目录

文档翻译组织：黄斌

文档统稿： 刘俊波

参与文档翻译：唐泽伟， 陈思帆， 冯帅辉， 张宇， 吴靖靖， 董飞， 郭航冶， 李俊诚， 林泽鹏， 刘俊波， 刘向同， 施继荣， 吴政， 黄豪

编程语言-C

1. 范围

1.1. 本国际标准规定了用 C 编程语言编写的程序的形式和解释，它规定：

- 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；
- 将由 C 程序处理的输入数据的表示；
- C 程序产生的输出数据的表示；
- C 的一致性实现所施加的约束和限制。

1.2. 本国际标准未明确规定：

- 转换 C 程序以供数据处理系统使用的机制；
- 数据处理系统调用 C 程序以供其使用的机制；
- 转换输入数据供 C 程序使用的机制；
- C 程序产生输出数据后进行转换的机制。
- 程序及其数据的大小或复杂性将超过任何特定数据处理系统或特定处理器的容量；
- 能够支持一致性实现的数据处理系统的所有最低要求。

2. 规范性引用

以下规范性文件所载规定，经本案文引用，构成本国际标准的规定。对于注明日期的参考文献，对任何这些出版物的后续修订或修订均不适用。但是，我们鼓励根据本国际标准达成协议的缔约方调查适用下列规范性文件的最新版本的可能性。对于未注明日期的参考文件，适用所参考的规范文件的最新版本。ISO 和 IEC 成员注册现行有效的国际标准。

ISO 31−11:1992, Quantities and units — Part 11: Mathematical signs and symbols for use in the physical sciences and technology.

ISO/IEC 646, Information technology — ISO 7-bit coded character set for information interchange.

ISO/IEC 2382−1:1993, Information technology — Vocabulary — Part 1: Fundamental terms.

ISO 4217, Codes for the representation of currencies and funds.

ISO 8601, Data elements and interchange formats — Information interchange — Representation of dates and times.

ISO/IEC 10646 (all parts), Information technology — Universal Multiple-Octet Coded Character Set (UCS).

IEC 60559:1989, Binary floating-point arithmetic for microprocessor systems (previously designated IEC 559:1989).

3. 术语、定义和符号

该国际标准使用了如下定义。其他术语在其出现的位置以斜体显示，或放在语法规则的左侧。该国际标准中显式定义的术语不被假设引用了其它地方隐式地声明的类似的术语。该国际标准中未定义的术语依据 ISO/IEC 2382-1 中的解释。该国际标准中未定义的数学符号依据 ISO 31-11 中的解释。

3.1. access 访问

<执行时间动作>要读取或修改对象的值

注 1 如果这两个操作中只有一个表示，则使用“read（读取）”或“modify（修改）”。

注 2 “Modify（修改）”包括正在存储的新值与前一个值相同的情况。

注 3 未求值的表达式不能访问对象。

3.2. alignment 对齐

要求特定类型的对象位于存储边界上，其地址是字节地址的特定倍数。

3.3. argument 参数

实变量

实际参数（不建议使用）

一个函数调用表达式中由圆括号包围的、由逗号分隔的表达式列表，或函数宏调用中有圆括号包围的、由逗号分隔的预处理记号序列

3.4. behavior 行为

外部表现或动作。

3.4.1. implementation-defined behavior

未指定的行为，其中每个实现都记录了是如何做出选择的。

例子 实现定义行为的一个例子是，当一个有符号整数右移，高位的传输。

3.4.2. locale-specific behavior

依赖于每个实施文件所记录的当地国籍、文化和语言惯例的行为。

例子 特定于区域的行为的一个例子是，对于 26 个小写拉丁字母以外的字符，islower 函数是否返回 true。

3.4.3. undefined behavior

使用该国际标准未强制要求的不便的或错误的程序构造、错误的数据时的行为。

注 可能的未定义行为包括完全忽略具有不可预测结果的情况，到在翻译或程序执行过程中以环境特征的记录方式行为（有或不发布诊断消息），到终止翻译或执行（发布一个诊断消息）。

例子 未定义行为的一个例子是在整数溢出上的行为。

3.4.4. unspecified behavior

本国际标准提供了两种或两种以上的可能性，且在任何情况下都没有提出进一步要求的行为。

例子 未指定行为的一个例子是计算对函数的参数的顺序。

3.5. bit 位

执行环境中的数据存储单元，它大到足以容纳可能具有两个值之一的对象。

注 不需要能够表示一个对象的每一个有效位的地址。

3.6. byte 字节

数据存储的可寻址单元，足以容纳执行环境的基本字符集的任何成员。

注 1 可以唯一地表示对象每个单独字节的地址。

注 2 字节由连续的位序列组成，其数量由实现定义。最低有效位称为低阶位；最高有效位称为高阶位。

3.7. character 字符

<摘要>用于组织、控制或表示数据的一组元素的成员。

3.7.1. character 字符

单字节字符

<C>适合于一个字节的位表示法

3.7.2. multibyte character 多字节字符

一个或多个字节的序列，表示源或执行环境的扩展字符集的成员。

注　扩展字符集是基本字符集的超集。

3.7.3. wide character 宽字符

适合于 wchar_t 类型的对象的位表示，能够表示当前区域环境中的任何字符。

3.8. constraint 约束

限制，句法的或语义的，用来解释语言元素的阐述。

3.9. correctly rounded result 正确的四舍五入结果

根据有效的舍入模式，以最接近值的结果格式表示，该结果的范围和精度不受限制。

3.10. diagnostic message 诊断消息

属于实现消息输出的实现定义子集的消息。

3.11. forward reference 向前引用

参考本国际标准后面的一个子条款，其中包含与本子条款相关的附加信息。

3.12. implementation 实现

特定的软件集，在特定控制选项下的特定翻译环境中运行，为特定执行环境执行程序的翻译，并支持执行环境中的功能。

3.13. implementation limit 实现限制

通过实现而对程序施加的限制。

3.14. object 数据对象

执行环境中的数据存储区域，其内容可以表示值。

注　当引用时，对象可以解释为具有特定类型；参见 6.3.2.1。

3.15. parameter 参数

形式参数（formal parameter）

形式参数（formal argument）（不建议使用）

作为函数声明或定义的一部分声明的对象，它获取函数的入口值，或从类似函数的宏定义中的宏名称后面的逗号分隔列表中获得的标识符。

3.16. recommended practice 操作规程建议（工业标准）

强烈建议符合本标准意图的规范，但对于某些实现可能不切实际。

3.17. value 数值

当被理解为具有特定类型时，对象内容的精确含义。

3.17.1. implementation-defined value 实现定义值

未指定的值，其中每个实现记录了如何进行选择。

3.17.2. indeterminate value 不确定值

一个未指定的值或一个陷阱的表示形式。

3.17.3. unspecified value 未申明的值

本国际标准在任何情况下都没有规定要选择该值的相关类型的有效值。

注 未指定的值不能是陷阱的表示形式。

3.18. ⌜x⌝

x 的上限：大于或等于 x 的最小整数。

例子 ⌜2.4⌝ 是 3， ⌜ -2.4 ⌝ 是-2。

3.19. ⌞x⌟

x 的下限：小于或等于 x 的最大整数。

例子 ⌞2.4⌟ 是 2， ⌞ -2.4 ⌟ 是-3。

4. 一致性

在本国际标准中，“shall”应被解释为对实施或计划的要求；相反，“shall not”应被解释为禁止。

如果违反了“shall”或“shall not”出现在约束之外的要求，则该行为未定义。在本国际标准中，未定义行为被表示为“未定义行为”或省略任何对行为的明确定义。这三个人在重要性上没有区别；它们都描述了“未定义的行为”。

在所有其他方面都正确的程序，根据正确的数据操作，包含未指明行为的程序应该是正确的程序，并按照 5.1.2.3 进行操作。

该实现不能成功地编译包含#error 预处理指令的预处理编译单元，除非它是被条件包含所跳过的组的一部分。

严格遵守标准的程序只能使用本国际标准中规定的语言和库的特点 2)。它不得根据任何未指明、未定义或实现定义的行为产生输出，也不得超过任何最低实现限度。

这两种相一致的实现形式是托管的和独立的。一个符合要求的主持实施人员应接受任何严格符合要求的程序。一个符合标准的独立实现应接受任何严格符合标准的程序，其中不使用复杂的类型，并且在库条款（第 7 条）中规定的特性的使用仅限于标准头文件<float.h>,<iso646.h>,<limits.h>,<stdarg.h>,<stdbool.h>, <stddef.h>,<stdint.h>的内容。一个符合要求的实现可能有扩展（包括额外的库函数），只要它们不改变任何严格符合要求的程序的行为 3)。

————————————

2) 严格符合条件的程序可以使用条件特性(如附件 F 中的特性)，前提是使用由带有适当宏的#ifdef 指令保护。例如：

#ifdef _ _STDC_IEC_559_ _ /* FE_UPWARD defined */

```
/* ... */
fesetround(FE_UPWARD);
/* ... */
#endif
```

3) 这意味着一致性实现除了在本国际标准中明确保留的标识符外，不保留任何标识符。

———————————

合格的程序是指合格的实施所可接受的程序 4)。

实现应附有定义所有实现定义和特定地区的特征和所有扩展的文档。

向前引用：条件包含（6.10.1），错误指令（6.10.5），浮点型<float.h>（7.7）的特征。替代拼写<iso646.h>（7.9），整数类型<limits.h>（7.10）的大小，变量参数<stdarg.h>（7.15），布尔类型和值<stdbool.h>（7.16），常见定义<stddef.h>（7.17），整数类型<stdint.h>（7.18）。

———————————

4) 严格符合要求的程序旨在在符合要求的实现中最大限度地可移植。一致性程序可能取决于一致性实现的不可移植特性。

5. 环境

一个实现在两个数据处理系统环境中转换 C 源文件并执行 C 程序，这在本国际标准中被称为翻译环境和执行环境。它们的特征定义和约束了根据语法和语义规则构建的一致性 C 程序的执行结果。

向前引用：在本条款中，只注意到许多可能的正向引用中的少数。

5.1. 概念模型

5.1.1. 翻译环境

5.1.1.1. 程序结构

一个 C 程序不需要全部同时被翻译。程序的文本以本国际标准中称为源文件（或预处理文件）的单位保存。通过预处理指令#include 所包含的源文件以及所有标头和源文件被称为预处理翻译单元。经过预处理后，预处理翻译单元称为翻译单元。以前翻译过的翻译单元可以单独保存，也可以保存在文库中。程序的独立翻译单元通过（例如）调用标识符具有外部链接的函数、标识符具有外部链接的对象的操作或数据文件的操作进行通信。翻译单元可以单独翻译，然后链接起来生成一个可执行程序。

前向引用：标识符的链接（6.2.2）、外部定义（6.9）、预处理指令（6.10）。

5.1.1.2. 翻译语法

翻译的语法规则之间的优先级由以下阶段指定。[5)]

1. 物理源文件多字节字符将以实现定义的方式映射到源字符集（为行末指示符引入新行字符）。三角图序列被相应的单字符内部表示所取代。
2. 删除紧跟反斜杠字符（\）和新行字符的每个实例，将物理源线拼接以形成逻辑源线。只有任何物理源线上的最后一个反斜杠才有资格成为此类拼接的一部分。非空的源文件应以新行字符结尾，在进行任何此类拼接之前，该新行字符前不得紧接着加反斜杠字符。
3. 源文件被分解为预处理标记[6)]和空白字符序列（包括注释）。源文件不得以部分预处理令牌或部分注释结尾。每个注释都被一个空格字符替换。此时将保留新行字符。除了新行之外的每个非空格字符序列是保留还是被一个空格字符替换，都是实现定义的。
4. 执行预处理指令，展开宏调用，并执行_Pragma 一元运算符表达式。如果与通用字符名称的字符序列匹配是由标记连接产生的（6.10.3.3），则该行为未定义。#include 预处理指令导致递归地处理命名头或源文件。然后将删除所有的预处理指令。
5. 字符常量和字符串文字中的每个源字符集成员和转义序列都被转换为执行字符集的对应成员；如果没有对应的成员，则它被转换为除空（宽）字符以外的实现定义的成员。[7)]
6. 相邻的字符串文字标记被连接起来。
7. 分隔标记的空白字符不再重要。每个预处理令牌都被转换为一个令牌。所得到的标记通过语法和语义进行分析，并翻译为一个翻译单元。
8. 所有的外部对象和函数引用都将被解析。链接库组件以满足对当前翻译中未定义的函数和对象的外部引用。所有这些转换器输出都被收集到一个程序图像中，其中包含在其执行环境中执行所需的信息。

前向引用：通用字符名称（6.4.3）、词汇元素（6.4）、预处理指令（6.10）、触发序列（5.2.1.1）、外部定义（6.9）。

————————————————

6) 如 6.4 中所述，将源文件的字符划分为预处理令牌的过程是与上下文相关的。例如，请参阅在#include 预处理指令中对<的处理。

7) 实现不需要将所有非对应的源字符转换为相同的执行字符。

5.1.1.3. 诊断

如果预处理翻译单元或翻译单元包含违反任何语法规则或约束，即使行为也被明确指定为未定义或实现定义的，一致性实现应产生至少一条诊断消息(以实现定义的方式识别)在其他情况下，不需要产生诊断消息。8)

例如 一个实施例应针对翻译单元发出诊断信息：

```
char i;
int i;
```

因为在本国际标准中的措辞将构造的行为描述为既是约束错误又导致未定义行为的情况下，应诊断约束错误。

5.1.2. 执行环境

定义了两个执行环境：独立式环境和托管式环境。在这两种情况下，当执行环境调用一个指定的 C 函数时，就会发生程序启动。所有具有静态存储持续时间的对象都应在程序启动前进行初始化（设置为其初始值）。这种初始化的方式和时间是未指定的。程序终止会将控制返回给执行环境。

前向引用：对象的存储持续时间（6.2.4）、初始化（6.7.8）。

5.1.2.1. 独立环境

在一个独立的环境中(C 程序的执行可以在没有任何操作系统的好处的情况下进行)，在程序启动时调用的函数的名称和类型是由实现定义的。除第 4 条要求的最小集外，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影响是实现定义的。

5.1.2.2. 主机环境

宿主式环境不一定需要有，但是如果有的话，需要遵守下面的规范。

5.1.2.2.1. 程序启动

1 程序启动时，被执行的函数叫做 MAIN 函数。这个函数的执行没有原型。这个函数在定义时应定义为没有参数的返回类型为整型（int）的函数：

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

或者具有两个参数（这里被定义为 argc 和 argv，尽管什么名称都可以被使用，他们在他们被声明的函数内是局部的）

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

或者等同的东西（int 可以被 typedef 定义的 int 型取代，argv 可以被写成 char ** argv，诸如此类），或者一些其他实现定义的方式。

2 如果这些 main 函数中的参数被声明的话，它们需要遵守如下的限制。

—argc 的值必须为非负数。

—argv[argc]需为一个空指针。

—如果 argc 的值大于 0，则数组成员 argv[0]到 argv[argc-1]需包含指向给定字符串的指针，该字符串是由宿主式环境在程序启动之前给定的定义实现的值。这么做的目的在于在程序启动之前向程序提供来自于宿主式环境中的其他地方的确定的信息。如果宿主式环境不能同时提供包含大小写字母的字符串，那么这次实现就必须保证字符串以小写的形式被接收。

—如果 argc 的值大于 0，argv[0]的字符串指针代表程序名称。如果程序名称不在该宿主式环境中的话，argv[0][0]需为空。如果 argc 的值大于 1，则 argv[1]到 argv[argc-1]的字符串指针代表程序参数。

—参数 argc 和 argv 以及 argv 数组指向的字符串应是可以被修改的。从程序开始到程序结束，他们应保持为他们最后被保存的值。

5.1.2.2.2. 程序执行

1 在宿主式环境中，一个程序可以使用库条款中的所有函数，宏命令，类型定义，对象。

5.1.2.2.3. 程序结束

1 如果 main 函数的返回类型与 int 兼容，初始调用 main 函数的返回相当于用 main 函数返回的值作为参数调用 exit 函数的返回。函数运行到}结束运行 main 函数后，返回 0。如果返回类型与 int 不兼容，返回给寄主式环境的中止状态就是不确定的。

5.1.2.3. 程序运行

1 这个国际标准中的语言描述描述了一个抽象计算机的行为，与优化无关。

2 访问一个变化的对象，修改一个对象，修改一个文件，调用一个函数，上述这些操作都是 C 语言中的副作用，是执行环境的状态的改变。对表达式进行求值可能会产生副作用。在执行序列中的某些指定点被称为序列点，序列点前的所有副作用已经结束，而序列点后的副作用还没有发生。（关于序列点的总结在 annex C 中被给出。）

3 在抽象计算机中，所有表达式都按照语义指定的方式计算。如果一个表达式的值没有被使用，且没有产生副作用，那么在实际的实现中，可以不去计算这个表达式（包括调用函数或是访问变化的对象引起的）。

4 当抽象计算机的进程被接收到的信号打断时，只有前一个序列点的对象的值可以被使用。在前一个序列点和下一个序列点之间可能被修改的对象不必获得正确的值。

5 C 语言的标准实行的最低需求有：

—在序列点时，可变对象是稳定的，因为之前的访问已经完成，后续的访问还没有开始。

—在程序终止时，所有写入文件的数据应与根据抽象语义所产生的程序执行结果相同。

—交互设备的输入和输出动态应按照 7.19.3 的规定进行。这些要求的目的是未缓存或行缓存的输出尽快产生，以确保提示消息实际出现在等待输入的程序之前。

交互设备的构成是由实际的实现定义的。

每一次实现都可能在抽象语义和实际语义之间构建更严格的联系。

示例 1：一次实现可以在抽象语义和实际语义之间定义一个一对一的对应关系：在每个序列点，实际对象的值

将与抽象语义指定的值一致。而可变这个词在此时就是多余的。

或者，一次实现可以在每个编译单元内执行各种优化，在这种情况下，只有在跨越编译单元边界进行函数调用时，实际语义才会与抽象语义一致。在这种实现中，当调用函数和被调用函数处于不同编译单元时，在每个函数入口和函数返回时，所有外部链接的对象以及通过其中的指针访问的所有对象的值都将与抽象语义一致。此外，在每一个这样的函数入口处，被调用函数的形参值和通过其中的指针访问的所有对象的值都符合抽象语义。在这样的实现中，由信号函数激活的中断服务例程引用的对象需要明确地指定可变存储，以及一些其他由实现定义的限制。

示例 2：在执行如下片段时

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

“整型提升”要求抽象计算机将每个变量变化为整形，然后将两个整型相加并截断总和。如果两个字符相加时不会发生溢出，或是溢出被静默的包装成正确的结果，那么实际的执行只需要生成正确的结果，忽略提升的过程。

11 示例 3：同样的，在下列片段中

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

如果实现时能够确定乘法执行的结果与使用双精度算法执行的结果相同，则可以使用单精度算法执行乘法（例如 d 被替换为 double 类型的常量 2.0）。

12 示例 4：使用宽寄存器的实现需遵守适当的语义。值与它是在寄存器中还是在内存中时无关的。例如，寄存器的隐性溢出是不能改变值的。并且，需要显性的存储和加载使值保留到存储的精度。尤其需要注意的是，指定的转换需要强制转换和赋值得以实现。在如下片段中：

```
double d1, d2;
float f;
d1 = f = expression;
d2 = (float) expression;
```

赋给 d1 和 d2 的值需被转化为浮点型。

13 示例 5：由于精度和范围的限制，浮点表达式的重新排列经常受到限制。由于在保留时发生错误，实现通常不能应用加法和乘法的相关数学规则，也不能应用分配规则，即使是在没有向上溢出或是向下溢出的情况下。同样的，实现通常不能替换十进制常量以实现表达式的重新排列。在下面的片段中，由数学规则产生的重新排列通常是无效的。

```
double x, y, z;
/* ... */
x = (x * y) * z; // 不等同于x *= y * z;
z = (x - y) + y ; // 不等同于 z = x;
z = x + x * y; // 不等同于 z=x* (1.0 + y);
y=x/ 5.0; // 不等同于 y=x* 0.2;
```

14 示例 6：为了解释表达式的分组行为，在下面的片段中

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

这个表达式实际上和下面的表达式意义相同

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

这是由于这些操作符具有的结合性和优先级。（a+32760）的和与 b 相加，和再与 5 相加，得到的结果重新赋给 a。但是在计算机中，数据溢出会产生显而易见的错误，例如整型仅包括范围在[-32768，32767]中的数，因此，一次实行不能将表达式重写为

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

如果 a 和 b 的值分别为-32754 和-15，a+b 的值就会溢出产生错误，然而原始的表达式则不会，这个表达式同样不能被重写成

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

或是

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

在 a 和 b 的值分别是 4 和-8 或是-17 和 12 的情况下。然而，在能以静默的方式产生一些值，并且正向溢出和负向溢出相互抵消的计算机中，可以用以上的任意一种方式重写表达式，因为他们的值实际上都是一样的。

15 示例 7：表达式的计算并不完全取决于它的分组。在下列的片段中

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

表达式就被分组写成了

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

但是 p 的实际自加可能发生在前一个序列点和下一个序列点之间的任意时间（;），而对 getchar 的调用可以发生在需要获得他的返回值的任意时间点。

5.2. 环境考虑

5.2.1. 字符集

1 应当定义两组字符及其相关的核对序列的集合：写入源文件的集合（源字符集），和执行环境的解释的集合（执行字符集）。每个字符集被进一步划分为一个基本字符集，它的内容由这个集合给出，和零个或者多个局部设置的成员（它们不是基本字符集的成员），被称为扩展字符。这个组合而成的字符集也被成为扩展字符集。执行字符集的值是由实现定义的。

2 在字符常量或是字符串常量中，执行字符集的成员应由源字符集的相应成员表示，或由反斜杠\后跟一个或多个字符的转义字符或序列表示。一个所有位都为 0 的字节，被称为空字符，应该存在与基本执行字符集中，用于中止字符串。

3 基本源字符集和基本执行字符集都应有如下成员：26 个大写的拉丁字母

```
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
```

26 个小写的拉丁字母

```
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

```
0 1 2 3 4 5 6 7 8 9
```

29 个图形文字

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

空格字符和表示水平制表，垂直制表和换页的控制字符。源字符集和基本执行字符集的每一个成员需用一个字节来表示。在源字符集和基本执行字符集中，上述的十进制数字列表中 0 后的每一个字符成员的值都要比前一个大

1。在源文件中每一行文本的结尾都应该有某种表示结束的方式，在本国际标准中，单个的换行字符被作为行结束指示符。在基本执行字符集中，应当由表示警告，退格，回车换行的控制字符。如果在源文件中有任何其他字符（除非在标识符，字符常量，字符串常量，头名称，注释或是永远不会变成标记的预处理标记），那么它就是未被定义的。

4 字母是上面定义的大小写字母；在这个国际标准中，字母不表示其他字母中的其他字符。

5 通用字符的名称构造方式提供了一种命名其他字符的方法。

5.2.1.1. 三字符序列

1 源文件中出现的以下三字符序列，都将被替换为对应的单个字符。

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

除此之外就没有其他三字符序列了。每一个没有对应上述列表的?是不会被替换的。

2 示例：下面的源代码行

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

会变成（在被上述列表替换后）

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

5.2.1.2. 多字节字符

1 源字符集可能包含多字节字符，用于表示扩展字符集的成员。执行字符集也可能包含多字节字符，但是它们不需要像在源字符集中的多字节字符那样被编码。在两种字符集中，需要遵守如下的规则

—必须要有基本字符集，且其中的每一个成员都应当被可以编码成单个字节。

—任何其他成员的存在，含义和表示都是局部特定的。

—多字节字符集可以拥有根据状态的编码，其中每个多字节字符序列以初始移位状态开始，并在遇到特定的多字节字符后进入局部特定的移位状态。在初始移位状态中，所有单字节字符保持他们原本的含义并且不会改变移位状态。序列中后续字节的含义是当前移位状态的函数。

—一个所有位为 0 的字节应被解释为一个独立于移位状态的空字符。

—所有位为 0 的字节不应出现在多字节字符的第二个及后续字节中。

2 对于头文件来说，应遵守如下规则：

—一个标识符，注释，字符串常量，字符常量，或是头名称应当在初始移位状态开始或是结束。

—一个标识符，注释，字符串常量，字符常量，或是头名称应当由有效的多字节字符序列组成。

5.2.2. 字符显示语义

1 活动位置是指显示设备上 fputc 函数输出的下一个字符出现的位置。将打印字符（由 isprint 函数定义）写入显示设备的目的是为了在活动位置显示该字符，然后将活动位置推进到当前行的下一个位置。写的方向是局部定义的。如果活动位置在一行的最后一个位置（如果有的话），那么显示设备的行为就是不确定的。

2 执行字符集中的非图形化的字母转义序列可用于在显示设备中产生如下操作

\a （alert）在不改变活动位置的前提下产生可听到或可看到的警报。

\b （backspace）将活动位置退回当前行的前一个位置。如果活动位置在当前行的初始位置，那么显示设备的行为就是不确定的。

\f (form feed)将活动位置移动到下一个逻辑页的初始位置。

\n (new line)将活动位置移到下一行的初始位置。

\r (carriage return)将活动位置移到当前行的初始位置。

\t (horizontal tab)将活动位置移动到当前行的下一个水平制表位置。如果活动位置已经在或者已经超过了当前行最后一个定义的水平制表位置，那么像是设备的行为就是不确定的。

\v (vertical tab)将活动位置移动到下一个垂直制表位置的初始位置。如果活动位置已经在或者已经超过了最后一个定义的垂直制表位置，那么像是设备的行为就是不确定的。

3 每一个转义序列都会产生一个独立的实现定义的值，该值能被存放在单个字符型中。文本文件中的外部表示和内部表示不必完全相同，并且这也超出了本国际标准的范围。

5.2.3. 信号和中断

1 函数的实现应当是这样的：函数运行会在任意时候被打断，或是被信号处理程序调用，或者两者都会，并且不会被打断执行的函数产生影响，该函数依然活跃，调用的控制流结束后，函数返回值或是具有自动存储功能的对象。在每次调用时，所有这些对象都应当在函数映像（使函数可执行的指令）外被保护起来。

5.2.4. 环境限制

1 编译和执行环境都限制了语言编译器和库的实现。下面的内容描述了语言相关的环境对规范执行的限制，与库相关的限制在 7 中讨论。

5.2.4.1. 转换限制

1 实现应当能够转换或是执行至少一个程序，该程序应至少包含以下每种限制的一个实例：

—127 个嵌套级的块

—63 个嵌套级的条件包含

—12 个修改了声明中的算术，结构，联合或是不完全类型的指针，数组和函数声明（以任意组合形式）

—63 个在一个完整声明符中的嵌套级的括弧声明

—63 个在一个完整表达式中的嵌套级的括弧表达式

—63 个重要的在内部标识符或宏名称中的初始字符（每个通用字符名或扩展源字符被视为单个字符）

—31 个重要的外部标识符中的初始字符（每个通用字符名称指定的短标识符 0000FFFF 或更少的被认定为 6 个字符，每个通用字符名称指定的短标识符 00010000 或更少的被认定为 10 个字符，每个扩展源字符被认定是对应的通用字符的名称的相同数量的字符，如果有的话）

—4095 个在一个编译单元中的外部标识符

—511 个在一个块中声明的作用域标识符

—4095 个在一个预处理编译单元中的宏标识符

—127 个在一个函数定义中的参数

—127 个在一次函数调用中的参数

—127 个在一个宏定义中的参数

—127 个在一次宏调用中的参数

—4095 个在一个逻辑行中的字符

—4095 个在字符串常量或是宽字符串常量中的字符（在相关后）

—65535 个在对象中的字节（仅在宿主式环境中）

—15 个嵌套级的包含（#include）文件

—1023 个 switch 语句的大小写标签（不包括任何嵌套 switch 语句的大小写标签）

—1023 个在结构体中或者联合体中的成员

—1023 个在单次枚举中的枚举常量

—63 个在一个结构声明列别中的嵌套级联合定义

5.2.4.2. 数值范围

需要一个实现来记录子句中指定的所有限制，这些限制在头文件<limits.h>和<float.h>中指定。其他限制在<stdint.h>中指定。

前向引用:整型< stint .h> (7.18)

5.2.4.2.1. 整数类型的大小<limits.h>

下面给出的值应该被替换为适合在#if 预处理指令中使用的常量表达式。此外，除 CHAR_BIT 和 MB_LEN_MAX 外，下列表达式应被替换为与根据整数升级转换的对应类型对象的表达式具有相同类型的表达式。它们的实现定义值在大小(绝对值)上应等于或大于所示的具有相同符号的值。

-最小的非位域(字节)对象的位数

CHAR_BIT 8

- signed char 类型对象的最小值

SCHAR_MIN -127 // −(2^7 − 1)

- signed char 类型对象的最大值

SCHAR_MAX +127 // 2^7 − 1

- unsigned char 类型对象的最大值

UCHAR_MAX 255 // 2^8 − 1

- char 类型对象的最小值

CHAR_MIN see below

- char 类型对象的最大值

CHAR_MAX see below

–对于任何支持的区域设置，多字节字符的最大字节数

MB_LEN_MAX 1

-对于任何支持的语言环境，多字节字符的最大字节数- short int 类型对象的最小值

Shrt_min -32767 //–(2^15–1)

- short int 类型对象的最大值

Shrt_max +32767 // 2^15–1

- unsigned short int 类型对象的最大值

Ushrt_max 65535 // 2^16–1

- int 类型对象的最小值

Int_min -32767 //(2^15–1)

- int 类型对象的最大值

Int_max +32767 // 2^15–1

- unsigned int 类型对象的最大值

Uint_max 65535 // 2^16–1 . Uint_max 65535

- long int 类型对象的最小值

LONG_MIN -2147483647 // $-(2^{31}-1)$

长整型对象的最大值

Long_max +2147483647 // $2^{31}-1$

- unsigned long int 类型对象的最大值

Ulong_max 4294967295 // $2^{32}-1$

- long long int 类型对象的最小值

Llong_min -9223372036854775807 //$(2^{63}-1)$

- long - long int 类型对象的最大值

Llong_max +9223372036854775807 // $2^{63}-1$

- unsigned long long int 类型对象的最大值

Ullong_max 18446744073709551615 // $2^{64}-1$

如果 char 类型的对象的值在表达式中被当作有符号整数使用，CHAR_MIN 的值应该和 CHAR_MIN 的值一样，CHAR_MAX 的值应该和 CHAR_MAX 的值一样。否则，CHAR_MIN 值为 0,CHAR_MAX 值与 UCHAR_MAX 相同。UCHAR_MAX 值应等于 2^{CHAR_BIT-1}。

前向引用:类型的表示(6.2.6)，条件包含(6.10.1)

5.2.4.2.2. 浮动类型的特点

1. 浮动类型的特征是根据描述浮点数和值的表示为模型被定义的，这些浮点数和值可以提供有关实现浮点算法的信息。（16) 以下参数用于为每个浮点类型定义模型：

S 符号 （±1）

b 基数 （一个大于 1 的整数）

e 指数 一个整数且$e_{min} < e < e_{max}$

p 有效数字中以 b 为基数的位数

f_k 小于 b 的非负整数

2 . 浮点数由以下模型定义：

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{\min} \le e \le e_{\max}$$

3. 除了标准化的浮点数（如果 x ≠ 0 时 $f_1 > 0$），浮点类型可能是能够包含其他类型的浮点数，例如非正规浮点数数字 (x ≠ 0, e = e_{min} , f_1 = 0) 和非正规归化浮点数 (x ≠ 0,

e > e_{min}, f_1 = 0)，以及不是浮点数的值，例如无穷大和 NaNs。 NaN 是表示一个非数的编码。一个安静的 NaN 传播通过几乎所有算术运算而不会引发浮点异常；一个信号 NaN 通常在发生算数操作数时引发浮点异常.

4. 浮点运算（+、-、*、/）和库函数的准确性返回浮点结果的 <math.h> 和 <complex.h> 是实现定义的。实现可能会声明准确性未知。

5. <float.h> 头文件中的所有整数值，除 FLT_ROUNDS 外，应为常量适合在#if 预处理指令中使用的表达式；所有浮动值应为常量表达式。除了 DECIMAL_DIG、FLT_EVAL_METHOD、FLT_RADIX 和 FLT_ROUNDS 外对所有三种浮点类型都有不同的名称。为为所有值提供浮点模型表示除了 FLT_EVAL_METHOD 和 FLT_ROUNDS。

6. 浮点加法的舍入模式由实现定义的 FLT_ROUNDS 值表征：

-1 不确定的

0 趋向于 0

1 到最近

2 向正无穷大

3 向负无穷大

FLT_ROUNDS 的所有其他值表征实现定义的舍入行为。

7. 具有浮动操作数的操作的值和服从通常算术的值转换和浮点常量被评估为一种格式，其范围和精度可能大于类型所需。评估格式的使用特点通过 FLT_EVAL_METHOD 的实现定义值：

-1 不确定的

0 仅根据范围和精度评估所有操作和常数类型

将 float 和 double 类型的运算和常量计算为 double 类型的范围和精度，计算 long doublelong double 的范围和精度的操作和常量类型

评估所有操作和常数的范围和精度长双型。

FLT_EVAL_METHOD 的所有其他负值表征实现定义行为。

8. 以下列表中给出的值应替换为常量表达式幅度大于或等于的实现定义的值（绝对值）显示的，带有相同符号的：

指数表示的基数，b

FLT_RADIX **2**

浮点有效数中的 base-FLT_RADIX 位数，p

FLT_MANT_DIG

DBL_MANT_DIG

LDBL_MANT_DIG

十进制位数，n，使得任何浮点数在最宽支持的浮点类型，p_{max} radix b 数字可以四舍五入为浮点数具有 n 个十进制数字的数字，然后再次返回而不更改值，

$$\begin{cases} p_{\max} \log_{10} b & \text{if } b \text{ is a power of 10} \\ \lceil 1 + p_{\max} \log_{10} b \rceil & \text{otherwise} \end{cases}$$

DECIMAL_DIG 10

小数位数 q，使得任何具有 q 个小数位数的浮点数可以用 p 基数 b 位四舍五入为浮点数，然后再返回不改变 q 个十进制数字，

$$\begin{cases} p \log_{10} b & \text{if } b \text{ is a power of 10} \\ \lfloor (p-1) \log_{10} b \rfloor & \text{otherwise} \end{cases}$$

FLT_DIG 6
DBL_DIG 10
LDBL_DIG 10

最小负整数，使得 FLT_RADIX 提高到比该幂小一是归一化浮点数，e_{min}

FLT_MIN_EXP
DBL_MIN_EXP
LDBL_MIN_EXP

最小负整数，使得 10 的该次幂在归一化浮点数 $\lceil \log_{10} b^{e_{\min}-1} \rceil$

FLT_MIN_10_EXP -37
DBL_MIN_10_EXP -37
LDBL_MIN_10_EXP -37

最大整数，使得 FLT_RADIX 提高到比该幂小一是可表示的有限浮点数,e_{max}

FLT_MAX_EXP
DBL_MAX_EXP
LDBL_MAX_EXP

最大整数，使得 10 的幂在可表示的范围内有限浮点数, $\lfloor \log_{10}((1 - b^{-p}) b^{e_{\max}}) \rfloor$

FLT_MAX_10_EXP +37
DBL_MAX_10_EXP +37
LDBL_MAX_10_EXP +37

9. 以下列表中给出的值应替换为常量表达式大于或等于所示值的实现定义值：

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

FLT_MAX	1E+37
DBL_MAX	1E+37
LDBL_MAX	1E+37

10. 以下列表中给出的值应替换为常量表达式小于或等于所示值的实现定义（正）值：

1 与大于 1 的最小值之间的差值给定浮点类型,b^{1-p}

FLT_EPSILON	1E-5
DBL_EPSILON	1E-9
LDBL_EPSILON	1E-9

最小归一化正浮点数,$b^{e_{min}-1}$

FLT_MIN	1E-37
DBL_MIN	1E-37
LDBL_MIN	1E-37

推荐做法

11. 使用 DECIMAL_DIG 数字从（至少）双精度数转换为十进制数并返回应该是恒等函数。

12. 示例 1 以下描述了满足最低要求的人工浮点表示本国际标准的要求，以及 <float.h> 标头中的适当值类型漂浮：

$$x = s16^e \sum_{k=1}^{6} f_k 16^{-k}, \quad -31 \le e \le +32$$

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

13. 示例 2 以下描述了也满足以下要求的浮点表示 IEC 60559,20 中的单精度和双精度归一化数）以及 a 中的适当值<float.h> 标头用于 float 和 double 类型：

$$x_f = s2^e \sum_{k=1}^{24} f_k 2^{-k}, \quad -125 \le e \le +128$$

$$x_d = s2^e \sum_{k=1}^{53} f_k 2^{-k}, \quad -1021 \le e \le +1024$$

```
FLT_RADIX                       2
DECIMAL_DIG                    17
FLT_MANT_DIG                   24
FLT_EPSILON         1.19209290E-07F // decimal constant
FLT_EPSILON               0X1P-23F // hex constant
FLT_DIG                         6
FLT_MIN_EXP                  -125
FLT_MIN             1.17549435E-38F // decimal constant
FLT_MIN                   0X1P-126F // hex constant

FLT_MIN_10_EXP                 -37
FLT_MAX_EXP                   +128
FLT_MAX             3.40282347E+38F // decimal constant
FLT_MAX             0X1.fffffeP127F // hex constant
FLT_MAX_10_EXP                 +38
DBL_MANT_DIG                   53
DBL_EPSILON 2.2204460492503131E-16 // decimal constant
DBL_EPSILON                0X1P-52 // hex constant
DBL_DIG                        15
DBL_MIN_EXP                 -1021
DBL_MIN    2.2250738585072014E-308 // decimal constant
DBL_MIN                  0X1P-1022 // hex constant
DBL_MIN_10_EXP                -307
DBL_MAX_EXP                  +1024
DBL_MAX    1.7976931348623157E+308 // decimal constant
DBL_MAX     0X1.fffffffffffffP1023 // hex constant
DBL_MAX_10_EXP                +308
```

如果支持比双精度宽的类型，则 DECIMAL_DIG 将大于 17。对于例如，如果最宽的类型使用最小宽度的 IEC 60559 双扩展格式（64 位精度），那么 DECIMAL_DIG 将为 21。

前向引用：条件包含 (6.10.1)、复数算术<complex.h> (7.3), 浮点环境 <fenv.h> (7.6), 数学<math.h> (7.12).

6. 语言

6.1. 符号

1 在本节使用的句法符号中，句法类别（非终结符）是用斜体表示，字面量和字符集成员（终端）用粗体表示类型。非终结符后面的冒号 (:) 介绍了它的定义。选择定义在单独的行中列出，除非以单词“one of”开头。一个可选符号由下标“opt”表示，因此

$$\{\ expression_{opt}\ \}$$

表示用大括号括起来的可选表达式.

2 当正文中提到句法类别时，它们不是斜体，而是单词由空格而不是连字符分隔。

3 语言句法摘要见附件 A

6.2. 概念

6.2.1. 标识符的范围

1 一个标识符可以表示一个对象；一个函数；结构、联合的标签或成员，或枚举；类型定义名称；标签名称；宏名；或宏参数。这相同的标识符可以在程序的不同点表示不同的实体。成员枚举的值称为枚举常数。宏名称和宏这里没有进一步考虑参数，因为在语义阶段之前

程序翻译源文件中出现的任何宏名称都被替换为预处理构成其宏定义的标记序列.

2 对于标识符指定的每个不同实体，标识符都是可见的（即，可以是使用）仅在称为其范围的程序文本区域内。指定的不同实体相同的标识符要么具有不同的范围，要么位于不同的名称空间中。那里有四种作用域：函数、文件、块和函数原型。 （一个函数原型是声明其参数类型的函数的声明。）

3 标签名称是唯一一种具有函数作用域的标识符。它可以使用（在一个 goto 语句）出现在函数中的任何位置，并且被隐式声明通过它的句法外观（后跟一个 : 和一个语句）。

4 每个其他标识符的范围由其声明的位置决定（在声明符或类型说明符）。如果声明标识符的声明符或类型说明符出现在任何块或参数列表之外，标识符具有文件范围，其中在翻译单元的末尾终止。如果声明符或类型说明符声明标识符出现在块内或参数声明列表中一个函数定义，标识符有块作用域，它终止于关联块。如果出现声明标识符的声明符或类型说明符在函数原型中的参数声明列表中（不是函数的一部分定义），标识符具有函数原型范围，它终止于函数声明器。如果一个标识符指定两个不同的同名实体空间，范围可能重叠。如果是这样，一个实体的范围（内部范围）将是另一个实体范围的严格子集（外部范围）。在内部范围内，标识符指定在内部范围内声明的实体；在外部声明的实体范围在内部范围内隐藏（不可见）。

5 除非另有明确说明，否则本国际标准使用该术语“标识符”指的是某个实体（与句法结构相反），它指的是相关名称空间中的实体，其声明在标识符处可见发生.

6 当且仅当它们的范围终止于相同时，两个标识符具有相同的范围观点。

7 结构体、联合体和枚举标记的范围紧随其出现之后开始声明标签的类型说明符中的标签。每个枚举常量的范围在其定义的枚举数出现在枚举数列表中之后开始。任何其他标识符的范围在其声明符完成后开始。

前向引用：声明（6.7）、函数调用（6.5.2.2）、函数定义(6.9.1)、标识符 (6.4.2)、标识符的名称空间 (6.2.3)、宏替换 (6.10.3)、源文件包含 (6.10.2)，语句 (6.8)。

6.2.2. 标识符的链接

1 在不同作用域或同一作用域中多次声明的标识符可以是通过称为链接的过程来引用相同的对象或函数。有三种联动方式：外联、内联和无联动。

2 在构成整个程序的一组翻译单元和库中，每个具有外部链接的特定标识符的声明表示相同的对象或功能。在一个翻译单元内，每个标识符的声明都带有内部链接表示相同的对象或功能。每个标识符的声明都没有链接表示一个唯一的实体.

3 如果对象或函数的文件范围标识符的声明包含 storageclass 说明符 static，则该标识符具有内部链接.

4 对于在一个范围内使用存储类说明符 extern 声明的标识符，该标识符的先前声明是可见的，如果先前声明指定内部或外部链接，后面声明时标识符的链接与在先前声明中指定的链接。如果没有可见的先前声明，或者如果先前的声明声明未指定链接，则标识符具有外部链接。

5 如果函数的标识符声明没有存储类说明符，则其链接完全确定就好像它是使用存储类说明符 extern 声明的。如果对象标识符的声明具有文件范围且没有存储类说明符，它的联系是外部的。

6 以下标识符没有链接：声明为除一个对象或一个函数；声明为函数参数的标识符；块作用域没有存储类说明符 extern 声明的对象的标识符.

7 如果在一个翻译单元内，相同的标识符同时出现在内部和外部链接，行为未定义.

前向引用：声明（6.7），表达式（6.5），外部定义（6.9），声明 (6.8)

6.2.3. 标识符的命名空间

1 如果一个特定标识符的多个声明在一个翻译单元，句法上下文消除了引用不同实体的用法的歧义。因此，各种类别的标识符都有单独的名称空间，如下所示：

标签名称（通过标签声明和使用的语法消除歧义）；

结构体、联合体和枚举的标签（遵循 any24 消除歧义）关键字 struct、union 或 enum）；、

结构或工会的成员；每个结构或联合都有一个单独的名称其成员的空间（通过用于访问的表达式的类型来消除歧义会员通过 .或 -> 运算符）；

所有其他标识符，称为普通标识符（在普通声明符中声明或作为枚举常量）。

前向引用：枚举说明符 (6.7.2.2)、标记语句 (6.8.1)、结构和联合说明符 (6.7.2.1)、结构和联合成员 (6.5.2.3)、标签(6.7.2.3)、goto 语句 (6.8.6.1)。

6.2.4. 对象的存储期限

1 对象具有决定其生命周期的存储持续时间。共有三个存储持续时间：静态、自动和分配。分配的存储在 7.20.3 中描述.

2 对象的生命周期是程序执行的一部分，在此期间存储是保证为它保留。一个对象存在，有一个常量地址，[25)] 并保留它在其整个生命周期中的最后存储值。[26)] 如果一个对象在其外部被引用生命周期，行为未定义。当指针的值变得不确定时它指向的对象到达其生命周期的尽头.

3 一个对象，其标识符是用外部或内部链接声明的，或者用存储类说明符 static 具有静态存储持续时间。它的生命周期是整个程序的执行及其存储的值仅在程序之前初始化一次启动.

4 一个对象，其标识符被声明为没有链接且没有存储类说明符 static 具有自动存储持续时间。

5 对于这样一个没有可变长度数组类型的对象，它的生命周期会延长从进入与其关联的块直到该块的执行结束反正。 （进入封闭块或调用函数会暂停，但不会结束，

执行当前块。）如果块是递归进入的，则该块的新实例每次都会创建对象。对象的初始值是不确定的。如果为对象指定初始化，每次声明时执行在执行块时达到；否则，每个值都变得不确定达到声明的时间。

6 对于这样一个具有可变长度数组类型的对象，它的生命周期从对象的声明直到程序执行离开了范围 declaration.27) 如果递归输入范围，则创建对象的新实例每一次。对象的初始值是不确定的。

前向引用：语句 (6.8)、函数调用 (6.5.2.2)、声明符 (6.7.5)、数组声明符（6.7.5.2），初始化（6.7.8）。

6.2.5. 类型

1 存储在对象中或由函数返回的值的含义由用于访问它的表达式的类型决定。(声明为对象的标识符是最简单的此类表达式;类型在标识符的声明中指定。)类型分为对象类型(完全描述对象的类型)、函数类型(描述函数的类型)和不完整类型(描述对象但缺乏确定其大小所需的信息的类型)。

2 声明为_Bool 类型的对象足够大，可以存储值 0 和 1

3 声明为 char 类型的对象足够大,可以存储基本执行字符集的任何成员。如果基本执行字符集的成员存储在 char 对象中，则其值保证为正。如果在 char 对象中存储任何其他字符，结果值是实现定义的，但应在该类型可以表示的值的范围内。

4 有五种标准的有符号整型，分别是 signed char、short int、int、long int 和 long long int。(如 6.7.2 所述，这些类型和其他类型可以通过几种其他方式指定。)也可能存在实现定义的扩展有符号整数类型。标准和扩展有符号整数类型统称为有符号整数类型。[29)]

28）实现定义的关键字应具有为 7.1.3 中所述的任何用途保留的标识符形式。

29）因此，本标准中关于有符号整数类型的任何语句也适用于扩展的有符号整数类型。

5 声明为 signed 类型 char 的对象占用的存储空间与“普通”char 对象相同。“普通”int 对象具有执行环境架构建议的自然大小(大到足以包含头文件<limits.h>中定义的 INT_MIN 到 INT_MAX 范围内的任何值)。

6 对于每一种有符号整数类型，都有一个对应的(但不同的)无符号整数类型(用关键字 unsigned 指定)，它使用相同的存储量(包括符号信息)并具有相同的对齐要求。类型_Bool 和与标准有符号整型对应的无符号整型是标准无符号整型。与扩展无符号整型相对应的无符号整型是扩展无符号整型。标准和扩展无符号整数类型统称为无符号整数类型。30)

30) 因此，本标准中关于无符号整数类型的任何语句也适用于扩展无符号整数类型。

7 标准有符号整数类型和标准无符号整数类型统称为标准整数类型，扩展有符号整数类型和扩展无符号整数类型统称为扩展整数类型。

8 对于任何两个具有相同符号但整数转换等级不同的整数类型(参见 6.3.1.1)，整数转换等级较小的类型的值的范围是另一类型值的子范围。

9 有符号整型的非负值范围是对应的无符号整型的子范围，并且每种类型中相同值的表示方式是相同的。31）涉及无符号操作数的计算永远不会溢出，因为无法由结果无符号整型表示的结果会对结果类型所能表示的最大值取 1 的数取模。

31)相同的表示和对齐要求意味着函数的参数、函数的返回值和联合成员的互换性。

10 真正的浮动类型有三种，分别是 float、double 和 long double。32) float 类型的值集是 double 类型值集的子集;double 类型的值集是长 double 类型值集的子集。

32)参见“未来的语言方向”(6.11.1)。

11 复杂类型有三种，分别为 float _Complex、double _Complex 和 long double _Complex。33)实浮动类型与复杂类型统称为浮动类型。

33)虚类型的规范见翔实的附件 G。

12 每个浮点类型都有一个对应的实类型，实类型总是一个实浮点类型。对于真正的浮点类型，它是相同的类型。对于复杂类型，它是通过从类型名称中删除关键字_Complex 而给出的类型。

13 每个复杂类型具有与包含两个对应实类型元素的数组类型相同的表示和对齐要求;第一个元素等于复数的实部，第二个元素等于虚部。

14char 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。即使实现定义了两个或两个以上的基本类型以具有相同的表示形式，它们仍然是不同的类型。34）

34) 实现可以定义新的关键字，提供指定基本(或任何其他)类型的替代方法;这并不违反所有基本类型都不同的要求。实现定义的关键字应具有为 7.1.3 中所述的任何用途保留的标识符形式。

15 char、signed char 和 unsigned char 这三种类型统称为字符类型。实现应该定义 char 与 signed char 或 unsigned char 具有相同的范围、表示和行为。35)

35)在<limits.h>中定义的 CHAR_MIN 将有一个值 0 或 SCHAR_MIN，这可以用来区分这两个选项。无论如何选择，char 类型都是与其他两种类型分开的类型，并且两者都不兼容。

16 枚举由一组命名整型常量值组成。每个不同的枚举构成不同的枚举类型。

17char 类型、有符号和无符号整数类型以及枚举类型统称为整数类型。整数和实浮点类型统称为实类型。

18 整数类型和浮点类型统称为算术类型。每个算术类型属于一个类型域:实类型域由实类型组成，复杂类型域由复杂类型组成。

19void 类型包括一组空值;它是无法完成的不完整类型。

20 从对象、函数和不完全类型可以构造任意数量的派生类型，如下所示:

—数组类型描述了一组连续分配的非空对象，该对象具有特定的成员对象类型，称为元素类型。36)数组类型的特征是元素类型和数组中元素的数量。一个数组类型被称为从它的元素类型派生而来，如果它的元素类型是 T，这个数组类型有时被称为“T 的数组”。从元素类型构造数组类型称为“数组类型派生”。

36)由于对象类型不包括不完整类型，不完整类型的数组不能被构造

—结构类型描述按顺序分配的非空成员对象集(在某些情况下，还描述不完整的数组)，每个成员对象都有一个可选的指定名称和可能不同的类型。

--联合类型描述一个重叠的非空成员对象集，每个成员对象都有一个可选的指定名称和可能不同的类型。

--函数类型描述具有指定返回类型的函数。函数类型的特征是它的返回类型及其参数的数量和类型。一个函数类型被称为派生自它的返回类型，如果它的返回类型是 T，函数类型有时被称为“函数返回 T”。从返回类型构造函数类型称为“函数类型派生”。

—指针类型可以派生自函数类型、对象类型或称为引用类型的不完整类型。指针类型描述了一个对象，其值提供了对所引用类型实体的引用。从引用类型 T 派生的指针类型有时被称为“指向 T 的指针”。从引用类型构造指针类型称为“指针类型派生”。

可以递归地应用这些构造派生类型的方法

21 算术类型和指针类型统称为标量类型。数组和结构类型统称为聚合类型。37)

22 大小未知的数组类型是不完整类型。对于该类型的标识符，通过在稍后的声明中(使用内部或外部链接)指定大小来完成。未知内容的结构或联合类型(如 6.7.2.3 所述)是不完整类型。对于该类型的所有声明，通过稍后在同一作用域中声明相同的结构或 union 标记及其定义内容，就完成了此操作。

23 数组、函数和指针类型统称为派生声明符类型。从类型 T 派生的声明符类型是通过应用数组类型、函数类型或指向 T 的指针类型派生来构造从 T 派生的声明符类型。

24 类型的特征是它的类型类别，它要么是派生类型的最外层派生类型(如上面的派生类型构造中所述)，要么是不包含派生类型的类型本身。

25 以上提到的任何类型都是非限定类型。每个非限定类型都有其类型的几个限定版本，对应于一个、两个或所

有三个 const、volatile 和 restrict 限定符的组合。类型的限定版本或非限定版本是属于同一类型类别的不同类型，具有相同的表示和对齐要求。派生类型不受派生类型的限定符(如果有的话)限定。

26 指向 void 的指针应具有与指向字符类型的指针相同的表示和对齐要求。类似地，指向兼容类型的合格或不合格版本的指针也应具有相同的表示和对齐要求。所有指向结构类型的指针应该具有相同的表示形式和对齐要求。所有指向联合类型的指针应该具有相同的表示形式和对齐要求。指向其他类型的指针不需要具有相同的表示形式或对齐要求。

27 例 1 指定为“float *”的类型具有“指针指向 float”的类型。它的类型类别是指针，而不是浮点类型。此类型的 const 限定版本被指定为“float * const”，而指定为“const float *”的类型不是限定类型——它的类型是“指向 const 限定 float 的指针”，是指向限定类型的指针。

28 例 2 指定为“struct tag (*[5])(float)”的类型为“指向返回 struct tag 的函数的指针数组”。数组长度为 5，函数只有一个 float 类型的参数。它的类型类别是数组。

前向引用:兼容类型和复合类型(6.2.7)，声明(6.7)。

6.2.6. 表示的类型

6.2.6.1. 常规

1 所有类型的表示都是不指定的，除非在本子条款中有规定。

2 除位字段外，对象由一个或多个字节的连续序列组成，其数字、顺序和编码都是明确指定的或由实现定义的。

3 存储在无符号位域中的值和 unsigned char 类型的对象应使用纯二进制表示。40）

40)一种用二进制数字 0 和 1 表示整数的位置表示法，其中由连续的位表示的值是可加的，从 1 开始，并乘以 2 的连续整数幂(最高位除外)。(改编自《美国国家信息处理系统词典》)一个字节包含 CHAR_BIT 位，unsigned char 类型的值从 0 到 2CHAR_BIT−1。

4 存储在任何其他类型的非位域对象中的值由 n × CHAR_BIT 位组成，其中 n 为该类型对象的大小，以字节为单位。该值可以被复制到 unsigned char [n]类型的对象中(例如，通过 memcpy);结果字节集称为值的对象表示。存储在位域中的值由 m 位组成，其中 m 为位域指定的大小。对象表示是位域包含在保存它的可寻址存储单元中的 m 位的集合。具有相同对象表示的 Tw o 值(非 nan)比较相等，但比较相等的值可能有不同的对象表示。

5 某些对象表示不需要表示对象类型的值。如果一个对象的存储值具有这样的表示，并且由没有字符类型的左值表达式读取，则该行为是未定义的。如果这样的表示是由一个没有字符类型的左值表达式修改对象的全部或任何部分的副作用产生的，则该行为是未定义的。41）这样的表示称为陷阱表示。

41)因此，一个自动变量可以被初始化为一个陷阱表示，而不会引起未定义的行为，但是变量的值不能被使用，直到一个适当的值被存储在其中。

6 当一个值存储在一个结构体或联合类型的对象中，包括成员对象中，该对象表示中与任何填充字节对应的字

节将取未指定的值。填充字节的值不应影响这样一个对象的值是否为 trap 表示。与位域成员在同一个字节中的结构或联合对象的那些位，但不是该成员的一部分，同样不应影响这样一个对象的值是否为 trap 表示。

7 当值存储在 union 类型对象的成员中时，该对象表示中不对应该成员而对应其他成员的字节将取未指定的值，但 union 对象的值不应因此成为 trap 表示。

8 当一个运算符应用于一个具有多个对象表示的值时，使用哪个对象表示不影响结果的值。当一个值使用一个具有多个对象表示的类型存储在一个对象中时，使用哪个表示是不确定的，但不应产生陷阱表示。

前向引用:声明(6.7)、表达式(6.5)、左值、数组和函数指示器(6.3.2.1)。

6.2.6.2. 整型类型

1 对于 unsigned char 以外的 unsigned integer 类型，对象表示的位应该分为两组:值位和填充位(填充位不需要存在)。如果有 N 个值位，每个位应该代表 1 到 2N - 1 之间 2 的不同幂，因此该类型的对象应该能够使用纯二进制表示从 0 到 2N - 1 的值;这就是所谓的值表示。未指定任何填充位的值。44)

2 对于有符号整数类型，对象表示的位分为三组:值位、填充位和符号位。不需要任何填充位;应该有一个标志位。值位的每一位在对应的无符号类型的对象表示中应具有相同的值(如果有符号类型中有 M 个值位，无符号类型中有 N 个，则 M≤N)，如果有符号位为零，则不应影响得到的值。如果符号位为 1，则可以通过以下方式之一修改该值:

--符号位为 0 的对应值为负(符号和幅度);

--符号位的值为−(2N)(2 的补码);

—符号位的值为−(2N−1)(1 的补码)。

其中哪一个是由实现定义的，就像符号位 1 和所有值位 0(对于前两个)的值，或者符号位和所有值位 1(对于一个人的补码)的值是陷阱表示还是正常值一样。在符号、大小和一的补语的情况下，如果这种表示是一个正常的值，它被称为负零。

3 如果实现支持负零，则只能通过以下方式产生负零:

- &， |， ^， ~， <<和>>操作符，带参数产生这样的值;

- +， -， *， /和%操作符，其中一个参数为负零，结果为零;

-基于上述情况的复合赋值操作符。

这些情况到底产生的是负零还是正常的零，

以及在存储到一个对象中时，负零是否会变成正常的零。

4 如果实现不支持负 0，那么&、|、^、~、<<和>>操作符带参数产生负 0 的行为是未定义的。

5 当符号位为 0 时，有符号整数类型的有效(非 trap)对象表示是对应的无符号类型的有效对象表示，并且应该表示相同的值。45）

45) 填充位的某些组合可能产生 trap 表示，例如，如果一个填充位是奇偶校验位。无论如何，除了作为异常条件(如溢出)的一部分之外，对有效值的任何算术操作都不能生成陷阱表示。所有其他填充位的组合都是由值位指定的值的替代对象表示。

6 整数类型的精度是它用来表示值的位数，不包括任何符号和填充位。整数类型的宽度相同，但包含任何符号位;因此，对于无符号整数类型，两个值是相同的，而对于有符号整数类型，宽度比精度大 1。

6.2.7. 兼容类型和复合类型

1 如果 Tw 的类型相同，那么它们的类型是兼容的。确定两种类型是否兼容的附加规则在 6.7.2 中描述了类型说明符，6.7.3 中描述了类型限定符，6.7.5 中描述了声明符。46）此外，如果在单独的翻译单元中声明的两个结构类型、联合类型或枚举类型的标签和成员满足以下要求，则它们是兼容的:如果一个用标记声明，另一个应用相同的标记声明。如果两者都是完整类型，则适用以下附加要求:它们的成员之间应该有一对一的对应关系，这样每一对对应的成员都用兼容的类型声明，而且如果对应的成员中的一个用名称声明，另一个成员也用相同的名称声明。对于两个结构，相应的构件应以相同的顺序声明。对于两个结构或联合体，对应的位域应该有相同的宽度。对于两个枚举，相应的成员应具有相同的值。

46) Tw o 类型不需要完全相同才能兼容。

2 所有引用同一对象或函数的声明应具有兼容的类型;否则，该行为是未定义的。

3 复合类型可以由兼容的两种类型构成;该类型与两种类型都兼容，且满足以下条件:

-如果一种类型是已知常量大小的数组，复合类型是该大小的数组;否则，如果其中一种类型是变长数组，复合类型就是该类型。

—如果只有一种类型是带有参数类型列表的函数类型(函数原型)，则复合类型是带有参数类型列表的函数原型。

—如果两个类型都是带有参数类型列表的函数类型，则复合参数类型列表中每个参数的类型都是对应参数的复合类型。

这些规则递归地应用于派生这两种类型的类型。

4 对于一个具有内部或外部链接的标识符，在该标识符的之前的声明是可见的作用域中，47）如果之前的声明指定了内部或外部链接，在之后的声明中标识符的类型将成为复合类型。

47)正如 6.2.1 中规定的那样，后面的声明可能会隐藏前面的声明

5 例 给定以下两个文件作用域声明:

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

该函数的结果复合类型为:

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

6.3. 转换

1 几个操作符自动地将操作数值从一种类型转换为另一种类型。这个子句指定这种隐式转换所需的结果，以及强制转换操作(显式转换)所产生的结果。6.3.1.8 中的列表总结了大多数普通操作符执行的转换;根据需要，6.5 中对每个操作符的讨论对其进行了补充。

2 将操作数值转换为兼容类型不会改变其值或表示形式。

前向引用:强制转换操作符(6.5.4)。

6.3.1. 算术操作数

6.3.1.1. 布尔型，字符型，整型

1 每个整数类型都有一个整数转换等级，定义如下:

-任何两个有符号整数类型都不能有相同的秩，即使它们有相同的表示。

-有符号整数类型的秩应大于精度较低的任何有符号整数类型的秩。

- long long int 的 rank 应该大于 long int 的 rank, long int 应该大于 int 的 rank, int 应该大于 short int 的 rank, short int 应该大于 signed char 的 rank。

-任何无符号整数类型的秩应等于相应的有符号整数类型的秩(如果有的话)。

-任何标准整数类型的等级均须大于任何相同宽度的扩展整数类型的等级。

- char 的 rank 应该等于 signed char 和 unsigned char 的 rank。

- _Bool 的 rank 值应该小于所有其他标准整数类型的 rank 值。

-任何枚举类型的等级须等于相容整数类型的等级(见 6.7.2.2)。

—任何扩展有符号整数类型相对于具有相同精度的另一个扩展有符号整数类型的秩是实现定义的，但仍然受确定整数转换秩的其他规则的影响。

—对于所有整数类型 T1、T2、T3，如果 T1 大于 T2，且 T2 大于 T3，则 T1 大于 T3。

2 当表达式中可以使用 int 或 unsigned int 时，可以使用以下语句:

-一个整数类型的对象或表达式，其整数转换级别小于 int 和 unsigned int 的级别。

-类型为_Bool、int、signed int 或 unsigned int 的位域。

如果一个 int 型的值可以表示原类型的所有值，则将该值转换为 int 型;否则，它将被转换为 unsigned int。这些被称为整数提升。48)所有其他类型都不受整数提升的影响。

48) 整数提升仅适用于:作为常用算术转换的一部分，对某些参数表达式，对一元运算符+、-和~的操作数，以及对移位运算符的两个操作数，由它们各自的子句指定。

3 整数升级保留包括符号在内的值。如前所述，“普通”字符是否被视为有符号的是由实现定义的。

前向引用:枚举说明符(6.7.2.2)、结构和联合说明符(6.7.2.1)。

6.3.1.2. 布尔类型

1 当任何标量值转换为_Bool 时，如果该值比较等于 0，则结果为 0;否则，结果为 1。

6.3.1.3. 有符号和无符号整型

1 当整型转换为除_Bool 类型外的其他整型时，如果该值可以用新类型表示，则不改变。

2 否则，如果新类型是无符号的，值将通过比新类型所能表示的最大值多加或减去 1 来进行转换，直到该值在新类型的范围内为止。49)

49)规则描述的是数学值的运算，而不是给定类型表达式的值。

3 否则，新类型被签名，值不能在其中表示;结果要么是实现定义的，要么是引发实现定义的信号。

6.3.1.4. 实数浮点和整型

1 当一个有限的实数浮点类型的值被转换为一个非_Bool 的整数类型时，小数部分被丢弃(即，值被截断为零)。如果整数部分的值不能用整数类型表示，则该行为是未定义的。50）

50)整型转换为无符号类型时执行的余数运算，在实数浮点型转换为无符号类型时无需执行。因此，可移植的真实浮点值的范围是(−1,Utype_MAX+1)。

2 当整型转换为实数浮点型时，如果转换后的值可以用新类型精确表示，则不会改变。如果转换的值在可表示但不能精确表示的值范围内，则结果是最接近较高或较低的可表示值，并以实现定义的方式选择。如果被转换的值超出了可表示的值范围，则该行为是未定义的。

6.3.1.5. 实数浮点类型

1 当 float 被提升为 double 或 long double，或者 double 被提升为 long double 时，其值不变。

2 当一个 double 类型被降级为 float 类型时，一个长 double 类型被降级为 double 或 float 类型，或者一个表示精度和范围大于其语义类型要求的值(参见 6.3.1.8)被显式转换为其语义类型，如果被转换的值可以在新类型中精确表示，则其不变。如果转换的值在可表示但不能精确表示的值范围内，则结果是最接近较高或较低的可表示值，并以实现定义的方式选择。如果被转换的值超出了可表示的值范围，则该行为是未定义的。

6.3.1.6. 复数类型

1 当一个复数类型的值转换为另一个复数类型时，实部和虚部都遵循对应实类型的转换规则。

6.3.1.7. 实数和复数

1 当一个实数类型的值转换为复数类型时，复数结果值的实部由转换为相应实数类型的规则决定，复数结果值的虚部为正零或无符号零。

2 当将复数类型的值转换为实类型时，复数的虚部将被丢弃，实部的值将按照对应实类型的转换规则进行转换。

6.3.1.8. 常用算数转换

1 许多期待算术类型操作数的操作符会以类似的方式引起转换和产生结果类型。目的是确定操作数和结果的通用实类型。对于指定的操作数，在不更改类型域的情况下，将每个操作数转换为对应的实类型为普通实类型的类型。除非另有明确说明，否则公共实类型也是结果的对应实类型，如果操作数相同，则其类型域为操作数的类型域，否则为复杂类型。这种模式被称为通常的算术转换:

首先，如果任一操作数对应的实类型为长双精度浮点数，则另一个操作数在不改变类型域的情况下转换为其对应的实类型为长双精度浮点数的类型。

否则，如果任意一个操作数对应的实类型为 double，则另一个操作数将转换为其对应的实类型为 double 的类型，而不更改类型域。

否则，如果任一操作数对应的实类型为 float，则另一个操作数将转换为其对应的实类型为 float 的类型，而不更改类型域。51)

51)例如，double _Complex 和 float 的相加只需要将 float 操作数转换为 double(并产生 double _Complex 结果)。

否则，对两个操作数执行整数提升。然后对提升的操作数应用以下规则:

如果两个操作数具有相同的类型，则不需要进一步的转换。

否则，如果两个操作数都是有符号整数类型或都是无符号整数类型，则转换级别较小的操作数将转换为级别较大的操作数的类型。

否则，如果具有无符号整型的操作数的级别大于或等于另一个操作数的类型的级别，那么具有有符号整型的操作数将被转换为具有无符号整型的操作数的类型。

否则，如果带符号整型的操作数的类型可以表示无符号整型操作数类型的所有值，那么带无符号整型的操作数将被转换为带符号整型操作数的类型。

否则，两个操作数将转换为与带符号整数类型操作数的类型对应的无符号整数类型。

2 浮点操作数的值和浮点表达式的结果可以比类型要求的更大的精度和范围表示;类型并没有因此而改变。52）

52)强制转换和赋值操作符仍然需要执行 6.3.1.4 和 6.3.1.5 中描述的指定转换。

6.3.2. 再其他操作数

6.3.2.1. 左值、数组和函数指示器

1 左值是一个具有对象类型或非 void 类型的不完整类型的表达式;[53)]如果左值在计算时没有指定对象，则该行为是未定义的。当一个对象被称为具有特定类型时，该类型由用于的左值来指定指定的对象。可修改左值是没有数组类型的左值没有不完整的类型，没有 const 限定的类型，如果是结构或联合，没有任何成员(递归地包括的任何成员或元素)所有包含聚合或联合)的常量限定类型。

53）名称“左值”最初来自赋值表达式 E1 = E2，其中左操作数 E1 必须是一个(可修改的)左值。也许它代表的是对象“价值定位”。在本国际标准中所描述的所谓“右值”是什么作为“表达式的值”。左值的一个明显例子是对象的标识符。再举一个例子，如果 E 是一元表达式是指向对象的指针，*E 是左值，指定 E 所指向的对象。

2 除非它是 sizeof 操作符、一元&操作符、++操作符的操作数操作符的左操作数。或赋值运算符，没有数组类型的左值将转换为存储在指定数组中的值对象(并且不再是左值)。如果左值具有限定类型，则该值具有左值类型的非限定版本;否则，该值的类型为左值。如果左值的类型不完整且没有数组类型，则行为为未定义的。

3 除非它是 sizeof 操作符或一元&操作符的操作数，或者是 a 用于初始化数组的字符串字面值，具有类型“数组的类型”的表达式是转换为类型为“指向类型的指针”的表达式，该表达式指向的初始元素数组对象，不是左值。如果数组对象具有注册存储类，则行为是未定义的。

4 函数指示器是具有函数类型的表达式。除非它是 sizeof 操作符的操作数 [54)]或一元&操作符的操作数将“函数返回类型”转换为具有“指针指向”类型的表达式函数返回类型”。

54) 由于没有发生这种转换，sizeof 操作符的操作数仍然是一个函数指示符，违反了 6.5.3.4 中的约束。

向前引用:地址和间接操作符(6.5.3.2)、赋值操作符(6.5.16)，通用定义<stddef.h>(7.17)，初始化(6.7.8)，后缀递增和递减操作符(6.5.2.4)，前缀递增和递减操作符(6.5.3.1)、sizeof 操作符(6.5.3.4)、结构和 union 成员(6.5.2.3)。

6.3.2.2. void

1 表达式(void 类型的表达式)的(不存在的)值不可以以任何方式使用，隐式或显式转换(无效除外)不得使用用于这种表达式的如果将任何其他类型的表达式计算为 void 表达式时，其值或指示符将被丢弃。(void 表达式的值为副作用)。

6.3.2.3. 指针

1 指向 void 的指针可以转换为指向任何未完成或对象的指针类型。指向任何未完成类型或对象类型的指针都可以转换为指向 void 的指针回来;比较结果应与原始指针相等。

2 对于任何限定符 q，指向非限定类型的指针可以转换为指向的指针 q 型合格版型;存储在原始指针和转换后的指针中的值应比较平等的。

3 一个值为 0 的整数常量表达式，或者将这样的表达式转换为类型 Void *，被称为空指针常量。[55)]如果将空指针常量转换为指针类型，产生的指针(称为空指针)保证比较不相等指向任何对象或函数的指针。

55）宏观 NULL 在<stddef.h>(和其他头文件)中定义为空指针常量;见 7.17。

4 将一个空指针转换为另一个指针类型会得到该类型的空指针。任何两个空指针应该是相等的。

5 整数可以转换为任何指针类型。除非如前所述，否则结果是实现定义的，可能没有正确对齐，也可能没有指向被引用类型的实体，可能是 trap 的表示形式。

6 任何指针类型都可以转换为整数类型。除非如前所述，否则结果是由实现定义的。如果结果不能用整数类型表示，这种行为是未定义的。结果不需要在任何整数的值范围内类型。[56)]

56)用于将指针转换为整数或将整数转换为指针的映射函数旨在与执行环境的寻址结构保持一致。

7 指向对象或未完成类型的指针可以转换为指向其他类型的指针对象或不完整类型。如果结果指针没有正确对齐 [57)] 指向类型，行为是未定义的。否则，当再次转换回来时，比较结果应与原始指针相等。当将指向对象的指针转换为指向字符类型的指针时，结果将指向的地址最低的字节对象。结果的连续递增直到对象的大小，将产生指针到对象的剩余字节。

57)一般来说，“正确对齐”的概念是可传递的:如果指向类型 a 的指针正确对齐 a 指向类型 B 的指针，与指向类型 C 的指针正确对齐，然后指向类型 a 的指针是正确对齐指向类型 C 的指针。

8 指向一种类型函数的指针可以转换为指向另一种类型函数的指针输入并返回;比较结果应与原始指针相等。如果一个转换指针用于调用与指向类型不兼容的函数，这种行为是未定义的。前向引用:强制转换操作符(6.5.4)、相等操作符(6.5.9)、整型能够保存对象指针(7.18.1.4)，简单的赋值(6.5.16.1)。

6.4. 词汇元素

语法

1 标记:

关键字

标识符

常数

字符串文字

加标点者

预处理标记:

头名称

标识符

预处理数字

字符常数

字符串文字

加标点者

不能属于上述字符之一的非空白字符

约束

2 每个被转换为令牌的预处理令牌应具有 a 的词法形式关键字、标识符、常量、字符串字面值或标点符号。

语义

3 在翻译的第 7 和第 8 阶段，标记是语言中最小的词汇元素。标记的类别有:关键字、标识符、常量、字符串文字和标点符号。预处理标记是翻译中语言的最小词法元素阶段 3 到 6。预处理标记的类别有:头名称，标识符、预处理数字、字符常量、字符串字面值、标点符号和在词汇上与其他预处理不匹配的单个非空白字符 [58)]如果一个'或' '字符匹配最后一个类别，则行为为未定义的。预处理标记可以用空格分隔;这包括注释(后面会介绍)，或者空白字符(空格、水平制表符、换行符、垂直选项卡和表单提要)，或两者兼有。如 6.10 所述，在某些情况下在翻译阶段 4，空白(或没有空白)的作用不仅仅是预处理标记分离。空白可以出现在预处理标记中只能作为头名称的一部分或字符常量中的引号字符之间或字符串。

58）在翻译阶段 4 内部使用另一个类别——位置标记(见 6.10.3.3);它不能发生在源文件中。

4 如果输入流被解析为预处理标记，直到给定字符，则下一个预处理标记是可以构成预处理标记。这个规则有一个例外:头名称预处理令牌只能在#include 预处理指令中识别，而在#include 预处理指令中指令，可以是头文件名称或字符串字面量的字符序列是公认为前者。

5 实例 1 程序片段 1Ex 被解析为预处理数字令牌(一个不是有效的浮点或整数常量标记)，即使解析为一对预处理标记 1 和 Ex 可能会产生一个有效的表达式(例如，如果 Ex 是一个定义为+1 的宏)。同样,程序片段 1E1 被解析为一个预处理号(一个有效的浮动常量标记)，无论或不是 E 是一个宏名。

6 实例 2 程序片段 x+++++y 被解析为 x++++y，这违反了一个约束自增操作符，即使 parse x++ + ++ y 可能会产生正确的表达式。

前向引用:字符常量(6.4.4.4)，注释(6.4.9)，表达式(6.5)，浮动常量(6.4.4.2)，头名称(6.4.7)，宏替换(6.10.3)，后缀递增和递减操作符(6.5.2.4)，前缀递增和递减操作符(6.5.3.1)、预处理指令(6.10)、预处理数字(6.4.8)、字符串字面值(6.4.5)。

6.4.1. 关键词

语法

1 关键词：包括

关键字	数据类型	类型限定符	无符号数
跳出并结束循环	引用	返回	任意类型
程序的入口	单精度浮点型	短整形	易改变的
字符指针类型	循环语句的执行	有符号	循环函数

常量	跳转语句	单目操作符	布尔型变量
结束本次循环	条件从句	静态变量	复数
默认的	用于实现的关键字	结构体	声明虚数类型
执行	整形数类	切换	
双精度浮点型	长整形	类型定义	
其他的	寄存器存储类型	联合体	

语义

2 以上的标记(区分大小写)是保留的(在翻译阶段 7 和 8)用作关键字，不得以其他方式使用。

6.4.2. 标识符

6.4.2.1 一般

语法

1 标识符：

非数字标识符

标识非数字标识符

数字标识符

非数字标识符：

非数字字符

通用字符名

其他实现定义字符

非数字字符：包括

— 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：包括

0 1 2 3 4 5 6 7 8 9

语义

2 标识符是由非数字字符组成的序列(包括“_”、“_”、“。”)小写和大写的拉丁字母，以及其他字符)和数字，这表示一个或多个实体，如 6.2.1 所述。小写字母和大写字母是不同的。

3 标识符的最大长度没有特定的限制。标识符中的每个通用字符名称应指定其编码的字符 ISO/ iec10646 的首字母，属于附件 D。[59]所订明的范围之一字符不能是指定数字的通用字符名称。一个实现可以允许不属于基本源字符集的多字节字符吗出现在标识符;哪些字符及其对应的通用字符名字是由实现定义的。

59)在链接器不能接受扩展字符的系统上，通用字符的编码名称可用于形成有效的外部标识符。例如，一些其他未使用的字符或字符序列可用于在通用字符名称中编码\u。扩展字符可能产生较长的外部标识符。

4 当预处理标记在翻译阶段 7 转换为标记时，如果 a 预处理令牌可以转换为关键字或标识符，它被转换一个关键字。

实施限制

5 如 5.2.4.1 所讨论的，实现可以限制重要初始值的数量标识符中的字符;外部名称的限制(具有外部名称的标识符链接)可能比内部名称(宏名称或没有外部链接的标识符)。中有效字符的数量标识符是由实现定义的。

6 任何在有效字符中不同的标识符都是不同的标识符。如果两个标识符只在不重要的字符上有区别，行为未定义。前向引用:通用字符名称(6.4.3)，宏替换(6.10.3)。6.4.2.2 预定义的标识符

语义

1 标识符_ _func_应由译者隐式声明，就像在每个函数定义的左大括号后面紧跟着的是声明静态 const char _ _func_ _[] = "函数名";出现了，其中 function-name 是包含词法的函数的名称。[60)]

2 这个名字被编码，好像隐式声明已经写在源代码中字符集，然后翻译成翻译中指定的执行字符集阶段 5。

3 实例 显示代码片段:

```
#include <stdio.h>
void myfunc(void)
{
printf("%s\n", _ _func_ _);
/* ... */
}
```

每次函数被调用，它都会输出到标准输出流： myfunc

前向引用:函数定义(6.9.1)。

6.4.3. 通用字符名

语法

1 通用字符名：

\u 十六进制四元组

\U 十六进制四元组 十六进制四元组

十六进制四元组：

十六进制数字 十六进制数字

十六进制数字 十六进制数字

约束

2 通用字符名不能指定短标识符小于的字符 00A0 除了 0024($)、0040(@)或 0060(')，也不是 D800 到 D800 范围内的一个 DFFF inclusive.[61)]

61) 禁用字符是指基本字符集中的字符和保留的代码位置的控制字符、字符 DELETE 和 s 区(保留给 UTF-16)。

描述

3 通用字符名称可用于标识符、字符常量和字符串字面值，以指定不在基本字符集中的字符。

语义

4 通用字符名\ unnnnnnn 指定字符的 8 位数字短标识符(由 ISO/IEC 10646 指定)是 nnnnnnnn。[62)]同样，普遍性字符名\unnnn 指定四位数短标识符为 NNNN 的字符(其 8 位短标识符为 0000nnnn)。

62）字符的短标识符首先在 ISO/IEC 10646-1/AMD9:1997 中规定。

6.4.4. 常量

语法

1 常量：

整形常量

浮点型常量

枚举型常量

字符型常量

约束

2 常数的值应在其类型的可表示值的范围内。

语义

3 每个常数都有一个类型，由它的形式和值决定，后面会详细说明。

6.4.4.1. 整型常量

语法

1 整型常量：

十进制常量 任选的整型后缀

八进制常量 任选的整型后缀

十六进制常量 任选的整型后缀

十进制常量：

非零数字

十进制常量 数字

八进制常量：

0

八进制常量 八进制数字

十六进制常量：

十六进制前缀 十六进制数字

十六进制常量 十六进制数字

十六进制前缀：其中之一

0x 0X

非零数字：其中之一

1 2 3 4 5 6 7 8 9

八进制数字：其中之一

0 1 2 3 4 5 6 7

十六进制数字：其中之一

0 1 2 3 4 5 6 7 8 9

a b c d e f

A B C D E F

整数后缀：

无符号后缀 任选的长后缀

无符号后缀 长长后缀

长后缀 任选的无符号后缀

长长后缀 任选的无符号后缀

无符号后缀：包括

u U

长后缀：包括

l L

长长后缀：包括

ll LL

描述

2 整数常数以数字开头，但没有周期部分或指数部分。它可能有前缀指定它的基，后缀指定它的类型。

3 十进制常数以一个非零位数字开头，由一串小数组成位数。八进制常量由前缀 0(可选)后跟一个序列组成只能输入 0～7 的数字。十六进制常数由后面的前缀 0x 或 0x 组成由十进制数字和字母 a(或 a)到 f(或 f)组成的数值序列分别为 10～15。

语义

4 十进制常数的值以 10 为基数计算;以 8 为基数的八进制常数;以 16 为底的十六进制常数。第一个数字是最重要的。

5 整数常量的类型是其值可以在对应列表中的第一个是代表。

后缀	衰减常数	八进制或十六进制常数
名词	整型数 长整型 长整型	整型数 无符号整型数 长整型 无符号长整型 长整型 无符号长整型
u 或 U	无符号整型数 无符号长整型 无符号长整型	无符号整型数 无符号长整型 无符号长整型

l 或 L	长整型 长整型	长整型 无符号长整型 长整型 无符号长整型
u 或 U 和 l 或 L	无符号长整型 无符号长整型	无符号长整型 无符号长整型
ll 或 LL	长整型	长整型 无符号长整型
u 或 U 和 ll 或 LL	无符号长整型	无符号长整型

如果整数常量不能用其列表中的任何类型表示，则它可能具有扩展整数类型，如果扩展整数类型可以表示其值。如果所有的列表中常量的类型是有符号的，扩展整数类型应该有符号。如果该常量列表中的所有类型都是无符号的，扩展整数类型应为无符号。如果列表同时包含有符号和无符号类型，则为扩展整数类型可以有符号或无符号。

6.4.4.2. 浮点型常量

语法

1 浮点常量：

十进制浮点常量

十六进制浮点常量

十进制浮点常量：

小数常量 任选的指数部分 任选的浮点后缀

数字序列 指数部分 任选的浮点后缀

十六进制浮点常量：

十六进制前缀 十六进制小数常量

二进制指数部分 任选的浮点后缀

十六进制前缀 十六进制数字序列

二进制指数部分 任选的浮点后缀

小数常量：

数字序列 . 数字序列

数字序列 .

指数部分：

e 任选的正负号 数字序列

E 任选的正负号 数字序列

正负号：其中之一

+ -

数字序列：

数字

数字序列 数字

十六进制小数常量：

任选的十六进制数字序列 . 十六进制数字序列

十六进制数字序列 .

二进制指数部分：

p 正负号 可选 数字序列

P 正负号 可选 数字序列

十六进制数字序列：

十六进制数字

十六进制数字序列 十六进制数字

浮点后缀：包括

f l F L

描述

2 一个浮动常数有一个显著部分，其后可以是一个指数部分和 A 指定其类型的后缀。有效部分的组成部分可以包括一个数字表示整数部分的序列，后跟一个句点(.)，再后跟一个表示分数部分的数字序列。指数部分的分量是 ane, e, p，或 p 后跟一个指数，由一个可选的有符号的数字序列组成。要么是整数部分要么是分数部分必须存在; 为十进制浮动常数，周期或指数部分必须出现。

语义

3 有效部分被解释为一个(十进制或十六进制)有理数;的指数部分的数字序列被解释为十进制整数。为小数浮动常数，指数表示有效部分所对应的 10 的次方按比例缩小的。对于十六进制浮点常量，指数表示 2 的幂有意义的

部分将被缩放。为十进制浮点常量，也为 16 进制浮点常量，当 FLT_RADIX 不是 2 的幂时，结果是任意一个最接近的可表示值，或较大或较小的可立即表示值与最近的可表示值相邻，以实现定义的方式选择。对于 16 进制浮点常量，当 FLT_RADIX 是 2 的幂时，结果为正确的。

4 无后缀的浮动常数具有 double 类型。如果以字母 f 或 f 作为后缀，则有浮子类型。如果后缀为字母 l 或 l，则为打字长双引号。

5 浮动常量被转换为内部格式，就像在转换时一样。的浮点常量的转换在执行时不会引发异常条件或浮点异常。推荐的做法

6 如果是十六进制常量，实现应该产生一个诊断消息不能准确地用它的评估格式表示;然后实现应该进行程序的翻译。

7 浮点常量的平移-时间转换应该与执行时间匹配给定匹配的标准库函数(如 strtod)对字符串的转换适合两种转换的输入、相同的结果格式和默认的执行时间舍入。[63)]

63)库函数规范推荐比所需的更准确的转换浮动常量(参见 7.20.1.3)。

6.4.4.3. 枚举常数

1 枚举常数标识符

语义

2 声明为枚举常量的标识符具有类型 int。

提前引用：枚举说明符（6.7.2.2）

6.4.4.4. 字符常数

语法

字符常数：

c 字符序列

L c 字符序列

c 字符序列：

c-char

序列 c-char

c 字符： 除单引号‘、反斜杠\或新行字符外的源字符集的任何成员

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

通用字符名称

简单转义序列

\' \" \? \\

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

八进制转义序列

\八进制数字

\八进制数字八进制数字

\八进制数字八进制数字八进制数字

十六进制转义序列

\x 十六进制数字

十六进制转义序列 十六进制数字

说明

2、 整数字符常量是一个或多个多字节的字符的序列，如‘x’所示。宽字符常量是相同的，除了前缀为字母 l。除了后面详细说明的一些例外情况，序列的元素是源字符集的任何成员；它们以实现定义的方式映射到执行字符集的成员。

3 单引号“、双引号”、问号？、反斜杠和任意整数值可根据以下转义序列表进行表示：

单引号' \'

双引号 " \"

问号? \?

反斜线符号 \ \\

八进制字符 \八进制字符

十六进制字符 \x 十六进制字符

4 双引号”和问号？可以分别用自身或转义序列和？表示，但单引号和反斜杠应分别用转义序列和\\表示

5 八进位转义序列中反斜杠后面的八进位数字被认为是整数字符常数的单个字符或宽字符常数的单个宽字符构造的一部分。这样形成的八进制整数的数值指定所需字符或宽字符的值。

6 反斜杠和转义序列中的十六进制数字和字母 x 后面的十六进制数字被认为是整数字符常数的单个字符或宽字符常数的单个宽字符构造的一部分。这样形成的十六进制整数的数值指定所需字符或宽字符的值。

7 每个八进制或十六进制转义序列都是可以构成转义序列的最长的字符序列。

8 此外，不在基本字符集中的字符可以用通用字符名称表示，某些非图形字符可以由反斜杠表示，后面跟着一个小写字母：\a、\b、\f、\n、\r、\t 和\v

64）这些字符的语义在 5.2.2 中讨论。如果任何其他字符遵循反斜杠，则结果不是标记，需要进行诊断。请参见“未来语言方向”（6.11.4）。

约束条件

9 八进制或十六进制转义序列的值应在整数字符常数的类型无符号字符的可表示值范围内，或与宽字符常数的 wchar_t 对应的无符号类型的范围内。

语法

10 一个整数字符常量的类型为 int。包含映射到单字节执行字符的单个字符的整数字符常量的值是被解释为整数的映射字符表示的数值。一个包含多个字符(例如，“ab”)的整数字符常量，或包含一个字符或转义序列的值，是由实现定义的。如果整数字符常量包含单个字符或转义序列，则其值是将字符类型的值为单个字符或转义序列的对象转换为 int 类型时产生的值。

11 宽字符常量具有类型 wchar_t，即在标头中定义的整数类型。包含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与该多字节字符对应的宽字符，这由 mbtowc 函数定义，具有实现定义的当前区域环境。包含多个多字节字符，或包含在扩展执行字符集中未表示的多字节字符或转义序列的宽字符常量的值是由实现定义的。

12 示例 1 构造“\0”通常用于表示空字符

13 示例 2 考虑对整数使用 2 位的补数表示，对具有 char 类型的对象使用 8 位的实现。在类型字符与字符有符号字符的值范围相同的实现中，整数字符常量“\xFF”的值为−1；如果类型字符与无符号字符的值范围相同，则字符常量“\xFF”的值为+255。

14 示例 3 即使对具有字符类型的对象使用 8 位，构造‘\x123’也指定一个仅包含一个字符的整数字符常量，因为十六进制转义序列仅由非十六进制字符终止。要指定一个包含“\x12”和“3”的整数字符常量，可以使用构造“\0223”，因为八进制转义序列在三个八进制数字之后终止。（此两个字符的整数字符常量的值是由实现定义的。）

15 示例 4 即使对类型为 wchar_t 的对象使用 12 位或更多位，构造 L‘\1234’也指定由值 0123 和‘4’组合产生的实现定义值。

提前引用：通用定义（7.17），mbtowc 函数（7.20.7.2）

6.4.5. 字符串文字

语法

字符串序列：S-char 序列 L“s-char-序列”

s 字符序列 s-char s-char-sequence s-char

s-char: 除双引号、反斜杠或新行字符外的源字符集的任何成员

转义序列

说明

2、字符串文字是由 0 个或多个多字节字符用双引号括起来的序列，如“xyz”。宽字符串文字是相同的，除了前缀是字母 L。

3、同样的考虑因素适用于字符串文字或宽的字符串序列，就像整数字符常数或宽字符常数一样，除了单引号本身或转义序列表示，但双引号“应由转义序列表示”。

语义

4、在翻译阶段 6 中，由任何相邻字符和宽字符串文字标记的序列指定的多字节字符序列被连接成单个多字节字符序列。如果任何标记是宽字符串文字标记，则生成的多字节字符序列将视为宽字符串文字；否则，它将视为字符串文字。

5、在转换阶段 7 中，一个值为 0 的字节或代码被追加到每个多字节由字符串字面值或字面值产生的字符序列 然后使用 Sequence 初始化一个包含静态存储持续时间和长度的数组 justsequence，这是由 mbstowcs 函数用实现定义的 current 定义的 语言环境。包含多字节字符或转义序列的字符串字面量的值 执行字符集中没有表示的是实现定义的。 足以包含序列的。对于字符串字面值，数组元素具有，类型为 char，并使用多字节字符的单个字节初始化 序列;对于宽字符串字面值，数组元素的类型为 wchar_t，并且是 用与多字节字符对应的宽字符序列初始化。

6 如果这些数组的元素具有适当的值，则指定这些数组是否不同。如果程序试图修改这样的数组，则该行为将未定义。

7 这对相邻字符串文字“\x12”“3”生成一个单一字符串文字，其中包含两个值为“\x12”和“3”的字符，因为转义序列在相邻字符串文字连接之前被转换为执行字符集的单个成员。

提前引用：通用定义<stddef.h>（7.17），mbstowcs 函数（7.20.8.1）。

6.4.6. 标点符号

语法

标点符号：[ ] ( ) { } . ->

++ -- & * + - ~ !

/ % << >> < > <= >= == != ^ | && ||

? : ; ...

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

, # ##

<: :> <% %> %: %:%:

语义

2、标点符号是一种具有独立的句法和语义意义的符号。根据上下文，它可以指定要执行的操作（进而可以产生值或函数指示符，产生副作用，或其某种组合），在这种情况下，它被称为操作符（在某些上下文中也存在其他形式的操作符）。操作数是操作员所要操作的实体

3、在该语言的所有方面，这六个标记的行为

<: :> <% %> %: %:%:

分别与这六个标记相同，

[ ] { } # ##

除了拼写不同。

提前引用：表达式（6.5）、声明（6.7）、预处理指令（6.10）、语句（6.8）

6.4.7. 标题名称

语法

header-name:
< h-char-sequence >
" q-char-sequence "
h-char-sequence:
h-char
h-char-sequence h-char
h-char: 除新行字符和>之外的源字符集的任何成员
q-char-sequence:
q-char
q-char-sequence q-char
q-char:源字符集的任何成员，除了新行字符和“

语义

2、两种头名形式中的序列都以实现定义的方式映射到 6.10.2 中指定的头名或外部源文件名。

3、如果字符'、\、"、//或/*出现在<和>分隔符之间的序列中，则该行为未定义。类似地，如果字符'、\、//或/*出现在分隔符"之间的序列中，则行为未定义。头名称预处理标记仅在#include 预处理指令中可识别。

4、示例：以下字符序列：

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

形成以下序列的预处理标记（每个单独的预处理标记由左边的 a{和右边的 a}分隔）。

{0x3}{<}{1}{/}{a}{.}{h}{>}{1e2}

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

提前引用：源文件包含（6.10.2）。

6.4.8. 预处理编号

语法

1 预处理数字

digit
. digit
pp-number digit
pp-number identifiifier-nondigit
pp-number e sign
pp-number E sign
pp-number p sign
pp-number P sign
pp-number .

描述

2、预处理数字以一个数字开始，前面可以有一个句号（。）和后面可以是有效的标识符字符和字符序列 e+、E-、E+、E-、p+、p-、P+、或 P-。

3、预处理数字令牌在词法上包括所有浮动和整数常数令牌。

语义

4、预处理编号没有类型或值；在成功转换（作为转换阶段 7 的一部分）到浮动常数标记或整数常数标记之后，它将获取两者。

68)因此，类似于转义序列的字符序列会导致未定义的行为

6.4.9. 说明

1、除了在字符常量、字符串文字或注释中外，字符/*将引入注释。检查这类注释的内容只是为了识别多字节字符并找到终止它的字符*/。

2、除了在字符常量、字符串文字或注释中之外，字符//引入的注释包括到下一个新行字符的所有多字节字符。检查这类注释的内容只是为了识别多字节字符和查找终止的新行字符。

3、示例

"a//b" // four-character string literal

```
#include "//e" // undefifined behavior
// */ // comment, not syntax error
f = g/**//h; // equivalent to f = g / h;
//\
i(); // part of a two-line comment
/\
/ j(); // part of a two-line comment
#define glue(x,y) x##y
glue(/,/) k(); // syntax error, not comment
/*//*/ l(); // equivalent to l();
m = n//**/o
+ p; // equivalent to m = n + p;
```

69)因此，/*……*/评论不嵌套。

6.5. 表达式

1、表达式是一组操作符和操作数序列，它指定值的计算，或指定对象或函数，或产生副作用，或执行它们的组合。

2、在上一个序列点和下一个序列点之间，一个对象的存储值最多只能通过对一个表达式的计算来修改一次。此外，只能读取优先值，以确定要存储的值。

3、运算符和操作数的分组由语法表示。除非后来指定(对于函数调用()，&&，||，？：，和逗号运算符)，子表示式的计算顺序和副作用发生的顺序都未说明。

4、一些运算符（一元运算符~和二进制运算符>、&、^和|，统称为位运算符）需要具有具有整数类型的操作数。这些运算符返回的值依赖于整数的内部表示，并且对有符号类型具有实现定义和未定义的方面。

5、如果在计算表达式的过程中出现了异常条件（即，如果结果没有被数学定义或不在其类型的可表示值范围内），则该行为将未被定义。

6、访问其存储值的对象的有效类型是对象的声明类型，如果有的话。如果一个值通过具有非字符类型的左值存储到一个没有声明类型的对象中，那么该左值的类型成为该访问和后续访问的对象的有效类型

70)本段呈现未定义的语句表达式，例如

```
i = ++i + 1;
a[i++] = i;
```

同时允许

```
i = i + 1;
a[i] = i;
```

71) 语法指定表达式求值中运算符的优先级，与本子句的顺序相同，最高优先级优先。因此，例如，允许作为二进制+操作符（6.5.6）的操作数的表达式是在 6.5.1 到 6.5.6 中定义的表达式。例外情况是一元运算符（6.5.3）的转换运算式（6.5.4），以及包含在以下一对操作符之间的运算数：分组括号()（6.5.1）、下标括号[]（6.5.2.1）、函数调用括号()（6.5.2.2），以及条件运算符？:(6.5.15).在每个主要子句中，操作符具有相同的优先级。在每个子句中用其中讨论的表达式的语法表示。

72)已分配的对象没有已声明的类型。

存储值。如果使用 memcpy 或 memmove 将一个值复制到没有声明类型的对象中，或作为字符类型数组复制，则该访问和后续不修改值的修改对象的访问的有效类型是复制值的对象的有效类型。对于对没有声明类型的对象的所有其他访问，该对象的有效类型只是用于访问的左值的类型。

7、对象的存储值只能由具有以下类型之一的左值表达式访问：

——与对象的有效类型兼容的类型

——与对象的有效类型兼容的类型的限定版本

——与对象的有效类型对应的有符号或无符号类型、与对象的有效类型的限定版本对应的有符号或无符号类型

——包含上述类型（递归地包括子聚合的成员或包含的联合类型，或者一字符类型

8、浮点型表达式可以被压缩，也就是说，像它是原子操作来计算，从而省略源代码和表达式计算方法所隐含的舍入错误。提供了一种不允许收缩表达式的方法。否则，表达式是否收缩以及如何收缩是由实现定义的。

提前引用：FP_CONTRACT 实用（7.12.2），复制函数（7.21.2）。

73)本列表的目的是指定对象可能或可能不被别名的情况。

74)一个收缩表达式也可能省略浮点异常的提升。

75)此许可证专门用于允许实现利用结合多个 C 操作符的快速机器指令。由于收缩可能会破坏可预测性，甚至会降低包含表达式的准确性，因此它们的使用需要有明确的定义和清晰的记录。

6.5.1. 主要表达式

语法

1 主要表达式：

标识符

常量

字符串文字

（表达式）

语义

2、标识符是一个主表达式，只要它已被声明为指定一个对象（在这种情况下它是一个左值）或一个函数（在这种情况下它是一个函数指示符）76。

3、一个常量是一个主表达式。其类型取决于其形式和值，详见 6.4.4

4、一个字符串文字是一个主表达式。它是一个左值，其类型详见 6.4.5。

5、括号表达式是主要表达式。它的类型和值与未加括号的表达式相同。如果未加括号的表达式分别是左值、函数指示符或空表达式，则它为左值、函数指示符或空表达式。

6.5.2. 后缀操作符

语法

后缀表达式：

primary-expression
postfifix-expression [expression]
postfifix-expression (argument-expression-listopt)
postfifix-expression . identififier
postfifix-expression -> identififier
postfifix-expression ++
postfifix-expression --
(type-name) { initializer-list }
(type-name) { initializer-list , }

76)因此，未声明的标识符违反了语法

参数表达式列表：

赋值表达式

参数表达式列表、赋值表达式

6.5.2.1. 数组下标

约束

1、其中一个表达式具有类型“指向对象类型“”的指针”，另一个表达式具有整数类型，结果具有类型“类型”。

语义

2、后缀表达式和方括号中的表达式是数组对象中元素的下标名称。下标运算符[]的定义是，E1[E2]与（*（（E1）+（E2））相同。由于适用于二进制+运算符的转换规则，如果 E1 是一个数组对象（等价地，一个指向数组对象的初始元素的指针），而 E2 是一个整数，则 E1[E2]指定 E1 的第 E2 个元素（从零开始计数）。

3、连续下标运算符指定多维数组对象中的一个元素。如果 E 是一个维数为×的 n 维数组(n≥2)，×j×…×k，那么 E（用作左值以外）被转换为一个指向(n−1)维数组的指针 j×…×k。如果一元*运算符显式地应用于这个指针，或者隐式地作为下标的结果，结果是指向(n−1)维数组，如果用作左值，它本身会被转换为一个指针。由此得出，数组以行大写顺序存储（最后一个下标变化最快）。

4、示例：考虑由声明定义的数组对象

int x[3][5];

这里 x 是一个 3×5 的 int 数组；更准确地说，x 是由三个元素对象组成的数组，每个元素对象都是由五个 int 组成的数组。在表达式 x[i]中，它相当于(*((x)+(i)))，x 首先被转换为一个指向 5 个 int 的初始数组的指针。然后，i 根据 x 的类型进行调整，从概念上讲，x 需要将 i 乘以指针所指向的对象的大小，即一个由五个 int 对象组成的数组。结果被添加并应用间接来产生一个包含五个 int 的数组。当在表达式 x[i][j]中使用时，该数组依次转换为指向第一个整数的指针，因此 x[i][j]产生一个 int。

提前引用：附加运算符（6.5.6）、地址和间接运算符（6.5.3.2）、数组声明器（6.7.5.2）

6.5.2.2. 函数调用

约束条件

1. 表示被调用函数 77）的表达式必须具有指向返回 void 或返回数组类型以外的对象类型的函数的类型指针。

77）通常，这是将一个标识符转换为函数指示器的结果。

2. 如果表示被调用函数的表达式具有包含原型的类型，则参数的数量应与形参的数量一致。每个实参都应有一个类型，其值可以赋给一个具有对应形参类型的非限定版本的对象。

语义

3. 后跟圆括号()的后缀表达式（可能包含空的、用逗号分隔的表达式列表）是一个函数调用。后缀表达式表示被调用的函数。表达式列表指定函数的参数。

4. 参数可以是任何对象类型的表达式。在准备调用函数时，对参数求值，并给每个形参赋相应实参的值 78）。

78）函数可以更改其形参的值，但这些更改不能影响实参的值。另一方面，可以将指针传递给对象，函数也可以更改所指向对象的值。声明为数组或函数类型的形参将被调整为指针类型，如 6.9.1 所述。

5. 如果表示被调用函数的表达式具有指向返回对象类型的函数的类型指针，则函数调用表达式具有与该对象类型相同的类型，并具有 6.8.6.4 中指定的值。否则，函数调用的类型为 void。如果试图修改函数调用的结果或在下一个序列点之后访问它，则该行为是未定义的。

6. 如果表示被调用函数的表达式具有不包含原型的类型，则对每个参数执行整型提升，而具有 float 类型的参数被提升为 double 类型。这些被称为默认参数提升。如果参数的数量不等于参数的数量，则该行为是未定义的。如果函数定义为包含原型的类型，并且原型以省略号(，…)结尾，或者提升后的实参类型与形参类型不兼容，则该行为未定义。如果函数定义为不包含原型的类型，并且提升后的实参类型与提升后的形参类型不兼容，则该行为未定义，以下情况除外:

— 一种提升类型是有符号整型，另一种提升类型是相应的无符号整型，其值在两种类型中均可表示;

— 这两个类型都是指向字符类型或 void 的限定或限定版本的指针。

7. 如果表示被调用函数的表达式具有包含原型的类型，则参数将被隐式转换为相应形参的类型，就像通过赋值一样，每个形参的类型都是其声明类型的非限定版本。函数原型声明符中的省略号表示法导致实参类型转换在最后一个声明的形参之后停止。默认的参数提升是对尾随的参数执行的。

8. 不隐式执行其他转换;特别是，在不包含函数原型声明符的函数定义中，实参的数量和类型不会与形参的数量和类型进行比较。

9. 如果函数定义的类型与表示被调用函数的表达式所指向的(表达式的)类型不兼容，则该行为未定义。

10. 函数指示符、实际参数和实际参数中的子表达式的求值顺序没有指定，但是在实际调用之前有一个序列点。

11. 允许通过任何其他函数链直接或间接地进行递归函数调用。

12. 例子 在函数调用中(*pf[f1()]) (f2(), f3() + f4())

函数 f1、f2、f3 和 f4 可以以任何顺序调用。所有副作用必须在 pf[f1()]所指向的函数被调用之前完成。

提前引用:函数声明符(包括原型)(6.7.5.3)、函数定义(6.9.1)、返回语句(6.8.6.4)、简单赋值(6.5.16.1)。

6.5.2.3. 结构体和联合成员

约束条件

1. 操作符的第一个操作数应具有合格或不合格的结构体或联合类型，而第二个操作数应指定该类型的成员。

2. 操作符的第一个操作数应具有“指向合格或不合格结构的指针”或“指向合格或不合格联合的指针”类型，而第二个操作数应指定所指向类型的成员。

语义

3. 后跟运算符和标识符的后缀表达式指定结构体或联合对象的成员。该值是命名成员的值，如果第一个表达式是左值，则为左值。如果第一个表达式具有限定类型，则结果具有指定成员类型的限定版本。

4. 后跟运算符和标识符的后缀表达式指定结构体或联合对象的成员 79)。如果第一个表达式是指向限定类型的指针，则结果具有指定成员类型的限定版本。

79)如果&E 是一个有效的指针表达式(其中&是“地址”操作符，它生成指向其操作数的指针)，则表达式(&E)->MOS 与 E.MOS 相同。

5.一个特殊的为了简化联合的使用约定:如果一个联合包含多个结构体，它们共享一个共同的初始序列(见下文)，并且如果联合对象当前包含其中一个结构，则允许检查其中任何一个结构的共同初始部分，以确保联合的完整类型的声明是可见的。如果对应的成员对于一个或多个初始成员的序列具有兼容的类型(对于位域，具有相同的宽度)，则两个结构体将共享一个公共的初始序列。

6.例子 1：如果 f 是返回结构体或联合的函数，x 是该结构体或联合的成员，则 f().X 是一个有效的后缀表达式，但不是左值。

7.例子 2：

```
struct s { int i; const int ci; };
struct s s;
```

```
const struct s cs;
volatile struct s vs;
```

各种成员有以下类型:

```
s.i int
s.ci const int
cs.i const int
cs.ci const int
vs.i volatile int
vs.ci volatile const int
```

8.例子 3 以下是一个有效的片段:

```
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)
if (sin(u.nf.doublenode) == 0.0)
/* ... */
```

以下不是一个有效的片段(因为联合类型在函数 f 中是不可见的):

```
struct t1 { int m; };
struct t2 { int m; };
int f(struct t1 * p1, struct t2 * p2)
{
    if (p1->m < 0)
    p2->m = -p2->m;
    return p1->m;
}
```

```
int g()
{
    union {
        struct t1 s1;
        struct t2 s2;
    } u;
    /* ... */
    return f(&u.s1, &u.s2);
}
```

提前引用:地址和间接操作符(6.5.3.2)、结构体和联合说明符(6.7.2.1)。

6.5.2.4. 后缀自增和自减操作符

约束条件

后缀自增或自减操作符的操作数必须是合格的或不合格的实数或指针类型，并且必须是可修改的左值。

语义

后缀++操作符的结果是操作数的值。获取结果后，操作数的值将递增。(也就是说，将适当类型的值 1 添加到它。)有关约束、类型和转换以及操作对指针的影响的信息，请参阅加法操作符和复合赋值的讨论。更新操作数的存储值的副作用将发生在前一个序列点和下一个序列点之间。

后缀--操作符类似于后缀++操作符，不同的是操作数的值是递减的(也就是说，从操作数中减去相应类型的值 1)。

提前引用:加法操作符(6.5.6)、复合赋值(6.5.16.2)。

6.5.2.5. 复合文字

约束条件

1.类型名称必须指定对象类型或未知大小的数组，但不能指定可变长度的数组类型。

2.任何初始化式都不应试图为复合文字指定的整个未命名对象中不包含的对象提供值。

3.如果复合文字出现在函数体之外，则初始化器列表应由常量表达式组成。

语义

4.后缀表达式由圆括号括起来的类型名后跟大括号括起来的初始化器列表组成，它是一个复合文字。它提供了一个未命名的对象，其值由初始化器列表给出。

5.如果类型名指定了一个未知大小的数组，则大小由 6.7.8 中指定的初始化器列表确定，复合文字的类型是完整数组类型的类型。否则(当类型名指定对象类型时)，复合文字的类型为类型名指定的类型。在这两种情况下，结果都是左值 [80]。

80）注意，这与强制转换表达式不同。例如，强制转换指定只转换到标量类型或 void，强制转换表达式的结果不是左值。

6.复合文字的值是由初始化器列表初始化的未命名对象的值。如果复合文字出现在函数体之外，则该对象具有静态存储持续时间;否则，它具有与封闭块相关联的自动存储持续时间。

7. 6.7.8 中初始化器列表的所有语义规则和约束都适用于复合文字 [81]。

81） 例如，没有显式初始化器的子对象被初始化为零。

8.字符串文字和具有 const 限定类型的复合文字不需要指定不同的对象 [82]。

82） 这允许实现共享具有相同或重叠表示的字符串字面值和常量复合字面值的存储。

9.例子 1　文件作用域定义

```
int *p = (int []){2, 4};
```

初始化 p，使其指向两个 int 型数组的第一个元素，第一个 int 型的值为 2，第二个 int 型的值为 4。这个复合文字中的表达式必须是常量。未命名对象具有静态存储时间。

10.例子 2　相比之下，在下面的语句中

```
void f(void)
{
    int *p;
    /*...*/
    p = (int [2]){*p};
    /*...*/
}
```

P 被赋值为两个 int 型数组的第一个元素的地址，第一个元素的值是 P 之前指向的，第二个元素的值是 0。这个复合文字中的表达式不必是常量。未命名对象具有自动存储期限。

11.例子 3　带有指定的初始化器可以与复合文字组合使用。使用复合文字创建的结构对象可以传递给函数，而不需要依赖成员顺序:

```
drawline((struct point){.x=1, .y=1},
    (struct point){.x=3, .y=4});
Or, if drawline instead expected pointers to struct point:
    drawline(&(struct point){.x=1, .y=1},
                &(struct point){.x=3, .y=4});
```

12.例子 4　只读复合文字可以通过如下结构来指定:

```
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

13.例子 5　以下三种表达有不同的含义:

```
"/tmp/fileXXXXXX"
(char []){"/tmp/fileXXXXXX"}
(const char []){"/tmp/fileXXXXXX"}
```

第一种类型的存储时间总是静态的，并且类型数组为 char，但不需要修改;后两个函数在函数体中发生时具有自动存储持续时间，而前两个函数是可修改的。

14.例子 6　与字符串字面值一样，const 限定的复合文字可以放在只读内存中，甚至可以共享。例如,

```
(const char []){"abc"} == "abc"
```

如果字面值的存储是共享的，可能会产生 1。

15.例子 7　由于复合文字是未命名的，单个复合文字不能指定循环链接的对象。例如，没有办法编写一个自我引用的复合文字来代替下面的命名对象 endless_zeros 作为函数参数:

```
struct int_list { int car; struct int_list *cdr; };
struct int_list endless_zeros = {0, &endless_zeros};
eval(endless_zeros);
```

16.例子 8　每个复合文字只在给定范围内创建一个对象:

```
struct s { int i; };
int f (void)
{
    struct s *p = 0, *q;
    int j = 0;
    again:
    q = p, p = &((struct s){ j++ });
    if (j < 2) goto again;
    return p == q && q->i == 1;
}
```

函数 f()总是返回值 1。

注意，如果使用迭代语句而不是显式的 goto 语句和标签语句，那么未命名对象的生命周期将仅是循环体，并且在下一次进入 p 时将有一个不确定的值，这将导致未定义的行为。

提前引用：类型名称(6.7.6)，初始化(6.7.8)。

6.5.3. 一元操作符

语法

一元表达式：

后缀表达式

++ 一元表达式

-- 一元表达式

一元运算符 强制转换表达式

sizeof 一元表达式

Sizeof（类型名）

一元运算符：

& * + - ~ !

6.5.3.1. 前缀自增和自减操作符

约束条件

1.前缀自增或自减操作符的操作数必须是合格的或不合格的实数或指针类型，并且必须是可修改的左值。

语义

2.前缀++操作符的操作数的值是递增的。结果是操作数加 1 后的新值。表达式++E 等价于(E+=1)。有关约束、类型、副作用、转换以及操作对指针的影响的信息，请参阅加法操作符和复合赋值的讨论。

3.前缀——操作符类似于前缀++操作符，只是操作数的值是递减的。

提前引用：加性运算符(6.5.6)，复合赋值(6.5.16.2)。

6.5.3.2. 地址和间接操作符

约束条件

1.一元&操作符的操作数可以是一个函数指示符、[]或一元*操作符的结果，或者是一个左值，该左值指定的对象不是位域，也不是用寄存器存储类说明符声明的。

2.一元*操作符的操作数应具有指针类型。

语义

3.一元&操作符返回其操作数的地址。如果操作数具有类型“type”，则结果具有类型“指向类型的指针”。如果操作数是一元*操作符的结果，则该操作符和&操作符都不求值，结果就好像都省略了一样，只是操作符的约束仍然适用，且结果不是左值。类似地，如果操作数是[]操作符的结果，则&操作符和[]所暗示的一元*都不会被求值，结果就好像&操作符被删除了，而[]操作符被改为+操作符。否则，结果是指向其操作数指定的对象或函数的指针。

4.一元*操作符表示间接。如果操作数指向函数，则结果为函数指示符;如果它指向一个对象，结果是一个指定该对象的左值。如果操作数的类型为"指向类型的指针"，则结果为" type "类型。如果给指针赋了无效值，则一元*操作符的行为是未定义的。83）

提前引用：存储类说明符(6.7.1)、结构和联合说明符(6.7.2.1)。

6.5.3.3. 一元运算符

约束条件

1.一元+或-操作符的操作数应具有算术类型；~操作符的操作数应具有整数类型；！操作符的操作数应具有标量类型。

语义

2.一元+操作符的结果是其(提升的)操作数的值。对操作数执行整数提升，结果具有提升的类型。

3.一元-操作符的结果是其(提升的)操作数的负数。对操作数执行整数提升，结果具有提升的类型。

4.~操作符的结果是其(提升)操作数的位补(也就是说，当且仅当转换后的操作数中对应的位未设置时，才设置结果中的每一位)。对操作数执行整数提升，结果具有提升的类型。如果提升类型是无符号类型，表达式~E 等价于该类型中可表示的最大值减 E。

5.逻辑否定运算符的结果!如果操作数的值不等于 0，则为 0;如果操作数的值等于 0，则为 1。结果类型为 int。表达式!E 等价于(0==E)。

因此，&*E 等价于 E(即使 E 是空指针)，而&(E1[E2])等价于((E1)+(E2))。如果 E 是一个函数指示符，或者是一元运算符&的有效操作数的左值，*&E 是一个函数指示符，或者等于 E 的左值，则总是成立。如果*P 是一个左值，而 T 是一个对象指针类型的名称，则*(T)P 是一个与 T 所指向的类型兼容的左值。

在用一元*操作符解除对指针引用的无效值中，有一个空指针、一个与所指向对象类型不正确对齐的地址，以及一个对象生命周期结束后的地址。

6.5.3.4. sizeof 操作符

约束条件

1.sizeof 操作符不得应用于具有函数类型或不完整类型的表达式、此类类型的圆括号名称或指定位域成员的表达式。

语义

2.sizeof 操作符产生其操作数的大小(以字节为单位)，该操作数可以是表达式或带圆括号的类型名称。大小由操作数的类型决定。结果是一个整数。如果操作数的类型是变长数组类型，则对操作数求值;否则，操作数不计算，结果是一个整型常量。

3.当应用于 char 类型、unsigned char 类型或 signed char 类型(或其限定版本)的操作数时，结果为 1。当应用于具有数组类型的操作数时，其结果为数组中的总字节数。[84)]当应用于具有结构类型或联合类型的操作数时，其结果为该对象中的总字节数，包括内部填充和尾随填充。

84） 当应用于声明为数组或函数类型的形参时，sizeof 操作符将得到调整后(指针)类型的大小(参见 6.9.1)。

4.结果的值是实现定义的，它的类型(无符号整数类型)是 size_t，在<stddef.h>(和其他头文件)中定义。

5.例子 1　sizeof 操作符的主要用途是与存储分配器和 I/O 系统等例程通信。存储分配函数可以接受要分配的对象的大小(以字节为单位)，并返回指向 void 的指针。例如:

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

alloc 函数的实现应该确保其返回值与转换为双精度浮点数指针时的值保持适当的对齐。

6.例子 2　sizeof 操作符的另一个用途是计算数组中元素的数量:

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

7.例子 3　在本例中，可变长度数组的大小是由函数计算并返回的:

```
#include <stddef.h>
size_t fsize3(int n)
{
    char b[n+3]; // variable length array
    return sizeof b; // execution time sizeof
}

int main()
{
    size_t size;
    size = fsize3(10); // fsize3 returns 13
    return 0;
}
```

提前引用：常见定义<stddef.h>(7.17)、声明(6.7)、结构和联合说明符(6.7.2.1)、类型名称(6.7.6)、数组声明符(6.7.5.2)。

6.5.4. 类型转换运算符

语法

1.强制转换表达式：

一元表达式

（类型名）强制转换表达式

约束条件

2.除非类型名指定 void 类型，否则类型名应指定限定或非限定标量类型，操作数应具有标量类型。

3.除了 6.5.16.1 的约束所允许的以外，涉及指针的转换应通过显式类型转换来指定。

语义

4.在表达式前面加上带圆括号的类型名称将表达式的值转换为已命名的类型。[85)] 这种结构称为铸型。没有指定转换的强制转换对表达式的类型或值没有影响。86）

提前引用：相等操作符(6.5.9)、函数声明符(包括原型)(6.7.5.3)、简单赋值(6.5.16.1)、类型名称(6.7.6)。

85） 强制转换不会产生左值。因此，强制转换为限定类型与强制转换为该类型的限定版本具有相同的效果。

如果表达式的值的表示精度或范围大于通过强制转换命名的类型(6.3.1.8)所要求的，则强制转换指定了转换，即使表达式的类型与指定的类型相同。

6.5.5. 乘法运算符

语法

1.乘法表达式：

强制转换表达式

乘法表达式 * 强制转换表达式

乘法表达式 / 强制转换表达式

乘法表达式 % 强制转换表达式

约束条件

2.每个操作数应具有算术类型。操作符%的操作数必须是整数类型。

语义

3.通常对操作数执行算术转换。

4.二元*操作符的结果是两个操作数的乘积。

5./运算符的结果是第一个操作数与第二个操作数除法的商;%操作符的结果是余数。在这两个操作中，如果第二个操作数的值为零，则该行为未定义。

6.当整数被除时，/运算符的结果是任何小数部分被丢弃的代数商。87）如果商 a/b 可表示，则表达式(a/b)*b + a%b 应等于 a。

87）这通常被称为“向零截断”。

6.5.6. 加法运算符

语法

1.加法表达式：

乘法表达式

加法表达式 + 乘法表达式

加法表达式 - 乘法表达式

约束条件

2.对于加法，要么两个操作数都应该是算术类型，要么一个操作数应该是指向对象类型的指针，而另一个操作数应该是整数类型。(递增等于加 1。)

3.对于减法，应保持下列任一项:

——两个操作数都具有算术类型;

——两个操作数都是指向兼容对象类型的限定或非限定版本的指针;或

——左操作数是指向对象类型的指针，右操作数为整型。

(递减等于减 1。)

语义

4.如果两个操作数都具有算术类型，则对它们执行通常的算术转换。

5.二元+操作符的结果是两个操作数的和。

6.二元-运算符的结果是第二个操作数与第一个操作数相减所产生的差。

7.对于这些操作符，指向非数组元素的对象的指针与指向长度为 1 的数组中以该对象的类型作为元素类型的第一个元素的指针的行为相同。

8.当在指针上加或减整数类型的表达式时，结果为指针操作数的类型。如果指针操作数指向数组对象的一个元素，并且数组足够大，则结果指向与原始元素偏移的元素，这样得到的结果和原始数组元素的下标之差等于整型表达式。换句话说，如果表达式 P 指向数组对象的第 i 个元素，表达式(P)+N(相当于 N+(P))和(P)-N(其中 N 的值为 N)分别指向数组对象的第 i+ N 和第 i -N 个元素，前提是它们存在。此外，如果表达式 P 指向数组对象的最后一个元素，则表达式(P)+1 指向数组对象的最后一个元素的下一位置

9.当两个指针相减时，两个指针都指向同一个数组对象的元素，或指向数组对象最后一个元素的下一位置;结果是两个数组元素下标的差值。结果的大小是实现定义的，它的类型(有符号整数类型)是在<stddef.h>头文件中定义的ptrdiff_t。如果结果不能在该类型的对象中表示，则该行为是未定义的。换句话说，如果表达式 P 和 Q 分别指向数组对象的第 i 和第 j 个元素，则表达式(P)-(Q)的值为 i−j，前提是该值适合 ptrdiff_t 类型的对象。此外,如果表达式 P 点一个元素的数组对象或一个过去的最后一个元素的数组对象,和表情问指向最后一个元素的数组对象相同,表达式((Q) + 1) - (P)有相同的值((Q) - (P)) + 1 和- ((P) - ((Q) + 1)),并且值 0 如果表达式 P 点 1 过去的最后一个元素的数组对象,即使表达式(Q) + 1 不指向一个数组的元素对象。88)

10.例子　指针算术通过指向可变长度数组类型的指针很好地定义。

```
{
    int n = 4, m = 3;
    int a[n][m];
    int (*p)[m] = a; // p == &a[0]
    p += 1; // p == &a[1]
    (*p)[2] = 99; // a[1][2] == 99
    n = p - a; // n == 1
}
```

如果在上面的例子中，数组 a 被声明为已知常量大小的数组，而指针 p 被声明为指向相同常量大小的数组(指向a)的指针，结果将是相同的。

提前引用：数组声明符(6.7.5.2)，通用定义<stddef.h>(7.17)。

6.5.7.　移位运算符

语法结构;

移位表达式：

加法表达式

移位表达式 << 附加表达式

移位表达式 >> 附加表达式

约束：

每个操作数必须为整数类型。

语意：

对每个操作数执行整数提升。 运算结果的类型是左边操作数在提升之后的类型， 如果右边操作数的值为负数或大于等于提升后的左边操作数的宽度，则该行为是未被定义的。

E1 << E2 的结果是 E1 左移 E2 位，移位后的空位由 0 进行填补。如果 E1 为无符号类型，则运算结果的值为 E1

× 2^{E2}，运算结果为与结果类型中可表示的最大位数加 1 的模。如果 E1 为有符号类型且为非负值，那么 E1 × 2^{E2} 在结果类型中是可以表示的，那么这就是运算的结果值。否则，该行为是未被定义的。

E1 >> E2 的结果是 E1 右移 E2 位。如果 E1 为无符号类型或者 E1 为有符号类型且为非负值，则运算结果的值为 E1 / 2^{E2} 的商的整数部分。如果 E1 为有符号类型且为负值，则结果值是未明确定义的。

6.5.8. 关系运算符

语法结构;

关系表达式：

移位表达式

关系表达式 < 移位表达式

关系表达式 > 移位表达式

关系表达式 <= 移位表达式

关系表达式 >= 移位表达式

约束：

必须遵守以下其中一个条件：

-两个操作数都具有实数类型；

-两个操作数都是指向兼容对象类型的限定或非限定类型的指针；

-两个操作数都是指向兼容不完整类型的限定或非限定类型的指针。

语意：

如果这两个操作数都具有算术类型，则执行通常的算术转换。

对于这些操作符，指向非数组元素对象的指针与指向长度为 1 的以该对象类型作为元素类型的第一个元素的指针的行为相同。

比较两个指针时，结果取决于所指向对象在地址空间中的相对位置。 如果两个指向对象类型或不完整类型的指针都指向同一个对象，或者都指向同一个数组对象的最后一个元素，它们在比较中相等。如果指向的对象是同一个联合对象的成员，则指向稍后声明的结构成员的指针比较起来大于指向结构中较早声明的成员的指针，并且指向具有较大下标值的数组元素的指针比较起来大于指向同一数组的元素中具有较低的下标值的指针。

指向同一个联合对象成员的所有指针比较起来相等。如果表达式 P 指向数组对象的一个元素，而表达式 Q 指向同一数组对象的最后一个元素，则指针表达式 Q+1 的比较值大于 P。在所有其他情况下，这种行为都是未定义的。

每个算子<（小于），>（大于），<=（小于等于），>=（大于等于），如果指定的关系为真，将产生 1，如果为假，将产生 0，结果类型为整型 [89)]。

89）a < b < c 这一表达式不像普通数学那样解释。从语法上看，它的意思是 (a < b) < c ，换句话说，如果 a 小

于 b ，比较 1 和 c ，否则，比较 0 和 c 。

6.5.9. 等式运算符

语法结构;

等式表达式：

关系表达式

等式表达式 == 关系表达式

等式表达式 != 关系表达式

约束：

必须遵守以下其中一个条件：

-两个操作数都是算术类型；

-两个操作数都是指向兼容类型的限定或非限定类型的指针；

-一个操作数是指向对象类型或不完整类型的指针，另一个是指向 void 的限定或非限定类型的指针；

-一个操作数是指针，另一个是空指针常量。

语意：

==(等于)和!=(不等于)操作符类似于关系操作符，只是它们的优先级较低 [90)] 。如果指定的关系为真，每个操作符产生 1，如果为假，则产生 0。结果类型为整型。对于任意一对操作数，只有一个关系为真。

90）由于优先级，当 a < b 和 c < d 具有相同的真值时， a < b == c < d 为 1。

如果两个操作数都为算术类型，则执行通常的算术转换。复杂类型的值当且仅当它们的实部相等且虚部也相等时是相等的。来自不同类型的任意两个算术类型值，当且仅当它们转换的由常用算术转换确定的（复杂）结果类型的结果相等时才相等。

否则，至少有一个操作数是指针。如果一个操作数是指针，另一个是空指针常量，则空指针常量被转换为该指针的类型。如果一个操作数是指向对象类型或不完整类型的指针，而另一个操作数是指向 void 的限定或非限定类型的指针，则前者被转换为后者的类型。

两个指针当且仅当在这两个指针都是空指针，都指向同一个对象（包括一个指向对象的指针和子对象的开始）或函数，且都是指向一个过去的最后一个元素的数组对象的情况下，或者一个指针指向一个数组对象的末尾，另一个指针指向另一个数组对象的开始，这个数组对象恰好紧跟着地址空间中的第一个数组对象的情况下相等 [91)] 。

91）两个对象在内存中可能是相邻的，因为它们是较大数组的相邻元素或结构的相邻成员，但它们之间没有填充，或者因为实现选择这样放置它们，即使它们是不相关的。如果之前的无效指针操作（比如访问数组边界外）产生了未定义的行为，那么后续的比较也会产生未定义的行为。

6.5.10. 按位“与”运算符

语法结构;

与表达式：

等式表达式

与表达式 & 等式表达式

约束：

每个操作数都应为整数类型。

语意：

通常对操作数执行算术转换。

按位与操作符&的结果是操作数的按位与（即，当且仅当转换的操作数中的每个对应位都被设置时，结果中的每一位都被设置）。

6.5.11. 按位“异或”运算符

语法结构:

异或表达式：

与表达式

异或表达式 ^ 与表达式

约束：

每个操作数都应为整数类型。

语意：

通常对操作数执行算术转换。

^ 运算符的结果是操作数的按位异或运算（即，当且仅当转换后的操作数中恰好有一个对应的位被设置时，将设置结果中的每一位）。

6.5.12. 按位“或”运算符

语法结构;

或表达式：

异或表达式

或表达式 | 异或表达式

约束：

每个操作数都应为整数类型。

语意：

通常对操作数执行算术转换。

| 运算符的结果是操作数的按位或运算（即，当且仅当转换后的操作数中至少有一个对应的位被设置时，结果中的每一位都被设置）。

6.5.13. 逻辑“与”运算符

语法结构;

逻辑与表达式：

或表达式

逻辑与表达式 && 或表达式

约束：

每个操作数都应该是数值类型。

语意：

如果&&操作符的两个操作数都不等于 0，则该操作符将产生结果 1；否则，它将产生结果 0，结果类型为整型。

和按位与操作符&不同，&&操作符保证从左到右求值，在第一个操作数求值之后有一个序列点，如果第一个操作数等于 0，则不计算第二个操作数。

6.5.14. 逻辑“或”运算符

语法结构;

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：

每个操作数都应该是数值类型。

语意：

如果 || 操作符的两个操作数不等于 0，则该操作符将产生结果 1，否则它将产生结果 0，结果类型为整型。

和按位或运算符 | 不同， || 运算符保证从左到右求值，在第一个操作数求值之后有一个序列点，如果第一个操作数不等于 0，则不计算第二个操作数。

6.5.15. 条件运算符

语法结构;

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：

第一个操作数应该是标量类型。

对于第二个和第三个操作数，必须遵守以下其中一个条件:

-两个操作数都是算术类型；

-两个操作数具有相同的结构或联合类型；

-两个操作数都是 void 类型；

-两个操作数都是指向兼容类型的限定或非限定类型的指针；

-一个操作数是指针，另一个是空指针常量；

-一个操作数是指向对象类型或不完整类型的指针，另一个是指向 void 的限定或非限定类型的指针。

语意：

对第一个操作数求值;求值后有一个序列点。只有当第一个操作数不等于 0 时，才计算第二个操作数；只有当第一个操作数等于 0 时，才计算第三个操作数;结果是第二个或第三个操作数的值（以计算结果为准），转换为下面描述的类型 92）。如果试图修改条件运算符的结果或在下一个序列点之后访问它，则该行为是未定义的。

92）条件表达式不生成左值。

如果第二个和第三个操作数都具有算术类型，则通常的算术转换应用于这两个操作数时所确定的结果类型即为结果类型。如果两个操作数都具有结构类型或联合类型，则结果具有该类型。如果两个操作数都是 void 类型，则结果为 void 类型。

如果第二个和第三个操作数都是指针，或者一个是空指针常量，另一个是指针，则结果类型是一个指针，指向一个由两个操作数指向的类型的所有类型限定符限定的类型。此外，如果两个操作数都是指向兼容类型或指向兼容类型的不同限定版本的指针，则结果类型是指向复合类型的适当限定类型的指针；如果一个操作数是空指针常量，则结果为另一个操作数的类型；除此之外，一个操作数是指向 void 或 void 的限定类型的指针，在这种情况下，结果类型是指向适当限定类型的 void 的指针。

示例：当第二个和第三个操作数是指针时产生的通用类型在两个独立阶段确定。例如，适当的限定符不取决于两个指针是否具有兼容的类型。

给出声明：

```
const void *c_vp;
void *vp;
const int *c_ip;
volatile int *v_ip;
int *ip;
const char *c_cp;
```

下表中的第三列是通用类型，它是条件表达式的结果，其中前两列是第二个和第三个操作数（按任意顺序）：

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 *

6.5.16. 赋值运算符

语法结构;

赋值表达式：

条件表达式

一元表达式 赋值运算符 赋值表达式

赋值运算符：下列中的其中之一

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

约束：

赋值运算符应具有一个可修改的左值作为其左操作数。

语意：

赋值操作符将值存储在由左操作数指定的对象中，赋值表达式在赋值后具有左操作数的值，但不是左值。赋值表达式的类型是左操作数的类型，除非左操作数具有限定类型，在这种情况下，它是左操作数类型的限定类型。更新左操作数的存储值的操作将发生在前一个序列点和下一个序列点之间。

操作数的求值顺序未指定。如果试图修改赋值操作符的结果或在下一个序列点之后访问它，则该行为是未定义的。

6.5.16.1. 简单赋值

约束：

必须遵守以下其中一个条件 [93]：

93）这些约束对于类型限定符的不对称外观是由于将左值更改为“表达式的值”的转换（在 6.3.2.1 中指定），从而从表达式的类型类别中删除任何类型限定符。

-左操作数具有限定或非限定算术类型，右操作数具有算术类型；

-左操作数具有与右操作数类型兼容的结构或联合类型的限定或非限定类型；

-两个操作数都是指向兼容类型的限定或非限定类型的指针，左边指向的类型具有右边指向的类型的所有限定符；

-一个操作数是指向对象或不完整类型的指针，另一个是指向限定或不限定 void 的指针，左指向的类型具有右指向的类型的所有限定符；

-左操作数是一个指针，右操作数是一个空指针常量；

-左操作数为_Bool 类型，右操作数为指针。

语意：

如果存储在一个对象中的值是从以任何方式与第一个对象的存储重叠的另一个对象读取的，则重叠应准确，两个对象应该具有兼容类型的有条件或无条件的类型；否则，该行为是未定义的。

示例 1：在程序片段

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

中函数返回的 int 值在存储到 char 类型时可能会被截断，然后在比较之前被转换回 int 宽度。在" plain " char 与 unsigned char 具有相同的值范围（而且 char 比 int 更窄）的情况下，转换的结果不能为负，因此比较的操作数永远不能相等。因此，为了完全可移植性，变量 c 应该声明为 int。

示例 2：在片段

```
char c;
```

```
int i;
long l;
l = (c = i);
```

中 i 的值被转换为赋值表达式 c= i 的类型，即 char 类型。然后，括号括起来的表达式的值被转换为外部赋值表达式的类型，即 long int 类型。

示例 3：考虑片段

```
const char **cpp;
char *p;
const char c = 'A';
cpp = &p;  // 违反约束
*cpp = &c; // 有效的
*p = 0;    // 有效的
```

第一个赋值是不安全的，因为它将允许以下有效的代码尝试更改 const 对象 c 的值。

6.5.16.2. 复合赋值

约束：

对于操作符+=和-=，要么左操作数为指向对象类型的指针，右操作数为整数类型，要么左操作数为有条件或无条件算术类型，右操作数为算术类型。

对于其他操作符，每个操作数应具有与对应的二元操作符所允许的算术类型一致的算术类型。

语意：

复合赋值表达式 E1 op = E2 与简单赋值表达式 E1 = E1 op (E2) 的区别仅在于左值 E1 只计算一次。

6.5.17. 逗号运算符

语法结构：

表达式：

赋值表达式

表达式 ， 赋值表达式

语意：

逗号操作符的左操作数计算为 void 表达式，求值后有一个序列点，然后对右操作数求值，结果有其类型和值[94)]。如果试图修改逗号操作符的结果或在下一个序列点之后访问它，则该行为是未定义的。

94）逗号运算符不生成左值。

示例：如语法结构所示，逗号操作符（如本子句所述）不能出现在使用逗号分隔列表中的项的上下文中（如函数的参数或软件初始化）。另一方面，它可以用在带括号的表达式中，也可以用在这种上下文中条件操作符的第二个表达式中。在函数调用 f(a, (t=3, t+2), c) 中，函数有三个参数，第二个参数的值为 5。

引用：初始化(6.7.8)

6.6. 常量表达式

语法结构：

常量表达式：

条件表达式

说明：

常量表达式可以在转换期间计算，而不是在运行时计算，因此可以在任何有常量的地方使用。

约束：

常量表达式不应包含赋值、自增、自减、函数调用或逗号操作符，除非它们包含在没有计算的子表达式中[95)]。

95）sizeof 操作符的操作数通常不计算(6.5.3.4)。

每个常数表达式的值应在其类型的可表示值范围内。

语意：

在多个上下文中需要求值为常量的表达式，如果在转换环境中计算浮点表达式，则算术精度和范围应至少与在执行环境中计算该表达式一样大。

整型常量表达式[96)]应具有整型类型，且其操作数只能为整型常量、枚举常量、字符常量、结果为整型常量的 sizeof 表达式，以及作为强制转换的直接操作数的浮点常量。整型常量表达式中的强制转换操作符只能将算术类型转换为整型，除非作为操作数的一部分转换为 sizeof 操作符。

96）整数常数表达式用于指定结构的位字段成员的大小、枚举常数的值、数组的大小或大小写常数的值，应用于条件包含预处理指令中的整型常量表达式的进一步约束将在 6.10.1 中讨论。

对于初始化中的常量表达式，允许有更多的自由度。该常数表达式应为或计算为以下方式之一：

-一个算术常数表达式；

-一个空指针常量；

-一个地址常量；

-对象类型的地址常数加或减一个整数常数表达式。

算术常量表达式必须具有算术类型，且操作数只能为整型常量、浮点常量、枚举常量、字符常量和 sizeof 表达式。算术常量表达式中的强制转换操作符只能将算术类型转换为算术类型，除非作为操作数的一部分转换为结果是整型常量的 sizeof 操作符。

地址常量是空指针、指向指定静态存储持续时间对象的左值的指针或指向函数指示符的指针，它应该使用单目运算符 & 或转换为指针类型的整型常量显式创建，或者使用数组或函数类型的表达式隐式创建。数组下标 [] 、成员访问和 -> 操作符、地址 & 和间接 * 单目运算符以及指针强制转换可以用于地址常量的创建，但对象的值不能通过这些操作符访问。

一个实现可以接受其他形式的常数表达式。

常量表达式求值的语义规则与非常量表达式相同[97)]。

97）因此，在接下来的初始化中，

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

该表达式是一个值为 1 的有效整数常量表达式。

引用：数组声明符(6.7.5.2)、初始化(6.7.8)。

6.7. 声明

语法

声明：

声明说明符 初始化声明器---listopt;

声明说明符：

存储类说明符 声明说明符 opt

类型说明符 声明说明符 opt

类型限定符 声明说明符 opt

函数说明符 声明说明符 opt

初始化声明器列表：

初始化声明器

初始化声明器列表，初始化声明器

初始化声明：

声明符

声明符=初始化器

约束

2 声明应至少声明一个声明符（函数的参数或结构或联合的成员）、标签或枚举的成员。

3 如果标识符没有链接，则标识符的声明不得超过一个(在声明符或类型说明符中)具有相同的作用域和相同的名称空间，除了对于 6.7.2.3 中指定的标签。

4 同一范围内引用同一对象或函数的所有声明应指定兼容的类型。

语义

5 声明指定一组标识符的解释和属性。 定义标识符的声明应该这样声明：

对于一个对象，使得为该对象保留存储空间；

对于函数，包括其函数体[98]；

对于枚举常量或 typedef 名称，是（唯一的）声明标识符

6 声明说明符由指示链接的说明符序列组成，存储持续时间，以及声明符表示的实体类型的一部分。 init declarator-list 是一个逗号分隔的声明符序列，每个声明符可能有额外的类型信息，或初始化器，或两者兼有。 声明器包含声明的标识符（如果有的话)。

98) 函数定义有不同的语法，在 6.9.1 中描述。

7 如果一个对象的标识符被声明为没有链接，则该对象的类型应在其声明符的末尾完成，如果它具有初始化器，则在其 init-declarator 的末尾完成； 在函数参数（包括原型）的情况下，需要完整的是调整后的类型（见 6.7.5.3）。

前向引用：声明符（6.7.5）、枚举说明符（6.7.2.2）、初始化（6.7.8）

6.7.1. 存储类说明符

语法

1 存储类说明符：

typedef
extern
static
auto
Register

约束

2 最多可以在声明的声明说明符中给出一个存储类说明符[99]。

语义

3 typedef 说明符被称为“存储类说明符”只是为了语法方便； 将在 6.7.7 中讨论。 在 6.2.2 和 6.2.4 中讨论了各种链接和存储持续时间的含义.

4 具有存储类说明符寄存器的对象标识符声明表明对对象的访问尽可能快。 此类建议的有效程度由实施定

义.[100]

5 具有块作用域的函数的标识符声明不应具有除 extern 之外的显式存储类说明符.

见“未来语言方向”(6.11.5)。

实现可以将任何寄存器声明简单地视为自动声明。 但是，无论是否实际使用可寻址存储，使用声明的对象的任何部分的地址存储类说明符寄存器不能显式计算（通过使用 6.5.3.2 中讨论的一元 & 运算符）或隐式计算（通过将数组名称转换为指针，如 6.5.3.2 中讨论的）6.3.2.1)。 因此，唯一可以应用于使用存储类说明符寄存器声明的数组的运算符是 sizeof。

6 如果使用 typedef 以外的存储类说明符声明聚合或联合对象，则存储类说明符产生的属性（除了链接方面）也适用于对象的成员，以此类推任何聚合

或联合成员对象也适用。

前向引用：类型定义 (6.7.7)

6.7.2. 类型说明符

语法

1 类型说明符：

void
char
short
int
long
float
double
signed
unsigned
_Bool
_Complex
_Imaginary

结构或联合说明符

枚举说明符

类型定义名称

约束

在每个声明的声明说明符中，以及每个结构声明和类型名称的说明符-限定符列表中，至少应给出一个类型说明符。 每个类型说明符列表应为以下集合之一（当一行中有多个集合时，以逗号分隔）； 类型说明符可以以任何顺序出现，可能会与其他声明说明符混合。

— void
— char
— signed char
— unsigned char
— short, signed short, short int, or signed short int
— unsigned short, or unsigned short int
—int, signed, or signed int
— unsigned, or unsigned int
— long, signed long, long int, or signed long int
— unsigned long, or unsigned long int
— long long, signed long long, long long int, or signed long long int
— unsigned long long, or unsigned long long int
— float
— double
— long double
— _Bool
— float _Complex
— double _Complex
— long double _Complex
— float _Imaginary
— double _Imaginary
— long double _Imaginary

结构或联合说明符

枚举说明符

类型定义名称

3 如果实现不使用这些类型，则不应使用类型说明符 _Complex 和 _Imaginary[101]。

语义

4 结构、联合和枚举的说明符在 6.7.2.1 到 6.7.2.3 中讨论。 typedef 名称的声明在 6.7.7 中讨论。 其他类型的特性在 6.2.5 中讨论。

5 每个逗号分隔的集合都指定相同的类型，除了位域，说明符 int 指定 signed int 与 unsigned int 相同的类型。

前向引用：枚举说明符 (6.7.2.2)、结构和联合说明符 (6.7.2.1)、标签 (6.7.2.3)、类型定义 (6.7.7)

101) 实现不需要提供虚构的类型。 不需要独立的实现来提供复杂的类型

6.7.2.1. 结构和联合说明符

语法

1.结构或联合说明符：

Struct-or-union 标识符

结构或联合：

struct

Union

结构声明列表：

结构声明

结构声明列表 结构声明：

类型说明符说明符限定符列表选择

类型限定符说明符限定符列表选择

结构声明器列表：

结构声明符

结构声明器列表，结构声明器

结构声明器：

声明者

declaratoropt : 常量表达式

约束

2 结构或联合不应包含不完整或函数类型的成员（因此，结构不应包含自身的实例，但可以包含指向自身实例的指针），除了结构的最后一个成员具有更多 比一个具名的成员可能有不完整的数组类型； 这样的结构（以及任何可能递归地包含此类结构的成员的联合）不应是结构的成员或数组的元素。

3 指定位域宽度的表达式应为具有非负值的整数常量表达式，如果省略冒号和表达式，则该值不得超过指定类型的对象中的位数。 如果值为零，则声明不应有声明符。

4 位域的类型应为 _Bool 的合格或不合格版本，带符号 int、unsigned int 或其他一些实现定义的类型。

语义

5 如 6.2.5 所述，结构是由一系列成员组成的类型，其存储按有序顺序分配，而联合是由一系列成员组成的类型，其存储重叠。

6 结构和联合说明符具有相同的形式

7 struct-or-union-specifier 中的 struct-declaration-list 的存在声明了翻译单元内的新类型。 struct-declaration-list 是结构或联合成员的一系列声明。如果 struct-declaration-list 不包含命名成员，则行为未定义。在终止列表的 } 之后，类型是不完整的。

8 结构或联合的成员可以具有除可变修改类型之外的任何对象类型[102]) 此外，可以声明成员由指定数量的位

组成（包括符号位，如果有的话）。 这样的成员称为一个位域.[103]) 它的宽度前面有一个冒号。

9 位域被解释为由指定位数组成的有符号或无符号整数类型.[104] 如果值 0 或 1 存储到类型的非零宽度位域中_Bool，位域的值应该等于存储的值。

10 实现可以分配任何大到足以容纳位字段的可寻址存储单元。 如果有足够的空间剩余，紧跟在结构中另一个位域之后的位域将被打包到同一单元的相邻位中。 如果剩余空间不足，是否将不适合的位域放入下一个单元或与相邻单元重叠是实现定义的。 单元内位域的分配顺序（高位到低位或低位到高位）是实现定义的。 未指定可寻址存储单元的对齐方式。

11 没有声明符但只有冒号和宽度的位域声明表示未命名的位域.[105]。作为一种特殊情况，宽度为 0 的位域结构成员表示没有进一步的位域 将被打包到放置前一个位字段（如果有）的单元中。

102）结构或联合不能包含具有可变修改类型的成员，因为成员名称不是 6.2.3 中定义的普通标识符。

103) 一元 & (address-of) 运算符不能应用于位域对象； 因此，没有指向位域对象的指针或数组。

104) 如上面 6.7.2 所述，如果实际使用的类型说明符是 int 或定义为 int 的 typedef-name，则位域是有符号还是无符号是实现定义的。

105) 未命名的位域结构成员可用于填充以符合外部强加的布局

12 结构或联合对象的每个非位域成员以适合其类型的实现定义的方式对齐.

13 在结构对象中，非位域成员和位域所在的单元的地址按声明顺序递增。 一个指向结构对象的指针，经过适当的转换，指向它的初始成员（或者如果该成员是位域，则指向它所在的单元），反之亦然。 结构对象内可能有未命名的填充，但不是在其开头。

14 这个单元的规模足以容纳其最大的成员。 任何时候最多可以将其中一个成员的值存储在联合对象中。 一个指向联合对象的指针，经过适当的转换，指向它的每个成员（或者如果一个成员是一个位域，然后到它所在的单元），反之亦然。

15 在结构或联合的末尾可能有未命名的填充。

16 作为一种特殊情况，具有多个命名成员的结构的最后一个元素可能具有不完整的数组类型；这称为灵活数组成员。除了两个例外，灵活数组成员被忽略。首先，结构的尺寸应为等于另一个相同结构的最后一个元素的偏移量，该结构用未指定长度的数组替换灵活数组成员.[106]) 其次，当 a .（或->）运算符的左操作数是（指向）具有灵活数组成员的结构，右操作数命名该成员，它的行为就像该成员被替换为最长的数组（具有相同的元素类型) 不会使结构大于被访问的对象；数组的偏移量应保持灵活数组成员的偏移量，即使这与替换数组的偏移量不同。如果这个数组没有元素，它的行为就好像它有一个元素，但如果尝试访问该元素或生成一个越过它的指针，则行为是不确定的。

17 示例 假设所有数组成员的对齐方式相同，在声明之后：

```
Struct s { int n; double d[]; };
Struct ss { int n; double d[1]; };
```

三种表达方式：

```
sizeof (struct s)
```

```
offsetof(struct s, d)
offsetof(struct ss, d)
```

具有相同的价值。 结构 struct s 有一个灵活的数组成员 d。

106) 长度未指定，以允许实现可能给数组成员不同的长度来进行对齐。

18 如果 sizeof(double) 为 8，则在执行以下代码后：

```
struct s *s1;
struct s *s2;
s1 = malloc(sizeof (struct s) + 64);
s2 = malloc(sizeof (struct s) + 46);
```

并假设对 malloc 的调用成功，s1 和 s2 指向的对象的行为就像标识符已被声明为：

```
struct { int n; double d[8]; } *s1;
struct { int n; double d[5]; } *s2;
```

19 继进一步成功的任务之后：

```
s1 = malloc(sizeof (struct s) + 10);
s2 = malloc(sizeof (struct s) + 6);
```

然后它们的行为就好像声明是：

```
struct { int n; double d[1]; } *s1, *s2;
```

和

```
double *dp;
dp = &(s1->d[0]); // valid
*dp = 42; // valid
dp = &(s2->d[0]); // valid
*dp = 42; // undefifined behavior
```

20 任务

```
*s1 = *s2;
```

只复制成员 n 而不是任何数组元素。 相似地：

```
struct s t1 = { 0 }; // valid
struct s t2 = { 2 }; // valid
struct ss tt = { 1, { 4.2 }}; // valid
struct s t3 = { 1, { 4.2 }}; // invalid: there is nothing for the 4.2 to initialize
```

```
t1.n = 4; // valid
t1.d[0] = 4.2; // undefifined behavior
```

前向引用：标签 (6.7.2.3)。

6.7.2.2. 枚举说明符

语法

1 枚举说明符：

enum identififieropt { enumerator-list }
enum identififieropt { enumerator-list , }
enum identififier

枚举器列表：

枚举器

枚举器列表,枚举器

枚举器：

枚举常量

枚举常量 = 常量表达式

约束

2 定义枚举常量值的表达式应为整数常量表达式，其值可表示为 int

语义

3 枚举器列表中的标识符被声明为具有 int 类型的常量，并且可以出现在任何允许的地方.[107]) 带有 = 的枚举器将其枚举常量定义为常量表达式的值。如果第一个枚举数没有 =，则其枚举常数的值为 0。随后的每个没有 = 的枚举数将其枚举常数定义为由下式获得的常量表达式的值将前一个枚举常量的值加 1。（使用带= 的枚举数可能会产生枚举常量，其值与同一枚举中的其他值重复。）枚举的枚举数也称为其成员。

4 每个枚举类型应与 char、有符号整数类型或无符号整数类型兼容。 类型的选择是实现定义的.[108]， 但应能够表示所有枚举成员的值。 在终止枚举器声明列表的 } 之后，枚举类型是不完整的。

5 例子 以下片段:

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

/* ... */

使 hue 成为枚举的标记，然后将 col 声明为具有该类型的对象，并将 cp 声明为指向具有该类型的对象的指针。 枚举值在集合 { 0, 1, 20, 21 }

因此，在同一作用域中声明的枚举常量的标识符应彼此不同，并且与在普通声明符中声明的其他标识符不同。

108) 一个实现可能会延迟选择哪种整数类型，直到所有枚举常量都被看到。

6.7.2.3. 标签

约束

1 一个特定类型的内容最多只能定义一次。

2 没有枚举数列表的形式枚举标识符的类型说明符仅应在其指定的类型完成后出现。

语义

具有相同范围并使用相同标记的结构、联合或枚举类型的所有声明都声明相同的类型。类型在定义内容的列表的右大括号之前是不完整的[109]，直到定义内容的列表的右大括号，然后完成。

在不同范围内或使用不同标记的结构、联合或枚举类型的两个声明声明了不同的类型。 不包含标记的结构、联合或枚举类型的每个声明都声明了不同的类型。

一种形式的类型说明符

struct-or-union identififieropt { struct-declaration-list }
or
enum identififier { enumerator-list }
or
enum identififier { enumerator-list , }

声明结构、联合或枚举类型。 列表定义结构内容、联合内容或枚举内容。 如果提供了标识符[110]，类型说明符还将标识符声明为该类型的标记。

6 表格声明

结构或联合标识符；

指定结构或联合类型并将标识符声明为该类型的标记[111]。

109）不完整类型只能在不需要该类型对象的大小时使用。 例如，当 typedef 名称被声明为结构或联合的说明符时，或者在声明指向或返回结构或联合的函数时，不需要它。 （参见 6.2.5 中的不完整类型。）在调用或定义这样的函数之前，规范必须是完整的。

110) 如果没有标识符，则该类型在翻译单元内只能由它作为一部分的声明来引用。 当然，当声明是 typedef 名称时，后续声明可以使用该 typedef 名称来声明具有指定结构、联合或枚举类型的对象。

111) 不存在与枚举类似的构造。

7 如果一个类型说明符的形式

结构或联合标识符 identififier

除了作为上述形式之一的一部分出现，并且没有其他标识符作为标记的声明是可见的，那么它声明了一个不完整的结构或联合类型，并将标识符声明为该类型的标记.[111].

8 如果形式的类型说明符

结构或联合标识符

或者

枚举标识符

不是作为上述形式之一的一部分出现，并且标识符作为标记的声明是可见的，则它指定与其他声明相同的类型，并且不重新声明标记.

9 示例 1 该机制允许声明自引用结构。

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

指定一个结构，该结构包含一个整数和两个指向相同类型对象的指针。 此声明一经发出，声明

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

将 s 声明为给定类型的对象，并将 sp 声明为指向给定类型的对象的指针。 使用这些声明，表达式 sp->left 引用对象的左结构 tnode 指针

sp 指向哪个表达式 s.right->count 指定从 s 指向的右 structtnode 的 count 成员。

10 以下替代公式使用 typedef 机制：

```
typedef struct tnode TNODE;
struct tnode {
int count;
TNODE *left, *right;
};
TNODE s, *sp;
```

11 示例 2 说明使用标签的预先声明来指定一对相互引用结构，声明

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

struct s2 { struct s1 *s1p; /* ... */ }; // D2

指定一对包含彼此指针的结构。 但是请注意，如果 s2 已被声明为封闭范围内的标记，则声明 D1 将引用它，而不是 D2 中声明的标记 s2。 为了消除这种上下文敏感性，

声明 struct s2;

可以在 D1 之前插入。 这在内部范围内声明了一个新标签 s2； 然后，声明 D2 完成了新类型的规范。

前向引用：声明符 (6.7.5)、数组声明符 (6.7.5.2)、类型定义(6.7.7)。

6.7.3. 类型限定符

语法

类型限定符：
const
restrict
Volatile

约束

2 从对象或不完整类型派生的指针类型以外的类型不应被限制限定

语义

3 与限定类型关联的属性仅对左值表达式有意义·[112]。

4 如果同一个限定符在同一个说明符限定符列表中出现多次，无论是直接出现还是通过一个或多个 typedef，行为与它只出现一次相同。

5 如果尝试通过使用具有非 const 限定类型的左值来修改使用 const 限定类型定义的对象，则这种行为未被定义。 如果尝试通过使用具有非 volatile 限定类型的左值来引用具有 volatile 限定类型定义的对象，则这种行为未被定义·[113]。

6 具有 volatile 限定类型的对象可能会以实现未知的方式进行修改或具有其他未知的副作用。 因此，任何引用此类对象的表达式都应严格按照抽象机的规则进行评估，如 5.1.2.3 所述。 此外，在每个序列点，最后存储在对象中的值应与抽象机规定的值一致，除非被前面提到的未知因素修改·[114]。 构成对具有 volatile 限定类型的对象的访问是实现 -定义。

112) 实现可以将非易失性的 const 对象放置在存储的只读区域中。 此外，如果从未使用过此类对象的地址，则实现不需要为此类对象分配存储空间。

113）这适用于那些表现得好像它们是用限定类型定义的对象，即使它们实际上从未定义为程序中的对象（例如内存映射输入/输出地址处的对象）

7 通过限制限定指针访问的对象与该指针具有特殊关联。 这种关联，在下面的 6.7.3.1 中定义，要求对该对象

的所有访问直接或间接使用该特定指针的值.[115]。 预期使用限制限定符（如寄存器存储类）是为了促进优化，并且从构成符合程序的所有预处理翻译单元中删除限定符的所有实例不会改变其含义（即，可观察的行为）。

8 如果数组类型的规范包括任何类型限定符，则元素类型是这样限定的，而不是数组类型。 如果函数类型的规范包含任何类型限定符，则这种行为未被定义.[116]。

9 对于要兼容的两个合格类型，两者都应具有兼容类型的相同合格版本； 说明符或限定符列表中类型限定符的顺序不影响指定的类型。

10 示例 1 声明的对象

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

.可以通过硬件修改，但不能分配、递增或递减。

11 示例 2 以下声明和表达式说明了类型限定符时的行为

修改聚合类型：

```
const struct s { int mem; } cs = { 1 };
struct s ncs; // the object ncs is modifiable
typedef int A[2][3];
const A a = {{4, 5, 6}, {7, 8, 9}}; // array of array of const int
int *pi;
const int *pci;
ncs = cs; // valid
cs = ncs; // violates modifiable lvalue constraint for =
pi = &ncs.mem; // valid
pi = &cs.mem; // violates type constraints for =
pci = &cs.mem; // valid
pi = a[0]; // invalid: a[0] has type “const int *”
```

volatile 声明可用于描述对应于内存映射输入/输出端口的对象或异步中断函数访问的对象。作用在如此声明的对象不应被实现“优化”或重新排序，除非评估表达式的规则允许。

例如，将 malloc 返回的值分配给单个指针的语句在分配的对象和指针之间建立了这种关联。

这两种情况都可以通过使用 typedef 来实现。

6.7.3.1. 正式定义 restrict

1 假设 D 是一个普通标识符的声明，该标识符提供了一种方法，可以将对象 P 指定为指向类型 T 的限制性指针。

2 如果 D 出现在一个代码块内，并且没有存储类别 extern，则用 B 表示该代码块。如果 D 出现在函数定义的参数声明列表中，则用 B 表示相关的代码块。否则，用 B 表示 main 函数代码块(或独立环境中程序启动时调用的任何函数的代码块)。

3 接下来，指针表达式 E 被称为基于对象 P 的，如果(在执行 B 的某个序列点，在 E 求值之前)修改 P 指向它之前指向的数组对象的副本，将改变 E 的值。注意，“based”只针对指针类型的表达式定义。

4 在每次执行 B 时，设 L 为基于 P 的具有&L 的任意左值。如果 L 被用于访问它指定的对象 X 的值，并且 X 也被修改(以任何方式)，则适用以下要求:T 不能是 const 限定的。每一个用于访问 X 值的其他左值，其地址也应基于 P。在本小节中，每一个修改 X 的访问也应被视为修改 P。如果 P 被赋值给一个指针表达式 E，该表达式基于另一个与代码块 B2 关联的受限指针对象 P2，那么要么 B2 的执行应该在 B 的执行之前开始，要么 B2 的执行应该在赋值之前结束。如果这些要求没有得到满足，那么这种行为是未定义的。

5 在这里，B 的执行意味着程序执行的那一部分将对应于与 B 相关联的标量类型和自动存储持续时间的对象的生存周期。

6 编译器可以随意忽略使用 restrict 的任何或所有混叠含义。

7 例 1 文件作用域声明

```
int * restrict a;
int * restrict b;
extern int c[];
```

可以断言，如果一个对象是使用 a、b 或 c 中的一个访问的，并且该对象在程序中的任何地方都被修改了，那么它永远不会使用其他两个中的任何一个访问。

8 例 2 函数参数声明在下面的例子中

```
void f(int n, int * restrict p, int * restrict q)
{
    while (n-- > 0)
    *p++ = *q++;
}
```

可以断言，在每次函数执行期间，如果一个对象是通过其中一个指针形参访问的，那么它就不会通过另一个指针形参访问。

9 restrict 修饰符的好处是，它们使编译器能够对函数 f 进行有效的依赖性分析，而无需检查程序中对 f 的任何调用。代价是程序员必须检查所有的这些调用，以确保没有任何调用给出未定义的行为。例如，g 中 f 的第二次调用具有未定义的行为，因为 d[1]到 d[49]都通过 p 和 q 访问。

```
void g(void)
{
    extern int d[100];
    f(50, d + 50, d); // 有效的
    f(50, d + 1, d); // 未定义的行为
}
```

10 例 3 函数形参声明

```
void h(int n, int * restrict p, int * restrict q, int * restrict r)
{
    int i;
    for (i = 0; i < n; i++)
    p[i] = q[i] + r[i];
}
```

说明未修改的对象如何通过两个受限指针起别名。特别是，如果 a 和 b 是不相交的数组，h(100, a, b, b)形式的调用具有已定义的行为，因为数组 b 在函数 h 内没有修改。

11 例 4 限制受限制指针之间赋值的规则不区分函数调用和等效嵌套块。只有一个例外，在嵌套块中声明的受限指针之间，只有“从外到内”的赋值具有定义的行为。

```
{
    int * restrict p1;
    int * restrict q1;
    p1 = q1; // 未定义的行为
    {
        int * restrict p2 = p1; // 有效的
        int * restrict q2 = q1; // 有效的
        p1 = q2; // 未定义的行为
        p2 = q2; // 未定义的行为
    }
}
```

12 一个例外是允许在块执行结束时声明受限制指针的值(或者更准确地说，是用于指定指针的普通标识符)。例如，这允许 new_vector 返回一个 vector。

```
typedef struct { int n; float * restrict v; } vector;
vector new_vector(int n)
{
    vector t;
    t.n = n;
    t.v = malloc(n * sizeof (float));
    return t;
}
```

6.7.4. 函数说明符

语法结构

1 function-specifier:
inline

约束条件

2 函数说明符应仅在函数标识符的声明中使用。

3 具有外部链接的函数的内联定义不应包含具有静态存储周期的可修改对象的定义，也不应包含对具有内部链接的标识符的引用。

4 在主机环境中，内联函数说明符不能出现在 main 的声明中。

语义

5 用内联函数说明符声明的函数是内联函数。函数说明符可以出现多次;这种行为就像它只出现过一次一样。将函数设置为内联函数意味着对该函数的调用要尽可能快 118)。这种建议的有效程度由具体实现决定。119)

118) 例如，通过使用通常的函数调用机制的替代，如“内联替换”。内联替换不是文本替换，也不会创建新函数。因此，例如，在函数体中使用的宏的扩展使用的是函数体出现时的定义，而不是函数被调用的地方;而标识符指的是发生主体的作用域中的声明。同样，函数只有一个地址，而不管外部定义之外的内联定义有多少。

119) 例如，实现可能永远不会执行内联替换，或者可能只对内联声明范围内的调用执行内联替换。

6 任何具有内部链接的函数都可以是内联函数。对于具有外部链接的函数，适用以下限制:如果一个函数使用内联函数说明符声明，那么它也应该在同一个编译单元中定义。如果编译单元中一个函数的所有文件范围声明都包含内联函数说明符，而没有 extern，那么该编译单元中的定义就是内联定义。内联定义不为函数提供外部定义，也不禁止在另一个编译单元中使用外部定义。内联定义提供了外部定义的替代方法，编译器可以使用外部定义实现同一编译单元中对函数的任何调用。函数调用使用内联定义还是外部定义是非特定的。

120) 由于内联定义不同于相应的外部定义，也不同于其他编译单元中任何其他相应的内联定义，因此在每个定义中，具有静态存储周期的所有对应对象也不同。

7 例子 带有外部链接的内联函数的声明可以产生外部定义，也可以产生只能在编译单元内使用的定义。使用 extern 的文件作用域声明将创建外部定义。下面的例子展示了一个完整的编译单元。

```
inline double fahr(double t)
{
    return (9.0 * t) / 5.0 + 32.0;
}
    inline double cels(double t)
{
    return (5.0 * (t - 32.0)) / 9.0;
```

```
}
    extern double fahr(double); // 创建一个外部定义
    double convert(int is_fahr, double temp)
{
    /* 编译器可以执行内联替换 */
    return is_fahr ? cels(temp) : fahr(temp);
}
```

8 注意，fahr 的定义是一个外部定义，因为 fahr 也使用 extern 声明，但 cels 的定义是内联定义。由于 cels 具有外部链接并被引用，外部定义必须出现在另一个编译单元中(见 6.9);该内联定义和外部定义是不同的，任何一个都可以用于调用。

前向引用:函数定义(6.9.1)。

6.7.5. 声明符

语法结构

```
1 declarator:
    pointer opt   direct-declarator
direct-declarator:
    identifiifier
    ( declarator )
    direct-declarator [ type-qualifiifier-listopt assignment-expressionopt ]
    direct-declarator [ static type-qualifiifier-listopt assignment-expression ]
    direct-declarator [ type-qualifiifier-list static assignment-expression ]
    direct-declarator [ type-qualifiifier-listopt * ]
    direct-declarator ( parameter-type-list )
    direct-declarator ( identifiifier-listopt )
pointer:
    * type-qualifiifier-listopt
    * type-qualifiifier-listopt pointer
type-qualifiifier-list:
    type-qualifiifier
    type-qualifiifier-list   type-qualifiifier
parameter-type-list:
    parameter-list
    parameter-list , ...
parameter-list:
    parameter-declaration
    parameter-list , parameter-declaration
parameter-declaration:
```

declaration-specifiers declarator
declaration-specifiers abstract-declaratoropt

identifier-list:
identifier
identifier-list , identifier

约束条件

2 每个声明符声明一个标识符，并断言当表达式中出现与声明符形式相同的操作数时，它指定一个函数或对象，该函数或对象具有声明符所指定的作用域、存储周期和类型。

3 完整声明符是一种不属于其他声明符部分的声明符。完整声明符的末尾是序列点。如果完整声明符中的嵌套声明符序列包含可变长度数组类型，则称完整声明符指定的类型是可变修改的。

4 在下面的子句中，考虑一个声明

T D1

其中 T 包含声明说明符，指定类型 T(比如 int)， D1 是包含标识符的声明符。在各种形式的声明符中为标识符标识指定的类型使用这种表示法进行归纳描述。

5 如在“T D1”声明中，D1 有以下构成

Identifier

则为 ident 指定的类型为 T

6 如在“T D1”声明中，D1 有以下构成

(D)

则 ident 具有声明“T D”指定的类型。因此，圆括号中的声明符与没有圆括号的声明符是相同的，但是复杂声明符的绑定可以通过圆括号改变。

使用限制

7 正如在 5.2.4.1 中讨论的，一个实现可以直接或通过一个或多个 typedef 限制用于修改算术、结构、联合或不完全类型的指针、数组和函数声明符的数量。

前向引用:数组声明符(6.7.5.2)、类型定义(6.7.7)。

6.7.5.1. 指针声明符

语法结构

1 如在“T D1”声明中，D1 有以下构成

***type-qualififier-listopt D**

在声明“T D”中为 ident 指定的类型是“派生声明符-类型列表 T”，那么为 ident 指定的类型是“派生声明符-类型列表类型限定符-指向 T 的列表指针”。对于列表中的每个类型限定符，ident 都是这样限定的指针。

2 对于兼容的两种指针类型，两者应具有相同的限定条件，且都应是指向兼容类型的指针。

3 例子 下面的两个声明说明了“指向常量的变量指针”和“指向变量值的常量指针”之间的区别。

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

ptr_to_constant 所指向的任何对象的内容都不能通过该指针修改，但 ptr_to_constant 本身可以被更改为指向另一个对象。类似地，constant_ptr 所指向的 int 类型的内容可以修改，但 constant_ptr 本身应始终指向相同的位置。

4 常量指针 constant_ptr 的声明可以通过包含类型“指向 int 的指针”的定义来阐明。

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

声明 constant_ptr 为一个具有"指向 int 的 const 限定指针"类型的对象。

6.7.5.2. 数组声明符

约束条件

1 除了可选的类型限定符和关键字 static 之外，[和]还可以分隔表达式或*。如果它们分隔表达式(指定数组的大小)，则表达式应具有整数类型。如果表达式是常数表达式，它应该有一个大于零的值。元素类型不能是不完整或功能类型。可选的类型限定符和关键字 static 只能出现在具有数组类型的函数形参声明中，然后只能出现在最外层的数组类型派生中。

2 只有既具有代码块范围又具有功能原型范围且没有联动关系的普通标识符(如 6.2.3 所定义)才具有可变修改类型。如果一个标识符被声明为具有静态存储周期的对象，它不应该具有变长数组类型。

语义

3 如在“T D1”声明中，D1 有以下构成

```
D[ type-qualififier-listopt assignment-expressionopt ]
D[ static type-qualififier-listopt assignment-expression ]
D[ type-qualififier-list static assignment-expression ]
D[ type-qualififier-listopt * ]
```

在声明“T D”中为 ident 指定的类型是“派生声明符-类型-列表 T”，那么为 ident 指定的类型是“派生声明符-类型-列表 T 数组”121)。 (可选类型限定符和关键字 static 的含义见 6.7.5.3)。

121) 当多个“数组的”规范相邻时，将声明一个多维数组。

4 如果 size 不存在，则数组类型为不完整类型。如果 size 为*而不是表达式，则数组类型是未指定大小的变长数组类型，只能在具有函数原型作用域的声明中使用 122);尽管如此，这样的数组仍然是完整类型。如果 size 是一个整型常量表达式，且元素类型已知常量大小，则数组类型不是变长数组类型;否则，数组类型为变长数组类型。

122) 因此，*只能用于非定义的函数声明(参见 6.7.5.3)。

5 如果 size 是一个不是整型常量表达式的表达式:如果它出现在函数原型范围的声明中，它被视为被*替换;否则，每次对它进行评估时，它的值都应该大于零。可变长度数组类型的每个实例的大小在其生命周期内不会改变。size 表达式是 sizeof 操作符的操作数的一部分，改变 size 表达式的值不会影响操作符的结果，因此 size 表达式是否被求值是不确定的。

6 要使两种数组类型兼容，必须具有兼容的元素类型，如果两个 size 说明符都存在，且都是整数常量表达式，则两个 size 说明符应具有相同的常量值。如果两个数组类型在要求它们兼容的上下文中使用，如果两个 size 说明符计算出的值不相等，则为未定义行为。

7 例 1

```
float fa[11], *afp[17];
```

声明一个浮点数数组和一个浮点数指针数组。

8 例 2 注意声明之间的区别

```
extern int *x;
extern int y[];
```

第一个声明 x 是指向 int 的指针;第二个声明 y 是一个未指定大小(不完整类型)的 int 数组，其存储空间在其他地方定义。

9 例 3 下面的声明演示了可变修改类型的兼容性规则。

```
extern int n;
extern int m;
void fcompat(void)
{
    int a[n][6][m];
    int (*p)[4][n+1];
    int c[n][n][6][m];
    int (*r)[n][n][n+1];
    p = a; // 无效:不兼容，因为 4 != 6
    r = c; // 兼容，但仅当
```

```
    // n == 6 and m == n+1
}
```

10 例 4 所有可变修改(VM)类型的声明必须在块范围或函数原型范围内。使用静态或外部存储类说明符声明的数组对象不能具有变长数组(VLA)类型。但是，使用静态存储类说明符声明的对象可以具有可变修改类型(即指向变长数组类型的指针)。最后，用可变修改类型声明的所有标识符必须是普通标识符，因此不能是结构或联合的成员。

```
extern int n;
int A[n]; // 无效:文件范围内的 VLA
extern int (*p2)[n]; // 无效:文件范围内的 VM
int B[100]; // 有效：文件范围内但不是 VM
void fvla(int m, int C[m][m]); // 有效 VLA 具有原型范围
void fvla(int m, int C[m][m]) // 有效: 调整为自动指针指向 VLA
{
    typedef int VLA[m][m]; // 有效:块作用域类型定义 VLA
    struct tag {
        int (*y)[n]; // 无效: y 不是普通的标识符
        int z[n]; // 无效: z 不是普通的标识符
    };
    int D[m]; // 有效: 自动 VLA
    static int E[m]; // 无效: 静态代码块范围内的 VLA
    extern int F[m]; // 无效: F 有链接且是 VLA
    int (*s)[m]; // 有效:自动指向 VLA
    extern int (*r)[m]; // 无效: r 有链接且指向 VLA
    static int (*q)[m] = &B; // 有效: q 是一个静态代码块指向 VLA
}
```

前向引用:函数声明符(6.7.5.3)、函数定义(6.9.1)、初始化(6.7.8)。

6.7.5.3 函数声明符(包括原型)

约束条件

1 函数声明符不能指定返回类型为函数类型或数组类型。

2 唯一应该出现在参数声明中的存储类型说明符是 register。

3 函数声明符中不属于该函数定义的标识符列表应为空。

4 调整后，函数声明符中作为该函数定义一部分的形参类型列表中的形参不能具有不完整类型。

语义

5 如在“T D1”声明中，D1 有以下构成

D(parameter-type-list)

或

D(identifiifier-listopt)

在声明“T D”中为 ident 指定的类型是“派生声明符-类型-列表 T”，然后为 ident 指定的类型是“派生声明符-类型-列表函数返回 T”。

6 形参类型列表指定函数形参的类型，并可以声明其标识符。

7 将形参声明为“类型数组”应调整为“限定类型指针”，其中类型限定符(如果有的话)是在数组类型派生的[和]中指定的。如果关键字 static 也出现在数组类型派生的[和]中，那么对于每次调用该函数，对应的实际参数值应提供对数组第一个元素的访问，该元素的数量至少与 size 表达式指定的元素数量相同。

8 将形参声明为“函数返回类型”应调整为“指向函数返回类型的指针”，如 6.3.2.1 所示。

9 如果列表以省略号(，…)结束，则不提供关于逗号后参数的数量或类型的信息。[123)]

123) 在<stdarg.h>头文件(7.15)中定义的宏可用于访问与省略号对应的参数。

10 未命名形参类型为 void(列表中唯一项)的特殊情况指定函数没有形参。

11 在形参声明中，圆括号中的单个 typedef 名称被当作一个抽象声明符，用于指定带有单个形参的函数，而不是作为声明符标识符周围的冗余圆括号。

12 如果函数声明符不是该函数定义的一部分，则形参的类型可能不完整，并可以在声明符说明符序列中使用[*]表示法来指定变长数组类型。

13 形参声明的声明说明符中的存储类型说明符如果存在，将被忽略，除非声明的形参是函数定义的形参类型列表的成员之一。

14 标识符列表只声明函数形参的标识符。作为该函数定义的一部分的函数声明符中的空列表指定该函数没有形参。函数声明符中不属于该函数定义的空列表指定不提供有关形参数目或类型的信息。[124)]

124) 参见“未来语言方向”(6.11.6)。

15 对于两个要兼容的函数类型，两者都应指定兼容的返回类型[125)]。此外，如果两者都存在，则参数类型列表应在参数的数量和省略号终止符的使用上达成一致;相应的参数应具有兼容的类型。如果一种类型有形参类型列表，而另一种类型由函数声明符指定，该声明符不属于函数定义，且包含一个空标识符列表，则形参列表不应有省略结束符，并且每个形参的类型应与应用默认实参提升产生的类型兼容。如果一种类型具有形参类型列表，而另一种类型由包含标识符列表(可能为空)的函数定义指定，则两者应在形参的数量上达成一致，并且每个原型形参的类型应与应用默认实参提升到相应标识符的类型所产生的类型兼容。(在确定类型兼容性和复合类型时，使用函数或数组类型声明的每个形参都被认为具有调整后的类型，而使用限定类型声明的每个形参都被认为具有其声明类型的非限定版本。)

125) 如果两个函数类型都是“旧类型”，则不比较形参类型。

16 例 1 声明

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

声明一个不带形参的函数 f 返回一个 int，一个不带形参说明的函数 fip 返回一个指向 int 的指针，一个指向不带

形参说明的函数的指针 pfi 返回一个 int。比较后两者尤其有用。*fip()的绑定是*(fip())，因此声明建议调用函数 fip，然后通过指针结果使用间接方法来生成一个 int，而在表达式中同样的构造也需要这样做。在声明符(*pfi)()中，额外的圆括号是必要的，以表明通过指向函数的指针间接产生一个函数指示符，然后用于调用函数;它返回一个 int。

17 如果声明发生在任何函数之外，则标识符具有文件作用域和外部链接。如果声明发生在函数内部，函数 f 和 fip 的标识符具有代码块作用域，并且有内部或外部链接(取决于这些标识符的文件作用域声明是可见的)，而指针 pfi 的标识符具有代码块作用域，并且没有链接。

18 例 2 声明

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

声明一个数组 apfi，其中包含三个指向返回 int 的函数的指针。每个函数都有两个形参，它们是指向 int 的指针。标识符 x 和 y 的声明仅用于描述性目的，在 apfi 声明结束时就超出了作用域。

19 例 3 声明

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

声明一个函数 fpfi，该函数返回指向返回 int 类型的函数的指针。fpfi 函数有两个形参:一个是指向返回 int 类型的函数的指针(其中一个形参为长 int 型)，另一个是 int 型。fpfi 返回的指针指向一个函数，该函数有一个 int 形参，并接受 0 个或多个任意类型的附加实参。

20 例 4 下面的原型有一个可变修改的参数。

```
void addscalar(int n, int m,
double a[n][n*m+300], double x);
int main()
{
    double b[4][308];
    addscalar(4, 2, b, 2.17);
    return 0;
}
void addscalar(int n, int m,
double a[n][n*m+300], double x)
{
    for (int i = 0; i < n; i++)
    for (int j = 0, k = n*m+300; j < k; j++)
    // a 是一个指向具有 n*m+300 个元素 VLA 的指针
    a[i][j] += x;
}
```

21 例 5 以下是所有兼容的函数原型声明符。

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

使用方式如下：

```
void f(double (* restrict a)[5]);
void f(double a[restrict][5]);
void f(double a[restrict 3][5]);
void f(double a[restrict static 3][5]);
```

(注意，最后一个声明还指定，在对 f 的任何调用中，对应于 a 的实参必须是指向至少三个由 5 个 double 组成的数组中的第一个的非空指针，其他数组则不是这样。)

前向引用:函数定义(6.9.1)、类型名称(6.7.6)。

6.7.6. 类型名称

语法

1.类型名称：**specififier-qualififier-list abstract-declaratoropt**

抽象声明符：**pointer**

pointeropt direct-abstract-declarator

直接抽象声明符（抽象声明符）：

direct-abstract-declaratoropt [assignment-expressionopt]
direct-abstract-declaratoropt [*]
direct-abstract-declaratoropt (parameter-type-listopt)

语义

2.在多个上下行中，必须指定类型，这是使用类型完成的名称，在语法上是函数或该类型的对象的声明，该类型是省略标识符。[126)]

126） 如语法所示，类型名称中的空括号被解释为"函数，没有参数规范"，而不是省略的标识符周围的冗余括号。

3.举例：

```
(h)    int (*const [])(unsigned int, ...)
```

结构分别命名类型（a）int，（b）指向 int 的指针，（c）指向 int 的三个指针数组，（d）指向三个整数的数组，（e）指向未指定数量的整数的可变长度数组的指针，（f）功能没有参数规范返回指向 int 的指针，（g）指向没有参数的函数的指针返回一个整型，（h）一个未指定数量的函数常量指针的数组，每个函数都有一个具有无符号 int 类型和未指定数量的其他参数的参数，返回 int。

6.7.7. 类型定义

语法

1. 类型名称：**identififier**

约束

2. 如果定义类型名称指定了可变修改的类型，则它应该具有块范围。

语义

3. 在存储类说明符为定义类型的声明中，每个声明符将一个标识符定义为定义类型名称，该名称表示以 6.7.5 中描述的方式为标识器指定的类型。每次按执行顺序到达定义类型名称的声明时，都会计算与可变长度数组声明符关联的任何数组大小表达式。类型定义声明不会引入新类型，仅是如此指定的类型的同义词。也就是说，在以下声明中：

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

type_ident 定义为具有声明指定的类型的类型定义名称 T 中的说明符（称为 T）和 D 中的标识符具有类型“派生声明符类型列表 T”，其中派生声明符类型列表由 D 的声明符指定。一个 typedef 名称与普通中声明的其他标识符共享相同的名称空间声明符。

4. 举例一：在

```
typedef int MILES, KLICKSP();
typedef struct { double hi, lo; } range;
```

之后的结构中，

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

都是有效的声明。距离的类型是 int，metricp 的类型是“指向函数的指针”，没有参数规范返回 int"，x 和 z 的参数规范是指定的结构;zp 是指向这样的结构。对象距离具有与任何其他整型对象兼容的类型。

5. 举例二：在声明后

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

类型 t1 和 tp1 所指向的类型是兼容的。t1 型也与结构型兼容 s1，但与类型 s2、t2、tp2 所指向的类型或 int 不兼容。

6. 举例三：以下晦涩难懂的结构

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

声明一个带有类型符号 int 的定义类型名称 t、一个带有类型 int 的定义类型名称 plain 和一个结构具有三个位字段成员，一个名为 t 的包含范围 [0，15]中的值，一个未命名的常量限定位字段（如果可以访问）将包含范围[−15， +15]中的值，或者[−16， +15]，以及一个名为 r 的值，其中包含其中一个范围[0，31]、[−15，+15]或[−16，+15]中的值。（范围的选择是实现定义的。）前两个位字段声明的不同之处在于 unsigned 是一个类型说明符（强制 t 是结构成员的名称），而常量是一个类型限定符（用于修改仍作为定义类型名称可见的 t）。如果遵循这些声明在内部范围中

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

则使用类型 "声明函数 f，该函数返回带有一个未命名参数的有符号 int 使用类型指针指向函数返回带符号的有符号 int，其中包含一个带有类型符号的未命名参数 int"，以及类型为 long int 的标识符 t。

7. 举例四：另一方面，定义类型名称可用于提高代码的可读性。所有三个信号函数的以下声明指定完全相同的类型，第一个不使用任何类型定义名称。

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

8. 举例五：如果定义类型名称表示一个可变长度的数组类型，那么数组的长度在定义类型名称时是固定的而不是在每次使用的时候。

```
void copyt(int n)
{
typedef int B[n]; // B is n ints, n evaluated now
n += 1;
B a; // a is n ints, n without += 1
Int b[n]; // a and b are different sizes
for (int i = 1; i < n; i++)
a[i-1] = b[i];
}
```

6.7.8. 初始化

语法

1. 初始化：

```
initializer:
    assignment-expression
    { initializer-list }
    { initializer-list , }
initializer-list:
    designationopt initializer
    initializer-list , designationopt initializer
designation:
    designator-list =
designator-list:
    designator
    designator-list designator
designator:
    [ constant-expression ]
    . Identifiifier
```

约束

2. 任何初始值设定项都不得尝试为实体中未包含的对象提供值正在初始化。
3. 要初始化的实体的类型应为大小未知的数组或对象类型不是可变长度数组类型。
4. 具有静态存储持续时间的对象的初始值设定项中的所有表达式应为常量表达式或字符串文本。
5. 如果标识符的声明具有块范围，并且标识符具有外部或内部联动，声明中应没有初始值设定项作为标识符。
6. 如果指示符具有以下形式

[constant-expression]

则当前对象（定义如下）应具有数组类型，表达式应为一个整数常量表达式。如果数组大小未知，则任何非负值为有效。

7. 如果指示符具有以下形式

. Identififier

则当前对象（定义如下）应具有结构或并集类型，并且标识符应为该类型成员的名称。

语义

8. 初始值设定项指定存储在对象中的初始值。

9. 除非另有明确说明，否则就本从句而言，未命名结构和联合类型的对象的成员不参与初始化。即使在初始化后，结构对象的未命名成员也具有不确定的值。

10. 如果具有自动存储持续时间的对象未显式初始化，则其值为定。如果具有静态存储持续时间的对象未显式初始化，然后：

--如果它具有指针类型，则将其初始化为空指针；

--如果它具有算术类型，则将其初始化为（正或无符号）零；

--如果它是聚合，则根据这些规则（递归）初始化每个成员；

--如果它是一个联合，则根据这些条件初始化（递归）第一个命名成员规则。

11. 标量的初始值设定项应为单个表达式，可选择括在大括号中。这对象的初始值是表达式的初始值（转换后）；相同类型适用于简单赋值的约束和转换，采用标量的类型成为其声明类型的非限定版本。

12. 此子句的其余部分处理具有聚合或并集的对象的初始值设定项类型。

13. 具有自动存储持续时间的结构或联合对象的初始值设定项应为如下所述的初始值设定项列表，或具有兼容的单个表达式结构或联合类型。在后一种情况下，对象的初始值，包括未命名的成员，是表达式。

14. 字符类型的数组可以由字符串文本初始化，也可以选择括在大括号中。字符串文本的连续字符（包括如果存在空间或数组大小未知，则终止空字符）初始化元素。

15. 元素类型与 wchar_t 兼容的数组可以通过宽字符串文字，可选择括在大括号中。宽字符串的连续宽字符文字（包括终止空宽字符，如果有空间或数组是未知大小）初始化数组的元素。

16. 否则，具有聚合或联合类型的对象的初始值设定项应为元素或命名成员的初始值设定项的大括号封闭列表。

17. 每个大括号括起来的初始值设定项列表都有一个关联的当前对象。当没有指定存在，当前对象的子对象按顺序初始化到当前对象的类型：按递增的下标顺序排列元素，结构成员按声明顺序排列，以及联合的第一个指定成员。相反，一个指定导致以下初始值设定项开始初始化子对象由指示符描述。然后，初始化按顺序继续，开始在指定符描述的子对象之后的下一个子对象。

18. 每个指示符列表都以与最近周围的大括号对。指示符列表中的每个项目（按顺序）指定一个其当前对象的特定成员，并将当前对象更改为下一个对象指示符（如果有）为该成员。在指示符清单是要由以下初始值设定项初始化的子对象。

19. 初始化应按初始值设定项列表顺序进行，每个初始值设定项为特定子对象覆盖同一子对象的任何先前列出的初始值设定项;都未显式初始化的子对象应隐式初始化，与具有静态存储持续时间的对象。

20. 如果聚合或联合包含作为聚合或联合的元素或成员，这些规则以递归方式应用于子聚合或包含的联合。如果初始值设定项子聚合或包含的并集以左大括号开头，初始值设定项由该大括号及其匹配的右大括号初始化子聚合或包含的并集。否则，列表中只有足够多的初始值设定项考虑子类集合的元素或成员或第一个成员包含的联盟;任何剩余的初始值设定项都留给初始化下一个元素或当前子聚合或包含的并集是其一部分的聚合的成员。

21. 如果大括号括起来的列表中的初始值设定项少于元素或成员的聚合，或字符串文本中用于初始化已知数组的较少字符大小大于数组中有元素，聚合的其余部分应为隐式初始化与具有静态存储持续时间的对象相同。

22. 如果初始化了未知大小的数组，则其大小由最大索引确定元素中具有显式初始值设定项。在其初始值设定项列表的末尾，数组不再具有不完整的类型。

23. 任何副作用在初始化列表表达式中发生的顺序为未指定。

24. 举例一 如果<complex.h>已被#included，则声明

```
int i = 3.5;
complex c = 5 + 3 * I;
```

定义并初始化 i 与值 3 和 c 与值 5.0+i3.0。

25. 举例二 声明

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

定义 x 并将其初始化为具有三个元素的一维数组对象，因为未指定大小
并且有三个初始值设定项。

26. 举例三 声明

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

是具有完全括弧初始化的定义：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，接下来的两行初始化 y[1] 和 y[2]初始值设定项提前结束，因此 y[3] 用零初始化。完全相同的效果可能具有由以下方式实现

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

y[0] 的初始值设定项不以左大括号开头，因此使用列表中的三个项目。同样

接下来的三个依次取 y[1] 和 y[2]。

27. 举例四 声明

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

初始化指定的 z 的第一列，并用零初始化其余列。

28. 举例五 声明

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

是具有不一致的括号初始化的定义。它定义了一个具有两个元素的数组结构：w[0].a[0]是 1 和 w[1].a[0]是 2;所有其他元素均为零。

29. 举例六 声明

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

包含不完整但始终用括号括起来的初始化。它定义了一个三维数组对象：q[0][0][0]为 1，q[1][0][0]为 2，q[1][0][1]为 3，4、5 和 6 初始化 q[2][0][0]、q[2][0][1]和 q[2][1][0];其余的都是零。的初始值设定项 q[0][0]不以左大括号开头，因此最多可以使用当前列表中的六个项目。有只有一个，因此，其余五个元素的值初始化为零。同样，初始值设定项对于 q[1][0]和 q[2][0]不以左大括号开头，因此每个都使用最多六个项目，初始化它们的各自的二维子聚合器。如果任何清单中有六个以上的项目，则将发出诊断信息。相同的初始化结果可以通过以下方式实现：

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

或通过以下方式：

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

以完全括起来的形式。

30. 请注意，通常，完全括弧和最小括弧的初始化形式不太可能引起混淆。

31. 举例七 完成数组类型的一种初始化形式涉及定义类型名称。鉴于声明

```
typedef int A[]; // OK - declared with block scope
```

声明时

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

与

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

取决于不完整类型的规则。

32. 举例八 声明

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

定义"plain"char 数组对象 s 和 t，其元素使用字符串文本初始化。此声明与

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

数组的内容是可修改的。另一方面，声明

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

使用类型“指向 char 的指针”定义 p，并将其初始化为指向类型为“char 数组”的对象长度为 4，其元素使用字符串文本初始化。如果尝试使用 p 修改数组的内容，行为未定义。

33. 举例九 可以使用以下命令初始化数组以对应于枚举的元素指示符：

```
enum { member_one, member_two };
const char *nm[] = {
    [member_two] = "member two",
    [member_one] = "member one",
};
```

34. 举例十 结构成员可以初始化为非零值，而不依赖于它们的顺序：

```
div_t answer = { .quot = 2, .rem = -1 };
```

35. 举例十一 指示符可用于在未修饰的初始值设定项列表时提供显式初始化可能被误解：

```
struct { int a[3], b; } w[] =
    { [0].a = {1}, [1].a[0] = 2 };
```

36. 举例十二 可以使用单个指示符从数组的两端“分配”空间：

```
int a[MAX] = {
    1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

37. 在上面，如果 MAX 大于十，中间会有一些零值元素;如果它更少如果超过 10 个，则前五个初始值设定项提供的某些值将被后五个初始值设定项覆盖。

38. 举例十三 可以初始化联合的任何成员：

```
union { /* ... */ } u = { .any_member = 42 };
Forward references: common defifinitions <stddef.h>
```

6.8. 语句和块

语法

```
    statement:
labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
```

jump-statement

语义

1. 语句指定要执行的操作。除另有说明外，声明是按顺序执行。

2. 块允许将一组声明和语句分组到一个语法单元中。具有自动存储持续时间和可变长度的对象的初始值设定项具有块作用域的普通标识符的数组声明符，被评估，并且值为存储在对象中（包括在对象中存储不确定值，而不使用初始值设定项））每次按执行顺序到达声明时，就好像它是语句，并在每个声明中按声明符出现的顺序排列。

3. 完整表达式是不属于另一个表达式或声明符的表达式。以下每个都是一个完整的表达式：一个初始值设定项;表达式中的表达式语句;选择语句的控制表达式（if 或 switch）;这控制一段时间的表达或做语句;的每个（可选）表达式 for 代表声明;return 语句中的（可选）表达式。完整结束表达式是一个序列点。

前导：表达式和空语句（6.8.3），选择语句（6.8.4）、迭代语句（6.8.5）、return 语句（6.8.6.4）。

6.8.1. 带标签语句

语法

1. **labeled-statement:**
identifiifier : statement
case constant-expression : statement
default : statement

2. 大小写或默认标签应仅出现在 switch 语句中，进一步在 switch 语句下讨论了对此类标签的约束。

3. 标签名称在函数中应是唯一的。

语义

4. 任何语句的前面都可以有一个前缀，该前缀将标识符声明为标签名称。标签本身不会改变控制流程，控制流程继续畅通无阻地跨越他们。

前导：goto 语句 6.8.6.1）、switch 语句（6.8.4.2）。

6.8.2. 复合语句

语法

1. **compound-statement:**
{ block-item-listopt }
block-item-list:
block-item
block-item-list block-item

block-item:
declaration
Statement

语义

2. 复合语句是一个块。

6.8.3. 表达式和空语句

语法

expression-statement:
expressionopt ;

语义

5. 表达式语句中的表达式将计算为其端的 void 表达式影响。

6. 空语句（仅由分号组成）不执行任何操作。

7. 举例一 如果函数调用仅作为表达式语句计算其副作用，则放弃其值可以通过将表达式转换为 void 表达式来显式内容：

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

8. 举例二 在程序片段中

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

9. null 语句用于为迭代语句提供空循环体。

10. 举例三 null 语句也可用于在化合物的结束 } 之前携带标签陈述。

```
while (loop1) {
/* ... */
while (loop2) {
/* ... */
if (want_out)
goto end_loop1;
/* ... */
```

```
}
/* ... */
end_loop1: ;
}
```

前导：迭代语句（6.8.5）。

6.8.4. 选择声明

语法

selection-statement:
if (expression) statement
if (expression) statement else statement
switch (expression) statement

语义

1. 选择语句根据控制表达式。

2. 选择语句是一个块，其作用域是其作用域的严格子集封闭块。每个关联的子语句也是一个块，其范围是严格的选择语句作用域的子集。

6.8.4.1. if 语句

约束

1. if 语句的控制表达式应具有标量类型。

2. 在这两种形式中，如果表达式与 0 的比较不相等，则执行第一个子语句。在 else 窗体中，如果表达式比较等于 0，则执行第二个子语句。如果第一个子语句是通过标签访问的，则第二个子语句不是执行。

3. 如果语法允许，则 else 与词汇上最近的前面相关联。

6.8.4.2. switch 语句

约束

1 switch 语句的控制表达式应当为整数类型。

2 如果一个 switch 语句在具有可变修改类型的标识符范围内具有关联的 case 或者 default 标签，那么整个 switch 语句应当在该标识符的范围内。132)

3 每个 case 标签的表达式应当为整数常量表达式，并且同一个 switch 语句中的任何两个 case 常量表达式在转换后的值不能相同。在 switch 语句中最多只能有一个 default 标签。（任何封闭的 switch 语句都可能有一个 default 标签或 case 常量表达式，其值与该封闭的 switch 语句中的 case 常量表达式相同。）

语义

4 根据控制表达式的值、default 标签的存在以及 switch 上任何 case 标签的值，switch 语句会控制跳转、进入或跳过 switch 主体的语句。case 或 default 标签只能在最近的封闭 switch 语句中访问。

5 整数的提升是在控制表达式上执行的。每个 case 标签中的常量表达式被转换成控制表达式的提升类型。如果转换后的值与提升的控制表达式的值相匹配，则控制跳转到匹配的 case 标签后面的语句。否则，如果有 default 标签，则控制跳转到 default 后面的语句。如果转换后的值不能与提升的控制表达式的值相匹配且没有 default 标签，则不执行 switch 主体的任何部分。

执行限制

6 在 5.2.4.1 节中讨论过，执行一个 switch 语句可能限制 case 值的数量。

132) 也就是说，声明要么在 switch 语句之前，要么在包含声明的块中的 switch 关联的最后一个 case 或 default 标签之后。

7 示例 在人写的程序片段中

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

标识符为 i 的对象具有自动存储期限（在块内），但从未初始化，因此如果控制表达式具有非零值，则调用 printf 函数将访问一个不确定的值。同样，无法调用函数 f。

6.8.5. 循环语句

语法

1

循环语句:

```
while ( 表达式 ) 语句
do 语句 while ( 表达式 ) ;
for ( 表达式（可选）; 表达式（可选） ; 表达式（可选） ) 语句
```

for (声明表达式（可选） ; 表达式（可选）) 语句

约束

2 循环语句的控制表达式必须有标量类型。

3 for 语句的声明部分只能声明具有 auto 或 register 这些存储类的对象的标识符。

语义

4 循环语句会导致重复执行被称为循环体的语句，直到控制表达式比较等于 0 为止。

5 循环语句是一个块，其作用域是其封闭块范围的严格子集。循环体也是一个块，其作用域范围为循环语句作用域的严格子集。

6.8.5.1. while 语句

1 控制表达式的计算在每次执行循环体之前进行。

6.8.5.2. do 语句

1 控制表达式的计算发生在每次执行循环体之后。

6.8.5.3. for 语句

1 例句

for (clause-1 ; expression-2 ; expression-3) statement
（子句 1；表达式 2；表达式 3）

就像下面所表示的那样：表达式 expression-2 是在每次执行循环体之前计算的控制表达式，表达式 expression-3 被认作是每次循环体执行后的 void 表达式。如果 clause-1 是一个声明，它声明的任何变量的作用域都是声明的剩余部分和整个循环，包括另外两个表达式；它在执行中到达的顺序是在控制表达式的第一次求值前。如果 clause-1 是一个表达式，那么将在控制表达式第一次进行求值前将其评估为 void 表达式。

2 clause-1 和 expression-3 都可以被省略。省略的 expression-2 可以被一个非零常数替代。

6.8.6. 跳转语句

语法

1 跳转语句:

goto 标识符;
continue ;

```
break ;
return 表达式 ;
```

语义

2 跳转语句会导致无条件跳转到另一个位置。

133) 因此，clause-1 规定了循环的初始化，可能声明一个或多个变量在循环中重复使用；控制表达式 expression-2 指定在每次循环之前求值，这样循环的执行就会继续，直到表达式的比较等于 0；而 expression-3 指定在每次循环之后执行操作（例如递增）。

6.8.6.1. goto 语句

约束

1 在 goto 语句中的标识符应命名在封闭函数中的某处的一个标签。goto 语句不应从具有可变修改类型的标识符的作用域外跳转到该标识符的作用域内。

语义

2 goto 语句会导致无条件跳转到以封闭函数中的命名标签作为前缀的语句。

3 示例 1 有的时候跳转到一组复杂的语句中间是很方便的。以下概述基于这三个假设提出了一种可能的解决问题的方法：

1. 通过初始化代码访问仅对当前函数可见的对象。
2. 通常的初始化代码太大以至于不允许重复。
3. 确定下一个可操作的代码位于循环的开头。（例如，允许通过 continue 语句访问它。）

```
/* ... */
goto first_time;
for (;;) {
    // 确定下一个操作
    /* ... */
    if (需要重新初始化) {
        // reinitialize-only 代码（重新初始化代码）
        /* ... */
        first_time:
        // 一般初始化代码
        /* ... */
        continue;
    }
    // 处理其他操作
    /* ... */
}
```

4 示例 2 一条 goto 语句不允许跳过任何具有可变修改类型的对象声明 。但是，允许在范围内进行跳转。

```
goto lab3;              // 无效：跳转进入长度可变的数组范围
{
    double a[n];
    a[j] = 4.4;
    lab3:
    a[j] = 3.3;
    goto lab4;          // 有效：在长度可变的数组范围内跳转
    a[j] = 5.5;
    lab4:
    a[j] = 6.6;
}
goto lab4;              // 无效：跳转进入长度可变的数组范围
```

6.8.6.2. continue 语句

约束

1 一条 continue 语句只能出现在循环体中或作为循环主体出现。

语义

2 一条 continue 语句导致跳转到最小的封闭循环语句的循环延续部分；即到达循环体的末尾。更准确地说，在每一个语句中

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

```
}
```

除非 continue 语句显示在封闭的循环语句中（在这种情况下，它在该语句中进行解释），它等价于 goto contin134)；

134) 在 contin 之后：标签是一个空的声明语句

6.8.6.3. break 语句

约束

1 一条 break 语句应仅出现在或作为 switch 主体或循环主体出现。

语义

2 一条 break 语句终止执行最小的封闭 switch 或循环语句的执行。

6.8.6.4. return 语句

约束

1 带有表达式的 return 语句不应出现在返回类型为 void 的函数中。妹有表达式的 return 语句只能出现在返回类型为 void 的函数中。

语义

2 return 语句终止当前函数的执行，并将控制权返回给它的调用者。一个函数可以有任意数量的 return 语句。

3 如果执行带有表达式的 return 语句，那么表达式的值将作为函数调用表达式的值返回给调用者。如果表达式的类型与它所出现的函数的返回类型不同，则该值将被转换，就好像是通过对具有该函数的返回类型的对象赋值一样。135)

4 示例 如下：

```
struct s { double i; } f(void);
union {
    struct {
        int f1;
        struct s f2;
    } u1;
    struct {
        struct s f3;
        int f4;
    } u2;
} g;
struct s f(void)
{
```

```
    return g.u1.f2;
}
/* ... */
g.u2.f3 = f();
```

没有未定义的行为，尽管如果赋值是直接完成的（不使用函数调用来获取值）。

135) return 语句不是赋值语句。第 6.5.16.1 小节的重叠限制不适用于函数返回的情况。

6.9. 外部定义

语法

1 翻译单元:

外部声明

翻译单元 外部声明

外部声明:

函数定义

声明

约束

2 存储类说明符 auto 和 register 不应出现在外部声明的声明说明符中。

3 对于在翻译单元中用内部链接声明的每个标识符，不应有超过一个的外部定义。此外，如果用内部链接的声明的标识符在表达式中使用（而不是作为结果为整数常量的 sizeof 运算符操作数的一部分），则该标识符在翻译单元中必须只有一个外部定义。

语义

4 如 5.1.1.1 所述，预处理后的程序文本单元是一个翻译单元，它由一系列外部声明组成。这些声明被描述为“外部”，因为它们出现在任何函数之外（因此具有文件范围）。正如 6.7 中所讨论的，如果一个声明会导致为标识符命名的对象或函数保留存储空间，则将其称为定义。

5 外部定义是一种外部声明，它同时也是函数（内联定义除外）或对象的定义。如果使用外部链接声明的标识符在表达式中使用（而不是作为结果为整数常量的 sizeof 运算符操作数的一部分），则在整个程序的某个地方，该标识符必须只有一个外部定义；否则，不得超过一个。136)

136) 因此，如果用外部链接声明的标识符未在表达式中使用，则不需要对其进行外部定义。

6.9.1. 函数定义

语法

1 函数定义:

声明说明符 声明符 声明列表（可选） 复合语句

声明列表:

声明符

声明列表声明

约束

2 在函数定义中声明的标识符（即函数的名称）应当有一个函数类型，由函数定义的声明符的部分来指定。137)

3 函数的返回类型应为 void 或非数组类型的对象类型。

4 声明说明符中的存储类说明符（如果存在的话）应为 extern 或 static。

5 如果声明符包含参数类型列表，则每个参数的声明都应包含一个标识符，但参数列表由单个 void 类型的参数组成的特殊情况除外，这种情况下不应有标识符。不得附有声明列表。

6 如果声明符包含标识符列表则声明列表中的每个声明都应至少有一个声明符，这些声明符应仅声明标识符列表中的标识符，并且应声明标识符列表中的每个标识符。声明为 typedef 名称的标识符不应被重新声明为参数。声明列表中的声明不应包含除寄存器和初始化以外的任何存储类说明符。

137) 目的是使得函数定义中的类型类别不能从 typedef 继承：

```
typedef int F(void);          // F 类型是无参数返回 int 的函数
F f, g;                       // f 和 g 的类型都与 F 兼容
F f { /* ... */ }             // 错误：语法/约束错误
F g() { /* ... */ }           // 错误：声明了 g 返回一个函数
int f(void) { /* ... */ }     // 正确：f 的类型与 F 兼容
int g() { /* ... */ }         // 正确：g 的类型与 F 兼容
F *e(void) { /* ... */ }      // e 返回一个指向函数的指针
F *((e))(void) { /* ... */ }  // 同上：括号无关
int (*fp)(void);              // fp 指向类型为 F 的函数
F *Fp;                        // Fp 指向类型为 F 的函数
```

语义

7 函数定义中的声明符指定了所定义函数的名称及其形参的标识符。如果声明符包含参数类型列表，则该列表还指定了所有参数的类型；这样的声明符还可以作为函数的原型，用于以后在同一个翻译单元中调用同一个函数。如果声明符包含一个标识符列表，138)那么参数的类型应当在下面的声明列表中进行声明。在任何一种情况下，每个参数的类型都会按照 6.7.5.3 中所描述的对参数类型列表的描述进行调整；产生的类型应为对象类型。

8 如果一个接受可变数量参数的函数没有定义以省略号符号结尾的形参类型列表，则该行为是未定义的。

9 每个参数都有自动存储时长，它的标识符是一个左值，它实际上是在构成函数体的复合语句的开头声明的（因此不能再函数体中重新声明，除非在一个封闭的块中）。参数的存储布局未指定。

10 在进入函数时，计算每个经过变量修改的形参的 size 表达式，并将每个实参表达式的值转换为对应形参的类型，就像通过赋值一样。（数组表达式和作为参数的函数指示符在调用之前被转换为指针。）

11 在赋值了所有的参数之后，将执行构成函数定义主题的复合语句。

12 如果到达了终止函数的}，并且调用者使用了函数调用的值，则该行为是未定义的。

13 示例 1 如下：

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

extern 是存储类说明符，int 是类型说明符；max(int a, int b)是函数声明符；同时

```
{ return a > b ? a : b; }
```

是函数体。以下类似的定义使用了标识符-列表的形式来声明参数。

138) 详见“未来语言方向” (6.11.7).

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

这里的 int a, b;是参数的声明列表。这两种定义的区别在于，第一种形式充当原型声明，强制类型转换函数后续调用函数的参数，而第二种形式则没有。

14 示例 2 将一个函数传递给另一个函数，可能会写

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

那么 g 的定义可以读作

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

或者，等效地，

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

6.9.2. 外部对象定义

语义

1 如果对象标识符的声明具有文件作用域和初始化程序，则该声明是标识符的外部定义。

2 对于具有文件作用域但没有初始化程序、没有存储类说明符或只有存储类说明符 static 的对象，声明标识符构成了暂定定义。如果一个翻译单元包含一个或多个标识符的暂定定义，而翻译单元不包含该标识符的外部定义，那么行为就像翻译单元包含该标识符的文件作用域声明一样，在翻译单元的末尾使用复合类型，则初始化式等于 0；

3 如果一个对象的标识符的声明是一个暂定的定义，并且具有内部链接，则声明的类型不应是一个不完整的类型。

4 示例 1

```
int i1 = 1;          // 定义，外部链接
static int i2 = 2;   // 定义，外部链接
extern int i3 = 3;   // 定义，外部链接
int i4;              // 暂定定义，外部链接
static int i5;       // 暂定定义，内部链接
int i1;               // 有效暂定定义，指向前一个
int i2;              // 6.2.2 呈现的未定义的链接分歧
int i3;              // 有效暂定定义，指向前一个
int i4;              // 有效暂定定义，指向前一个
int i5;              // 6.2.2 呈现的未定义的链接分歧
extern int i1;       // 引用前一个外部链接
extern int i2;       // 引用前一个内部链接
extern int i3;       // 引用前一个外部链接
extern int i4;       // 引用前一个外部链接
extern int i5;       // 引用前一个内部链接
```

5 示例 2 如果在翻译单元的末尾包含

```
int i[];
```

数组 i 仍然具有不完全类型，隐式初始化程序使它有一个元素，在程序启动时该元素被设为零。

6.10. 预处理指令

语法

1.

```
preprocessing-file:
    groupopt
group:
    group-part
    group group-part
group-part:
    if-section
    control-line
    text-line
    # non-directive
if-section:
    if-group elif-groupsopt else-groupopt endif-line
    if-group:
    # if       constant-expression new-line groupopt
    # ifdef    identifiifier new-line groupopt
    # ifndef identifiifier new-line groupopt
elif-groups:
    elif-group
    elif-groups elif-group
elif-group:
    # elif     constant-expression new-line groupopt
else-group:
    # else     new-line groupopt
endif-line:
    # endif    new-line
control-line:
    # include pp-tokens new-line
    # define   identifiifier replacement-list new-line
    # define   identifiifier lparen identifiifier-listopt ) replacement-list new-line
    # define   identifiifier lparen ... )   replacement-list new-line
    # define   identifiifier lparen identifiifier-list , ... ) replacement-list new-line
    # undef    identifiifier new-line
    # line     pp-tokens new-line
    # error    pp-tokensopt new-line
    # pragma   pp-tokensopt new-line
    #          new-line
text-line:
    pp-tokensopt new-line
non-directive:
```

pp-tokens new-line

lparen:

a (character not immediately preceded by white-space

replacement-list:

pp-tokensopt

pp-tokens:

preprocessing-token

pp-tokens preprocessing-token

new-line:

the new-line character

描述

2. 预处理指令由一系列预处理标记组成，以 # 预处理标记开始(在翻译阶段 4 的开始)，该标记要么是源文件中的第一个字符(可选在不包含换行字符的空白之后)，要么在空白之后至少包含一个换行字符，139)换行符结束预处理指令，即使它发生在调用类函数宏的过程中。

139)因此，预处理指令通常被称为“行”。这些“行”没有其他的语法重要性，因为所有空白都是等价的，除非在预处理的某些情况下(参见例如，6.10.3.2 中的 # 字符串字面量创建操作符)

3. 文本行不能以#预处理标记开始。非指令不能以语法中出现的任何指令名开头。

4. 当在一个被跳过的组中(6.10.1)，指令语法被放宽，允许在指令名和新行字符之间出现任何预处理标记序列。

约束

4. 在预处理指令中出现在预处理标记之间(从引入 # 预处理令牌之后到终止新行字符之前)应出现的唯一空白字符是空格和水平制表符(包括在翻译阶段 3 中替换注释或其他空白字符的空格）。

语义

6. 该实现可以有条件地处理和跳过源文件的各个部分，包括其他源文件，并替换宏。这些功能被称为预处理，因为从概念上讲，它们发生在结果翻译单元的翻译之前。

7. 除非另有说明，否则预处理指令中的预处理标记不受宏扩展的影响。

8. 例子:

```
#define EMPTY
EMPTY # include <file.h>
```

第二行上的预处理标记序列不是一个预处理指令，因为它在转换阶段 4 的开始时以 # 开始，即使它将在替换宏空之后这样做。

6.10.1. 条件包含

约束

1. 控制条件包含的表达式应为整数常数表达式，但除外：它不包含转换；标识符（包括与关键词相同的词汇）解释如下；140)，它可以包含表单的一元运算符表达式

defined identififier

or

defined (identififier)

如果标识符当前定义为宏名(也就是说，如果它是预定义的，或者它是#define 预处理指令的主题，而没有插入具有相同主题标识符的#undef 指令)，则该值为 1，如果不是，则为 0。

140) 因为控制常量表达式是在翻译阶段 4 计算的，所以所有标识符要么是宏名称，要么不是宏名称——没有关键字、枚举常量等。

语义

2 . 对表单指令进行预处理

 # if constant-expression new-line groupopt
 # elif constant-expression new-line groupopt

在求值之前，将成为控制常量表达式的预处理标记列表中的宏调用将被替换(由 define 的一元操作符修改的宏名称除外)，就像在普通文本中一样。如果所定义的 difine 是这个替换过程的结果，或者所 define 的一元操作符的使用与宏替换之前指定的两种形式中的一种不匹配，则该行为未定义。在执行了所有由宏展开和 define 的一元操作符引起的替换之后，所有剩余的标识符将被替换为 pp-number 0，然后将每个预处理标记转换为一个标记。产生的令牌组成了控制常量表达式，该表达式根据 6.6 的规则求值，除了所有有符号整数类型和所有无符号整数类型的行为好像它们分别具有与头文件 <stdint.h> ;中定义的类型 intmax_t 和 uintmax_t 相同的表示形式。这包括解释字符常量，这可能涉及将转义序列转换为执行字符集成员。这些字符常量的数值是否与在表达式中(除 #if 或 #elif 指令内)出现相同字符常量时获得的值匹配是由实现定义的。此外，单个字符常量是否可能有负值也是由实现定义的。

4 .对表单指令进行预处理

 # ifdef identififier new-line groupopt
 # ifndef identififier new-line groupopt

检查该标识符当前是否定义为宏名称。它们的条件分别相当于 #if define 的标识符和 #if !define 的标识符。

141) 因此，下面的 #if 指令和 if 语句中的常量表达式在这两种上下文中不能保证计算出相同的值。

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

5.每个指令的条件都按顺序检查。如果它的计算结果为 false(0)，则跳过它所控制的组:为了跟踪嵌套条件的级别，指令只通过决定指令的名称进行处理;其他指令的预处理标记会被忽略，组中的其他预处理标记也会被忽略。只处理控制条件计算为 true(非零)的第一个组。如果所有条件的计算结果都不为真，并且存在一个 #else 指令，则处理由 #else 控制的组;没有 #else 指令，在 #endif 之前的所有组都会被跳过。142)

142）如语法所示，在终止新行字符之前不得遵循 #else 或 #endif 指令。但是，注释可能出现在源文件中的任何地方，包括在预处理指令中。

前向引用:宏替换(6.10.3)、源文件包含(6.10.2)、最大整数类型(7.18.1.5)。

6.10.2. 包含源文件

约束

1. #include 指令应该标识一个可以被实现处理的头文件或源文件。

语义

2. 表单的预处理指令

 # include <h-char-sequence> new-line

在 < 和 > 分隔符之间搜索由指定序列唯一标识的头的实现定义的位置序列，并导致该指令被头的整个内容替换。如何指定这些位置或如何标识相应的标头是由实现定义的。

3. 表单的预处理指令

 # include "q-char-sequence" new-line

导致该指令被由“分隔符”之间的指定序列所标识的源文件的整个内容所替换。以实现定义的方式搜索已命名的源文件。如果不支持此搜索，或者如果搜索失败，则会像读取一样重新处理该指令

 # include <h-char-sequence> new-line

与原始指令中包含的相同序列（包括>字符，如果有的话）

4. 表单的预处理指令

 # include pp-tokens new-line

（这与前两种表格中的一种不匹配）是允许的。在指令中 include 之后的预处理令牌就像在正常文本中一样被处理。（当前定义为宏名称的每个标识符都被其预处理令牌的替换列表替换。）所有更换后产生的指令应与前两种表格中的一种相匹配。143)将 < 和 > 预处理标记对或一对“字符”之间的预处理标记序列组合成单个头名预处理标记的实现定义的方法。

143)注意，相邻的字符串文本没有连接到单个字符串文字中（参见 5.1.1.2 中的翻译阶段）；因此，导致两个字符串文本的扩展是无效的指令。

5. 实现应为由一个或多个字母或数字（如 5.2.1 中的定义）组成的序列提供唯一的映射，然后是一个周期（。）还有一个字母。第一个字符应该是一个字母。实现可能忽略字母大小的区别，并将映射限制在周期之前的 8 个重要字符。

6. 一个 #include 预处理指令可以达到实现定义的嵌套限制（参见 5.2.4.1）。

7. 示例 1 #include 预处理指令最常见的用法如下所示

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

8. 示例 2 这说明了宏替换的#include 指令：

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

前向引用:宏替换(6.10.3)。

6.10.3. 宏替换

约束

1. 当且仅当两者中的预处理标记具有相同的数字、顺序、拼写和空白分隔时，替换列表是相同的，其中所有的空白分隔都被认为是相同的。

2. 当前定义为一个类对象宏的标识符不应被另一个 #define 预处理指令重新定义，除非第二个定义是一个类对象的宏定义，并且两个替换列表是相同的。同样，当前 define 为类函数宏的标识符不应被另一个#定义预处理指令重新定义，除非第二个定义是一个类函数的宏定义，具有相同的参数数量和拼写，并且两个替换列表是相同的。

3. 在类对象宏的定义中，标识符和替换列表之间应该有空白。

4. 如果宏定义中的标识符列表未以省略号结束，则调用类似函数的宏中的参数（包括没有预处理令牌的参数）的数量应等于宏定义中的参数数量。否则，调用中的参数应大于宏定义中的参数（不包括....）。应存在终止调用的预处理令牌。

5. 标识符__VA_ARGS__应该只出现在一个在参数中使用省略号符号的类函数宏的替换列表中。

6. 类函数宏中的参数标识符应在其范围内唯一声明。

语义

紧邻 define 后面的标识符称为宏名称。宏名称有一个名称空间。在预处理令牌的替换列表之前或之后的任何空白字符都不被认为是任何一种形式的宏的替换列表的一部分。

如果在预处理指令可以开始的点上出现 # 预处理标记，后面跟着标识符，则该标识符不受宏替换。

表单的预处理指令

define 标识符替换列表 new-line

定义一个类似对象的宏，它使宏名称 144)的每个后续实例被构成指令的其余部分的预处理标记的替换列表所取代。

144）由于根据宏替换时间，所有字符常量和字符串文字都是预处理标记，而不是可能包含类似标识符的子序列的序列（参见 5.1.1.2，翻译阶段），因此它们永远不会被扫描为宏名称或参数。

10. 表单的预处理指令

 # define identififier lparen identififier-listopt) replacement-list new-line
 # define identififier lparen ...) replacement-list new-line
 # define identififier lparen identififier-list , ...) replacement-list new-line

定义了一个带有参数的类似于函数的宏，在语法上类似于函数调用。参数由可选的标识符列表指定，这些标识符的范围从标识符列表中的声明扩展到终止 #define 预处理指令的新行字符。类似函数的宏名称的每个后续实例后面跟着一个(作为下一个预处理令牌引入预处理令牌的序列，它被定义中的替换列表取代（宏的调用）。被替换的预处理标记序列被匹配的)预处理标记终止，跳过插入匹配的左右括号预处理标记对。在构成一个类似于函数的宏的调用的预处理标记序列中，新行被认为是一个正常的空白字符。

11. 由最外部匹配的圆括号所限定的预处理令牌序列形成了类函数宏的参数列表。列表中的单个参数由逗号预处理标记分隔，但在匹配的内括号之间的逗号预处理标记并不分隔参数。如果参数列表中有一系列预处理令牌，否则将作为预处理指令，则行为未定义。

12.如果有一个……在宏定义中的标识符列表中，那么后面的参数，包括任何分隔的逗号预处理标记，将被合并成一个单个项：变量参数。这样合并的参数的数量是这样的，在合并之后，参数的数量比宏定义中的参数的数量多一个（不包括……）。

6.10.3.1. 参数替换

1. 在确定调用类似函数宏的参数之后，将发生参数替换。替换列表中的参数，除非前面有 # 或 ## 预处理标记，或后面是 ## 预处理标记（见下文），将在展开其中包含的所有宏之后替换为相应的参数。在被替换之前，每个参数的预处理标记都被完全宏替换，就好像它们形成了预处理文件的其余部分一样；没有其他预处理标记可用。

2.在替换列表中出现的标识符__VA_ARGS__应被视为一个参数，而变量参数应构成用于替换它的预处理令牌。

6.10.3.2. #操作符

约束

1. 一个类函数宏的替换列表中的每个 # 预处理令牌后面都应该跟着一个参数，作为替换列表中的下一个预处理令牌。

语义

2.如果在替换列表中，参数前面有一个 # 预处理标记，则两者都被单个字符串文字预处理标记替换，该标记包含相应参数的预处理标记序列的拼写。参数预处理标记之间的每次空白都成为字符串文字中的一个空格字符。删除在第一个预处理令牌之前和组成该参数的最后一个预处理令牌之后的空格。否则，参数中每个预处理标记的原始拼写将保留在字符串字符串中，除了生成字符串文字和字符常量的拼写的特殊处理：字符插入字符和字符串或字符串文字的字符之前，除了实现定义是否在通用字符名称开头插入字符之前。如果导致的替换不是有效的字符串文字，则该行为未定义。与空参数对应的字符串文字为“”。# 和 ## 操作符的计算顺序不详。

字符)，除了是否将\字符插入到通用字符名开头的\字符之前是由实现定义的。如果结果的替换不是有效的字符串字面量，则该行为未定义。与空参数对应的字符串字面量是""。#和##操作符的求值顺序没有指定。

6.10.3.3. ##操作符

约束

1.对于任何一种宏定义，不得在替换列表的开头或结束处出现 ## 预处理令牌。

语义

2.如果在类似函数的宏的替换列表中，参数前面或后面有 ## 预处理标记，则该参数将被相应参数的预处理标记序列替换；但是，如果参数不包含预处理标记，则该参数将被列符标记预处理标记替换。145)

145) Placemarker 预处理标记不会出现在语法中，因为它们是只存在于翻译阶段 4 中的临时实体。

3.对于类对象和类函数的宏调用，在重新检查替换列表以替换更多的宏名称之前，将删除替换列表中的 ## 预处理令牌的每个实例，并且前面的预处理令牌与以下预处理令牌连接起来。标记预处理标记被特殊处理：两个标记的连接产生一个标记预处理令牌，一个标记与非标记预处理标记的连接导致非标记预处理令牌。如果结果不是有效的预处理令牌，则该行为将未定义。生成的令牌可用于进一步的宏替换。未指定 ## 操作符的计算顺序。

4.在以下片段中:

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)
```

```
char p[] = join(x, y); // equivalent to
                       // char p[] = "x ## y";
```

这次扩张在不同的阶段产生了：

```
join(x, y)
in_between(x hash_hash y)
in_between(x ## y)
mkstr(x ## y)
"x ## y"
```

换句话说，扩展 hash_hash 会产生一个新的标记，由两个相邻的尖锐符号组成，但这个新的标记不是 ## 运算符。

6.10.3.4. 重新扫描与进一步替换

1 在替换列表中的所有参数均已替换且#和##处理都已完成后，所有占位符预处理标记都会删除。然后，生成的预处理标记序列会连同源文件的所有后续 预处理标记被重新扫描，以便替换更多的宏名称。

2 如果在这次替换列表扫描期间发现了被替换宏的名称(不包括源文件的其余预处理标记)，则不替换它。此外，如果任何嵌套替换遇到要替换的宏名称，则不会替换它。这些未替换的宏名称预处理标记不再用于进一步替换，即使稍后在宏名称预处理标记本应被替换的上下文中(重新)检查它们。

3 产生的完全宏替换的预处理令牌序列不作为预处理指令处理，即使它类似于预处理指令，但它中的所有编译指示一元操作符表达式将按照下面 6.10.9 中指定的方式处理。

6.10.3.5. 宏定义的范围

1 宏定义会一直持续(独立于块结构)直到遇到对应的#undef 指令，或者(如果没有遇到)直到预处理翻译单元结束。在翻译阶段 4 之后，宏定义不再具有意义。

2 一种常用的预处理指令

```
# undef   identififier   new-line
```

这会使得指定的标识符不再定义为宏名称。如果指定的标识符目前没有定义为宏名称，则忽略该标识符。

3 例 1 此功能的最简单用法是定义一个“显式常量”，如

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

4 例 2 下面定义了一个类函数的宏，其值为其参数的最大值。它具有适用于任何兼容类型的参数和生成内联代码而不需要调用函数的优点。它的缺点是二次计算其中一个参数(包括副作用)，如果多次调用，会生成比函数更多的代码。它的地址也不能被取走，因为它没有地址

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

圆括号确保参数和结果表达式被正确绑定

5 例 4 为了说明重新定义和重新审查的规则，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
#define p() int
#define q(x) x
#define r(x,y) x ## y
#define str(x) # x

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
(f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };
char c[2][6] = { str(hello), str() };
```

相当于

```
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);
int i[] = { 1, 23, 4, 5, };
char c[2][6] = { "hello", "" };
```

6 例 4 为了说明创建字符串字面值和连接标记的规则，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
#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') // 此处消失
== 0) str(: @\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

相当于

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

或者，在代入字符串字面值之后，

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

宏定义中#和##标记周围的空格是可选择的。

7 例 5 为了说明占位符预处理标记的规则，序列

```
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,),
t(10,,), t(,11,), t(,,12), t(,,) };
```

相当于

```
int j[] = { 123, 45, 67, 89,
10, 11, 12, };
```

8 例 6 为了演示重定义规则，以下序列是有效的

```
#define OBJ_LIKE (1-1)
#define OBJ_LIKE /* 空白 */ (1-1) /* 其他 */
#define FUNC_LIKE(a) ( a )
#define FUNC_LIKE( a )( /*标记空白部分 */ \
a /* 这条线上的其他东西
*/ )
```

但以下重定义是无效的：

```
#define OBJ_LIKE (0) // 不同标记序列
#define OBJ_LIKE (1 - 1) // 不同的空白
#define FUNC_LIKE(b) ( a ) // 使用不同的参数
#define FUNC_LIKE(b) ( b ) // 不同的参数拼写
```

9 例 7 最后，为了展示变量参数列表宏工具：

```
#define debug(...) fprintf(stderr, _ _VA_ARGS_ _)
#define showlist(...) puts(#_ _VA_ARGS_ _)
#define report(test, ...) ((test)?puts(#test):\
printf(_ _VA_ARGS_ _))
debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

相当于

```
fprintf(stderr, "Flag" );
fprintf(stderr, "X = %d\n", x );
puts( "The first, second, and third items." );
((x>y)?puts("x>y"):
printf("x is %d but y is %d", x, y));
```

6.10.4. 行控制

约束

1 #line 指令的字符串字面量，如果存在，应该是一个字符型字符串字面量

语义

2 当前源行的行号比翻译阶段 1(5.1.1.2)在源文件处理到当前标记时读取或引入的新行字符数大 1。

3 以下形式的预处理指令

 # line digit-sequence new-line

其实现的行为类似于使后续的源行序列以一个源行开始，该源行具有由数字序列指定的行号(解释为十进制整数)。数字序列不能指定零，也不能指定大于 2147483647 的数字。

4 以下形式的预处理指令

 # line digit-sequence "s-char-sequenceopt" new-line

它以类似的方式设置假定的行号，并将假定的源文件名称更改为字符串文字的内容

5 以下形式的预处理指令

 # line pp-tokens new-line

它(与前两种形式不匹配的形式)是允许的。指令行后的预处理标记会像处理普通文本一样处理(当前定义为宏名的每个标识符会被其预处理标记的替换列表所取代)。所有替换后产生的指令应匹配前两种形式中的一种，然后进行适当处理。

6.10.5. 错误指令

语义

1 以下形式的预处理指令

 # error pp-tokensopt new-line

其用于实现产生包含指定的预处理令牌序列的诊断消息。

6.10.6. 编译指示指令

语义

1 以下形式的预处理指令

 # pragma pp-tokensopt new-line

其中预处理标记 STDC 不立即跟随指令中的**编译指示**(在任何宏替换之前)[146]，导致功能实现以实现定义的方式运行。这种行为可能导致翻译失败，或者导致翻译人员或结果程序以不符合规范的方式运行。没有被功能实现识别的任何此类**编译指示**都会被忽略。

2 如果预处理标记 STDC 在指令中紧跟**编译指示**(在任何宏替换之前)，那么不会对指令执行宏替换，该指令应该有以下形式之一，其含义会在其他地方描述:

#pragma STDC FP_CONTRACT on-off-switch
#pragma STDC FENV_ACCESS on-off-switch
#pragma STDC CX_LIMITED_RANGE on-off-switch
on-off-switch: one of
ON OFF DEFAULT

向前引用：FP_CONTRACT 编译指示（7.12.2），**FENV_ACCESS** 编译指示（7.6.1），**CX_LIMITED_RANGE** 编译指示（7.3.4）。

6.10.7. 空指令

语义

1 以下形式的预处理指令

 # new-line

是无效的

6.10.8. 预定义的宏名称

1 以下宏名称[148]应由实现定义:

__DATE__ 预处理翻译单元的翻译日期:格式为“Mmm dd yyyy”的字符串字面量，其中月份名称与 asctime 函数生成的月份名称相同，如果值小于 10,dd 的第一个字符为空格字符。如果没有翻译日期，则应提供实现定义的有效日期。

__FILE__ 当前源文件的假定名称(字符串字面量)。[149]

__LINE__ 当前源行的假定行号(在当前源文件中)(一个整数常量)。[149]

__STDC__ 整数常量 1，用于指示符合要求的实现。

__STDC_HOSTED__ 如果实现是宿主实现，则为整型常量 1，否则为整型常量 0。

__STDC_VERSION__ 整数常量 199901L。

__TIME__ 预处理翻译单元的翻译时间:格式为“hh:mm:ss”的字符串字面量，与 asctime 函数生成的时间相同。如果没有翻译时间，则应提供实现定义的有效时间

2 下面的宏名称是由实现有条件地定义的:

__STDC_IEC_559__ 整数常数 1，用于表示符合附件 F (IEC 60559 浮点运算)的规范

__STDC_ISO_10646__ 形式为 yyyymmL(例如，199712L)的整数常量，用于指示类型为 wchar_t 的值是 ISO/IEC 10646 定义的字符的编 码表示，以及指定年份和月份的所有修订和技术勘误。

3 预定义宏的值(__FILE__和 __LINE__除外)在整个翻译单元中保持不变。

4 这些宏名称和**定义**的标识符都不能成为**#define** 或**#undef** 预处理指令的主题。任何其他预定义的宏名称应该以一个下划线开头，后面跟着一个大写字母或第二个下划线。

5 实现不应预先定义宏 __cplusplus，也不应在任何标准头文件中定义它。

前向引用: asctime 函数(7.23.3. 1)，标准头文件(7. 1. 2)

6.10.9. 编译指示符

6 以下形式的一元操作符表达式:

_Pragma (string-literal)

其处理方法如下:删除 L 前缀(如果存在)，删除开头和结尾的双引号，用双引号替换每个转义序列\"，用一个反斜杠替换每个转义序列\\。通过翻译阶 段 3 处理产生的字符序列，以产生预处理标记，这些标记将被执行，就像它 们是编译指示指令中的 pp 标记一样。一元运算符表达式中原来的四个预处理标记被删除。

7 例 以下形式的指令：

```
#pragma listing on "..\listing.dir"
```

也可以表示为：

```
_Pragma ( "listing on \"..\\listing.dir\"" )
```

后一种形式以相同的方式处理，无论是按字面意思显示，还是宏替换的结果，如:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)
LISTING ( ..\listing.dir )
```

6.11. 未来的语言方向

6.11.1. 浮点型

1 未来的标准化可能包括更多的浮点类型，包括那些具有更大的范围、精度或两者都大于长双精度型。

6.11.2. 标识符的联系

2 在文件范围内声明具有内部链接而不使用静态存储类说明符的标识符是一种被淘汰的特性。

6.11.3. 外部名

3 将外部名称的重要性限制在255个字符以内(将每个通用字符名称或扩展源字符视为单个字符)是一种被淘汰的特性，是对现有实现的让步。

6.11.4. 字符转义序列

4 作为转义序列的小写字母为将来的标准化保留。其他字符可以在扩展中使用。

6.11.5. 存储类关键字

5 将存储类说明符放置在声明中声明说明符的开头之外是一种被废弃的特性。

6.11.6. 函数声明符

6 使用带空圆括号的函数声明符(而不是原型格式的参数类型声明符)是一种被废弃的特性。

6.11.7. 函数定义

7 使用单独的参数标识符和声明列表(而不是原型格式的参数类型和标识符声明符)的函数定义是一种被废弃的特性。

6.11.8. 编译指示指令

8 第一个预处理标记是 STDC 的编译指令是为将来的标准化所保留

6.11.9. 预定义的宏名称

9 以_ _STDC_开头的宏名称是为将来的标准化所保留。