为如同无任何库函数调用这些函数一样。

7.12.3.1 函数 asctime

调用序列规格说明

```
#include <time.h>
char *asctime(const struct tm *timeptr);
```

描述

函数 asctime 将 timeptr 所指向的结构中的分段表示的时间转换为如下形式的串：

```
Sun Sep 16 01:03:52 1973\n\0
```

转换时使用等价于下述的算法：

```
char *asctime(const struct tm *timeptr)
{
    static const char wday_name[7][3]={
        "Sun","Mon","Tue","Wed","Thu","Fri","Sat"
    };
    static const char mon_name[12][3]={
        "Jan","Feb","Mar","Apr","May","Jun",
        "Jul","Aug","Sep","Oct","Nov","Dec"
    };
    static char result[26];
    sprintf(result, "%.3s %.3s %3d %.2d:%.2d:%.2d %d\n",
            wday_name[timeptr->tm_wday],
            mon_name[timeptr->tm_mon],
            timeptr->tm_mday,timeptr->tm_hour,
            timeptr->tm_min,timeptr->tm_sec,
            1900+timeptr->tm_year);
    return result;
}
```

返回值

函数 asctime 返回一个指向该串的指针。

7.12.3.2 函数 ctime

调用序列规格说明

```
#include <time.h>
char *ctime(const time_t *timer);
```

描述

函数 ctime 将 timer 所指向的日历时间转换为串形式的当地时间。它等价于：

```
asctime(localtime(timer))
```

返回值

函数 ctime 返回由函数 asctime 以分段表示的时间为实参时所返回的指针。

提前引用的条文：函数 localtime(7.12.3.4条)。

7.12.3.3 函数 gmtime

调用序列规格说明

```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

描述

函数 gmtime 将由 timer 所指向的日历时间转换为以国际标准时(UTC)表达的分段时间。

返回值

函数 gmtime 返回一个指向该对象的指针，或当不能用 UTC 时返回一个空指针。

7.12.3.4 函数 localtime

调用序列规格说明

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

描述

函数 localtime 将由 timer 所指向的日历时间转换为分段表示的当地时间。

返回值

函数 localtime 返回一个指向该对象的指针。

7.12.3.5 函数 strftime

调用序列规格说明

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,const char *format,
                                        const struct tm *timeptr);
```

描述

函数 strftime 按由 format 所指向的串的控制将字符存入由 s 所指向的数组中。格式应是一个以初始转义状态开始和结束的多字节字符序列。format 串由零个或多个转换说明符及普通多字节字符组成。转换说明符由一个%字符后跟一个确定该转换说明符行为的字符组成。将所有普通多字节字符(包括结尾的空字符)不作改变复写到该数组中。若出现两个重叠的对象间的复写，则函数 strftime 的行为是未定义的。存入数组的字符数不能超过 maxsize。每个转换说明符将由在下表中描述的适当的字符替换。这些字符由当前地域环境的 LC_TIME 类别以及包含在由 timeptr 所指向的结构中的值所确定。

%a 用当地缩写的星期名替换。

%A 用当地完整的星期名替换。

%b 用当地缩写的月份名替换。

%B 用当地完整的月份名替换。

%c 用当地合适的日期和时间表示替换。

%d 用以十进制数表示的一月中的日期(01—31)替换。

%H 用以十进制数表示的小时(24小时制)(00—23)替换。

%I 用以十进制数表示的小时(12小时制)(01—12)替换。

%j 用以十进制数表示的一年中的日期(001—366)替换。

%m 用以十进制数表示的月份(01—12)替换。

%M 用以十进制数表示的分(00—59)替换。

%P 用当地等价于12小时制加上 AM/PM 的表记形式替换。

%S 用以十进制数表示的秒(00—61)替换。

%U 用以十进制数表示一年中的星期号(第一个星期日是第一个星期的第一天)(00—53)替换。

%w 用以十进制数表示的星期中的哪一天[0(星期日)—6]替换。

%W 用以十进制数表示一年中的星期号(第一个星期一是第一个星期的第一天)(00—53)替换。

%x 用当地合适的日期表示替换。

%X 用当地合适的时间表示替换。

%y 用以十进制数表示的不带世纪的年份(00—99)替换。

%Y　用以十进制数表示的带世纪的年份替换。

%Z　用时区名称或其缩写替换，或不能确定时区时替换为无任何字符。

%%　用%字符替换。

若转换说明符不是上述之一，则函数 strftime 的行为是未定义的。

返回值

若结果的字符数，包括结尾的空字符在内，不超过 maxsize，则函数 strftime 返回存入 s 所指向的数组中的字符数，不包括结尾的空字符。否则返回零，且数组的内容是不确定的。

7.13　库的发展趋向

是为了方便才将下列名字组合在个别的前导文卷中。无论程序并入哪个前导文卷，以下描述的所有外部名都是保留名。

7.13.1　出错处理程序库前导文卷<errno.h>

在前导文卷<errno.h>的声明中可以加入以 E 和一个数字字符，或 E 和一个大写字母开头，后跟数字字符、字母和下横线的任意组合的宏。

7.13.2　字符处理程序库前导文卷<ctype.h>

在前导文卷<ctype.h>的声明中可以加入以 is 或 to 以及一个小写字母开头，后跟数字字符、字母和下横线的任意组合的函数名。

7.13.3　本地化程序库前导文卷<locale.h>

在前导文卷<locale.h>的定义中可以加入以 LC_和一个大写字母开头，后跟数字字符、字母和下横线的任意组合的宏。

7.13.4　数学程序库前导文卷<math.h>

所有已在前导文卷<math.h>中声明的函数名，加上后缀 f 或 l，分别保留为以 float 和 long double 作为实参和返回值的相应函数用。

7.13.5　信号处理程序库前导文卷<signal.h>

在前导文卷<signal.h>的定义中可以加入以 SIG 和一个大写字母，或 SIG 和一个大写字母开头，后跟数字字符、字母和下横线的任意组合的宏。

7.13.6　输入输出程序库前导文卷<stdio.h>

在函数 fprintf 和 fscanf 的转换说明符中可以加入小写字母。在扩展中可用其他字符。

7.13.7　通用实用程序库前导文卷<stdlib.h>

在前导文卷<stdlib.h>的声明中可以加入以 str 和一个小写字母开头，后跟数字字符、字母和下横线的任意组合的函数名。

7.13.8　串处理程序库前导文卷<string.h>

在前导文卷<string.h>的声明中可以加入以 str、mem 或 wcs 和一个小写字母开头，后跟数字字符、字母和下横线的任意组合的函数名。

附 录 A[1]
语言语法汇总
（参考件）

注：记法在标准文本的引言至第6章(语言)中描述。

A1 词法部分文法

A1.1 单词

(6.1条)*单词：*

关键字
标识符
常量
串字面值
算符
标点符号

(6.1条)*预处理单词：*

前导文卷名
标识符
预处理数
字符常量
串字面值
算符
标点符号
不在上述范围内的任一非白空类字符

A1.2 关键字

(6.1.1条)*关键字：*下列英文单词之一：

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

A1.3 标识符

(6.1.2条)*标识符：*

非数字字符
标识符 非数字字符

采用说明：

1] 本标准中删去了原ISO/IEC 9899:1990中的附录A“参考文献”，原附录B改为附录A，附录C改为附录B，以下类推。原索引改为附录G。

标识符 数字字符

(6.1.2条)*非数字字符*：下列字符之一：

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

(6.1.2条)*数字字符*：下列字符之一：

0 1 2 3 4 5 6 7 8 9

A1.4 常量

(6.1.3条)*常量*：

浮点常量
整数常量
枚举常量
字符常量

(6.1.3.1条) *浮点常量*：

小数部分常量 任选的指数部分 任选的浮点后缀字符
数字字符序列 指数部分 任选的浮点后缀字符

(6.1.3.1条) *小数部分常量*：

任选的数字字符序列 数字字符序列
数字字符序列

(6.1.3.1条) *指数部分*：

e 任选的符号字符 数字字符序列
E 任选的符号字符 数字字符序列

(6.1.3.1条) *符号字符*：下列字符之一：

\+ −

(6.1.3.1条) *数字字符序列*：

数字字符
数字字符序列 数字字符

(6.1.3.1条) *浮点后缀字符*：下列字符之一：

f l F L

(6.1.3.2条) *整数常量*：

十进制常量 任选的整数后缀字符
八进制常量 任选的整数后缀字符
十六进制常量 任选的整数后缀字符

(6.1.3.2条) *十进制常量*：

非零数字字符
十进制常量 数字字符

(6.1.3.2条) *八进制常量*：

字符0
八进制常量 八进制数字字符

(6.1.3.2条) *十六进制常量*：

字符0x 十六进制数字字符
字符0X 十六进制数字字符

十六进制常量　十六进制数字字符

（6.1.3.2条）*非零数字字符*：下列字符之一：

1　2　3　4　5　6　7　8　9

（6.1.3.2条）*八进制数字字符*：下列字符之一：

0　1　2　3　4　5　6　7

（6.1.3.2条）*十六进制数字字符*：下列字符之一：

0　1　2　3　4　5　6　7　8　9

a　b　c　d　e　f

A　B　C　D　E　F

（6.1.3.2条）*整数后缀字符*：

无符号后缀字符　任选的长整数后缀字符

长整数后缀字符　任选的无符号后缀字符

（6.1.3.2条）*无符号后缀字符*：下列字符之一：

u　U

（6.1.3.2条）*长整数后缀字符*：下列字符之一：

l　L

（6.1.3.3条）*枚举常量*：

标识符

（6.1.3.4条）*字符常量*：

'C-字符序列'

L*'C-字符序列'*

（6.1.3.4条）*C-字符序列*：

C-字符

C-字符序列　C-字符

（6.1.3.4条）*C-字符*：

源字符集中除单引号字符'、反斜线字符\、或新行字符外的任何字符

转义序列

（6.1.3.4条）*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

（6.1.3.4条）*简单转义序列*：下列字符序列之一：

\'　\"　\?　\\

\a　\b　\f　\n　\r　\t　\v

（6.1.3.4条）*八进制转义序列*：下列字符序列之一：

八进制数字字符

八进制数字字符　八进制数字字符

八进制数字字符　八进制数字字符　八进制数字字符

（6.1.3.4条）*十六进制转义序列*：下列字符序列之一：

\x *十六进制数字字符*

十六进制转义序列　十六进制数字字符

A1.5　串字面值

（6.1.4条）*串字面值*：

"任选的串字符序列"

L"任选的串字符序列"

(6.1.4条) *串字符序列:*

串字符

串字符序列 串字符

(6.1.4条) *串字符:*

源字符集中除双引号字符"、反斜线字符\、或新行字符外的任何字符

转义序列

A1.6 算符

(6.1.5条) *算符:* 下列字符(或字符序列)之一:

[ ] ( ) . ->

++ -- & * + - ~ ! sizeof

/ % << >> < > <= >= != ^ | && ||

? :

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

, # ##

A1.7 标点符号

(6.1.6条) *标点符号:* 下列字符之一:

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

A1.8 前导文卷名

(6.1.7条) *前导文卷名:*

<前导文卷字符序列>

"引号内字符序列"

(6.1.7条) *前导文卷字符序列:*

前导文卷字符

前导文卷字符序列 前导文卷字符

(6.1.7条) *前导文卷字符:*

源字符集中除新行字符和右尖括号字符>外的任何字符

(6.1.7条) *引号内字符序列:*

引号内字符

引号内字符序列 引号内字符

(6.1.7条) *引号内字符:*

源字符集中除新行字符和双引号字符"外的任何字符

A1.9 预处理数

(6.1.8条) *预处理数:*

数字字符

数字字符

预处理数 数字字符

预处理数 非数字字符

预处理数 e符号字符

预处理数 E符号字符

预处理数

A2 短语结构文法

A2.1 表达式

(6.3.1条) *初等表达式:*

标识符

常量

串字面值

(表达式)

(6.3.2条) *后缀表达式:*

初等表达式

后缀表达式 [表达式]

后缀表达式 (任选的实参表达式表)

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 --

(6.3.2条) *实参表达式表:*

赋值表达式

实参表达式表,赋值表达式

(6.3.3条) *一元表达式:*

后缀表达式

++一元表达式

--一元表达式

一元算符 强制(转换)表达式

sizeof*一元表达式*

sizeof*(类型名)*

(6.3.3条) *一元算符:* 下列字符之一:

& * + - !

(6.3.4条) *强制(转换)表达式:*

一元表达式

(类型名)强制(转换)表达式

(6.3.5条) *乘除类表达式:*

强制(转换)表达式

*乘除类表达式***强制(转换)表达式*

乘除类表达式/强制(转换)表达式

乘除类表达式%强制(转换)表达式

(6.3.6条) *加减类表达式:*

乘除类表达式

加减类表达式+乘除类表达式

加减类表达式—乘除类表达式

(6.3.7条) *移位类表达式:*

加减类表达式

移位类表达式≪加减类表达式

移位类表达式≫加减类表达式

(6.3.8条) *关系表达式：*

移位类表达式

关系表达式＜移位类表达式

关系表达式＞移位类表达式

关系表达式＜＝移位类表达式

关系表达式＞＝移位类表达式

(6.3.9条) *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

(6.3.10条)*按位与表达式：*

相等类表达式

按位与表达式 & 相等类表达式

(6.3.11条)*按位加表达式：*

按位与表达式

按位加表达式 ^ 按位与表达式

(6.3.12条)*按位或表达式：*

按位加表达式

按位或表达式/按位加表达式

(6.3.13条)*逻辑与表达式：*

按位或表达式

逻辑与表达式 && 按位或表达式

(6.3.14条)*逻辑或表达式：*

逻辑与表达式

逻辑或表达式//逻辑与表达式

(6.3.15条)*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

(6.3.16条)*赋值表达式：*

条件表达式

一元表达式 赋值算符 赋值表达式

(6.3.16条)*赋值算符*：下列字符序列之一：

＝ ＊＝ /＝ %＝ ＋＝ －＝ ≪＝ ≫＝ &＝ ^＝ ｜＝

(6.3.17条)*表达式：*

赋值表达式

表达式,赋值表达式

(6.3.4条) *常量表达式：*

条件表达式

A2.2 声明

(6.5条) *声明：*

声明区分符 任选的初值声明符表;

(6.5条) *声明区分符：*

存储类区分符　任选的声明区分符
类型区分符　任选的声明区分符
类型限定词　任选的声明区分符

(6.5条) *初值声明符表:*
初值声明符
初值声明符表,初值声明符

(6.5条) *初值声明符:*
声明符
声明符=初值符

(6.5.1条) *存储类区分符:*
typedef
extern
static
auto
register

(6.5.2条) *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
自定义类型名

(6.5.2.1条) *结构或联合区分符:*
标识结构或联合的关键字　任选的标识符{结构声明表}
标识结构或联合的关键字　标识符

(6.5.2.1条) *标识结构或联合的关键字:*
struct
union

(6.5.2.1条) *结构声明表:*
结构声明
结构声明表　结构声明

(6.5.2.1条) *结构声明:*
区分符或限定词表　结构声明符表;

(6.5.2.1条) *区分符或限定词表:*
类型区分符　任选的区分符或限定词表
类型限定词　任选的区分符或限定词表

(6.5.2.1条) *结构声明符表:*

结构声明符

结构声明符表,结构声明符

(6.5.2.1条) *结构声明符:*

声明符

任选的声明符:常量表达式

(6.5.2.2条) *枚举区分符:*

enum*任选的标识符* 〔*枚举符表*〕

enum*标识符*

(6.5.2.2条) *枚举符表:*

枚举符

枚举符表,枚举符

(6.5.2.2条) *枚举符:*

枚举常量

枚举常量=常量表达式

(6.5.3条) *类型限定词:*

const

volatile

(6.5.4条) *声明符:*

任选的指针 直接声明符

(6.5.4条) *直接声明符:*

标识符

(声明符)

直接声明符 〔任选的常量表达式〕

直接声明符 (形参类型表)

直接声明符 (任选的标识符表)

(6.5.4条) *指针:*

* *任选的类型限定词表*

* *任选的类型限定词表 指针*

(6.5.4条) *类型限定词表:*

类型限定词

类型限定词表 类型限定词

(6.5.4条) *形参类型表:*

形参表

形参表,…

(6.5.4条) *形参表:*

形参声明

形参表,形参声明

(6.5.4条) *形参声明:*

声明区分符 声明符

声明区分符 任选的抽象声明符

(6.5.4条) *标识符表:*

标识符

标识符表,标识符

(6.5.5条) *类型名:*

区分符或限定词表　任选的抽象声明符

(6.5.5条) *抽象声明符:*

指针

任选的指针　直接抽象声明符

(6.5.5条) *直接抽象声明符:*

(抽象声明符)

任选的直接抽象声明符　[任选的常量表达式]

任选的直接抽象声明符　(任选的形参类型表)

(6.5.6条) *自定义类型名:*

标识符

(6.5.7条) *初值符:*

赋值表达式

{初值符表}

{初值符表,}

(6.5.7条) *初值符表:*

初值符

初值符表,初值符

A2.3 语句

(6.6条) *语句:*

带标号语句

复合语句

表达式语句

选择语句

循环语句

跳转语句

(6.6.1条) *带标号语句:*

标识符:语句

case*常量表达式:语句*

default:*语句*

(6.6.2条) *复合语句:*

{任选的声明表　任选的语句表}

(6.6.2条) *声明表:*

声明

声明表　声明

(6.6.2条) *语句表:*

语句

语句表　语句

(6.6.3条) *表达式语句:*

任选的表达式:

(6.6.4条) *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*

(6.6.5条) *循环语句*:

while(*表达式*)*语句*

do*语句* while(*表达式*);

for(*任选的表达式*;*任选的表达式*;*任选的表达式*)*语句*

(6.6.6条) *跳转语句*:

goto*标识符*;

continue;

break;

return*任选的表达式*;

A2.4 外部定义

(6.7条) *翻译单位*:

外部声明

翻译单位 *外部声明*

(6.7条) *外部声明*:

函数定义

声明

(6.7.1条) *函数定义*:

任选的声明区分符表 *声明符* *任选的声明表* *复合语句*

A3 预处理指示

(6.8条) *预处理文卷*:

任选的程序组

(6.8条) *程序组*:

程序组成分

程序组 *程序组成分*

(6.8条) *程序组成分*:

任选的pp单词 *新行*

条件节

控制行

(6.8.1条) *条件节*:

条件组 *任选的多重嵌套条件组* *任选的否则组* *条件终行*

(6.8.1条) *条件组*:

if *常量表达式* *新行* *任选的程序组*

ifdef *标识符* *新行* *任选的程序组*

ifndef*标识符* *新行* *任选的程序组*

(6.8.1条) *多重嵌套条件组*:

嵌套条件组

多重嵌套条件组 *嵌套条件组*

(6.8.1条) *嵌套条件组*:

#elif *常量表达式* *新行* *任选的程序组*

(6.8.1条) *否则组*:

#else *新行* *任选的程序组*

(6.8.1条) *条件终行:*

#endif *新行*

控制行:

(6.8.2条) #include *pp 单词 新行*

(6.8.3条) #define *标识符 替换表 新行*

(6.8.3条) #define *标识符 左括号 任选的标识符表)替换表 新行*

(6.8.3条) #undef *标识符 新行*

(6.8.4条) #line *pp 单词 新行*

(6.8.5条) #error *任选的 pp 单词 新行*

(6.8.6条) #pragma *任选的 pp 单词 新行*

(6.8.7条) # *新行*

(6.8.3条) *左括号:*

不带前导白空类字符的左圆括号字符

(6.8.3条) *替换表:*

任选的 pp 单词

(6.8条) *pp 单词:*

预处理单词

pp 单词 预处理单词

(6.8条) *新行:*

新行字符

A4 Lexical gramma

A4.1 Tokens

(6.1) *token:*

keyword
identifier
const
string-literal
operator
punctuator

(6.1) *preprocessing-token:*

header-name
identifier
pp-number
character-constant
string-literal
operator
punctutuator
each none-white-space character that cannot be one of the above

A4.2 keywords

(6.1.1) *keyword:* one of

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

A4.3 Identifier

(6.1.2) *identifier:*

nodigit

identifier nodigit

identifier digit

(6.1.2) *nodigit:* one of

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

(6.1.2) *digit:* one of

0 1 2 3 4 5 6 7 8 9

A4.4 Constants

(6.1.3) *constant:*

floating-constant

integer-constant

enumeration-constant

character-constant

(6.1.3.1) *floating-constant:*

fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$

digit-sequence exponent-part floating-suffix$_{opt}$

(6.1.3.1) *fractional-constant:*

digit-sequence$_{opt}$. *digit-sequence*

digit-sequence.

(6.1.3.1) *exponent-part:*

e sign$_{opt}$ *digit-sequence*

E sign$_{opt}$ *digit-sequence*

(6.1.3.1) *sign:* one of

\+ —

(6.1.3.1) *digit-sequence:*

digit

digit-sequence digit

(6.1.3.1) *floating-suffix:* one of

f l F L

(6.1.3.2) *integer-constant:*

decimal-constant integer-suffix$_{opt}$

octal-constant integer-suffix$_{opt}$

hexadecimal-constant integer-suffix$_{opt}$

(6.1.3.2) *decimal-constant:*

nonzero-digit

decimal-constant digit

(6.1.3.2) *octal-constant:*

0

octal-constant octal-digit

(6.1.3.2) *hexadecimal-constant:*

0x hexadecimal-digit

0X hexadecimal-digit

hexadecimal-constant hexadecimal-digit

(6.1.3.2) *nonzero-digit:* one of

1 2 3 4 5 6 7 8 9

(6.1.3.2) *octal-digit:* one of

0 1 2 3 4 5 6 7

(6.1.3.2) *hexadecimal-digit:* one of

0 1 2 3 4 5 6 7 8 9

a b c d e f

A B C D E F

(6.1.3.2) *integer-suffix:*

unsigned-suffix long-suffix$_{opt}$

long-suffix unsigned-suffix$_{opt}$

(6.1.3.2) *unsigned-suffix:* one of

u U

(6.1.3.2) *long-suffix:* one of

l L

(6.1.3.3) *enumeration-constant:*

identifier

(6.1.3.4) *character-constant:*

'c-char-sequence'

L*'c-char-sequence'*

(6.1.3.4) *c-char-sequence:*

c-char

c-char-sequence c-

(6.1.3.4) *c-char:*

any member of the source character set except

the single-quote',backslash\,or new-line character

escape-sequence

(6.1.3.4) *escape-sequence:*

simple-escape-sequence

octal-escape-sequence

hexadecimal-escape-sequence

(6.1.3.4) *simple-escape-sequence:* one of

\' \" \? \\

\a \b \f \n \r \t \v

(6.1.3.4条) *octal-escape-sequence:* one of

|*octal-digit*

|*octal-digit octal-digit*

|*octal-digit octal-digit octal-digit*

(6.1.3.4条) *hexadecimal-escape-sequence:* one of

|*x hexadecimal-digit*

hexadecimal-escape-sequence hexadecimal-digit

A4.5 String literals

(6.1.4) *string-literal:*

"s-char-sequence$_{opt}$"

L*"s-char-sequence$_{opt}$"*

(6.1.4) *s-char-sequence:*

s-char

s-char-sequence s-char

(6.1.4) *s-char:*

any member of the source character set except
the double-quote",backslash\,or new-line character

escape-sequence

A4.6 Operators

(6.1.5) *operator:* one of

[ ] ( ) . ->

++ -- & * + - ~ ! sizeof

/ % << >> < > <= >= != ^ | && ||

? :

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

, # ##

A4.7 Punctuators

(6.1.6) *punctuator:* one of

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

A4.8 Header names

(6.1.7) *header-name:*

<h-char-sequence>

"q-char-sequence"

(6.1.7) h-char-sequence:

h-char

h-char-sequence h-char

(6.1.7) *h-char:*

any member of the source character set except the new-line character and>

(6.1.7) *q-char-sequence:*

q-char

q-char-sequence q-char

(6.1.7) *q-char:*

any member of the source character set except the new-line character and"

A4.9 Preprocessing numbers

(6.1.8) *pp-number:*

digit
. digit
pp-number digit
pp-number nondigit
pp-number e sign
pp-number E sign
pp-number .

A5 Phrase structure gramma

A5.1 Expressions

(6.3.1) *primary-expression:*

identifier
constant
string-literal
(expression)

(6.3.2) *postfix-expression:*

primary-expression
postfix-expression [expression]
postfix-expression (argumant-expression-list$_{opt}$)
postfix-expression . identifier
postfix-expression —> identifier
postfix-expression ++
postfix-expression --

(6.3.2) *argumant-expression-list:*

assignmant-expression
argumant-expression-list, assignmant-expression

(6.3.3) *unary-expression:*

unary-expression
++unary-expression
--unary-expression
unary-operator cast-expression
sizeof *unary-expression*
sizeof *(type-name)*

(6.3.3) *unary-operator:* one of

& * + − ~ !

(6.3.4) *cast-expression:*

unary-expression
(type-name) cast-expression

(6.3.5) *multiplicative-expression:*

cast-expression

multiplicative-expression * *cast-expression*

multiplicative-expression / *cast-expression*

multiplicative-expression % *cast-expression*

(6.3.6) *additive-expression*:

multiplicative-expression

additive-expression + *multiplicative-expression*

additive-expression − *multiplicative-expression*

(6.3.7) *shift-expression*:

additive-expression

shift-expression ≪ *additive-expression*

shift-expression ≫ *additive-expression*

(6.3.8) *relational-expression*:

shift-expression

relational-expression < *shift-expression*

relational-expression > *shift-expression*

relational-expression<=*shift-expression*

relational-expression>=*shift-expression*

(6.3.9) *equality-expression*:

relational-expression

equality-expression==*relational-expression*

equality-expression != *relational-expression*

(6.3.10) *AND-expression*:

equality--expression

AND-expression & *equality-expression*

(6.3.11) *exclusive-OR-expression*:

AND-expression

exclusive-OR-expression ^ *AND-expression*

(6.3.12) *inclusive-OR-expression*:

exclusive-OR-expression

inclusive-OR-expression | *exclusive-OR-expression*

(6.3.13) *logical-AND-expression*:

inclusive-OR-expression

logical-AND-expression && *inclusive-OR-expression*

(6.3.14) *logical-OR-expression*:

logical-AND-expression

logical-OR-expression || *logical-AND-expression*

(6.3.15) *conditional-expression*:

logical-OR-expression

logical-OR-expression ? *expression* : *conditional-expression*

(6.3.16) *assignment-expression*:

conditional-expression

unary-expression assignment-operator assignment-expression

(6.3.16) *assignment-operator*: one of

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

(6.3.17) *expression*:

assignment-expression

expression, *assignment-expression*

(6.3.4) *constant-expression*:

conditional-expression

A5.2 Declarations

(6.5) *declaration*:

declaration-specifiers init-declarator-list$_{opt}$;

(6.5) *declaration-specifiers*:

storage-class-specifier declaration-specifiers$_{opt}$

type-specifier declaration-specifiers$_{opt}$

type-qualifier declaration-specifiers$_{opt}$

(6.5) *init-declarator-list*:

init-declarator

init-declarator-list, *init-declarator*

(6.5) *init-declarator*:

declarator

declarator=*initializer*

(6.5.1) *storage-class-specifier*:

typedef

extern

static

auto

register

(6.5.2) *type-specifier*:

void

char

short

int

long

float

double

signed

unsigned

struct-or-union-specifier

enum-specifier

typedef-name

(6.5.2.1) *struct-or-union-specifier*:

struct-or-union identifier$_{opt}$*{struct-declaration-list}*

struct-or-union identifier

(6.5.2.1) *struct-or-union*:

struct

union

(6.5.2.1) *struct-declaration-list:*

struct-declaration

struct-declaration-list struct-declaration

(6.5.2.1) *struct-declaration:*

specifier-qualifier-list struct-declarator-list;

(6.5.2.1) *specifier-qualifier-list:*

type-specifier specifier-qualifier-list$_{opt}$

type-qualifier specifier-qualifier-list$_{opt}$

(6.5.2.1) *struct-declarator-list:*

struct-declarator

struct-declarator-list, struct-declarator

(6.5.2.1) *struct-declarator:*

declarator

declarator$_{opt}$ *: constant-expression*

(6.5.2.2) *rnum-specifier:*

enum *identifier*$_{opt}$ *{enumerator-list}*

enum *identifoer*

(6.5.2.2) *enumerator-list:*

enumerator

enumerator-list, enumerator

(6.5.2.2) *enumerator:*

enumeration-constant

enumeration-constant=constant-expression

(6.5.3) *type-qualifier:*

const

volatile

(6.5.4) *declarator:*

pointer$_{opt}$ *direct-declarator*

(6.5.4) *direct-declarator:*

identifier

(declarator)

direct-declarator [constant-expression$_{opt}$ *]*

direct-declarator (parameter-type-list)

direct-declarator (identifier-list$_{opt}$ *)*

(6.5.4) *pointer:*

** type-qualifier-list*$_{opt}$

** type-qualifier-list*$_{opt}$ *pointer*

(6.5.4) *type-qualifier-list:*

type-qualifier

type-qualifier-list type-qualifier

(6.5.4) *parameter-type-list:*

parameter-list

parameter-list, ···

(6.5.4) *parameter-list*:

parameter-declaration

parameter-list,

(6.5.4) *parameter-declaration*:

declaration-specifier declarator

declaration-specifier abstract-declarator$_{opt}$

(6.5.4) *identifier-list*:

identifier

identifier-list, *identifier*

(6.5.5) *type-name*:

specifier-qualifier-list abstract-declarator$_{opt}$

(6.5.5) *abstract-declarator*:

pointer

pinter$_{opt}$ *direct-abstract-declarator*

(6.5.5) *direct-abstract-declarator*:

(abstract-declarator)

direct-abstract-declarator$_{opt}$ *[constant-expression*$_{opt}$ *]*

direct-abstract-declarator$_{opt}$ *(parameter-type-list*$_{opt}$ *)*

(6.5.6) *typedef-name*:

identifier

(6.5.7) *initializer*:

assignment-expression

{initializer-list}

{initializer-list,}

(6.5.7) *initializer-list*:

initializer

initializer-list, *initializer*

A5.3 Statements

(6.6) *statement*:

labeled-statement

compound-statement

expression-statement

selection-statement

iteration-statement

jump-statement

(6.6.1) *labeled-statement*:

identifier : *statement*

case *constant-expression* : *statement*

default :*statement*

(6.6.2) *compound--statement*:

{declaration-list$_{opt}$*statement-list*$_{opt}$ *}*

(6.6.2) *declaration-list:*

declaration

declaration-list declaration

(6.6.2) *statement -list:*

statement

statement-list statement

(6.6.3) *expression-statement:*

expression$_{opt}$;

(6.6.4) *selection-statement:*

if *(expression) statement*

if *(expression) statement* else *statement*

switch *(expression) statement*

(6.6.5) *iteration-statement:*

while *(expression) statement*

do *statement* while *(expression)* ;

for *(expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$ *) statement*

(6.6.6) *jump-statement:*

goto *identifier;*

continue;

break;

return *expression*$_{opt}$;

A5.4 External definitions

(6.7) *translation-unit:*

external-declaration

translation-unit external-declaration

(6.7) *external-declaration:*

function-definition

declaration

(6.7.1) *function-definition:*

declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$ *compound ststement*

A6 Preprocessing directives

(6.8) *preprocessing-file:*

group$_{opt}$

(6.8) *group:*

group-part

group group-part

(6.8) *group-part:*

pp-tokens$_{opt}$ *new-line*

if-section

control-line

(6.8.1) *if-section:*

if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

(6.8.1) *if-group:*

#if *constant-expression new-line group*$_{opt}$

#ifdef *identifier new-line group*$_{opt}$

#ifndef *identifier new-line group*$_{opt}$

(6.8.1) *elif-groups:*

elif-group

elif-groups elif-group

(6.8.1) *elif-group:*

#elif *constant-expression new-line group*$_{opt}$

(6.8.1) *else-group:*

#else *new-line group*$_{opt}$

(6.8.1) *endif-line:*

#endif *new-line*

control-line:

(6.8.2) #include *pp-tokens new-line*

(6.8.3) #define *identifier replacement-list new-line*

(6.8.3) #define *identifier lparen identifierlist*$_{opt}$) *replacement-list new-line*

(6.8.3) #undef *identifier new-line*

(6.8.4) #line *pp-tokens new-line*

(6.8.5) #error *pp-tokens*$_{opt}$ *new-line*

(6.8.6) #pragma *pp-tokens*$_{opt}$ *new-line*

(6.8.7) # *new-line*

(6.8.3) *lparen:*

the left-parenthesis character without preceding white space

(6.8.3) *replacement-list:*

pp-tokens$_{opt}$

(6.8) *pp-token:*

preprocessing-token

pp-token preprocessing-token

(6.8) *new-line:*

the new-line character

附 录 B
序 点
(参考件)

下列是5.1.2.3条中所描述的序点(对表达式求值的顺序控制点):

——在调用函数时对实参求值以后(见6.3.2.2条)。

——下列算符的第一个操作数的末尾:逻辑与 &&(见6.3.13条);逻辑或||(见6.3.14条);条件?(见6.3.15条);逗号,(见6.3.17条)。

——下列完全表达式的末尾:初值符(见6.5.7条);表达式语句中的表达式(见6.6.3条);选择语句(if 或 switch 语句)中的控制表达式(见6.6.4条);while 或 do 语句中的控制表达式(见6.6.5条);for 语

句三个表达式中的每一个(见6.6.5.3条);return 语句中的表达式(见6.6.6.4条)。

附 录 C
库 汇 总
（参考件）

C1 出错处理程序库前导文卷<errno.h>

EDOM
ERANGE
errno

C2 公用定义库前导文卷<stddef.h>

NULL
offsetof(*类型,成员指示符*)
ptrdiff_t
size_t
wchar_t

C3 诊断程序库前导文卷<assert.h>

NDEBUG
void assert(int 表达式);

C4 字符处理程序库前导文卷<ctype.h>

```
int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int tolower(int c);
int toupper(int c);
```

C5 地域特性程序库前导文卷<locale.h>

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY

LC_NUMERIC
LC_TIME
NULL
struct lconv
char *setlocale(int category, const char *locale);
struct lconv *localeconv(void);

C6 数学程序库前导文卷<math.h>

HUGE_VAL
double acos(double x);
double asin(double x);
double atan(double x);
double atan2(double y,double x);
double cos(double x);
double sin(double x);
double tan(double x);
double cosh(double x);
double sinh(double x);
double tanh(double x);
double exp(double x);
double frexp(double value, int *exp);
double ldexp(double x,int exp);
double log(double x);
double log 10(double x);
double modf(double value, double *iptr);
double pow(double x,double y);
double sort(double x);
double ceil(double x);
double fabs(double x);
double floor(double x);
double fmod(double x,double y);

C7 非局部跳转库前导文卷<setjmp.h>

jmp_buf
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);

C8 信号处理程序库前导文卷<signal.h>

sig_atomic_t
SIG_DFL
SIG_ERR
SIG_IGN
SIGABRT

SIGFPE
SIGILL
SIGINT
SIGSEGV
SIGTERM
void(*signal (int sig, void(*' int)))(int);
int raise(int sig);

C9 变长实参库前导文卷<stdar

va_list
void va_start(va_list ap, *paramN*);
类型 va_arg(va_list ap, *类型*);
void va_end(va_list ap);

C10 输入输出程序库前导文卷<stdio.h>

_IOFBF
_IOLBF
_IONBF
BUFSIZ
EOF
FILE
FILENAME_MAX
FOPEN_MAX
fpos_t
L_tmpnam
NULL
SEEK_CUR
SEEK_END
SEEK_SET
size_t
stderr
stdin
stdout
TMP_MAX
int remove(const char *filename);
int rename(const char *old, const char *new);
FILE *tmpfile(void);
char *tmpnam(char *s);
int fclose(FILE *stream);
int fflush(FILE *stream);
FILE *fopen(const char *filename, const char *mode);
FILE *freopen(const char *filename, const char *mode, FILE *stream);

```
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
int fprintf(FILE *stream, const char *format,...);
int fscanf(FILE *stream, const char *format,...);
int printf(const char *format,...);
int scanf(const char *format,...);
int sprintf(char *s, const char *format,...);
int sscanf(const char *s, const char *format,...);
int vfprintf(FILE *stream, const char *format, va_list arg);
int vprintf(const char *format, va_list arg);
int vsprintf(char *s, const char *format, va_list arg);
int fgetc(FILE *stream);
char *fgets(char *s, int n,FILE *stream);
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
int getc(FILE *stream);
int getchar(void);
char *gets(char *s);
int putc(int c, FILE *stream);
int putchar(int c);
int puts(const char *s);
int ungetc(int c, FILE *stream);
size_t fread(void *ptr, size_t size,size_t nmemb,FILE *stream);
size_t fwrite(const void *ptr, size_t size,size_t nmemb, FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fseek(FILE *stream, long int offset, int whence);
int fsetpos(FILE *stream,const fpos_t *pos);
long int ftell(FILE *stream);
void rewind(FILE *stream);
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
void perror(const char *s);
```

C11 通用实用程序库前导文卷<stdlib.h>

```
EXIT_FAILURE
EXIT_SUCCESS
MB_CUR_MAX
NULL
RAND_MAX
div_t
ldiv_t
size_t
```

```
wchar_t
double atof(const char *nptr);
int atoi(const char *nptr);
long int atol(const char *nptr);
double strtod(const char *nptr, char **endptr);
long int strtol(const char *nptr, char **endptr, int base);
unsigned long int strtoul(const char *nptr, char **endptr, int base);
int rand(void);
void srand(unsigned int seed);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void abort(void);
int atexit(void(*func)(void));
void exit(int status);
char *getenv(const char *name);
int system(const char *string);
void *bsearch(const void *key, const void *base, size_t nmemb,
    size_t size, int (*compar)(const void *,const void *));
void qsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));
int abs(int j);
div_t div(int numer, int denom);
long int labs(long int j);
ldiv_t ldiv(long int numer, long int denom);
int mblen(const char *s, size_t n);
int mbtowc(wchar_t *pwc, const char *s, size_t n);
int wctomb(char *s, wchar_t wchar);
size_t mbstowcs(wchar_t *pwcs,const char *s, size_t n);
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

C12 串处理程序库前导文卷<string.h>

```
NULL
size_t
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1,const void *s2,size_t n);
char *strcpy(char *s1,const char *s2);
char *strncpy(char *s1,const char *s2, size_t n);
char *strcat(char *s1,const char *s2);
char *strncat(char *s1,const char *s2,size_t n);
int memcmp(const void *s1,const void *s2, size_t n);
int strcmp(const char *s1,const char *s2);
```

```
int strcoll(const char *s1,const char *s2);
int strncmp(const char *s1, const char *s2,size_t n);
size_t strxfrm(char *s1,const char *s2,size_t n);
void *memchr(const void *s, int c, size_t n);
char *strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
char *strpbrk(const char *s1,const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1,const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char *s1,const char *s2);
void *memset(void *s, int c ,size_t n);
char *strerror(int errnum);
size_t strlen(const char *s);
```

C13 日期与时间函数库前导文卷<time.h>

```
CLOCKS_PER_SEC
NULL
clock_t
time_t
size_t
struct tm
clock_t clock(void);
double difftime(time_t time1,time_t time0);
time_t mktime(struct tm *timeptr);
time_t time(time_t *timer);
char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timer);
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
size_t strftime(char *s, size_t maxsize ,const char *format,
                                const struct tm *timeptr);
```

附 录 D
实现规定的限定值
（参考件）

前导文卷<limits.h>的内容按字母顺序在下表中列出。表中所示的最小量值应以符号相同的实现定义的量值替换。所有的值均应是适合于在#if 预处理指示中使用的常量表达式。各分量在5.2.4.2.1条中作了进一步的描述。

#define CHAR_BIT	8
#define CHAR_MAX	UCHAR_MAX 或 SCHAR_MAX

#define CHAR_MIN	0或 SCHAR_MIN
#define INT_MAX	+32767
#define INT_MIN	−32767
#define LONG_MAX	+2147483647
#define LONG_MIN	−2147483647
#define MB_LEN_MAX	1
#define SCHAR_MAX	+127
#define SCHAR_MIN	−127
#define SHRT_MAX	+32767
#define SHRT_MIN	−32767
#define UCHAR_MAX	255
#define UINT_MAX	65535
#define ULONG_MAX	4294967295
#define USHRT_MAX	65535

前导文卷<float.h>的内容在下表中列出。FLT_RADIX 的值应是适合于在#if 预处理指示中使用的常量表达式。将不必是常量表达式的值提供给其他分量。各分量在5.2.4.2.2条中作了进一步的描述。

#define FLT_ROUNDS

下面所给出的各值应以符号相同、量值等于或大于表中所示绝对值的实现定义的表达式替换：

#define DBL_DIG	10
#define DBL_MANT_DIG	
#define DBL_MAX_10_EXP	+37
#define DBL_MAX_EXP	
#define DBL_MIN_10_EXP	−37
#define DBL_MIN_EXP	
#define FLT_DIG	6
#define FLT_MANT_DIG	
#define FLT_MAX_10_EXP	+37
#define FLT_MAX_EXP	
#define FLT_MIN_10_EXP	−37
#define FLT_MIN_EXP	
#define FLT_RADIX	2
#define LDBL_DIG	10
#define LDBL_MANT_DIG	
#define LDBL_MIN_10_EXP	+37
#define LDBL_MAX_EXP	
#define LDBL_MIN_10_EXP	−37
#define LDBL_MIN_EXP	

下面所给出的各值应以等于或大于表中所示值的实现定义的表达式替换：

#define DBL_MAX	1E+37
#define FLT_MAX	1E+37
#define LDBL_MAX	1E+37

下面所给出的各值应以等于或小于表中所示值的实现定义的表达式替换：

#define DBL_EPSILON	1E—9
#define DBL_MIN	1E—37
#define FLT_EPSILON	1E—5
#define FLT_MIN	1E—37
#define LDBL_EPSILON	1E—9
#define LDBL_MIN	1E—37

附 录 E
常见的告诫消息
（参考件）

实现应在许多情况下产生告诫消息，本标准文本中并未规定这些情况。下面列出一些最常见的情况：

——转入对一个属于自动存储类的对象进行初始化的程序块（见6.1.2.4条）。

——整型字符常量包含多于一个字符或宽字符常量包含多于一个多字节字符（见6.1.3.4条）。

——在注释中发现字符对/*（见6.1.7条）。

——遇到隐式的由存储区大的类型向存储区小的类型转换，如将 long int 或 double 赋予 int，或将指向 void 的指针转换为指向除字符类型外的其他类型等（见6.2条）。

——“无序的”二元算符（不是逗号、&&或||）中包含对某个操作数的左值的副作用，以及对另一个操作数的同样的左值的另一个副作用或访问另一操作数的同样左值的值（见6.3条）。

——调用了一个函数但未提供原型（见6.3.2.2条）。

——函数调用的实参的数量和类型与不是原型的函数定义中的形参的数量和类型不一致（见6.3.2.2条）。

——定义了一个对象但并未用到（见6.5条）。

——不是通过作为一个枚举类型成员的枚举常量或具有相同类型的枚举变量，或返回相同枚举类型的函数值赋值的方法对某个枚举类型的对象赋值（见6.5.2.2条）。

——聚集只有部分用方括号括起来的初始化说明（见6.5.7条）。

——不能到达某一语句（见6.6条）。

——遇到没有明显作用的语句（见6.6条）。

——将常量表达式用作选择语句的控制表达式（见6.6.4条）。

——函数中有带和不带表达式的 return 语句（见6.6.6.4条）。

——在跳过预处理组时遇到不正确组成的预处理组（见6.8.1条）。

——遇到不可识别的#pragma 指示（见6.8.6条）。

附 录 F
与可移植性有关的问题
（参考件）

本附录中收集了在本标准中出现过的有关可移植性的一些信息。

F1 未规定的行为

下列行为属于未规定的：

——静态初始化的方式与时间(见5.1.2条)。

——当活动位置已处于行的最后一个位置时写出一个可印刷字符的行为(见5.2.2条)。

——当活动位置已处于行的第一个位置时写出一个退格字符的行为(见5.2.2条)。

——当活动位置已处于或越过所定义的最后一个横向表格位置时写出一个横向制表字符的行为(见5.2.2条)。

——当活动位置已处于或越过所定义的最后一个纵向表格位置时写出一个纵向制表字符的行为(见5.2.2条)。

——浮点类型的表示(见6.1.2.5条)。

——对表达式求值的顺序——以任何符合优先级规则的次序,即使有括号出现时也是这样(见6.3条)。

——副作用出现的次序。(见6.3条)。

——函数调用中对函数指示符和实参求值的顺序(见6.3.2.2条)。

——为了存放一个位段而分配的可编址存储单位的对齐(见6.5.2.1条)。

——用于形参的存储区的布局(见6.7.1条)。

——在宏替换时#和##算符的求值顺序(见6.8.3.3条)。

——errno 是宏还是外部标识符(见7.1.4条)。

——setjmp 是宏还是外部标识符(见7.6.1.1条)。

——va_end 是宏还是外部标识符(见7.8.1.3条)。

——在对文本流成功地调用函数 ungetc 后,直到所有放回流中的字符都被读入或抛弃之前,文卷位置指示符的值(见7.9.7.11条)。

——由函数 fgetpos 成功地存储的值的细节(见7.9.9.1条)。

——对文本流成功地调用函数 ftell 所返回的值的细节(见7.9.9.4条)。

——由函数 calloc、malloc 和 realloc 所分配的存储区的次序及邻接性(见7.10.3条)。

——由函数 bsearch 比较相等的两元素中返回哪一个(见7.10.5.1条)。

——函数 qsort 对比较相等的两元素数组的排列次序(见7.10.5.2条)。

——由函数 time 返回的日历时间的编码(见7.12.2.3条)。

F2 未定义的行为

下列情况下的行为是未定义的:

——非空的源文卷不以新行字符结束,或以在反斜线字符后紧接一个新行字符结束,或在不完整的预处理单词或注释的中间结束(见5.1.1.2条)。

——在源文卷中遇到了除永不转换为单词的预处理单词、字符常量、串字面值、前导文卷名或注释外的不在所要求的字符集中的字符(见5.2.1条)。

——注释、串字面值、字符常量或前导文卷名中包含无效的多字节字符或不以初始转义状态开始和结束的多字节字符(见5.2.1.2条)。

——识别单词时在一个逻辑源行中遇到不匹配的'或"字符(见6.1条)。

——在同一函数中,多次使用同一标识符作标号(见6.1.2.1条)。

——使用了在当前作用域中不可见的标识符(见6.1.2.1条)。

——意图指示同一实体的多个标识符在最少要求的有效字符之后的字符有差别(见6.1.2条)。

——在同一翻译单位内,同一标识符既有内部链接又有外部链接(见6.1.2.2条)。

——使用了存放在引用自动存储期对象的指针中的值(见6.1.2.4条)。

——同一函数或对象的两个声明中规定了不相容的类型(见6.1.2.6条)。

——在字符常量或串字面值中遇到未规定的转义序列(见6.1.3.4条)。

——企图修改任何形式的串字面值(见6.1.4条)。

——字符串字面值单词与宽串字面值单词邻接(见6.1.4条)。

——在两种形式的前导文卷名预处理单词中,在定界符<和>之间或者定界符"之间遇到字符'、\、或/*(见6.1.7条)。

——算术转换产生了不能在所提供的空间中表示的结果(见6.2.1条)。

——在要求所表记的对象的值的上下文中使用了类型不完全的左值(见6.2.2.1条)。

——使用了 void 型表达式的值,或对 void 型表达式应用了(除 void 外的)隐式转换(见6.2.2.2条)。

——在两个序点之间多次修改某个对象,或对该对象进行除确定新值以外的修改和访问(见6.3条)。

——无效的算术运算(例如除以0或对0求模),或算术运算产生的结果不能在所提供的空间中表示的结果(例如上溢或下溢)(见6.3条)。

——一个对象的存储值被不是下列类型的左值访问:该对象的所声明的类型,该对象的所声明的类型的某种限定形式,对应于该对象所声明类型的有或无符号的类型,对应于该对象所声明类型的某种限定形式的有符号或无符号的类型,其成员中递归地包含的上述类型中的一种聚集或联合类型,或字符类型(见6.3条)。

——函数的实参是 void 型表达式(见6.3.2.2条)。

——对于无函数原型的函数调用,实参数目与形参数目不一致(见6.3.2.2条)。

——对于无函数原型的函数调用,若函数定义时无函数原型,且升格后的实参类型与升格后的行参类型不一致(见6.3.2.2条)。

——带函数原型调用某函数,而该函数不是定义为一个相容类型(见6.3.2.2条)。

——使用不以省略号结束的函数原型调用接受可变数目实参的函数(见6.3.2.2条)。

——在终止程序块中出现无效数组引用、空指针引用或对声明为自动存储期的对象的引用(见6.3.3.2条)。

——指向一个函数的指针被转换为指向不同类型的函数,且被用于调用其类型与原先类型不相容的函数(见6.3.4条)。

——指向一个函数的指针被转换为指向一个对象的指针,或者指向一个对象的指针被转换为指向一个函数的指针(见6.3.4条)。

——指针被转换为除整型或指针型外的其他类型(见6.3.4条)。

——对不是数组对象元素指针的指针进行加、减运算(见6.3.6条)。

——不是指向同一数组对象的指针相减(见6.3.6条)。

——表达式移位的位数是负数,或要求移的位数大于等于要进行移位的表达式本身的位宽(见6.3.7条)。

——对并不指向同一聚集或联合的指针使用关系算符进行比较(见6.3.8条)。

——将一对象赋于一个重叠的对象(见6.3.16.1条)。

——一个对象的标识符声明为无链接,且在声明符后该对象的类型不完全,或若它有初值符时,在初值声明符后该对象的类型不完全(见6.5条)。

——在块作用域中声明了一个函数,该函数的存储类区分符不是 extern(见6.5.1条)。

——结构或联合定义为只包含未命名的成员(见6.5.2.1条)。

——声明了类型不是 int、signed int 或 unsigned int 的位段(见6.5.2.1条)。

——企图通过非 const 限定类型的左值修改 const 限定类型的对象(见6.5.3条)。

——企图通过非 volatile 限定类型的左值引用 volatile 限定类型的对象(6.5.3条)。

——在对未初始化的自动存储期对象赋值以前使用该对象的值(见6.5.7条)。

——静态存储期的聚集或联合类型的对象具有不是用花括号括起来的初值符，或者自动存储期的聚集或联合类型对象具有类型与该对象类型不同的单表达式初值符，或不是用花括号括起来的初值符(见6.5.7条)。

——用到了函数的值，但先前并未返回任何值(见6.6.6.4条)。

——用到了有外部链接的标识符，但在程序中不确切存在一个对该标识符的外部定义(见6.7条)。

——定义接受可变数目实参的函数时没有以省略号结束的形参类型表(见6.7.1条)。

——以临时性定义声明一个表示具有内部链接和不完全类型的对象的标识符(见6.7.2条)。

——在展开预处理指示#if 或#elif 期间产生了单词 defined(见6.8.1条)。

——展开后的#include 预处理指示不与两种形式的前导文卷名中任何一种匹配(见6.8.2条)。

——不是以预处理单词组成的宏实参(见6.8.3条)。

——宏实参表中存在一系列在其他地方将作为预处理指示行的预处理单词(见6.8.3条)。

——预处理算符#的结果不是有效的字符串字面值(见6.8.3.2条)。

——预处理串接算符##的结果不是有效的预处理单词(见6.8.3.3条)。

——展开后的#line 预处理指示不与两种周密定义的形式中任何一种匹配(见6.8.4条)。

——下列标识符之一是预处理指示#define 或#undef 的主体：defined、_LINE_、FILE_、DATE_、TIME_或_STDC_(见6.8.8条)。

——企图使用除函数 memmove 外的其他库函数将一个对象复写到另一个重叠的对象中(见7条)。

——下列情况下的效果：外部定义中包含标准前导文卷；在首次引用了它所声明的任何函数或对象，或它所定义的任何类型或宏之后第一次包含标准前导文卷；或当以与某个关键字相同的名字定义一个宏时包含标准前导文卷(见7.1.2条)。

——若程序重定义一个保留的外部标识符所产生的效果(见7.1.3条)。

——抑制宏定义 errno 以便访问一个实际对象(见7.1.4条)。

——宏 offsetor 的形参*成员指示符*是*类型*形参的，算符的一个无效右操作数，或指示该结构的一个位段成员(见7.1.6条)。

——除非已显式规定其行为，库函数的实参具有无效值(见7.1.7条)。

——没有事先声明接受可变数目实参的库函数(见7.1.7条)。

——抑制宏定义 assert 以便访问一个实际函数(见7.2条)。

——字符处理函数的实参超出定义域(见7.3条)。

——抑制宏定义 setjmp 以便访问一个实际函数(见7.6条)。

——宏 setjmp 的调用不是出现在作为选择语句或循环语句的控制表达式的语境中，也不出现在作为选择语句或循环语句的控制表达式的整型常量表达式(该常量表达式可能由一元算符!隐含)的比较中，或者不作为一个表达式语句(可能强制转换为 void 型)出现(见7.6.1.1条)。

——已在 setjmp 调用和 longjmp 调用之间改变了不是 volatile 限定类型的自动存储类的对象，然后访问该对象的值(见7.6.2.1条)。

——在嵌套的信号例行程序中调用函数 longjmp(见7.6.2.1条)。

——出现了不是作为调用函数 abort 或 raise 的结果的信号，且信号处理程序调用标准库中除函数 signal 本身外的其他函数，或除通过对 volatile sig_atomic_t 型的静态存储期的变量赋值手段外引用一个静态存储期的对象时出现信号(时7.7.1.1.条)。

——在除作为调用函数 abort 或 raise 的结果外出现信号，且相应的信号处理程序调用函数 signal 使其返回值 SIG_ERR 时引用 errno 的值(见7.7.1.1条)。

——以形参 ap 调用宏 va_arg，而 ap 已被传递给以相同形参调用宏 va_arg 的一个函数(见7.8条)。

——抑制宏定义 va_start、va_arg、va_end 或它们的组合以访问一个实际函数(见7.8.1条)。

——宏 va_start 的形参*paramN* 声明为 register 存储类，或函数，或数组类型，或是一种与应用默

认的实参升格规则后所得到的类型不相容的类型(见7.8.1.1条)。

——调用宏 va_arg 时不存在下一个实参(见7.8.1.2条)。

——变长的实参表中下一个实在实参的类型与由宏 va_arg 所规定的类型不一致(见7.8.1.2条)。

——没有先调用对应的宏 va_start 的情况下调用宏 va_end(见7.8.1.3条)。

——在调用宏 va_end 之前从由宏 va_start 初始化的具有变长实参表的函数出现返回值(见7.8.1.3条)。

——函数 fflush 的流指向一个输入流或者最近一次操作是输入操作的更新流(见7.9.5.2条)。

——在对更新流的输出操作后跟随一个输入操作,而两者之间没有调用函数 fflush 或文卷定位函数;或者在对更新流的输入操作后跟随一个输出操作,而两者之间没有调用文卷定位函数(见7.9.5.3条)。

——函数 fprintf 或 fscanf 的格式与实参表不匹配(见7.9.6条)。

——在函数 fprintf 或 fscanf 的格式中发现无效的转换规格说明(见7.9.6条)。

——函数 fprintf 或 fscanf 的%%转换规格说明中在一对%字符间包含字符(见7.9.6条)。

——函数 fprintf 的转换规格说明中包含字符 h 或 l,以及除 d、i、n、o、u、x 或 X 以外的转换说明符,或者包含字符 L 以及除 e、E、f、g 或 G 以外的转换说明符(见7.9.6.1条)。

——函数 fprintf 的转换规格说明中包含#标记,以及除 o、x、X、e、E、f、g 或 G 以外的转换说明符(见7.9.6.1条)。

——函数 fprintf 的转换规格说明中包含0标记,以及除 d、i、o、u、x、X、e、E、f、g 或 G 以外的转换说明符(见7.9.6.1条)。

——除作为转换说明符%s(对字符型数组)或%p(对指向 void 的指针)的实参外,聚集或联合,或指向聚集或联合的指针是函数 fprintf 的实参(见7.9.6.1条)。

——由函数 fprintf 的单个转换产生的输出多于509个字符(见7.9.6.1条)。

——函数 fscanf 的转换规格说明中包含字符 h 或 l,以及除 d、i、n、o、u 或 x 以外的转换说明符,或者包含字符 L 以及除 e、f 或 g 以外的转换说明符(见7.9.6.2条)。

——在先前的程序执行期间由函数 fprintf 以%p 转换显示出的指针值是函数 fscanf 的%p 转换的实参(见7.9.6.2条)。

——由函数 fscanf 转换的结果不能用所提供的空间表示,或者接收结果的对象的类型不合适(见7.9.6.2条)。

——不能表示将一个串用函数 atof、atoi 或 atol 转换为一个数的转换结果(见7.10.1条)。

——引用一个指针值,该指针指向已由调用函数 free 或 realloc 释放了的空间(见7.10.3条)。

——函数 free 或 realloc 的指针实参与先前由函数 calloc、malloc 或 realloc 返回的指针不匹配,或该指针实参所指向的对象早已经调用函数 free 或 realloc 释放(见7.10.3条)。

——一个程序多次执行对函数 exit 的调用(见7.10.4.3条)。

——不能表示整型算术函数(abs、div、labs、或 ldiv)的结果(见7.10.6条)。

——当改变当前地点的 LC_CTYPE 类别时,函数 mblen、mbtowc 和 wctomb 的转义状态没有显式地复位到初始状态(见7.10.7条)。

——复写类函数或串接函数所要写的数组太小(见7.11.2条,7.11.3条)。

——在函数 strftime 的格式中发现无效的转换规格说明(见7.12.3.5条)。

F3 实现定义的行为

每个实现均应用文档说明在下列条文中所列出的每一方面的行为。下列子条目所述的行为是实现定义的。

F3.1 翻译

——如何标识诊断(见5.1.1.3条)。

F3.2 环境

——函数 main 实参的语义(见5.1.2.1条)。

——交互设备的组成(见5.1.1.3条)。

F3.3 标识符

——无外部链接的标识符中(超过31个的部分)有效初始字符的个数(见6.1.2条)。

——有外部链接的标识符中(超过6个的部分)有效初始字符的个数(见6.1.2条)。

——有外部链接的标识符中大小写有无区别(见6.1.2条)。

F3.4 字符

——除已在本标准中显式规定了的以外,源字符集和执行字符集中的成员(见5.2.1条)。

——用于对多字节字符编码的转义状态(见5.2.1.2条)。

——执行字符集中每个字符的位数(见5.2.4.2.1条)。

——(在字符常量与串字面值中)源字符集成员至执行字符集成员的映射(见6.1.3.4条)。

——包含有不在基本执行字符集中表示的字符或转义序列的整型字符常量的值,或包含有不在扩展字符集中表示的字符或转义序列的宽字符常量的值(见6.1.3.4条)。

——包含多于一个字符的整型字符常量值或包含多于一个多字节字符的宽字符常量值(见6.1.3.4条)。

——用于对宽字符常量将多字节字符转换为对应的宽字符(代码)的当前地域环境(见6.1.3.4条)。

——"普通"char 型的值的范围是否与 signed char 或 unsigned char 型的相同(见6.2.1.1条)。

F3.5 整数

——各类整数的表示和值集(见6.1.2.5条)。

——若不能表示将整数转换为较短的有符号整数,或将无符号整数转换为等长的有符号整数的结果值时(见6.2.1.2条)。

——对有符号整数进行按位运算的结果(见6.3条)。

——整数除法所得余数的符号(见6.3.5条)。

——值为负的有符号整数右移后的结果(见6.3.7条)。

F3.6 浮点数

——各类浮点数的表示和值集(见6.1.2.5条)。

——当将整型数转换为不能确切表示原先值的浮点数时截断的方向(见6.2.1.3条)。

——将一个浮点数转换为较短的浮点数时截断或舍入的方向(见6.2.1.4条)。

F3.7 数组与指针

——容纳最大尺寸的数组所需的整数类型——即 sizeof 算符的类型 size_t(见6.3.3.4条,7.1.1条)。

——将指针强制转换为整数,或将整数强制转换为指针的结果(见6.3.4条)。

——容纳指向同一数组中的元素的两个指针之差所需的整数类型 ptrdiff_t(见6.3.6条,7.1.1条)。

F3.8 寄存器

——对象能通过使用 register 存储类区分符而实际存放在寄存器中的程度(见6.5.1条)。

F3.9 结构、联合、枚举与位段

——使用不同类型的成员访问一个联合对象的成员(见6.3.2.3条)。

——结构成员的对齐与填充(见6.5.2.1条)。除非由一个实现写出的二进制数据将由另一个实现读入,否则这不应存在问题。

——"普通"的 int 型位段是按 signed int 位段对待还是按 unsigned int 位段对待(见6.5.2.1条)。

——单位内位段的分配次序(见6.5.2.1条)。

——位段是否可以跨越存储单位的边界(见6.5.2.1条)。

——所选择的表示枚举型值的整类型(见6.5.2.2条)。

F3.10 限定词

——对 volatile 限定类型对象的访问的构成(见6.5.5.3条)。

F3.11 声明符

——可修改算术、结构或联合类型的声明符的最大数目(见6.5.4条)。

F3.12 语句

——一个 switch 语句中 case 值的最大数目(见6.6.4.2条)。

F3.13 预处理指示

——控制条件并入的常量表达式中的单字符字符常量值是否与在执行字符集中的相同字符常量值匹配。这种字符常量能否为负值(见6.8.1条)。

——定位可并入的源文卷的方法(见6.8.2条)。

——对可并入的源文卷使用加引号的名字的支持(见6.8.2条)。

——源文卷字符序列的映射(见6.8.2条)。

——每个识别出的 #pragma 指示的行为(见6.8.6条)。

——当翻译的日期和时间不可用时, DATE_和 TIME_的定义(见6.8.8条)。

F3.14 库函数

——宏 NULL 所展开的空指针常量(见7.1.5条)。

——函数 assert 显示出的诊断消息及函数 assert 终止时的行为(见7.2条)。

——由函数 isalnum、isalpha、iscntrl、islower、isprint 和 isupper 所测试的字符集(见7.3.1条)。

——当定义域出错时由数学函数返回的值(见7.5.1条)。

——当出现下溢错时,数学函数是否将整型表达式 errno 置为宏 ERANGE 的值(见7.5.1条)。

——当函数 fmod 的第二个实参为0时,是出现定义域错还是返回0值(见7.5.6.4条)。

——函数 signal 的信号集合(见7.7.1.1条)。

——由函数 signal 识别出的每种信号的语义(见7.7.1.1条)。

——对由函数 signal 识别出的每种信号的默认处理和在程序启动时的处理(见7.7.1.1条)。

——若在调用信号处理程序前没有执行等价于 signal(sig,SIG_DFL);的操作,所实现的对信号的阻塞(见7.7.1.1条)。

——对函数 signal 规定的信号处理程序收到 SIGILL 信号时,是否复位默认的信号处理(见7.7.1.1条)。

——文本流的最后一行是否需要一个结束用的新行字符(见7.9.2条)。

——写出到文本流中的紧先于新行字符的空格字符在读入时是否出现(见7.9.2条)。

——写到二进制流中的数据后可以添加的空字符数(见7.9.2条)。

——对属于添加模式的流,文卷位置指示符在初始时是定位在文卷的开端还是在文卷的末尾(见7.9.3条)。

——对文本流的写操作是否会截断相关的文卷中超过该写出位置的内容(见7.9.3条)。

——文卷缓冲的特性(见7.9.3条)。

——长度为零的文卷是否有可能实际存在(见7.9.3条)。

——组成有效文卷名的规则(见7.9.3条)。

——同一文卷能否多次打开(见7.9.3条)。

——函数 remove 对开文卷的影响(见7.9.4.1条)。

——在使用该新名字的文卷早已存在的情况下,调用函数 rename 的效果(见7.9.4.2条)。

——函数 fprintf 中%p 转换的输出(见7.9.6.1条)。

——函数 fscanf 中%p 转换的输入(见7.9.6.2条)。

——对既不是在函数 fscanf 中%[转换的扫描表中的第一个字符,又不是最后一个字符的字符一的解释(见7.9.6.2条)。

——函数 fgetpos 或 ftell 在失败时对宏 errno 所置的值(见7.9.9.1,7.9.9.4条)。

——函数 perror 所产生的信息(见7.9.10.4条)。

——函数 calloc、malloc 和 realloc 在所请求分配的存储区尺寸为零时的行为(见7.10.3条)。

——函数 abort 对于开文卷和临时性文卷的行为(见7.10.4.1条)。

——实参值不为零、EXIT_SUCCESS 或 EXIT_FAILURE 时函数 exit 返回的状态(见7.10.4.3条)。

——环境名集合和改变由函数 getenv 所使用的环境表的方法(见7.10.4.4条)。

——由函数 system 传递的串的内容和执行模式(见7.10.4.5条)。

——由函数 strerror 返回的出错消息串的内容(见7.11.6.2条)。

——本地时区和夏令时(见7.12.1条)。

——函数 clock 所使用的纪元(见7.12.2.1条)。

F4 地域特定的行为

宿主环境的下列特性是地域特定的:

——除必需的成员外,执行字符集的其他内容(见5.2.1条)。

——印刷的方向(见5.2.2条)。

——十进制小数点字符(见7.1.1条)。

——字符测试函数和大小写映射函数的实现定义的概貌(见7.3条)。

——执行字符集的理序序列(见7.11.4.4条)。

——日期和时间的格式(见7.12.3.5条)。

F5 常见的扩展

下列扩展广泛使用在许多系统中,但不是对所有的实现都是可移植的。并入任何扩展将使一个严格符合程序成为无效。这类扩展的例子有:新的关键字,在标准前导文卷中声明的新的库函数,或名字不以下横线开始的预定义的宏。

F5.1 环境实参

在宿主环境中,函数 main 接受第三个实参 char *envp[ ],它指向一个以空结尾的指向 char 的指针数组,数组中每个指针指向一个串,该串提供了本次进程执行的有关环境信息(见5.1.2.2.1条)。

F5.2 特定标识符

在所要求的源字符集中未定义的,除下横线_、字母和数字字符以外的字符(例如币符¥,或民族字符集中的字符)可以出现在标识符中(见6.1.2条)。

F5.3 标识符的长度和大小写

无论有无外部链接,标识符中的所有字符均有效,且大小写表示不同的字符(见6.1.2条)。

F5.4 标识符的作用域

函数标识符、或其声明中包含关键字 extern 的对象的标识符的作用域为整个文卷(见6.1.2.1条)。

F5.5 可写的串字面值

可修改串字面值。等同的串字面值应可区别(见6.1.4条)。

F5.6 其他算术类型

定义了其他的算术类型,例如 long long int,以及它们的转换规则(见6.2.2.1条)。

F5.7 函数指针类型强制转换

指向一个对象或 void 型的指针可被强制转换成函数的指针，以便象函数一样调用数据（见6.3.4条）。函数指针可强制转换成指向一个对象或 void 型的指针，以便（例如用排错程序）检查或修改函数（见6.3.4条）。

F5.8 非整数位段类型

除 int、unsigned int 或 signed int 以外的类型可声明为具有合适最大宽度的位段（见6.5.2.1条）。

F5.9 关键字 fortran

可在函数声明中使用声明区分符 fortran，指示应产生适合于 FORTRAN 的调用，或要对外部名产生不同的表示（见6.5.4.3条）。

F5.10 关键字 asm

可使用关键字 asm 在翻译程序的输出中直接插入汇编语言代码。最常见的实现是通过形式为

asm(*字符串字面值*)；

的语句（见6.6条）。

F5.11 多重外部定义

对一个对象的标识符可有多个外部定义，定义时可显式使用也可不显式使用关键字 extern。若定义之间不一致，或对多个对象进行了初始化，则行为是未定义的（见6.7.2条）。

F5.12 空的宏实参

宏实参可不由预处理单词组成（见6.8.3条）。

F5.13 预定义的宏名

实现可在开始翻译前定义不以下横线开头的宏名来描述翻译和执行环境（见6.8.8条）。

F5.14 信号处理程序的额外实参

调用指定信号的处理程序时，除信号编号外，还可有额外的实参（见7.7.1.1条）。

F5.15 附加的流类型和文卷打开模式

可支持附加的从文卷到流的映射（见7.9.2条），还可通过在函数 fopen 的 mode 实参后添加字符来提供附加的文卷打开模式（见7.9.5.3条）。

F5.16 给出定义的文卷位置指示符

除在调用前其值为零的情况外，每次对文本流成功地调用函数 ungetc，文卷位置指示符均减1（见7.9.7.11条）。

附 录 G
索 引[1]
（参考件）

采用说明：

1] 括号内为 ISO/IEC 9899:1990中的英文索引名。

&	地址算符	(address operator	)6.3.3.2
&	按位与算符	(bitwise AND operator	)6.3.10
&&	逻辑与算符	(logical AND operator	)6.3.13
&=	按位与后赋值算符	(bitwise AND assignment operator	)6.3.16.2
( )	强制(类型)转换算符	(cast operator	)6.3.4
( )	函数调用算符	(function-call operator	)6.3.2.2
( )	圆括号(标点符号)	(parentheses punctuator	)6.1.6,6.5.4.3
*	间接算符	(indirection operator	)6.3.3.2
*	乘算符	(multiplication operator	)6.3.5
*	*号(标点符号)	(asterisk punctuator	)6.1.6,6.5.4.1
*=	乘后赋值算符	(multiplication assignment operator	)6.3.16.2
+	加算符	(addition operator	)6.3.6
+	一元加算符	(unary plus operator	)6.3.3.3
++	后缀增量算符	(postfix increment operator	)6.3.2.4
++	前缀增量算符	(prefix increment operator	)6.3.3.1
+=	加后赋值算符	(addition assignment operator	)6.3.16.2
,	逗号算符	(comma operator	)6.3.17
,…	省略,未规定的形参	(ellipsis, unspecified parameters	)6.5.4.3
-	减算符	(subtraction operator	)6.3.6
-	一元减算符	(unary minus operator	)6.3.3.3
--	后缀减量算符	(postfix decrement operator	)6.3.2.4
--	前缀减量算符	(prefix decrement operator	)6.3.3.1
-=	减后赋值算符	(subtraction assignment operator	)6.3.16.2
->	结构或联合指针算符	(structure/union pointer operator	)6.3.2.3
.	结构或联合成员算符	(structure/union member operator	)6.3.2.3
…	省略号(标点符号)	(ellipsis punctuator	)6.1.6,6.5.4.3
/	除算符	(division operator	)6.3.5
/**/	注释定界符	(comment delimiters	)6.1.7
/=	除后赋值算符	(division assignment operator	)6.3.16.2
:	冒号(标点符号)	(colon punctuator	)6.1.6,6.5.2.1
;	分号(标点符号)	(semicolon punctuator	)6.1.6,6.5,6.6.3
<	小于算符	(less-than operator	)6.3.8
<<	左移算符	(left_shift operator	)6.3.7
<<=	左移后赋值算符	(left_ shift assignment operator	)6.3.16.2
<=	小于等于算符	(less-than-or-equal-to operator	)6.3.8
=	等于号(标点符号)	(equal-sign punctuator	)6.1.6,6.5,6.5.7
=	简单赋值算符	(simple assignment operator	)6.3.16.1
==	等于算符	(equal-to operator	)6.3.9
>	大于算符	(greater-than operator	)6.3.8
>=	大于等于算符	(greater-than-or-equal-to operator	)6.3.8
>>	右移算符	(right-shift operator	)6.3.7
>>=	右移后赋值算符	(right-shift assignment operator	)6.3.16.2
?:	条件算符	(conditional operator	)6.3.15

附加说明：

本标准由中华人民共和国电子工业部提出。

本标准由电子工业部标准化研究所归口。

本标准由西安电子科技大学负责起草。

本标准主要起草人金益民、陈平、冯惠、孙玉方、段祥、黄嘉启、周明德。