# Contents


- C 语言文档
- C 语言参考
  - C 语言参考
  - 《C 语言参考》的组织
    - 《C 语言参考》的组织
    - 此手册的范围
    - ANSI 一致性
  - C 的元素
    - C 的元素
    - C 标记
      - C 标记
      - 空白字符
      - C 注释
      - 标记的计算
    - C 关键字
    - C 标识符
      - C 标识符
      - 多字节和宽字符
      - 三字符组
    - C 常量
      - C 常量
      - C 浮点常量
        - C 浮点常量
        - 对浮点常量的限制
      - C 整数常量
        - C 整数常量
        - 整数类型
        - C 和 C++ 整数限制
      - C 字符常量

C 字符常量
字符类型
执行字符集
转义序列
八进制和十六进制字符规范
C 字符串文本
C 字符串文本
字符串文本的类型
字符串文本的存储
字符串文本串联
最大字符串长度
标点和特殊字符
程序结构
程序结构
源文件和源程序
源文件和源程序
对预处理器的指令
C 杂注
C 声明和定义
函数声明和定义
Blocks
示例程序
main 函数和程序执行
main 函数和程序执行
使用 wmain
自变量说明
扩展通配符自变量
分析 C 命令行自变量
自定义 C 命令行处理
生存期、范围、可见性和链接
生存期、范围、可见性和链接
生存期

范围和可见性
生存期和可见性的摘要
链接
链接
内部链接
外部链接
无链接
命名空间
声明和类型
对齐 (C11)
声明和类型
声明概述
C 存储类
C 存储类
外部级别声明的存储类说明符
内部级别声明的存储类说明符
内部级别声明的存储类说明符
auto 存储类说明符
register 存储类说明符
静态存储类说明符
extern 存储类说明符
具有函数声明的存储类说明符
C 类型说明符
C 类型说明符
数据类型说明符和等效项
类型限定符
声明符和变量声明
声明符和变量声明
简单变量声明
C 枚举声明
结构声明
结构声明

C 位域
结构的存储和对齐
联合声明
联合声明
联合的存储
数组声明
数组声明
数组的存储
指针声明
指针声明
地址存储
基指针 (C)
C 抽象声明符
解释更复杂的声明符
初始化
初始化
初始化标量类型
初始化聚合类型
初始化字符串
基本类型的存储
基本类型的存储
char 类型
int 类型
C 调整了大小的整型
float 类型
类型 double
长双精度类型
不完整类型
Typedef 声明
C 扩展的存储类特性
C 扩展的存储类特性
DLL 导入和导出

Naked (C)
线程本地存储
表达式和赋值
表达式和赋值
操作数和表达式
操作数和表达式
C 主要表达式
C 主要表达式
主要表达式中的标识符
主要表达式中的常量
主要表达式中的字符串文本
括号中的表达式
一般选择 (C11)
左值和右值表达式
C 常量表达式
表达式计算 (C)
表达式计算 (C)
副作用
C 序列点
C 运算符
C 运算符
计算的优先级和顺序
常用算术转换
后缀运算符
后缀运算符
一维数组
多维数组 (C)
函数调用 (C)
结构和联合成员
C 后缀增量和减量运算符
C 一元运算符
C 一元运算符

前缀增量和减量运算符
间接寻址运算符和 address-of 运算符
一元算术运算符
sizeof 运算符 (C)
强制转换运算符
C 乘法运算符
C 加法运算符
C 加法运算符
加 (+)
减法 (–)
使用加法运算符
指针算术
按位移位运算符
C 关系和相等运算符
C 按位运算符
C 逻辑运算符
条件表达式运算符
C 赋值运算符
C 赋值运算符
简单赋值 (C)
C 复合赋值
有序评估运算符
类型转换 (C)
类型转换 (C)
赋值转换
赋值转换
从带符号整型的转换
从无符号整型的转换
从浮点类型的转换
指针类型之间的转换
从其他类型的转换
类型强制转换的转换

函数调用转换
语句 (C)
语句 (C)
C 语句概述
break 语句 (C)
复合语句 (C)
continue 语句 (C)
do-while 语句 (C)
表达式语句 (C)
for 语句 (C)
goto 和标记语句 (C)
if 语句 (C)
Null 语句 (C)
return 语句 (C)
static_assert statement (C11)
switch 语句 (C)
try-except 语句 (C)
try-finally 语句 (C)
While 语句 (C)
函数 (C)
函数 (C)
函数概述
函数概述
函数声明和定义的过时形式
C 函数定义
C 函数定义
函数特性
函数特性
指定调用约定
内联函数
内联汇编程序 (C)
_Noreturn (C)

DLL 导入和导出函数
DLL 导入和导出函数
定义和声明 (C)
使用 dllexport 和 dllimport 定义内联 C 函数
dllimport-dllexport 的规则和限制
Naked 函数
裸函数
针对使用 Naked 函数的规则和限制
编写 Prolog-Epilog 代码时的注意事项
存储类
返回类型
参数
函数体
函数原型
函数调用
函数调用
自变量
使用数目可变的自变量调用
递归函数
C 语言语法摘要
C 语言语法摘要
定义和约定
词法语法
短语结构语法
短语结构语法
表达式摘要
声明摘要
语句摘要
外部定义
实现定义的行为
实现定义的行为
翻译:诊断

环境
环境
要保留的自变量
交互式设备
标识符的行为
标识符的行为
没有外部链接的重要字符
带有外部链接的重要字符
大写和小写
字符
字符
ASCII 字符集
多字节字符
每字符的位数
字符集 1
无代表的字符常量
宽字符
转换多字节字符
字符值的范围
整数
整数
整数值的范围
整数的降级
带符号的按位运算
余数
右移
浮点数学
浮点数学
值
将整数转换为浮点值
浮点值的截断
数组和指针

数组和指针
最大数组大小
指针减法
寄存器：寄存器的可用性
结构、联合、枚举和位域
结构、联合、枚举和位域
对联合的不正确的访问
结构成员的填充和对齐
位域的符号
位域的存储
枚举类型
限定符：访问易失对象
声明符：最大数量
语句：对 Switch 语句的限制
预处理指令
预处理指令
字符常量和条件包含
包含用括号括起来的文件名
包含带引号的文件名
字符序列
杂注
默认日期和时间
库函数
库函数
NULL 宏
assert 函数输出的诊断
字符测试
域错误
浮点值的下溢
fmod 函数
signal 函数 (C)
默认信号

终止换行符
空行
Null 字符
追加模式中的文件位置
文本文件的截断
文件缓冲
零长度文件
文件名
文件访问限制
删除打开的文件
使用已存在的名称进行重命名
读取指针值
读取范围
文件位置错误
由 perror 函数生成的消息
分配零内存
abort 函数 (C)
atexit 函数 (C)
环境名称
系统函数
strerror 函数
时区
clock 函数 (C)
C/C++ 预处理器参考
C 运行时库 (CRT) 参考

# C 语言参考

2021/8/16 •

《C 语言参考》介绍了在 Microsoft C 中实现的 C 编程语言。本书的组织基于 ANSI C 标准（有时称为 C89）以及关于 ANSI C 标准的 Microsoft 扩展的其他材料。

- 《C 语言参考》的组织

有关 C++ 和预处理器的其他参考资料，请参阅：

- C++ 语言参考
- 预处理器参考

C/C++ 生成参考中记录了编译器和链接器选项。

## 请参阅

C++ 语言参考

# 《C 语言参考》的组织

2021/8/15 •

- C 的元素
- 程序结构
- 声明和类型
- 表达式和赋值
- 语句
- 函数
- C 语言语法摘要
- 实现定义的行为

## 请参阅

C 语言参考

# 此手册的范围

2021/8/12 •

C 是一种灵活的语言，可让您决定多个编程。遵循该原理，C 在事件上极少施加限制，如类型转换。尽管该语言的此特性可以使编程工作更加容易，但是必须很好地了解该语言以确定程序的行为方式。本书提供了有关 C 语言组件和 Microsoft 实现的功能的信息。C 语言的语法来自 ANSI X3.159-1989，American National Standard for Information Systems - 编程语言 - C（以后称为 ANSI C 标准），不过它不是 ANSI C 标准的一部分。C 语言语法摘要提供语法和有关如何读取和使用语法定义的说明。

本书没有讨论如何使用 C++ 进行编程。有关 C++ 语言的信息，请参阅 C++ 语言参考。

## 请参阅

《C 语言参考》的组织

# ANSI 一致性

2021/8/14 •

Microsoft C 遵循在 ANSI C 标准的 9899:1990 版本中规定的 C 语言的标准。

本书的文本和语法中以及联机引用中介绍了 Microsoft ANSI C 标准扩展。由于此扩展不是 ANSI C 标准的一部分，因此对它们的使用可能会限制系统间程序的可移植性。默认情况下，将启用 Microsoft 扩展。若要禁用此扩展，请指定 /Za 编译器选项。使用 /Za，所有非 ANSI 代码将生成错误或警告。

## 请参阅

《C 语言参考》的组织

# C 的元素

2021/8/13 •

本节介绍 C 编程语言的元素，包括用于构造 C 程序的名称、数字和字符。ANSI C 语法标记这些组件标记。

本节介绍如何定义标记以及编译器如何计算它们。

本文讨论了以下主题：

- 标记
- 注释
- 关键字
- 标识符
- 常量
- 字符串文本
- 标点和特殊字符

该部分还包括三元组、浮点常量限制、C 和 C++ 整数限制和转义序列的引用表。

运算符是指定如何操作值的符号（单个字符和字符组合）。将每个符号解释为单个单元，称为令牌。有关详细信息，请参阅运算符。

## 请参阅

C 语言参考

# C 标记

2021/8/13 •

在 C 源程序中，编译器识别的基本元素是"标记"。标记是编译器不会分解为组件元素的源程序文本。

## 语法

*token*: *keyword*

*identifier*

*constant*

*string-literal*

*operator*

punctuator

> **NOTE**
> 有关 ANSI 语法约定的说明，请参阅 C 语言语法摘要的简介。

本节描述的关键字、标识符、常量、字符串文本和运算符是标记的示例。标点符号（如方括号 ([ ])、大括号 ({ })、圆括号 ( ( ) ) 和逗号 (, ) 也是标记。

## 请参阅

C 的元素

# 空白字符

2021/8/12 •

空格、制表符、换行符(创建新行)、回车符、换页符、垂直制表符称为"空白字符",因为它们与打印页上的单词和行之间的空格一样都是起到方便阅读的作用。标记由空白字符和其他标记分隔(划分边界),如运算符和标点。在分析代码时,C 编译器将忽略空白字符,除非您将它们用作分隔符或者字符常量或字符串文本的组成部分。使用空白字符可以让程序更易于阅读。请注意,编译器也将注释视为空白。

## 请参阅

C 标记

# C 注释

2021/8/11 •

"注释"是一个以正斜杠/星号组合(/*)开头的字符序列，编译器会将正斜杠/星号组合视为单个空白字符，要不然就忽略它。注释可以包含可表示字符集中的任意字符组合，包括换行符，但"结束注释"分隔符 (*/) 除外。注释可以占用多行，但无法嵌套。

注释可以出现在允许使用空白字符的任何地方。由于编译器将注释视为空白字符，因此不能将注释包含在标记中。编译器将忽略注释中的字符。

使用注释来记录您的代码。本示例是编译器接受的注释：

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

注释可以与代码语句出现在同一行中：

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

可以选择在函数或程序模块前面放置一个描述性注释块：

```
/* MATHERR.C illustrates writing an error routine
* for math functions.
*/
```

由于注释不能包含嵌套注释，因此本示例导致了一个错误：

```
/* Comment out this routine for testing

   /* Open file */
    fh = _open( "myfile.c", _O_RDONLY );
    .
    .
    .
*/
```

该错误发生的原因是编译器将单词 `*/` 后的第一个 `Open file` 识别为注释的末尾。编译器尝试处理剩余的文本，当它在注释外找到 `*/` 时，便产生了错误。

尽管您可以出于测试目的使用注释来呈现某些处于非活动状态的代码行，但预处理器指令 `#if` 和 `#endif` 以及条件编译都是执行此任务的有用的替代选择。有关详细信息，请参阅预处理器参考 中的预处理器指令。

### Microsoft 专用

Microsoft 编译器还支持前面有两个正斜杠 ( // ) 的单行注释。如果使用 /Za(ANSI 标准)进行编译，这些注释将产生错误。这些注释不能扩展到第二行。

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

以两个正斜杠 ( // ) 开头的注释被前面没有转义字符的下一个换行符终止。在下一个示例中，换行符的前面有一个反斜杠 ( \ )，这将创建"转义序列"。此转义序列会使编译器将下一行视为上一行的一部分。(有关详细信息，请

参阅转义序列。）

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

因此，`i++;` 语句被注释掉。

Microsoft C 的默认设置是启用 Microsoft 扩展。请使用 /Za 禁用这些扩展。

结束 Microsoft 专用

## 请参阅

C 标记

# 标记的计算

2021/8/15 •

当编译器解释标记时，它在移到下一个标记之前，会在单个标记中包括尽可能多的字符。由于此行为，编译器可能不会按预期方式解释标记（如果没有用空格正确分隔标记）。考虑下面的表达式：

```
i+++j
```

在此示例中，编译器首先从三个加号生成可能最长的运算符 ( `++` )，然后将剩余的加号视为加法运算符 ( `+` )。因此，该表达式将解释为 `(i++) + (j)` 而不是 `(i) + (++j)` 。在此情况以及类似的情况下，使用空格和括号以避免多义性，并确保适当的表达式计算。

**Microsoft 专用**

C 编译器将 CTRL+Z 字符视为文件尾指示符。它忽略 CTRL+Z 后的所有文本。

**结束 Microsoft 专用**

## 请参阅

C 标记

# C 关键字

2021/8/16 •

“关键字”是对 C 编译器具有特殊含义的单词。在翻译的第 7 和第 8 阶段中，标识符不能具有与 C 关键字相同的拼写和大小写。有关详细信息，请参阅《预处理器参考》中的翻译阶段。有关标识符的详细信息，请参阅标识符。

## 标准 C 关键字

C 语言使用下列关键字：

`auto`
`break`
`case`
`char`
`const`
`continue`
`default`
`do`
`double`
`else`
`enum`

`extern`
`float`
`for`
`goto`
`if`
`inline` [1, a]
`int`
`long`
`register`
`restrict` [1, a]
`return`

`short`
`signed`
`sizeof`
`static`
`struct`
`switch`
`typedef`
`union`
`unsigned`
`void`
`volatile`

`while`
`_Alignas` [2, a]
`_Alignof` [2, a]

`_Atomic` [2, b]
`_Bool` [1, a]
`_Complex` [1, b]
`_Generic` [2, a]
`_Imaginary` [1, b]
`_Noreturn` [2, a]
`_Static_assert` [2, a]
`_Thread_local` [2, b]

[1] ISO C99 中引入的关键字。

[2] ISO C11 中引入的关键字。

[a] 从 Visual Studio 2019 版本 16.8 开始，如果指定了 `/std:c11` 或 `/std:c17` 编译器选项，将在编译为 C 的代码中支持这些关键字。

[b] 从 Visual Studio 2019 版本 16.8 开始，如果指定了 `/std:c11` 或 `/std:c17` 编译器选项，这些关键字将由编译器在编译为 C 的代码中识别，但不受支持。

不能重新定义关键字。但是，你可以在编译前通过使用 C 预处理器指令指定文本来替换关键字。

## Microsoft 专用 C 关键字

ANSI 和 ISO C 标准允许为编译器实现保留带有两个前导下划线的标识符。Microsoft 的惯例是在 Microsoft 专用关键字名称前加上双下划线。这些单词不能用作标识符名称。有关标识符命名规则的说明，包括双下划线的使用，请参阅标识符。

下列关键字和特殊标识符由 Microsoft C 编译器识别：

`__asm` [5]
`__based` [3, 5]
`__cdecl` [5]
`__declspec` [5]
`__except` [5]
`__fastcall`
`__finally` [5]

`__inline` [5]
`__int16` [5]
`__int32` [5]
`__int64` [5]
`__int8` [5]
`__leave` [5]
`__restrict`

`__stdcall` [5]
`__try` [5]
`dllexport` [4]
`dllimport` [4]
`naked` [4]
`static_assert` [6]
`thread` [4]

[3] `__based` 关键字对 32 位和 64 位目标编译的用途有限。

[4] 当与 `__declspec` 一起使用时，这些关键字是特殊的标识符；它们在其他情况下的使用不受限制。

[5] 为了与以前的版本兼容，当启用 Microsoft 扩展时，这些关键字既可以使用两个前导下划线，也可以使用一个前导下划线。

[6] 如果不包括 <assert.h>，则 Microsoft Visual C 编译器会将 `static_assert` 映射到 C11 `_Static_assert` 关键字。

默认情况下将启用 Microsoft 扩展。为了帮助创建可移植的代码，可以在编译过程中指定 /Za (禁用语言扩展) 选项来禁用 Microsoft 扩展。如果使用此选项，将禁用某些 Microsoft 专用关键字。

启用 Microsoft 扩展时，您可在程序中使用上面列出的关键字。为了符合标准，这些关键字大多使用前导双下划线。`dllexport`、`dllimport`、`naked` 和 `thread` 这 4 个关键字除外，它们只与 `__declspec` 一起使用，不需要前导双下划线。为了向后兼容，支持其余的关键字的单下划线版本。

## 请参阅

C 的元素

# C 标识符

2021/8/11 •

“Identifiers”或“symbols”是您为程序中的变量、类型、函数和标签提供的名称。标识符名称在拼写和大小写上必须与任何关键字都不同。不能将关键词(C 或 Microsoft)用作标识符;将它们保留以用于特殊用途。通过在变量、类型或函数的声明中指定标识符来创建标识符。在此示例中，`result` 是整型变量的标识符，`main` 和 `printf` 是函数的标识符名称。

```
#include <stdio.h>

int main()
{
    int result;

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

声明后，您可在以后的程序语句中使用标识符来引用关联值。

可以在 `goto` 语句中使用一种特殊的标识符，称为“语句标签”。(声明和类型中介绍了声明，goto 和标记语句中介绍了语句标签。)

## 语法

identifier：
*nondigit*
*identifier nondigit*
*identifier digit*

*nondigit*: 以下之一：
_ a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L MN O P Q R S T U V W X Y Z

*digit*: 以下之一：
0 1 2 3 4 5 6 7 8 9

标识符名称的第一个字符必须是 `nondigit`(即，第一个字符必须是下划线、大写字母或小写字母)。ANSI 允许外部标识符名称包含 6 个有效字符，内部(一个函数中)标识符名称包含 31 个有效字符。外部标识符(在全局范围声明的或使用存储类 `extern` 声明的标识符)可能会受到额外的命名限制，因为这些标识符必须由其他软件(如链接器)处理。

**Microsoft 专用**

尽管 ANSI 允许外部标识符名称包含 6 个有效字符，内部标识符(在一个函数内)名称包含 31 个有效字符，但 Microsoft C 编译器允许内部或外部标识符名称包含 247 个字符。如果您不担心 ANSI 兼容性，则可使用 /H(限制外部名称的长度)选项将此默认值修改为更小或更大的数字。

**结束 Microsoft 专用**

C 编译器会将大写和小写字母视为不同的字符。利用此功能(称为“区分大小写”)，您可以创建拼写相同但一个或多个字母的大小写不同的不同标识符。例如，下列每个标识符都是唯一的：

```
add
ADD
Add
aDD
```

**Microsoft 专用**

请勿为标识符选择以两条下划线开头或者以一条下划线后跟一个大写字母开头的名称。ANSI C 标准允许保留以这些字符组合开头的标识符名称以供编译器使用。不应将一条下划线和一个小写字母作为前两个字母来命名具有五级范围的标识符。以这些字符开头的标识符名称也将保留。按照约定，Microsoft 使用下划线和大写字母作为宏名称的开头，并使用双下划线作为 Microsoft 特定关键字名称的开头。若要避免任何命名冲突，请始终选择不是以一条或两条下划线开头的标识符名称，或选择以一条下划线后跟一个大写字母开头的名称。

**结束 Microsoft 专用**

下面是符合 ANSI 或 Microsoft 命名限制的有效标识符的示例：

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

**Microsoft 专用**

尽管默认情况下源文件中的标识符区分大小写，但对象文件中的符号不区分大小写。Microsoft C 将编译单元中的标识符视为区分大小写。

Microsoft 链接器区分大小写。您必须根据情况一致地指定所有标识符。

"源字符集"是可能出现在源文件中的合法字符集。对于 Microsoft C，源集是标准 ASCII 字符集。源字符集和执行字符集包含用作转义序列的 ASCII 字符。有关执行字符集的信息，请参阅字符常量。

**结束 Microsoft 专用**

标识符具有"范围"（在程序中发现标识符的区域）和"链接"（确定其他范围中的同一名称是否引用同一标识符）。生存期、范围、可见性和链接中介绍了这些主题。

# 请参阅

C 的元素

# 多字节和宽字符

2021/8/13 •

多字节字符是由一个或多个字节的序列构成的字符。每个字节序列表示扩展字符集中的单个字符。多字节字符用于字符集(如日文汉字)中。

宽字符是宽度始终为 16 位的多语言字符代码。字符常数的类型是 `char`;对于宽字符,类型是 `wchar_t`。由于宽字符始终具有固定大小,因此使用宽字符集可以简化使用国际字符集进行的编程。

宽字符串文本 `L"hello"` 成为类型为 `wchar_t` 的包含六个整数的数组。

```
{L'h', L'e', L'l', L'l', L'o', 0}
```

Unicode 规范是宽字符的规范。用于多字节和宽字符之间的转换的运行库例程包括 `mbstowcs`、`mbtowc`、`wcstombs` 和 `wctomb`。

## 请参阅

C 标识符

# 三字符组

2021/8/13 •

C 源程序的源字符集包含在 7 位 ASCII 字符集中，但它是 ISO 646-1983 固定语言代码集的超集。三元组序列仅允许使用 ISO（国际标准组织）固定语言代码集编写 C 程序。三元组是三字符序列（由两个连续问号引入），编译器会将它替换为其相应的标点字符。可以将 C 源文件中的三元组与不包含某些标点字符的方便图形表示形式的字符集一起使用。

C++ 17 从语言中删除三字符组。实现可能会继续支持三元组作为从物理源文件到 *基本源字符集* 的实现定义的映射的一部分，但标准不鼓励实现这样做。通过 C++ 14，三元组受到如在 C 中一样的支持。

Visual C++ 继续支持三元组替换，但默认处于禁用状态。若要了解如何启用三元组替换，请参阅 `/Zc:trigraphs`（三元组替换）。

下表显示了九个三元组序列。第一列中的标点字符的源文件中的所有匹配项都将替换为第二列中的相应字符。

**三元组序列**

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

三元组始终被视为单个源字符。在识别字符串和字符常数中的转义符前，三元组的转换会在第一个转换阶段出现。仅识别上表中显示的九个三元组。所有其他字符序列都未转换。

字符转义序列 `\?` 可防止对类似于三元组的字符序列进行错误解释。（有关转义序列的信息，请参阅转义序列。）例如，如果您尝试打印具有 `What??!` 语句的字符串 `printf`

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

打印的字符串为 `What|`，因为 `??!` 是一个三元组序列，它将替换为 `|` 字符。编写如下语句以正确打印字符串：

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

在此 `printf` 语句中，第二个问号的前面的反斜杠转义符可防止将 `??!` 错误解释为三元组。

## 请参阅

`/Zc:trigraphs`（三元组替换）
C 标识符

# C 常量

2021/8/13 •

常数是可以在程序中用作值的数字、字符或字符串。使用常量可表示不能修改的浮点、整数、枚举或字符值。

## 语法

`constant`:

- `floating-point-constant`
- `integer-constant`
- `enumeration-constant`
- `character-constant`

常量的特征是具有值和类型。接下来三节将讨论浮点、整数和字符常量。枚举常量将在枚举声明中介绍。

## 请参阅

C 的元素

# C 浮点常量

2021/8/11 •

"浮点常量"是表示带符号实数的十进制数字。带符号实数的表现形式包括整数部分、小数部分和指数。浮点常量用于表示不可更改的浮点值。

## 语法

*floating-point-constant*:
*fractional-constant* *exponent-part* opt *floating-suffix* opt
*digit-sequence* *exponent-part* *floating-suffix* opt

*fractional-constant*:
*digit-sequence* opt . *digit-sequence*
*digit-sequence* .

*exponent-part*:
e *sign* opt *digit-sequence*
E *sign* opt *digit-sequence*

*sign* :以下之一
+ -

*digit-sequence*:
*digit*
*digit-sequence* *digit*

*floating-suffix* :以下之一
f l F L

你可以省略小数点前面的数字(整数部分)或小数点后面的数字(小数部分),但不能同时省略。仅在包括一个指数时可省略小数点。空白字符不能分隔常量的数字或字符。

以下示例阐释了某些形式的浮点常量和表达式:

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

浮点常量为正数,除非它们的前面有减号 ( - )。在这种情况下,减号将视为一元算术求反运算符。浮点常数的类型为 `float` 、`double` 或 `long double` 。

不带 `f` 、`F` 、`l` 或 `L` 后缀的浮点常量为 `double` 。如果后缀是字母 `f` 或 `F` ,则常量的类型为 `float` 。如果后缀是字母 `l` 或 `L` ,则常数的类型为 `long double` 。例如:

```
10.0L  /* Has type long double  */
10.0   /* Has type double       */
10.0F  /* Has type float        */
```

Microsoft C 编译器在内部表示 `long double` ,与 `double` 类型相同 。但其实类型是不同的。若要了解类型

`double`、`float` 和 `long double`，请查看基本类型的存储 。

如下例所示，可以省略浮点常量的整数部分。可通过多种方式表达数字 0.75，包括以下示例：

```
.0075e2
0.075e1
.075e1
75e-2
```

## 请参阅

C 常量

# 对浮点常量的限制

2021/8/15 •

**Microsoft 专用**

下表中提供了对浮点常量值的限制。头文件 FLOAT.H 包含此信息。

**对浮点常量的限制**

| tttttt | tt | "t" |
|---|---|---|
| FLT_DIG<br>DBL_DIG<br>LDBL_DIG | 位数 q，以便 q 十进制数的浮点数可以被舍入到浮点表示形式并返回，而不会丢失精度。 | 6<br>15<br>15 |
| FLT_EPSILON<br>DBL_EPSILON<br>LDBL_EPSILON | 最小正数 x，以便 x + 1.0 不等于 1.0 | 1.192092896e-07F<br>2.2204460492503131e-016<br>2.2204460492503131e-016 |
| FLT_GUARD | | 0 |
| FLT_MANT_DIG<br>DBL_MANT_DIG<br>LDBL_MANT_DIG | 由浮点有效位数中的 FLT_RADIX 指定的基数中的位数。基数为 2；因此这些值指定位。 | 24<br>53<br>53 |
| FLT_MAX<br>DBL_MAX<br>LDBL_MAX | 可表示的最大浮点数。 | 3.402823466e+38F<br>1.7976931348623158e+308<br>1.7976931348623158e+308 |
| FLT_MAX_10_EXP<br>DBL_MAX_10_EXP<br>LDBL_MAX_10_EXP | 最大整数，以便 10 的该数字的幂是一个可表示的浮点数。 | 38<br>308<br>308 |
| FLT_MAX_EXP<br>DBL_MAX_EXP<br>LDBL_MAX_EXP | 最大整数，以便 FLT_RADIX 的该数字的幂是一个可表示的浮点数。 | 128<br>1024<br>1024 |
| FLT_MIN<br>DBL_MIN<br>LDBL_MIN | 最小正值。 | 1.175494351e-38F<br>2.2250738585072014e-308<br>2.2250738585072014e-308 |
| FLT_MIN_10_EXP<br>DBL_MIN_10_EXP<br>LDBL_MIN_10_EXP | 最小负整数，以便 10 的该数字的幂是一个可表示的浮点数。 | -37<br>-307<br>-307 |
| FLT_MIN_EXP<br>DBL_MIN_EXP<br>LDBL_MIN_EXP | 最小负整数，以便 FLT_RADIX 的该数字的幂是一个可表示的浮点数。 | -125<br>-1021<br>-1021 |
| FLT_NORMALIZE | | 0 |
| FLT_RADIX<br>_DBL_RADIX<br>_LDBL_RADIX | 基数的指数表示形式。 | 2<br>2<br>2 |

| ɛɛɛɛɛ | ɛɛ | "ɛ" |
| --- | --- | --- |
| FLT_ROUNDS<br>_DBL_ROUNDS<br>_LDBL_ROUNDS | 浮点加法的舍入模式。 | 1(相邻)<br>1(相邻)<br>1(相邻) |

请注意，上表中的信息可能在未来的实现中不同。

结束 Microsoft 专用

## 请参阅

C 浮点常量

# C 整数常量

2021/8/12 •

**整数常量 是表示整数值的十进制(基数为 10)、八进制(基数为 8)或十六进制(基数为 16)数字。使用整数常量表示不能更改的整数值。**

## 语法

*integer-constant*:
  *decimal-constant integer-suffix*$_{opt}$
  *octal-constant integer-suffix*$_{opt}$
  *hexadecimal-constant integer-suffix*$_{opt}$

*decimal-constant*:
  nonzero-digit
  *decimal-constant digit*

*octal-constant*:
  0
  *octal-constant octal-digit*

*hexadecimal-constant*:
  *hexadecimal-prefix hexadecimal-digit*
  *hexadecimal-constant hexadecimal-digit*

hexadecimal-prefix : one of
  **0x 0X**

*nonzero-digit*: one of
  1 2 3 4 5 6 7 8 9

*octal-digit*: one of
  0 1 2 3 4 5 6 7

*hexadecimal-digit*: one of
  0 1 2 3 4 5 6 7 8 9
  a b c d e f
  A B C D E F

*integer-suffix*:
  *unsigned-suffix long-suffix*$_{opt}$
  *unsigned-suffix long-long-suffix*
  *unsigned-suffix 64-bit-integer-suffix*
  *long-suffix unsigned-suffix*$_{opt}$
  *long-long-suffix unsigned-suffix*$_{opt}$
  64-bit-integer-suffix

*unsigned-suffix*: one of
  u U

*long-suffix*: one of
  l L

long-long-suffix : one of
ll LL

64-bit-integer-suffix : one of
i64 I64

i64 和 I64 后缀为 Microsoft 专用。

整数常量为正数，除非它们的前面有减号 ( - )。减号解释为一元算术求反运算符。（有关此运算符的信息，请参阅一元算术运算符。）

如果整数常量以 0x 或 0X 开始，则它是十六进制。如果它以数字 0 开始，则为八进制。否则，将其假定为十进制。

以下整数常量是等效的：

```
28
0x1C   /* = Hexadecimal representation for decimal 28 */
034    /* = Octal representation for decimal 28 */
```

空白字符不能分隔整数常量的数字。这些示例显示了一些有效的十进制、八进制和十六进制常量。

```
    /* Decimal Constants */
    int                dec_int    = 28;
    unsigned           dec_uint   = 4000000024u;
    long               dec_long   = 2000000022l;
    unsigned long      dec_ulong  = 4000000000ul;
    long long          dec_llong  = 9000000000LL;
    unsigned long long dec_ullong = 900000000001ull;
    __int64            dec_i64    = 9000000000002I64;
    unsigned __int64   dec_ui64   = 90000000000004ui64;

    /* Octal Constants */
    int                oct_int    = 024;
    unsigned           oct_uint   = 04000000024u;
    long               oct_long   = 02000000022l;
    unsigned long      oct_ulong  = 04000000000UL;
    long long          oct_llong  = 04400000000000ll;
    unsigned long long oct_ullong = 044400000000000001Ull;
    __int64            oct_i64    = 0444400000000000000i64;
    unsigned __int64   oct_ui64   = 0444400000000000004uI64;

    /* Hexadecimal Constants */
    int                hex_int    = 0x2a;
    unsigned           hex_uint   = 0XA0000024u;
    long               hex_long   = 0x20000022l;
    unsigned long      hex_ulong  = 0XA0000021uL;
    long long          hex_llong  = 0x8a00000000000ll;
    unsigned long long hex_ullong = 0x8A40000000000010uLL;
    __int64            hex_i64    = 0x4a44000000000020I64;
    unsigned __int64   hex_ui64   = 0x8a44000000000040Ui64;
```

## 请参阅

C 常量

# 整型

2021/8/15 •

每个整型常数都根据其值及其表达方式被赋予一个类型。可通过将字母 `l` 或 `L` 追加到任何整型常数的末尾，将常数强制转换为类型 `long`；可通过将 `u` 或 `U` 追加到值中，将常数强制转换为类型 `unsigned`。小写字母 `l` 可能会与数字 1 混淆，应避免使用。一些形式的 `long` 整型常数如下所示：

```
/* Long decimal constants */
10L
79L

/* Long octal constants */
012L
0115L

/* Long hexadecimal constants */
0xaL or 0xAL
0X4fL or 0x4FL

/* Unsigned long decimal constant */
776745UL
778866LU
```

您分配给常量的类型取决于常量表示的值。常量的值必须在其类型的可表示值的范围内。常量的类型用于确定在表达式中使用常量或在应用减号 ( `-` ) 时执行的转换类型。此列表汇总了整数常量的转换规则。

- 不带后缀的十进制常数的类型是 `int`、`long int` 或 `unsigned long int`。可用来表示常量值的三种类型中的第一个类型是分配给常量的类型。
- 分配给不带后缀的八进制常数和十六进制常数的类型是 `int`、`unsigned int`、`long int` 或 `unsigned long int`，具体视常数大小而定。
- 分配给带 `u` 或 `U` 后缀的常数的类型是 `unsigned int` 或 `unsigned long int`，具体视常数大小而定。
- 分配给带 `l` 或 `L` 后缀的常数的类型是 `long int` 或 `unsigned long int`，具体视常数大小而定。
- 分配给带 `u` 或 `U` 以及 `l` 或 `L` 后缀的常数的类型是 `unsigned long int`。

## 请参阅

C 整数常量

# C 和 C++ 整数限制

2021/8/16 •

Microsoft 专用

下表列出了 C 和 C++ 整数类型的限制。这些限制在 C 标准标头文件 `<limits.h>` 中定义。C++ 标准库标头 `<limits>` 包括 `<climits>`，其中包括 `<limits.h>`。

Microsoft C 还允许声明固定大小的整数变量，即大小为 8 位、16 位、32 位或 64 位的整数类型。有关 C 中固定大小整数的详细信息，请参阅固定大小整数类型。

## 对整数常量的限制

| “ | “ | “[” |
|---|---|---|
| CHAR_BIT | 不是位域的最小变量中的位数。 | 8 |
| SCHAR_MIN | 类型为 `signed char` 的变量的最小值。 | -128 |
| SCHAR_MAX | 类型为 `signed char` 的变量的最大值。 | 127 |
| UCHAR_MAX | 类型为 `unsigned char` 的变量的最大值。 | 255 (0xff) |
| CHAR_MIN | 类型为 `char` 的变量的最小值。 | -128；如果使用了 /J 选项，则为 0 |
| CHAR_MAX | 类型为 `char` 的变量的最大值。 | 127；如果使用了 /J 选项，则为 255 |
| MB_LEN_MAX | 多字符常量中的最大字节数。 | 5 |
| SHRT_MIN | 类型为 `short` 的变量的最小值。 | -32768 |
| SHRT_MAX | 类型为 `short` 的变量的最大值。 | 32767 |
| USHRT_MAX | 类型为 `unsigned short` 的变量的最大值。 | 65535 (0xffff) |
| INT_MIN | 类型为 `int` 的变量的最小值。 | -2147483647 - 1 |
| INT_MAX | 类型为 `int` 的变量的最大值。 | 2147483647 |
| UINT_MAX | 类型为 `unsigned int` 的变量的最大值。 | 4294967295 (0xffffffff) |
| LONG_MIN | 类型为 `long` 的变量的最小值。 | -2147483647 - 1 |
| LONG_MAX | 类型为 `long` 的变量的最大值。 | 2147483647 |

| “[” | “[” | “[” |
| --- | --- | --- |
| ULONG_MAX | 类型为 `unsigned long` 的变量的最大值。 | 4294967295 (0xffffffff) |
| LLONG_MIN | 类型为 `long long` 的变量的最小值。 | -9,223,372,036,854,775,807 - 1 |
| LLONG_MAX | 类型为 `long long` 的变量的最大值。 | 9,223,372,036,854,775,807 |
| ULLONG_MAX | 类型为 `unsigned long long` 的变量的最大值。 | 18,446,744,073,709,551,615 (0xffffffffffffffff) |

如果值超出了最大整数表示形式，则 Microsoft 编译器会产生错误。

结束 Microsoft 专用

## 请参阅

C 整数常量

# C 字符常量

2021/8/16 •

"字符常量"通过在单引号 (' ') 内封闭可表示的字符集中的单个字符来构成。字符常量用于表示执行字符集内的字符。

## 语法

*character-constant*: ' *c-char-sequence* '

L' *c-char-sequence* '

*c-char-sequence*: *c-char*

*c-char-sequence c-char*

*c-char*:除单引号 (' )、反斜杠 (\ ) 或者换行符以外的所有源字符集成员

escape-sequence

*escape-sequence*: *simple-escape-sequence*

*octal-escape-sequence*

*hexadecimal-escape-sequence*

*simple-escape-sequence*: one of **\a \b \f \n \r \t \v**

**\' \" \\ \?**

*octal-escape-sequence*: **** *octal-digit*

**** *octal-digit octal-digit*

**** *octal-digit octal-digit octal-digit*

*hexadecimal-escape-sequence*: **\x** *hexadecimal-digit*

*hexadecimal-escape-sequence hexadecimal-digit*

## 请参阅

C 常量

# 字符类型

2021/8/12 •

前面没有字母 L 的整型字符常数的类型为 `int`。包含单个字符的整数字符常量的值是解释为整数的字符的数字值。例如，字符 `a` 的数字值在十进制和十六进制下分别为 97 和 61。

从语法上来说，"宽字符常量"是带有字母 L 前缀的字符常量。宽字符常数的类型为 `wchar_t`（在 STDDEF.H 头文件中定义的整型类型）。例如：

```
char    schar =  'x';   /* A character constant          */
wchar_t wchar = L'x';   /* A wide-character constant for
                           the same character           */
```

宽字符常量的宽度为 16 位，用于指定扩展执行字符集的成员。借助它们，可以用字母表示因太大而无法用类型 `char` 表示的字符。有关宽字符的详细信息，请参阅多字节和宽字符。

## 请参阅

C 字符常量

# 执行字符集

2021/8/13 •

此内容通常指“执行字符集”。执行字符集不一定与用于编写 C 程序的源字符集相同。执行字符集包含源字符集中的所有字符以及 null 字符、换行符、退格符、水平制表符、垂直制表符、回车和转义序列。源和执行字符集在其他实现中可能不同。

## 请参阅

C 字符常量

# 转义序列

2021/8/17 •

由反斜杠 (\) 后接字母或数字组合构成的字符组合称为“转义序列”。要在字符常量中表示换行符，单引号或某些其他字符，你必须使用转义序列。转义序列被视为单个字符，因此，它是有效的字符常量。

转义序列通常用于指定操作，例如终端和打印机上的回车和制表符移动。它们还用于提供非打印字符的文本表现形式和通常具有特殊意义的字符，例如双引号 (") 。下表列出 ANSI 转义序列以及它们所表示的内容。

请注意，在字符序列被错误地解释为三元组的情况下，前接反斜杠 (\?) 的问号指定文本问号。有关详细信息，请参阅三元组。

**转义序列**

| tttt | tt |
| --- | --- |
| \a | 响铃(警报) |
| \b | Backspace |
| \f | 换页 |
| \n | 换行 |
| \r | 回车 |
| \t | 水平制表符 |
| \v | 垂直制表符 |
| \' | 单引号 |
| \" | 双引号 |
| \\ | 反斜杠 |
| \? | 文本问号 |
| \ *ooo* | 八进制表示法的 ASCII 字符 |
| \x *hh* | 十六进制表示法的 ASCII 字符 |
| \x *hhhh* | 十六进制表示法的 Unicode 字符(如果此转义序列用于宽字符常量或 Unicode 字符串文本)。<br><br>例如，`WCHAR f = L'\x4e00'` 或<br>`WCHAR b[] = L"The Chinese character for one is \x4e00"`<br>。 |

**Microsoft 专用**

如果反斜杠在表中未显示的字符前面，则编译器将未定义的字符作为字符本身进行处理。例如，`\c` 被视为 `c`。

结束 Microsoft 专用

转义序列允许你发送非图形控制字符到显示设备。例如，ESC 字符 (\033 ) 通常用作终端或打印机的控制命令的第一个字符。一些转义序列特定于设备。例如，垂直制表符和换页符转义序列(\v 和 \f )不会影响屏幕输出，但它们会执行适当的打印机操作。

还可以将反斜杠 (\ ) 用作继续符。当换行符(等效于按 RETURN 键)紧接反斜杠时，编译器会忽略反斜杠和换行符并将下一行作为前一行的一部分。这主要对长于一行的预处理器定义有用。例如：

```
#define assert(exp) \
( (exp) ? (void) 0:_assert( #exp, __FILE__, __LINE__ ) )
```

## 请参阅

C 字符常量

# 八进制和十六进制字符规范

2021/8/12 •

序列 \ ooo 表示可以将 ASCII 字符集中的任何字符指定为三位数八进制字符代码。八进制整数的数字值用于指定所需字符或宽字符的值。

同样，序列 \x hhh 可使用户将任何 ASCII 字符指定为十六进制字符代码。例如，可以将 ASCII 退格符指定为常规 C 转义序列 (\b )，或者也可以将其编码为 \010（八进制）或 \x008（十六进制）。

在八进制转义序列中只能使用 0 到 7 的数字。八进制转义序列绝不能长于三位且不能由第一个不是八进制数字的字符结尾。虽然不需要使用所有三个数字，但必须至少使用其中一个。例如，ASCII 退格符的八进制表示形式是 \10，字母 A 的八进制表示形式是 \101，如 ASCII 图表中所提供。

同样，对十六进制转义序列必须至少使用一个数字，但是您可以忽略第二个和第三个数字。因此，可以将退格符的十六进制转义序列指定为 \x8 、\x08 或 \x008 。

八进制或十六进制转义序列的值必须在字符常数的类型 `unsigned char` 和宽字符常数的类型 `wchar_t` 的可表示值的范围内。有关宽字符常量的信息，请参阅多字节和宽字符。

不同于八进制转义常量，转义序列中的十六进制数字的数量不受限制。十六进制转义序列在第一个不是十六进制数字的字符处结尾。由于十六进制数字包含字母 a 到 f，因此必须小心谨慎，确保转义序列在预期的数字处终止。为了避免混淆，您可以将八进制或十六进制字符定义放入宏定义：

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

对于十六进制值，可以拆开字符串以清楚地显示正确的值：

```
"\xabc"    /* one character  */
"\xab" "c" /* two characters */
```

## 请参阅

C 字符常量

# C 字符串文本

2021/8/16 •

“字符串文本”是封闭在双引号 (" ") 内的源字符集中的字符序列。字符串用于表示可一起构成以 null 结尾的字符串的字符序列。必须在宽字符串文本前添加字母 L 作为前缀。

## 语法

string-literal：
  " *s-char-sequence*$_{opt}$ "
  L" *s-char-sequence*$_{opt}$ "

s-char-sequence：
  *s-char*

  *s-char-sequence s-char*

s-char：
  除双引号 (")、反斜杠 (\) 或者换行符以外的任何源字符集成员

  *escape-sequence*

## 备注

以下示例是一个简单的字符串：

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

在转义序列表中列出的所有转义码在字符串文本中均有效。若要表示字符串文本中的双引号，请使用转义序列 \" 。可在不使用转义序列的情况下表示单引号 (' )。反斜杠 (\ ) 在字符串中出现时必须后跟另一个反斜杠 (\\ )。当反斜杠出现在行的末尾时，始终解释为行继续符。

## 请参阅

C 的元素

# 字符串文本的类型

2021/8/13 •

字符串文本有 `char` 类型数组(即 char[ ])。(宽字符字符串有 `wchar_t` 类型数组，即 wchar_t[ ]。)也就是说，字符串是包含类型为 `char` 的元素的数组。数组中的元素数等于字符串中的字符数加上结尾的 null 字符。

## 请参阅

C 字符串文本

# 字符串文本的存储

2021/8/16 •

文本字符串的字符将按顺序存储在连续内存位置。字符串文本中的转义序列(例如, \\ 或 \" )将作为单个字符进行计数。null 字符(由 \0 转义序列表示)自动追加到每个字符串并标记该字符串的末尾。(这会在转换阶段 7 出现。)请注意, 编译器无法在两个不同的地址存储两个相同的字符串。/GF 强制编译器将相同字符串的单个副本置于可执行文件中。

## 备注

**Microsoft 专用**

字符串具有静态存储持续时间。有关存储持续时间的信息, 请参阅存储类。

**结束 Microsoft 专用**

## 请参阅

C 字符串文本

# 字符串文本串联

2021/8/14 •

若要形成占用多行的字符串文本，则可以将两个字符串串联起来。为此，请键入反斜杠，然后按 Return 键。反斜杠将使编译器忽略以下换行符。例如，字符串文本

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

等同于字符串

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

在之前可能已使用过反斜杠后跟换行符的任何地方，都可以使用字符串串联，用来输入长于一行的字符串。

若要在字符串文本中强制换行，请在字符串中要换行的位置输入换行转义序列 (\n)，如下所示：

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

由于字符串可以在源代码的任何列中开始，而长字符串可以在后面的行的任何列中继续，因此可以放置字符串以增强源代码可读性。在任一情况下，在输出时，字符串的屏幕表示形式都不受影响。例如：

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

只要将字符串中的每个部分都用双引号括起来，则各个部分都将作为单个字符串进行串联和输出。此串联根据转换阶段指定的编译期间的事件序列发生。

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

初始化为仅用空白分隔的两个不同的字符串文本的字符串指针将作为单个字符串存储（指针在指针声明中讨论）。当正确引用后（如以下示例所示），结果与上一示例的相同：

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

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

在翻译阶段 6 中，相邻字符串文本或相邻宽字符串文本的任意序列指定的多字节字符序列被串联为一个多字节字符序列。因此，不要设计在执行期间允许修改字符串文本的程序。ANSI C 标准规定，修改字符串的结果是不确定的。

## 请参阅

C 字符串文本

# 最大字符串长度

2021/8/16 •

**Microsoft 专用**

ANSI 兼容性要求编译器在串联后接受字符串中最多 509 个字符。Microsoft C 中允许的字符串的最大长度约为 2,048 个字节。但是，如果字符串由用双引号引起来的多个部分构成，则预处理器会将这些部分串联为一个字符串，对于串联的每个行，它会将一个额外的字节添加到总字节数。

例如，假设字符串包含 40 个行（其中，每行 50 个字符，共 2,000 个字符）和一个包含 7 个字符的行，并且每个行是由双引号引起来的。所有这些字符共有 2,007 个字节，加上终止 null 字符的 1 个字节，总共为 2,008 个字节。在串联时，为前 40 个行中的每个行添加一个额外字符。这样将共有 2,048 个字节。但请注意，如果使用行继续符 (\) 代替双引号，则预处理器不会为每个行添加一个额外字符。

单个带引号的字符串的长度不能多于 2048 个字节，可以通过串联多个字符串来构造一个长度约为 65535 个字节的字符串。

**结束 Microsoft 专用**

## 请参阅

C 字符串文本

# 标点和特殊字符

2021/8/12 •

C 字符集中的标点和特殊字符各有其用途，从组织程序文本到定义编译器或已编译程序所执行的任务。它们不指定要执行的操作。某些标点符号也是运算符（请参阅运算符）。编译器从上下文确定其用途。

## 语法

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

这些字符在 C 中具有特殊含义。本书中将介绍其用途。井号 ( # ) 只能出现在 预处理指令中。

## 请参阅

C 的元素

# 程序结构

2021/8/15 •

本节概述了 C 程序和程序执行。还引入了对于了解 C 程序和组件来说很重要的术语和功能。讨论的主题包括：

- 源文件和源程序
- 主函数和程序执行
- 分析命令行参数
- 生存期、范围、可见性和链接
- 命名空间

由于本节只是为了进行概述，因此讨论的主题仅包含介绍性材料。有关更详细的解释，请参阅交叉引用信息。

## 请参阅

C 语言参考

# 源文件和源程序

2021/8/12 •

源程序可以分为一个或多个"源文件"或"翻译单元"。编译器的输入称为"翻译单元"。

## 语法

translation-unit：

*external-declaration*

*translation-unit external-declaration*

external-declaration：

*function-definition*

*declaration*

声明概述提供了 `declaration` 非终止符的语法，并且《预处理器参考》解释了翻译单元的处理方式。

> **NOTE**
>
> 有关 ANSI 语法约定的说明，请参阅 C 语言语法摘要的简介。

翻译单元的组件是包括函数定义和标识符声明的外部声明。这些声明和定义可以位于源文件、头文件、库和程序需要的其他文件中。必须编译每个翻译单元，并将生成的对象文件链接起来以生成程序。

A C"源程序"是指令、杂注、声明、定义、语句块和函数的集合。若要成为 Microsoft C 程序的有效组件，每个组件必须具有本书中描述的语法，但它们可以按照程序中的任何顺序显示（受本书中概述的规则的限制）。但是，这些组件在程序中的位置会影响变量和函数在程序中的使用方式。（有关详细信息，请参阅生存期、范围、可见性和链接。）

源文件无需包含可执行语句。例如，您可能发现将变量的定义放置在一个源文件中，然后在使用这些变量的其他源文件中声明对这些变量的引用会很有用。必要时，可通过此方法轻松查找和更新定义。由于相同的原因，常量和宏通常会被归类为称为"包含文件"或"头文件"的独立文件中，可在源文件中将这些文件引用为所需文件。有关宏和包含文件的详细信息，请参阅《预处理器参考》。

## 请参阅

程序结构

# 对预处理器的指令

2021/8/13 •

“指令”指示 C 预处理器在编译之前先对程序的文本执行特定操作。《预处理器参考》中完整描述了预处理器指令。本示例使用预处理器指令 `#define`：

```
#define MAX 100
```

该语句告知编译器在编译前将 `MAX` 的每个匹配项替换为 `100`。C 编译器预处理器指令为：

| #DEFINE | #ENDIF | #IFDEF | #LINE |
| --- | --- | --- | --- |
| `#elif` | `#error` | #ifndef | **#pragma** |
| `#else` | `#if` | `#include` | `#undef` |

## 请参阅

源文件和源程序

# C 杂注

2021/8/17 •

### Microsoft 专用

“杂注”指示编译器在编译时执行特定操作 。杂注随编译器的不同而不同。例如，可以使用 `optimize` pragma 来设置要对程序执行的优化。Microsoft C 杂注为：

`alloc_text`
`auto_inline`
`bss_seg`
`check_stack`
`code_seg`
`comment`
`component`
`const_seg`
`data_seg`
`deprecated`

`detect_mismatch`
`fenv_access`
`float_control`
`fp_contract`
`function`
`hdrstop`
`include_alias`
`inline_depth`
`inline_recursion`

`intrinsic`
`make_public`
`managed`
`message`
`omp`
`once`
`optimize`
`pack`
`pop_macro`

`push_macro`
`region`, `endregion`
`runtime_checks`
`section`
`setlocale`
`strict_gs_check`
`system_header`
`unmanaged`
`warning`

有关 Microsoft C 编译器 pragma 的说明，请参阅 Pragma 指令和 `__Pragma` 关键字。

结束 Microsoft 专用

## 请参阅

源文件和源程序

# C 声明和定义

2021/8/13 •

“声明”在特定变量、函数或类型及其特性之间建立关联。声明概述为 `declaration` 非终止符提供了 ANSI 语法。声明还指定可访问标识符的位置和时间(标识符的“链接”)。有关链接的信息，请参阅生存期、可见性和链接。

变量的“定义”将建立与声明建立的相同的关联，但也会导致为变量分配存储。

例如，`main`、`find` 和 `count` 函数以及 `var` 和 `val` 变量在一个源文件中定义，顺序如下：

```
int main() {}

int var = 0;
double val[MAXVAL];
char find( fileptr ) {}
int count( double f ) {}
```

变量 `var` 和 `val` 可用于 `find` 和 `count` 函数中；无需进一步声明。但是，这些名称在 `main` 中不可见(无法访问)。

## 请参阅

源文件和源程序

# 函数声明和定义

2021/8/13 •

函数原型确定了函数的名称、返回类型以及形参的类型和数量。函数定义包括函数体。

## 备注

函数声明和变量声明均可出现在函数定义的内部或外部。函数定义中的所有声明应在“内部”或“局部”级别显示。所有函数定义之外的声明应在“外部”、“全局”或“文件范围”级别显示。变量定义(如声明)可在内部级别(在函数定义中)或在外部级别(在所有函数定义外)显示。函数定义始终会在外部级别显示。函数定义中进一步讨论了函数定义。函数原型中介绍了函数原型。

## 请参阅

源文件和源程序

# Blocks

2021/8/15 •

包含在大括号 ({ }) 中的声明、定义和语句的序列称为“块”。C 中有两种类型的块。由一个或多个语句构成的语句“复合语句”(请参阅 复合语句)是一种类型的块。另一种类型的块是“函数定义”，它由一个复合语句(函数的主体)和函数的关联的“标头”(函数名称、返回类型和形参)构成。一个块位于其他块中的情况称作“嵌套”。

请注意，当所有复合语句包含在大括号内时，并非大括号内的所有内容都构成复合语句。例如，虽然数组、结构或枚举元素的说明可出现在大括号内，但它们不是复合语句。

## 请参阅

源文件和源程序

# 示例程序

2021/8/17 •

下面的 C 源程序包括两个源文件。它尽可能在 C 程序中提供一些各种声明和定义的概述。本书后面的各部分将介绍如何编写这些声明、定义和初始化，以及如何使用 C 关键字（如 `static` 和 `extern`）。在 C 标头文件 STDIO.H. 中声明 `printf` 函数。

假定 `main` 和 `max` 函数在单独的文件中，且程序的执行开始于 `main` 函数。未在 `main` 之前执行任何显式用户函数。

```
/*****************************************************************
                    FILE1.C - main function
*****************************************************************/

#define ONE     1
#define TWO     2
#define THREE   3
#include <stdio.h>

int a = 1;                      // Defining declarations
int b = 2;                      //  of external variables

extern int max( int a, int b );  // Function prototype

int main()                      // Function definition
{                               //  for main function
    int c;                      // Definitions for
    int d;                      //  two uninitialized
                                //  local variables

    extern int u;               // Referencing declaration
                                //  of external variable
                                //  defined elsewhere
    static int v;               // Definition of variable
                                //  with continuous lifetime

    int w = ONE, x = TWO, y = THREE;
    int z = 0;

    z = max( x, y );            // Executable statements
    w = max( z, w );
    printf_s( "%d %d\n", z, w );
    return 0;
}

/*****************************************************************
            FILE2.C - definition of max function
*****************************************************************/

int max( int a, int b )         // Note formal parameters are
                                //  included in function header
{
    if( a > b )
        return( a );
    else
        return( b );
}
```

FILE1.C 包含 `max` 函数的原型。这种声明有时称为“前向声明”，因为该函数在使用之前被声明。`main` 函数的定

义包含对 `max` 的调用。

以 `#define` 开始的行是预处理指令。这些指令告知预处理器通过 FILE1.C.用数字 `ONE`、`TWO` 和 `THREE` 分别替换标识符 `1`、`2` 和 `3`。但是，此类指令不适用于 FILE2.C，将单独对其进行编译，然后将其与 FILE1.C 链接。以 `#include` 开始的行告知编译器包含文件 STDIO.H，该文件包含 `printf` 函数的原型。预处理器指令在《预处理器参考》中进行了解释。

FILE1.C 使用定义声明初始化全局变量 `a` 和 `b`。声明但不初始化局部变量 `c` 和 `d`。为所有这些变量分配存储。静态和外部变量 `u` 和 `v` 将自动初始化为 0。因此，仅 `a`、`b`、`u` 和 `v` 在被声明时包含有意义的值，因为它们被显式或隐式初始化。FILE2.C 包含 `max` 的函数定义。此定义满足 FILE1.C. 中对 `max` 的调用。

生存期、范围、可见性和链接中讨论了标识符的生存期和可见性。有关函数的详细信息，请参阅函数。

## 请参阅

源文件和源程序

# main 函数和程序执行

2021/8/14 •

每个 C 程序都有必须命名为 main 的主函数。如果你的代码遵循 Unicode 编程模型，则可以使用 main 的宽字符版本 wmain 。main 函数充当程序执行的起点。它通常通过将调用定向到程序中的其他函数来控制程序执行。尽管程序可以因为各种原因在程序的其他点上终止，但它通常在 main 的结尾处停止执行。有时，当检测到某一错误时，您可能希望强制终止程序。为此，请使用 exit 函数。有关使用 exit 函数的信息和示例，请参阅《运行时库参考》。

## 语法

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

## 备注

源程序中的函数执行一个或多个特定任务。main 函数可调用这些函数来执行其各自的任务。当 main 调用另一函数时，它会将执行控制权交给该函数，以便执行在该函数中的第一个语句处开始。当执行 `return` 语句或到达函数末尾时，函数将控制权返回给 main。

可以声明任何函数（包括 main ）以包含参数。术语“参数”或“形参”指的是接收传递到函数的值的标识符。有关将实参传递到形参的信息，请参阅参数。当一个函数调用另一个函数时，被调用的函数将从实施调用的函数接收其参数的值。这些值称为“自变量”。可以将形参声明为 main ，以便让它使用以下格式从命令行接收实参：

在将信息传递给 main 函数时，尽管 C 编译器不需要这些名称，但上述参数在传统上命名为 `argc` 和 `argv` 。`argc` 和 `argv` 的类型由 C 语言定义。传统上，如果将第三个参数传递给 main ，该参数将命名为 `envp` 。本节后面的示例演示如何使用这三个参数访问命令行自变量。以下各节说明了这些参数。

有关 main 的宽字符版本的说明，请参阅使用 wmain。

## 请参阅

main 函数和命令行参数 (C++)
分析 C 命令行参数

# 使用 wmain

2021/8/17 •

**Microsoft 专用**

在 Unicode 编程模型中，可以定义 **main** 函数的宽字符版本。如果要编写符合 Unicode 编程模型的可移植代码，请使用 wmain 而不是 main 。

## 语法

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

## 备注

使用与 main 的相似格式声明 wmain 的形参。然后可以将宽字符自变量和宽字符环境指针(可选)传递给该程序。wmain 的 `argv` 和 `envp` 参数为 `wchar_t*` 类型。例如：

如果程序使用 main 函数，则多字节字符环境由运行时库在程序启动时创建。环境的宽字符副本仅在需要时创建(如调用 `_wgetenv` 或 `_wputenv` 函数时)。在首次调用 `_wputenv` 或首次调用 `_wgetenv` 时(如果 MBCS 环境已存在)，会创建一个对应的宽字符字符串环境，然后通过 `_wenviron` 全局变量指向该环境，此变量是 `_environ` 全局变量的宽字符版本。此时，同时存在该环境的两个副本(MBCS 和 Unicode)，在程序的整个生存期这两个副本由操作系统维护。

同样，如果程序使用 wmain 函数，则在程序启动时创建宽字符环境并用 `_wenviron` 全局变量指向该环境。MBCS (ASCII) 环境是在首次调用 `_putenv` 或 `getenv` 时创建的，并由 `_environ` 全局变量指向。

有关 MBCS 环境的详细信息，请参阅《运行时库参考》中的国际化。

**结束 Microsoft 专用**

## 请参阅

main 函数和程序执行

# 自变量说明

2021/8/14 •

main 和 wmain 函数中的 `argc` 形参是一个整数，用来指定从命令行传递到程序的实参的数量。由于程序名被视为实参，因此 `argc` 的值至少有一个。

## 备注

`argv` 形参是一个指针数组，这些指针指向表示程序实参的以 null 结尾的字符串。该数组的每个元素指向传递给 main（或 wmain）的参数的字符串表示形式。（有关数组的信息，请参阅数组声明。）`argv` 参数可以声明为指向类型 `char` （`char *argv[]`）的指针数组，也可以声明为一个指针来指向多个指向类型 `char` （`char **argv`）的指针。对于 wmain，`argv` 参数可以声明为指向类型 `wchar_t` （`wchar_t *argv[]`）的指针数组，也可以声明为一个指针来指向多个指向类型 `wchar_t` （`wchar_t **argv`）的指针。

按照约定，`argv`[0] 是用于调用程序的命令。但是，可以使用 CreateProcess 来生成进程，并且如果同时使用了第一个和第二个参数（`lpApplicationName` 和 `lpCommandLine`），`argv`[0] 可能不是可执行名称；请使用 GetModuleFileName 来检索可执行名称。

最后一个指针（`argv[argc]`）是 NULL 。（有关获取环境变量信息的替代方法，请参阅《运行时库参考》中的 getenv。）

**Microsoft 专用**

`envp` 参数是以 null 结尾的字符串的数组，这些字符串表示在用户的环境变量中设置的值。`envp` 参数可以声明为指向 `char` （`char *envp[]`）的指针数组，也可以声明为一个指针来指向多个指向 `char` （`char **envp`）的指针。在 wmain 函数中，`envp` 参数可以声明为指向 `wchar_t` （`wchar_t *envp[]`）的指针数组，也可以声明为一个指针来指向多个指向 `wchar_t` （`wchar_t **envp`）的指针。数组的末尾由 NULL * 指针来指示 。请注意，传递给 main 或 wmain 的环境块是当前环境的"冻结"副本。如果随后通过调用 _putenv 或 `_wputenv` 更改环境，则当前环境（由 `getenv` / `_wgetenv` 以及 `_environ` 或 `_wenviron` 变量返回）将发生更改，但 `envp` 指向的块将不会更改。`envp` 参数在 C 中是与 ANSI 兼容的，但在 C++ 中却不是如此。

**结束 Microsoft 专用**

## 请参阅

main 函数和程序执行

# 扩展通配符自变量

2021/8/25 •

通配符参数扩展是特定于 Microsoft 的。

在运行 C 程序时，你可以使用两个通配符（问号 ( `?` ) 和星号 ( `*` )）之一，以便在命令行上指定文件名和路径参数。

默认情况下，命令行参数中不展开通配符。你可以通过链接 `setargv.obj` 或 `wsetargv.obj` 文件将普通参数向量 `argv` 加载例程替换为展开通配符的版本。如果程序使用 `main` 函数，请与 `setargv.obj` 链接。如果程序使用 `wmain` 函数，请与 `wsetargv.obj` 链接。它们具有等效的行为。

若要与 `setargv.obj` 或 `wsetargv.obj` 链接，请使用 `/link` 选项。例如：

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

按照与操作系统命令相同的方式展开通配符。

## 请参阅

链接选项
`main` 函数和程序执行

# 分析 C 命令行自变量

2021/8/13 •

### Microsoft 专用

在解释操作系统命令行上给出的自变量时，Microsoft C 启动代码使用下列规则：

- 参数用空白分隔，空白可以是一个空格或制表符。
- 第一个参数 ( `argv[0]` ) 是经过专门处理的。它表示程序名称。因为它必须是有效的路径名，因此允许用双引号 ( `"` ) 括起来一些部分。双引号不包含在 `argv[0]` 输出中。用双引号括起来的部分可以防止将空格或制表符解释为参数的末尾。此列表中的后续规则不适用。
- 无论其中是否包含空格，双引号括起来的字符串均被解释为单个参数。带引号的字符串可以嵌入在自变量内。插入点 ( `^` ) 未被识别为转义字符或者分隔符。在带引号的字符串中，一对双引号被解释为单个转义的双引号。如果在找到右双引号之前命令行结束，则到目前为止读取的所有字符都将输出为最后一个参数。
- 前面有反斜杠的双引号 ( `\"` ) 被解释为原义双引号 ( `"` )。
- 反斜杠按其原义解释，除非它们紧位于双引号之前。
- 如果偶数个反斜杠后跟双引号，则每对反斜杠 ( `\\` ) 中有一个反斜杠 ( `\` ) 被置于 `argv` 数组中，而双引号 ( `"` ) 被解释为字符串分隔符。
- 如果奇数个反斜杠后跟双引号，则每对反斜杠 ( `\\` ) 中有一个反斜杠 ( `\` ) 被置于 `argv` 数组中。双引号由剩余反斜杠解释为转义序列，导致原义双引号 ( `"` ) 被置于 `argv` 中。

此列表通过显示命令行参数的多个示例的传递到 `argv` 的解释结果来阐释上述规则。在第二列、第三列和第四列中列出的输出来自于遵循列表的 ARGS.C 程序。

| tttttt | ARGV[1] | ARGV[2] | ARGV[3] |
|---|---|---|---|
| `"a b c" d e` | `a b c` | `d` | `e` |
| `"ab\"c" "\\" d` | `ab"c` | `\` | `d` |
| `a\\\b d"e f"g h` | `a\\\b` | `de fg` | `h` |
| `a\\\"b c d` | `a\"b` | `c` | `d` |
| `a\\\\"b c" d e` | `a\\b c` | `d` | `e` |
| `a"b"" c d` | `ab" c d` | | |

## 示例

### 代码

```
// ARGS.C illustrates the following variables used for accessing
// command-line arguments and environment variables:
// argc  argv  envp
//

#include <stdio.h>

int main( int argc, // Number of strings in array argv
char *argv[],      // Array of command-line argument strings
char **envp )      // Array of environment variable strings
{
    int count;

    // Display each command-line argument.
    printf_s( "\nCommand-line arguments:\n" );
    for( count = 0; count < argc; count++ )
        printf_s( "  argv[%d]   %s\n", count, argv[count] );

    // Display each environment variable.
    printf_s( "\nEnvironment variables:\n" );
    while( *envp != NULL )
        printf_s( "  %s\n", *(envp++) );

    return;
}
```

## 注释

此程序中输出的一个示例是：

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

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

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

结束 Microsoft 专用

## 请参阅

main 函数和程序执行

# 自定义 C 命令行处理

2021/8/17 •

如果程序不采用命令行参数，则可以取消命令行处理例程来节省少量空间。若要禁止使用该方法，请在 `/link` 编译器选项或 `LINK` 命令行中包含 `noarg.obj` 文件（用于 `main` 和 `wmain`）。

同样，如果从不通过 `envp` 参数访问环境表，则可以取消内部环境处理例程。若要禁止使用该方法，请在 `/link` 编译器选项或 `LINK` 命令行中包含 `noenv.obj` 文件（用于 `main` 和 `wmain`）。

有关运行时启动链接器选项的详细信息，请参阅链接选项。

程序可以调用 C 运行时库中的 `spawn` 或 `exec` 系列例程。如果是这样，则不应取消环境处理例程，因为可使用它将环境从父进程传递到子进程中。

## 另请参阅

`main` 函数和程序执行
链接选项。

# 生存期、范围、可见性和链接

2021/8/13 •

若要了解 C 程序的工作方式，您必须了解确定如何在程序中使用变量和函数的规则。有多个概念对于理解这些规则是至关重要的：

- 生存期
- 范围和可见性
- 链接

## 请参阅

程序结构

# 生存期

2021/8/16 •

“生存期”是其中存在变量或函数的程序执行的时段。标识符的存储持续时间决定其生存期。

使用 storage-class-specifier `static` 声明的标识符有静态存储持续时间。具有静态存储持续时间的标识符（也称为“全局”）具有存储和程序持续时间的定义值。将保留存储，并且在程序启动前只将标识符的存储值初始化一次。使用外部或内部链接声明的标识符还具有静态存储持续时间（请参阅链接）。

在函数内部没有使用 `static` 存储类说明符声明的标识符有自动存储持续时间。具有自动存储持续时间的标识符（“本地标识符”）具有存储和已定义的值（仅在定义或声明该标识符的块中）。程序每次进入该块时都将为自动标识符分配新存储，并在程序退出该块时丢失其存储（及其值）。函数中声明的不具有链接的标识符也具有自动存储持续时间。

以下规则指定标识符是具有全局（静态）还是局部（自动）生存期：

- 所有函数都具有静态生存期。因此，它们在程序执行期间始终存在。在外部级别（即，在同一级别函数定义的程序中的所有块之外）声明的标识符始终具有全局（静态）生存期。
- 如果局部变量有初始值设定项，则此变量在每次创建时都会进行初始化（除非它被声明为 `static`）。函数参数也具有本地生存期。通过在标识符声明中添加 `static` 存储类说明符，可以为块中的标识符指定全局生存期。一旦声明了 `static`，变量就会将它的值从块中的一个条目保留到下一个条目。

尽管有全局生存期的标识符（例如，外部声明的变量或使用 `static` 关键字声明的局部变量）在源程序的整个执行过程中都存在，但它可能并不在程序的所有部分中都可见。有关可见性的信息，请参阅范围和可见性；有关 storage-class-specifier 非终止符的讨论，请参阅存储类。

如果通过使用特殊库例程（如 `malloc`）创建，则可以根据需要分配内存（动态）。由于动态内存分配使用库例程，因此它不被视为语言的一部分。请参阅《运行时库参考》中的 malloc 函数。

## 请参阅

生存期、范围、可见性和链接

# 范围和可见性

2021/8/13 •

标识符的“可见性”确定其可以引用的程序部分，即其“范围”。标识符仅在其“范围”包含的程序部分中可见（即可使用），这可能仅限于（按限制增长的顺序）它显示在其中的文件、函数、块或函数原型。标识符的范围是可使用名称的程序的一部分。这有时被称为“词法范围”。有四种范围：函数、文件、块和函数原型。

除标签之外，所有标识符的范围都由在其上进行声明的级别决定。以下针对每种范围的规则将管理标识符在程序中的可见性：

文件范围 带文件范围的标识符的声明符或类型说明符显示在任何块或参数列表的外部，并且在其声明后可从翻译单元的任何位置进行访问。带文件范围的标识符名称通常称为“全局”或“外部”。全局标识符的范围开始于其定义或声明的点，结束于翻译单元的末尾。

函数范围 标签是唯一一种具有函数范围的标识符。通过在语句中使用标签来隐式声明标签。标签名称在函数中必须是唯一的。（有关标签和标签名称的详细信息，请参阅 goto 和 Labeled 语句。）

块范围 带块范围的标识符的声明符或类型说明符显示在块中或函数定义中的形参声明列表中。它仅从其声明或定义的点到包含其声明或定义的块的结尾可见。其范围限制为该块以及嵌入该块中的任何块，并结束于封闭该关联块的大括号处。此类标识符有时称为“局部变量”。

函数原型范围 带函数原型范围的标识符的声明符或类型说明符显示在函数原型（不是函数声明的一部分）中的参数声明列表中。其范围在函数声明符的末尾终止。

存储类别中介绍了可使变量在其他源文件中可见的适当声明。不过，在外部级别使用 `static` 存储类说明符声明的变量和函数只在定义它们的源文件中可见。所有其他函数都是全局可见的。

## 请参阅

生存期、范围、可见性和链接

# 生存期和可见性的摘要

2021/8/14 •

下表是大多数标识符的生存期和可见性特征的摘要。前三列提供了定义生存期和可见性的特性。具有前三列提供的特性的标识符具有在第四和第五列中显示的生存期和可见性。但是，该表未涵盖所有可能的情况。有关详细信息，请参考存储类。

**生存期和可见性的摘要**

| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
|---|---|---|---|---|
| 文件范围 | 变量定义 | `static` | Global | 此项所在的源文件的剩余部分 |
| | 变量声明 | `extern` | Global | 此项所在的源文件的剩余部分 |
| | 函数原型或定义 | `static` | Global | 单个源文件 |
| | 函数原型 | `extern` | Global | 源文件的剩余部分 |
| 块范围 | 变量声明 | `extern` | Global | 块 |
| | 变量定义 | `static` | Global | 块 |
| | 变量定义 | `auto` 或 `register` | 本地 | 块 |

## 示例

**描述**

以下示例演示了变量的块、嵌套和可见性：

**代码**

```
// Lifetime_and_Visibility.c

#include <stdio.h>

int i = 1;  // i defined at external level

int main()  // main function defined at external level
{
    printf_s( "%d\n", i ); // Prints 1 (value of external level i)
    {                                  // Begin first nested block
        int i = 2, j = 3;          // i and j defined at internal level
        printf_s( "%d %d\n", i, j );  // Prints 2, 3
        {                              // Begin second nested block
            int i = 0;                 // i is redefined
            printf_s( "%d %d\n", i, j ); // Prints 0, 3
        }                              // End of second nested block
        printf_s( "%d\n", i );         // Prints 2 (outer definition
                                       //  restored)
    }                                  // End of first nested block
    printf_s( "%d\n", i );             // Prints 1 (external level
                                       // definition restored)
    return 0;
}
```

**注释**

在此示例中，有四个级别的可见性：外部级别和三个块级别。值将输出到屏幕中，如每个语句后面的注释中所述。

## 请参阅

生存期、范围、可见性和链接

# 链接

2021/8/15 •

标识符名称可引用不同范围内的各个标识符。在不同的范围内或在同一范围内多次声明的标识符可以通过称为"链接"的过程来引用同一标识符或函数。链接确定可在其中引用标识符的程序的部分(其"可见性")。有三种链接:内部、外部和无链接。

## 请参阅

使用 extern 指定链接

# 内部链接

2021/8/12 •

如果对象或函数的文件范围标识符声明包含 storage-class-specifier `static`，则标识符有内部链接。否则，该标识符具有外部链接。有关 storage-class-specifier 非终止符的讨论，请参阅存储类。

在一个翻译单元内，带内部链接的标识符的每个实例均表示相同的标识符或函数。内部链接的标识符对于翻译单元是唯一的。

## 请参阅

使用 extern 指定链接

# 外部链接

2021/8/12 •

如果标识符在文件范围级别的第一个声明没有使用 `static` 存储类说明符，则对象有外部链接。

如果函数标识符的声明没有 storage-class-specifier，则它的链接确定方式与使用 storage-class-specifier `extern` 声明它时完全一样。如果对象标识符的声明具有文件范围但没有 storage-class-specifier，则其链接为外部的。

具有外部链接的标识符的名称指定相同的函数或数据对象，这与具有外部连接的相同名称的任何其他声明一样。这两个声明可以在同一个翻译单元中，也可以在不同的翻译单元中。如果该对象或函数还具有全局生存期，则该对象或函数由整个程序共享。

## 请参阅

使用 extern 指定链接

# 无链接

2021/8/12 •

如果块内某标识符的声明不包括 `extern` 存储类说明符，则此标识符没有链接，并且对函数是唯一的。

以下标识符没有链接：

- 声明为除对象或函数以外的任何项的标识符
- 声明为函数参数的标识符
- 声明不包含 `extern` 存储类说明符的对象的块范围标识符

如果标识符没有链接，那么在相同的范围级别内再次声明相同的名称（在声明符或类型说明符中）将产生符号重新定义错误。

## 请参阅

使用 extern 指定链接

# 命名空间

2021/8/13 •

编译器设置"命名空间"来区分用于各种项的标识符。每个命名空间中的名称必须是唯一的以避免冲突，但相同的名称可出现在多个命名空间中。这意味着，可以对两个或更多不同的项使用同一个标识符，前提是这些项位于不同的命名空间中。编译器可以基于程序中标识符的语义上下文来解析引用。

> **NOTE**
> 不要将命名空间的有限 C 概念与 C++"命名空间"功能混淆。有关详细信息，请参阅"C++ 语言参考"中的命名空间。

此列表描述了 C 中使用的命名空间。

语句标签 命名的语句标签是语句的一部分。语句标签的定义始终后跟一个冒号，但它们不是 `case` 标签的一部分。始终紧跟在关键字 `goto` 后面使用语句标签。语句标签不必与其他名称或其他函数中的标签名称有所不同。

结构、联合和枚举标记 这些标记是结构、联合和枚举类型说明符的一部分，如果存在，总是紧跟在保留字 `struct`、`union` 或 `enum` 后面。标记名称必须不同于具有相同可见性的所有其他结构、枚举或联合标记。

结构或联合的成员 成员名称分配在与各结构和联合类型关联的命名空间中。即，同一标识符可以同时为任意数量的结构或联合的组件名称。组件名称的定义总是出现在结构或联合类型说明符中。组件名称的使用总是紧跟在成员选择运算符（`->` 和 `.`）之后。成员的名称在结构或联合中必须是唯一的，但它无需不同于程序中的其他名称（包括不同的结构和联合的成员或结构本身的名称）。

普通标识符 所有其他名称都属于一个包含变量、函数（包括形参和局部变量）和枚举常量的命名空间。标识符名称具有嵌套可见性，因此您可以在块内重新定义它们。

Typedef 名称 Typedef 名称不能用作同一作用域内的标识符。

例如，由于结构标记、结构成员和变量名位于三个不同的命名空间中，因此该示例中名为 `student` 的三个项不会发生冲突。在该程序中，每个项的上下文允许对 `student` 的每个匹配项进行正确解释。（有关结构的信息，请参阅结构声明。）

```
struct student {
   char student[20];
   int class;
   int id;
   } student;
```

如果 `student` 出现在 `struct` 关键字后面，编译器将它识别为结构标记。当 `student` 出现在成员选择运算符（`->` 或 `.`）的后面时，名称将引用结构成员。在其他上下文中，`student` 引用结构变量。但是，建议不要重载标记命名空间，因为它会使含义变得模糊。

## 请参阅

程序结构

# 对齐 (C11)

2021/8/25 •

C 的低级功能之一是能够指定内存中对象的精确对齐方式，以最大限度利用硬件体系结构。

当数据存储在成倍数据大小的地址中时，CPU 会更有效地读取和写入内存。例如，如果数据存储在倍数为 4 的地址中，则会更有效地访问 4 字节整数。如果数据未对齐，则 CPU 需要执行更多地址计算工作来访问数据。

默认情况下，编译器会根据数据的大小对齐数据：`char` 在 1 字节的边界上对齐，`short` 在 2 字节的边界上对齐，`int`、`long` 和 `float` 在 4 字节的边界上对齐，`double` 在 8 字节的边界上对齐，依次类推。

另外，通过将常用数据与处理器的缓存行大小对齐，可以提高缓存性能。例如，假设定义了一个大小小于 32 个字节的结构。可能需要使用 32 字节对齐方式，以确保有效缓存结构的所有实例。

通常，无需担心对齐方式。编译器通常在基于目标处理器和数据大小的自然边界上对齐数据。在 32 位处理器上，数据最多在 4 字节的边界上对齐，在 64 位处理器上，数据最多在 8 字节的边界对齐。但是，在某些情况下，你可以通过指定数据结构的自定义对齐方式获得性能提升或节约内存。

使用 C11 关键字 `_Alignof` 来获取类型或变量的首选对齐方式，使用 `_Alignas` 来指定变量或用户定义类型的自定义对齐方式。

`<stdalign.h>` 中定义的便捷宏 `alignof` 和 `alignas` 分别映射到 `_Alignof` 和 `_Alignas`。这些宏与 C++ 中使用的关键字匹配。因此，如果你在两种语言之间共享任何代码，则使用宏（而不是 C 关键字）可能会对代码可移植性有所帮助。

## `alignas` 和 `_Alignas` (C11)

使用 `alignas` 或 `_Alignas` 来指定变量或用户定义类型的自定义对齐方式。它们可以应用于结构、联合、枚举或变量。

### `alignas` 语法

```
alignas(type)
alignas(constant-expression)
_Alignas(type)
_Alignas(constant-expression)
```

**备注**

`_Alignas` 不能在 typedef、位域、函数、函数参数或使用 `register` 说明符声明的对象的声明中使用。

指定幂为 2（如 1、2、4、8、16 等）的对齐方式。不要使用小于类型大小的值。

`struct` 和 `union` 类型的对齐方式与任何成员的最大对齐方式相等。在 `struct` 中添加填充字节，以确保满足各个成员的对齐要求。

如果声明中有多个 `alignas` 说明符（例如，具有多个成员的 `struct` 具有不同的 `alignas` 说明符），则 `struct` 的对齐方式将至少是最大说明符的值。

### `alignas` 实例

此示例使用便捷宏 `alignof`，因为它可移植到 C++。如果使用 `_Alignof`，则行为相同。

```
// Compile with /std:c11

#include <stdio.h>
#include <stdalign.h>

typedef struct
{
    int value; // aligns on a 4-byte boundary. There will be 28 bytes of padding between value and alignas
    alignas(32) char alignedMemory[32]; // assuming a 32 byte friendly cache alignment
} cacheFriendly; // this struct will be 32-byte aligned because alignedMemory is 32-byte aligned and is the
largest alignment specified in the struct

int main()
{
    printf("sizeof(cacheFriendly): %d\n", sizeof(cacheFriendly)); // 4 bytes for int value + 32 bytes for
alignedMemory[] + padding to ensure  alignment
    printf("alignof(cacheFriendly): %d\n", alignof(cacheFriendly)); // 32 because alignedMemory[] is aligned
on a 32-byte boundary

    /* output
        sizeof(cacheFriendly): 64
        alignof(cacheFriendly): 32
    */
}
```

## `alignof` 和 `_Alignof` (C11)

`_Alignof` 和别名 `alignof` 返回指定类型的对齐方式（以字节为单位）。它返回类型为 `size_t` 的值。

`alignof` 语法

```
alignof(type)
_Alignof(type)
```

`alignof` 实例

此示例使用便捷宏 `alignof`，因为它可移植到 C++。如果使用 `_Alignof`，则行为相同。

```
// Compile with /std:c11

#include <stdalign.h>
#include <stdio.h>

int main()
{
    size_t alignment = alignof(short);
    printf("alignof(short) = %d\n", alignment); // 2
    printf("alignof(int) = %d\n", alignof(int)); // 4
    printf("alignof(long) = %d\n", alignof(long)); // 4
    printf("alignof(float) = %d\n", alignof(float)); // 4
    printf("alignof(double) = %d\n", alignof(double)); // 8

    typedef struct
    {
        int a;
        double b;
    } test;

    printf("alignof(test) = %d\n", alignof(test)); // 8 because that is the alignment of the largest element
in the structure

    /* output

       alignof(short) = 2
       alignof(int) = 4
       alignof(long) = 4
       alignof(float) = 4
       alignof(double) = 8
       alignof(test) = 8
    */
}
```

## 要求

使用 `/std:c11` 进行编译。

Windows SDK 10.0.20348.0（版本 2104）或更高版本。请参阅 Windows 10 SDK 以下载最新 SDK。有关安装和使用 SDK 进行 C11 和 C17 开发的说明，请参阅在 Visual Studio 中安装 C11 和 C17 支持。

## 另请参阅

`/std`（指定语言标准版本）
C++ `alignof` 和 `alignas`
数据对齐的编译器处理

# 声明和类型

2021/8/16 •

本节介绍了变量、函数和类型的声明和初始化。C 语言包括一组标准的基本数据类型。您还可以通过基于已定义的类型声明新的类型来添加自己的数据类型（称为“派生类型”）。本文讨论了以下主题：

- 声明概述
- 存储类
- 类型说明符
- 类型限定符
- 声明符和变量声明
- 解释更复杂的声明符
- 初始化
- 基本类型的存储
- 不完整类型
- Typedef 声明
- 扩展的存储类特性

## 请参阅

C 语言参考

# 声明概述

2021/8/11 •

“声明”指定一组标识符的解释和特性。还将导致针对标识符命名的对象或函数保留存储的声明将称为“定义”。用于变量、函数和类型的 C 声明都具有以下语法：

## 语法

*declaration*：
  declaration-specifiers attribute-seq opt init-declarator-list opt ;

/* attribute-seq opt 是特定于 Microsoft 的 */

*declaration-specifiers*：
  storage-class-specifier declaration-specifiers opt
  type-specifier declaration-specifiers opt
  type-qualifier declaration-specifiers opt

*init-declarator-list*：
  *init-declarator*
  *init-declarator-list* , *init-declarator*

*init-declarator*：
  *declarator*
  *declarator* = *initializer*

> **NOTE**
> 下面几个部分中不会重复这种 declaration 语法。下面几个部分中的语法通常以 declarator 非终止符开头。

init-declarator-list 中的声明包含要命名的标识符；init 是初始值设定项的缩写。init-declarator-list 是用逗号分隔的声明符序列，其中每个声明符都可以有额外的类型信息和/或初始值设定项。declarator 包含要声明的标识符（若有）。declaration-specifiers 非终止符是由类型和存储类说明符的序列组成，这些说明符指明链接、存储持续时间，以及声明符表示的实体的至少一部分类型。声明由存储类说明符、类型说明符、类型限定符、声明符和初始值设定项的某种组合组成。

声明可以包含 attribute-seq 中列出的一个或多个可选特性；seq 是序列的缩写。这些特定于 Microsoft 的特性执行多个函数，本书中将详细介绍这些函数。

在一般形式的变量声明中，type-specifier 给出变量的数据类型。type-specifier 可以是复合的，就像类型被 const 或 volatile 修改时。declarator 可为变量命名，并可能将其修改以用于声明数组类型或指针类型。例如，应用于对象的

```
int const *fp;
```

将名为 fp 的变量声明为指向不可修改的 ( const ) int 值的指针。可以使用多个声明符（用逗号分隔）来定义声明中的多个变量。

声明必须至少具有一个声明符，或者其类型说明符必须声明一个结构标记、联合标记或枚举的成员。声明符提供有关标识符的所有剩余信息。声明符是一种标识符，可以用方括号 ( [ ] )、星号 ( * ) 或圆括号 ( ( ) ) 进行修改，

以分别声明数组、指针或函数类型。在声明简单变量(例如字符、整数和浮点项)或简单变量的结构和联合时，`declarator` 只是一个标识符。有关声明符的详细信息，请参阅声明符和变量声明。

所有定义都为隐式声明，但并非所有声明都为定义声明。例如，以 `extern` 存储类说明符开头的变量声明是"引用"声明，而不是"定义"声明。如果要在定义外部变量之前引用它，或者如果在使用它的源文件之外的另一个源文件中定义它，则需要 `extern` 声明。存储无法由"引用"声明分配，也不能在声明中初始化变量。

变量声明中需要存储类或类型(或者这两者)。除了 `__declspec` 之外，声明中只允许使用一个存储类说明符，并不是每个上下文中都允许使用所有存储类说明符。允许将 `__declspec` 存储类与其他存储类说明符一起使用，并且允许使用不止一次。声明的存储类说明符会影响存储和初始化声明的项的方式，还会影响程序中的哪些部分可以引用该项。

在 C 中定义的 `storage-class-specifier` 终止符包括 `auto`、`extern`、`register`、`static` 和 `typedef`。Microsoft C 还包括 `storage-class-specifier` 终止符 `__declspec`。存储类中讨论了除 `typedef` 和 `__declspec` 之外的所有 `storage-class-specifier` 终止符。若要了解 `typedef`，请参阅 `typedef` 声明。若要了解 `__declspec`，请参阅扩展的存储类特性。

源程序中声明的位置以及变量的其他声明存在与否是确定变量生存期的重要因素。可以有多个重新声明，但只能有一个定义。但是，定义可以出现在多个翻译单元中。对于带内部链接的对象，此规则可分别应用到每个翻译单元，因为内部链接的对象对翻译单元是唯一的。对于带外部链接的对象，此规则适用于整个程序。若要详细了解可见性，请参阅生存期、范围、可见性和链接。

类型说明符提供了一些有关标识符的数据类型的信息。默认类型说明符是 `int`。有关详细信息，请参阅类型说明符。类型说明符还可以定义类型标记、结构和联合组件名称以及枚举常量。有关详细信息，请参阅枚举声明、结构声明和联合声明。

有两个 `type-qualifier` 终止符：`const` 和 `volatile`。这些限定符指定仅在通过左值访问该类型的对象时才相关的类型的其他属性。若要详细了解 `const` 和 `volatile`，请参阅类型限定符。有关左值的定义，请参阅左值和右值表达式。

## 请参阅

C 语言语法摘要
声明和类型
声明摘要

# C 存储类

2021/8/17 •

变量的"存储类"可确定项是具有"全局"还是"本地"生存期。C 将这两个生存期称为"静态"和"自动"。具有全局生存期的项存在且具有贯穿整个程序执行过程的值。所有函数都具有全局生存期。

每次执行控制权传递到从中定义它们的块时，都会为自动变量或具有本地生存期的变量分配新存储。当执行返回时，这些变量不再具有有意义的值。

C 提供了以下存储类说明符：

## 语法

*storage-class-specifier*:

`auto`

`register`

`static`

`extern`

`typedef`

`__declspec (` extended-decl-modifier-seq `)` /* 特定于 Microsoft */

除了 `__declspec` 之外，只能在声明中的 declaration-specifier 内使用一个 storage-class-specifier。如果没有制定存储类规范，块中的声明将创建自动对象。

使用 `auto` 或 `register` 说明符声明的项有本地生存期。使用 `static` 或 `extern` 说明符声明的项有全局生存期。

由于 `typedef` 和 `__declspec` 在语义上与其他四个 storage-class-specifier 终止符不同，因此将分开介绍它们。有关 `typedef` 的具体信息，请参阅 `typedef` 声明。有关 `__declspec` 的具体信息，请参阅扩展的存储类特性。

源文件中变量和函数声明的位置还会影响存储类和可见性。所有函数定义之外的声明据说显示在"外部级别"。函数定义中的声明显示在"内部级别"。

每个存储类说明符的确切含义取决于两个因素：

- 声明是显示在外部还是内部级别
- 要声明的项是变量还是函数

用于外部级别声明的存储类说明符和用于内部级别的存储类说明符介绍了每种声明中的 storage-class-specifier 终端并解释了从变量中省略 storage-class-specifier 时的默认行为。存储类说明符与函数声明讨论了与函数一起使用的存储类说明符。

## 请参阅

声明和类型

# 外部级别声明的存储类说明符

2021/8/12 •

外部变量是文件范围内的变量。它们在任何函数的外部定义，并且可能对许多函数可用。只能在外部级别定义函数，因此不能将其嵌套。默认情况下，所有对同名外部变量和函数的引用都是对同一个对象的引用，也就是说它们具有外部链接。（可以使用 `static` 关键字来替代此行为。）

外部级别的变量声明要么是变量的定义（定义声明），要么是对其他地方定义的变量的引用（引用声明）。

（隐式或显式）初始化变量的外部变量声明是变量的定义声明。外部级别的定义可采用多种形式：

- 使用 `static` 存储类说明符声明的变量。可以使用常数表达式来显式初始化 `static` 变量，如初始化中所述。如果省略初始值设定项，默认情况下变量将初始化为 0。例如，这两个语句都被视为变量 `k` 的定义。

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

- 在外部级别显式初始化的变量。例如，`int j = 3;` 是变量 `j` 的定义。

在外部级别（即在所有函数之外）的变量声明中，可以使用 `static` 或 `extern` 存储类说明符，也可以完全省略存储类说明符。不能在外部级别使用 `auto` 和 `register` `storage-class-specifier` 终止符。

一旦在外部级别定义变量，该变量在翻译单元的其余部分便可见。该变量在同一源文件中的声明之前不可见。此外，除非引用声明使其可见，否则它在程序的其他源文件中不可见，如下文所述。

与 `static` 相关的规则包括：

- 在所有块之外声明的、没有 `static` 关键字的变量在整个程序中始终保持自己的值不变。若要限制它们对特定翻译单元的访问，必须使用 `static` 关键字。这给了它们内部链接。若要让它们成为整个程序的全局变量，请省略显式存储类或使用关键字 `extern`（见下一个列表中的规则）。这给了它们外部链接。内部链接和外部链接也在链接中进行了讨论。

- 在程序中，只可在外部级别定义变量一次。可以在不同的翻译单元中定义另一个具有相同名称和 `static` 存储类说明符的变量。由于每个 `static` 定义只在它自己的翻译单元中可见，因此不会发生任何冲突。这提供了一种实用的方法来隐藏标识符名称，这些标识符名称必须在一个翻译单元的函数之间共用，但对其他翻译单元不可见。

- `static` 存储类说明符也可以应用于函数。如果你声明函数 `static`，那么它的名称在声明它的文件之外是不可见的。

`extern` 的使用规则为：

- `extern` 存储类说明符声明对在其他地方定义的变量的引用。可以使用 `extern` 声明让另一个源文件中的定义可见，或让变量先于同一源文件中的定义可见。一旦你在外部级别声明了对变量的引用，此变量就在声明的引用所在的整个翻译单元的其余部分都是可见的。

- 若要使 `extern` 引用有效，它所引用的变量必须在外部级别定义一次，且只定义一次。此定义（不包含 `extern` 存储类）可以位于组成程序的任何翻译单元中。

## 示例

下面的示例阐释了外部声明：

```
/*************************************************************************
                     SOURCE FILE ONE
*************************************************************************/
#include <stdio.h>

extern int i;                // Reference to i, defined below
void next( void );           // Function prototype

int main()
{
    i++;
    printf_s( "%d\n", i );   // i equals 4
    next();
}

int i = 3;                  // Definition of i

void next( void )
{
    i++;
    printf_s( "%d\n", i );  // i equals 5
    other();
}

/*************************************************************************
                     SOURCE FILE TWO
*************************************************************************/
#include <stdio.h>

extern int i;              // Reference to i in
                           // first source file
void other( void )
{
    i++;
    printf_s( "%d\n", i ); // i equals 6
}
```

此示例中的两个源文件包含 `i` 的总共三个外部声明。仅一个声明是"定义声明"。该声明

```
int i = 3;
```

定义全局变量 `i` 并使用初始值 3 对其进行初始化。使用 `extern` 的第一个源文件顶部的 `i` 的"引用"声明让全局变量先于文件中的定义声明可见。此外，第二个源文件中的 `i` 的引用声明使变量在该源文件中可见。如果翻译单元中未提供变量的定义实例，则编译器假定有

```
extern int x;
```

一个引用声明，并且定义引用

```
int x = 0;
```

出现在程序的另一个翻译单元中。

所有三个函数（`main`、`next` 和 `other`）都执行同一任务：它们增大 `i` 并将其打印。打印值 4、5 和 6。

如果变量 `i` 还没有被初始化，则会自动设置为 0。在这种情况下，值 1、2 和 3 可能已打印。有关变量初始化的信息，请参阅初始化。

请参阅

C 存储类

# 内部级别声明的存储类说明符

2021/8/14 •

可以使用四个 `storage-class-specifier` 终止符中的任何一个在内部级别声明变量。如果从这样的声明中省略 `storage-class-specifier`，默认存储类为 `auto`。因此，关键字 `auto` 在 C 程序中很少出现。

## 请参阅

C 存储类

# `auto` 存储类说明符

2021/8/13 •

`auto` 存储类说明符声明具有本地生存期的自动变量。 `auto` 变量只在声明它的代码块中可见。 `auto` 变量的声明可包含初始值设定项，如初始化中所述。由于包含 `auto` 存储类的变量不会自动初始化，因此应在声明这些变量时显式初始化它们，或在代码块的语句中将初始值赋给它们。未初始化的 `auto` 变量的值是未定义的。（如果给定了初始值设定项，则每次 `auto` 或 `register` 存储类的局部变量进入范围时都会被初始化。）

内部 `static` 变量（具有局部或块范围的静态变量）可以使用任何外部项或 `static` 项的地址进行初始化，但不能使用另一个 `auto` 项的地址进行初始化，因为 `auto` 项的地址不是常数。

## 请参阅

`auto` 关键字

# register 存储类说明符

2021/8/16 •

**Microsoft 专用**

Microsoft C/C++ 编译器不会按照用户请求来使用寄存器变量。不过，为了确保可移植性，编译器将遵循与 `register` 关键字关联的其他所有语义。例如，无法对寄存器对象应用一元取址运算符 (&)，也无法对数组使用 `register` 关键字。

**结束 Microsoft 专用**

## 请参阅

内部级别声明的存储类说明符

# 静态存储类说明符

2021/8/16 •

在内部级别使用 `static` 存储类说明符声明的变量有全局生存期，但它仅在声明它的块中可见。对于常数字符串，使用 `static` 是有用的，因为它减轻了频繁初始化经常调用的函数的开销。

## 备注

如果没有显式初始化 `static` 变量，则它默认初始化为 0。在函数内，`static` 会导致存储被分配并用作定义。内部静态变量仅提供对单个函数可见的私有永久存储。

## 请参阅

C 存储类
存储类 (C++)

# extern 存储类说明符

2021/8/16 •

使用 `extern` 存储类说明符声明的变量是对另一个源文件中定义的同名变量的引用。它用于显示外部级别变量定义。声明为 `extern` 的变量没有为自己分配存储；它只是名称。

## 示例

此示例阐释内部和外部级别的声明：

```
// Source1.c

int i = 1;

// Source2. c

#include <stdio.h>

// Refers to the i that is defined in Source1.c:
extern int i;

void func(void);

int main()
{
    // Prints 1:
    printf_s("%d\n", i);
    func();
    return;
}

void func(void)
{
    // Address of global i assigned to pointer variable:
    static int *external_i = &i;

    // This definition of i hides the global i in Source.c:
    int i = 16;

    // Prints 16, 1:
    printf_s("%d\n%d\n", i, *external_i);
}
```

在此示例中，变量 `i` 是在 Source1.c 中定义，初始值为 1。Source2.c 中的 `extern` 声明让“i”在此文件中可见。

在 `func` 函数中，全局变量 `i` 的地址用于初始化 `static` 指针变量 `external_i`。此用法之所以有效是因为，全局变量有 `static` 生存期；也就是说，它的地址在程序执行期间不会改变。接下来，变量 `i` 在 `func` 的范围内定义为初始值为 16 的局部变量。此定义不会影响外部级别 `i` 的值（通过对局部变量使用它的名称来隐藏它）。全局 `i` 的值现在只能通过指针 `external_i` 访问。

## 请参阅

内部级别声明的存储类说明符

# 具有函数声明的存储类说明符

2021/8/13 •

在函数声明中，可以使用 `static` 或 `extern` 存储类说明符。函数始终具有全局生存期。

**Microsoft 专用**

内部级别的函数声明的含义与外部级别的函数声明的含义相同。这意味着，在翻译单元的剩余部分中，该函数从其声明点可见，即使它是在局部范围内声明的。

**结束 Microsoft 专用**

函数的可见性规则与变量的规则略有不同，如下所示：

- 声明为 `static` 的函数只在定义它的源文件中可见。同一个源文件中的函数可以调用 `static` 函数，但其他源文件中的函数无法按名称直接访问它。可以在不同的源文件中声明另一个同名的 `static` 函数，而不会产生冲突。
- 声明为 `extern` 的函数在程序中的所有源文件中都可见（除非稍后将此类函数重新声明为 `static`）。任何函数都可以调用 `extern` 函数。
- 默认情况下，省略存储类说明符的函数声明为 `extern`。

**Microsoft 专用**

Microsoft 允许将 `extern` 标识符重新定义为 `static`。

**结束 Microsoft 专用**

## 请参阅

C 存储类

# C 类型说明符

2021/8/13 •

声明中的类型说明符定义变量或函数声明的类型。

## 语法

type-specifier: `void` `char` `short` `int` `long` `float` `double` `signed` `unsigned` struct-or-union-specifier enum-specifier typedef-name

`signed char`、`signed int`、`signed short int` 和 signed long int 类型以及它们对应的 `unsigned` 和 `enum` 一起称为"整型类型"。`float`、`double` 和 `long double` 类型说明符称为"浮点"或"浮点类型"。可在变量或函数声明中使用任何整型或浮点型说明符。如果在声明中没有提供 type-specifier，则将其视为 `int`。

可选关键字 `signed` 和 `unsigned` 可位于任何整型类型的前面或后面（`enum` 除外），还可以单独用作类型说明符（在这种情况下，它们分别被理解为 `signed int` 和 `unsigned int`）。单独使用时，关键字 `int` 被假定为 `signed`。单独使用时，关键字 `long` 和 `short` 被理解为 long int 和 `short int`。

枚举类型被视为基本类型。枚举声明中讨论了枚举类型的类型说明符。

关键字 `void` 有三种用途：一是指定函数返回类型，二是为不接受参数的函数指定参数类型列表，三是指定一个指向未指定的类型的指针。可以使用 `void` 类型来声明不返回值的函数，或声明指向未指定的类型的指针。若要了解单独出现在函数名称后面的括号中的 `void`，请参阅参数。

### Microsoft 专用

类型检查现在符合 ANSI；也就是说，`short` 类型和 `int` 类型是不同的类型。例如，下面是编译器的早期版本接受的 Microsoft C 编译器中的一个重新定义。

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

下一个示例还会生成有关到不同类型的间接寻址的警告：

```
int *pi;
short *ps;

ps = pi;  /* Now generates warning */
```

Microsoft C 编译器还生成保留符号的差异警告。例如：

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

pi = pu;  /* Now generates warning */
```

对类型 `void` 表达式计算副作用。不能以任何方式使用类型为 `void` 的表达式的（不存在的）值，也不能将 `void` 表达式转换为除 `void` 以外的任何类型（通过隐式或显式转换）。如果你确实在需要 `void` 表达式的上下文中使用了其他任何类型的表达式，则其值将被放弃。

为了符合 ANSI 规范，void** 不能用作 int**。只有 `void` * 才能用作指向未指定的类型的指针。

结束 Microsoft 专用

可以使用 `typedef` 声明创建额外的类型说明符，如 Typedef 声明中所述。有关每个类型的大小的信息，请参阅基本类型的存储。

## 请参阅

声明和类型

# 数据类型说明符和等效项

2021/8/12 •

该文档通常使用下表中列出的类型说明符的形式(而不是长形式)。该文档还假定默认情况下 `char` 类型是带符号的。在整个文档中，`char` 与 `signed char` 等效。

## 类型说明符和等效项

| | |
|---|---|
| `signed char`[1] | `char` |
| `signed int` | `signed`, `int` |
| `signed short int` | `short`, `signed short` |
| `signed long int` | `long`, `signed long` |
| `unsigned char` | — |
| `unsigned int` | `unsigned` |
| `unsigned short int` | `unsigned short` |
| `unsigned long int` | `unsigned long` |
| `float` | — |
| `long double`[2] | — |

[1] 当在默认情况下将 `char` 类型设为无符号时(通过指定 `/J` 编译器选项)，无法将 `signed char` 缩写为 `char`。

[2] 在 32 位和 64 位操作系统中，Microsoft C 编译器将 `long double` 映射到类型 `double`。

**Microsoft 专用**

可以通过指定 `/J` 编译器选项，将默认 `char` 类型从 `signed char` 更改为 `unsigned char`。当此选项生效时，`char` 的含义与 `unsigned char` 相同，你必须使用 `signed` 关键字来声明带符号字符值。如果 `char` 值被显式声明为 `signed`，则 `/J` 选项不影响它，并且当加宽为 `int` 类型时，值是符号扩展的。当加宽为 `int` 类型时，`char` 类型是零扩展的。

**结束 Microsoft 专用**

## 请参阅

C 类型说明符

# 类型限定符

2021/8/25 •

类型限定符为标识符提供两个属性之一。`const` 类型限定符将对象声明为不可修改。`volatile` 类型限定符声明了一个项，此项的值可以被超出此项所在程序的控制范围的某个项（如并发执行的线程）以合法方式更改。

`const`、`restrict` 和 `volatile` 这几个类型限定符只能在声明中出现一次。类型限定符可与任何类型说明符一起出现；但是，它们不能在多项声明中的第一个逗号的后面出现。例如，以下声明是合法的：

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

以下声明是非法的：

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

仅当访问作为表达式的左值的标识符时，类型限定符才会相关。有关左值和表达式的信息，请参阅左值表达式和右值表达式。

## 语法

*`type-qualifier`*：

`const`

`restrict`

`volatile`

## `const` 和 `volatile`

以下是合法的 `const` 和 `volatile` 声明：

```
int const *p_ci;      // Pointer to constant int
int const (*p_ci);    // Pointer to constant int
int *const cp_i;      // Constant pointer to int
int (*const cp_i);    // Constant pointer to int
int volatile vint;      // Volatile integer
```

如果数组类型的规范包括类型限定符，则将限定元素而不是数组类型。如果函数类型的规范包括限定符，则行为是不确定的。`volatile` 和 `const` 都不会影响值的范围或对象的算术属性。

- `const` 关键字可用于修改任何基本或聚合类型，或修改指向任何类型的对象的指针或 `typedef`。如果某项声明为只包含 `const` 类型限定符，则其类型被视为 const int。`const` 变量可以被初始化，也可以被置于存储的只读区域中。`const` 关键字对于声明指向 `const` 的指针很有用，因为这要求函数不以任何方式更改指针。
- 编译器假定，在程序运行的任何时刻，使用或修改 `volatile` 变量的值的未知进程都可以访问此变量。无论在命令行上指定了哪些优化，都必须为 `volatile` 变量的每次赋值或引用生成代码，即使它看起来没有任何效果。

如果单独使用 `volatile`，则假定为 `int`。`volatile` 类型说明符可用于提供对特殊内存位置的可靠访问。将

`volatile` 用于数据对象，这些对象可能会被信号处理程序、并行执行的程序或特殊硬件（如内存映射的 I/O 控制寄存器）访问或更改。可以将变量在其生存期内声明为 `volatile`，也可以将单个引用强制转换为 `volatile`。

- 一个项可以同时是 `const` 和 `volatile`，在这种情况下，此项不能被它自己的程序以合法方式修改，但能被一些异步进程修改。

## `restrict`

C99 中引入的 `restrict` 类型限定符（以 `/std:c11` 或 `/std:c17` 模式提供）可应用于指针声明。它限定指针，而不是它指向的内容。

`restrict` 是编译器的优化提示，当前作用域中没有其他指针引用相同的内存位置。也就是说，只使用指针或从其派生的值（例如指针 + 1）在指针的生存期内访问对象。这有助于编译器生成更多的优化代码。C++ 具有等效机制 `__restrict`

请注意，`restrict` 是你与编译器之间的协定。如果你对标记为 `restrict` 的指针使用了别名，则结果是不确定的。

下面是使用 `restrict` 的示例：

```
void test(int* restrict first, int* restrict second, int* val)
{
    *first += *val;
    *second += *val;
}

int main()
{
    int i = 1, j = 2, k = 3;
    test(&i, &j, &k);

    return 0;
}

// Marking union members restrict tells the compiler that
// only z.x or z.y will be accessed in any scope, which allows
// the compiler to optimize access to the members.
union z
{
    int* restrict x;
    double* restrict y;
};
```

# 另请参阅

`/std`（指定语言标准版本）
声明和类型

# 声明符和变量声明

2021/8/12 •

本节的其余部分描述了该列表中汇总的变量类型的声明的形式和含义。具体来说，其余各部分说明如何声明以下内容：

| | |
|---|---|
| 简单变量 | 带整型或浮点型的单值变量 |
| 数组 | 由类型相同的一系列元素构成的变量 |
| 指针 | 指向其他变量并包含变量位置（以地址的形式）而不是值的变量 |
| 枚举变量 | 带整型的简单变量，其中包含命名整数常量组中的某个值 |
| 结构 | 由一系列可具有不同类型的值构成的变量 |
| Unions | 由占用相同的存储空间的不同类型的多个值构成的变量 |

声明符是声明的一部分，它指定要引入程序中的名称。它可以包括修饰符（如 `*` (pointer-to)）和任何 Microsoft 调用约定关键字。

**Microsoft 专用**

在此声明符中，

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

**是类型说明符，** `__declspec(thread)` **和** `*` **是修饰符，** `var` **是标识符名称** `char` 。

**结束 Microsoft 专用**

可使用声明符声明值的数组、指向值的指针和返回指定类型的值的函数。声明符出现在本节后面描述的数组和指针声明中。

## 语法

*declarator*:
*pointer* opt *direct-declarator*

*direct-declarator*:
*identifier*
( *declarator* )
*direct-declarator* [ *constant-expression* opt ]
*direct-declarator* ( *parameter-type-list* )
*direct-declarator* ( *identifier-list* opt )

*pointer*:
* *type-qualifier-list* opt

`*` *type-qualifier-list* opt *pointer*

*type-qualifier-list*:

*type-qualifier*

*type-qualifier-list* *type-qualifier*

> **NOTE**
> 有关引用 `declarator` 的语法，请查看声明概述或 C 语言语法摘要中的 `declaration` 的语法 。

当声明符包含未修改的标识符时，正在声明的项将具有基类型。如果星号 ( `*` ) 显示在标识符的左侧，则将类型修改为指针类型。如果标识符后跟方括号 ( `[ ]` )，则将类型修改为数组类型。如果标识符后跟圆括号，则将类型修改为函数类型。若要详细了解如何解释声明中的优先级，请查看解释更复杂的声明符。

每个声明符至少声明一个标识符。声明符必须包含一个类型说明符才能成为完整声明。类型说明符显示：数组类型的元素的类型、指针类型寻址的对象的类型，或者函数的返回类型。

数组和指针声明将在本节后面做更详细地讨论。以下示例阐释声明符的几种简单形式：

```
int list[20]; // Declares an array of 20 int values named list
char *cp;     // Declares a pointer to a char value
double func( void ); // Declares a function named func, with no
                     // arguments, that returns a double value
int *aptr[10] // Declares an array of 10 pointers
```

**Microsoft 专用**

Microsoft C 编译器不限制可修改算法、结构和联合类型的声明符的数目。该数字仅受可用内存限制。

**结束 Microsoft 专用**

## 请参阅

声明和类型

# 简单变量声明

2021/8/12 •

简单变量的声明(直接声明符的最简单形式)指定变量的名称和类型。它还指定变量的存储类和数据类型。

变量声明需要存储类和/或类型。非类型化变量(如 `var;`)会生成警告。

## 语法

*declarator*:
  *pointer*$_{opt}$ *direct-declarator*

*direct-declarator*:
  *identifier*

identifier :
  *nondigit*
  *identifier nondigit*
  *identifier digit*

对于算术、结构、联合、枚举和 void 类型以及由 `typedef` 名称表示的类型，可以在声明中使用简单声明符，因为类型说明符提供了所有类型化信息。指针、数组和函数类型需要更复杂的声明符。

可以使用一系列由逗号 (,) 分隔的标识符来指定同一个声明中的多个变量。声明中定义的所有变量都具有相同的基类型。例如:

```
int x, y;        /* Declares two simple variables of type int */
int const z = 1; /* Declares a constant value of type int */
```

变量 `x` 和 `y` 可以保留由 `int` 类型为特定实现定义的集中的任何值。简单对象 `z` 将初始化为值 1 且不可修改。

如果 `z` 的声明针对未初始化的静态变量或在文件范围内，则它将接收初始值 0，并且该值是不可修改的。

```
unsigned long reply, flag; /* Declares two variables
                              named reply and flag     */
```

在此示例中，`reply` 和 `flag` 这两个变量都是 `unsigned long` 类型，并包含不带符号整数值。

## 请参阅

声明符和变量声明

# C 枚举声明

2021/8/14 •

枚举由一组命名整数常量构成。枚举类型声明提供(可选)枚举标记的名称。并且，它定义了一组已命名的整数标识符(称为"枚举集"、"枚举器常量"、"枚举器"或"成员")。枚举类型的变量存储该类型所定义的枚举集的值之一。

类型为 `enum` 的变量可用于索引表达式，并且可用作所有算术和关系运算符的操作数。枚举提供了 `#define` 预处理器指令的替代方法，带来的好处是可为您生成值并遵循一般范围规则。

在 ANSI C 中，定义枚举器常量值的表达式始终具有 `int` 类型。这意味着，与枚举变量关联的存储是单个 `int` 值所需的存储。可以在 C 语言允许整数表达式的任意位置使用枚举常量或枚举类型的值。

## 语法

*enum-specifier*:
- enum *identifier* opt { *enumerator-list* }
- enum *identifier*

*enumerator-list*:
- *enumerator*
- *enumerator-list* , *enumerator*

*enumerator*:
- *enumeration-constant*
- *enumeration-constant* = *constant-expression*

*enumeration-constant*:
- *identifier*

可选的 `identifier` 命名由 `enumerator-list` 定义的枚举类型 。此标识符通常称为列表指定的枚举的"标记"。类型说明符声明 `identifier` 是由 `enumerator-list` 非终止符指定的枚举的标记，如下所示：

```
enum identifier
{
    // enumerator-list
}
```

`enumerator-list` 定义枚举集的成员。

如果标记的声明可见，则后续使用标记但忽略 `enumerator-list` 的声明将指定之前声明的枚举的类型。标记必须引用定义的枚举类型，并且该枚举类型必须在当前范围内。由于在其他位置定义枚举类型，因此 `enumerator-list` 不会出现在此声明中。派生自枚举的类型的声明和枚举类型的 `typedef` 声明可以在定义枚举类型之前使用枚举标记。

`enumerator-list` 中的每个 `enumeration-constant` 命名一个枚举集的值 。默认情况下，第一个 `enumeration-constant` 与值 0 相关联。列表中的下一个 `enumeration-constant` 与 ( `constant-expression` + 1) 的值相关联，除非显式将其与另一个值相关联 。 `enumeration-constant` 的名称与其值等效。

可使用 `enumeration-constant` = `constant-expression` 替代值的默认序列 。也就是说，如果 `enumeration-constant` = `constant-expression` 出现在 `enumerator-list` 中，则 `enumeration-constant` 与 `constant-expression` 给定的值相关联 。 `constant-expression` 必须是 `int` 类型，并且可以为负。

下面的规则适用于枚举集的成员：

- 枚举集可以包含重复的常量值。例如，可以将值 0 与两个不同的标识符（例如同一集合中名为 `null` 和 `zero` 的成员）关联。
- 枚举列表中的标识符必须与同一范围中具有相同可见性的其他标识符不同。这包括普通变量名和其他枚举列表中的标识符。
- 枚举标记遵循一般范围规则。它们必须不同于具有相同可见性的其他枚举、结构和联合标记。

## 示例

这些示例阐释枚举声明：

```
enum DAY            /* Defines an enumeration type    */
{
    saturday,       /* Names day and declares a       */
    sunday = 0,     /* variable named workday with    */
    monday,         /* that type                      */
    tuesday,
    wednesday,      /* wednesday is associated with 3 */
    thursday,
    friday
} workday;
```

默认情况下，值 0 与 `saturday` 关联。标识符 `sunday` 将显式设置为 0。默认情况下，将为剩余标识符提供从 1 到 5 的值。

在此示例中，将集 `DAY` 中的值赋给变量 `today`。

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

可使用枚举常量的名称进行赋值。由于之前声明了 `DAY` 枚举类型，因此仅枚举标记 `DAY` 是必需的。

若要显式将整数值赋给枚举数据类型的变量，请使用类型转换：

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

建议在 C 中进行此转换，但这不是必需的。

```
enum BOOLEAN  /* Declares an enumeration data type called BOOLEAN */
{
    false,     /* false = 0, true = 1 */
    true
};

enum BOOLEAN end_flag, match_flag; /* Two variables of type BOOLEAN */
```

还可将此声明指定为

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

或指定为

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

使用上述变量的示例可能类似于以下内容：

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

还可以声明未命名的枚举器数据类型。忽略数据类型的名称，但可以声明变量。变量 `response` 是已定义的类型的变量：

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

## 请参阅

枚举

# 结构声明

2021/8/13 •

"结构声明"用于为类型命名和指定一系列可具有不同类型的变量值(称为结构的"成员"或"字段")。可选标识符(称为"标记")为结构类型命名并可用于结构类型的后续引用。该结构类型的变量保留该类型定义的整个序列。C 中的结构类似于其他语言中称为"记录"的类型。

## 语法

*struct-or-union-specifier*:
  *struct-or-union identifier*$_{opt}$ { *struct-declaration-list* }
  *struct-or-union identifier*

*struct-or-union*:
  `struct`
  `union`

*struct-declaration-list*:
  *struct-declaration*
  *struct-declaration-list struct-declaration*

*struct-declaration*:
  *specifier-qualifier-list struct-declarator-list* ;

*specifier-qualifier-list*:
  *type-specifier specifier-qualifier-list*$_{opt}$
  *type-qualifier specifier-qualifier-list*$_{opt}$

*struct-declarator-list*:
  *struct-declarator struct-declarator-list* , *struct-declarator*

*struct-declarator*:
  *declarator*
  *type-specifier declarator*$_{opt}$ : *constant-expression*

结构类型的声明不为结构预留空间。它只是结构变量的最新声明的模板。

前面定义的 *identifier*(标记)可用于引用在其他位置定义的结构类型。在本例中,只要该定义可见,*struct-declaration-list* 就不能重复。在定义结构类型之前,指向结构类型的结构和 typedef 的指针的声明可使用结构标记。但是,在实际使用字段大小之前,一定会先遇到结构定义。这是类型和类型标记的不完整定义。为了完成此定义,类型定义之后必须出现在相同的范围中。

*struct-declaration-list* 指定结构成员的类型和名称。*struct-declaration-list* 参数包含一个或多个变量或位域声明。

每个在 *struct-declaration-list* 中声明的变量都被定义为结构类型的成员。*struct-declaration-list* 中的变量声明与本节中讨论的其他变量声明的形式相同,只不过声明不能包含存储类说明符或初始值设定项。结构成员可以具有任何变量类型(类型 `void` 除外)、不完整类型或函数类型。

不能将成员声明为具有其出现的结构的类型。但是,可将成员声明为指向结构类型(只要该结构类型具有标记,该成员就会出现在其中)的指针。这使您可以创建结构的链接列表。

结构的范围与其他标识符的一样。结构标识符必须不同于具有相同可见性的其他结构、联合和枚举标记。

*struct-declaration-list* 中的每个 *struct-declaration* 在列表中都必须是唯一的。但是，*struct-declaration-list* 中的标识符名称不必与普通变量名称或其他结构声明列表中的标识符名称不同。

嵌套结构也可以访问，就像它们是在文件范围级别声明的一样。例如，给定以下声明：

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

以下两个声明都是合法的：

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

## 示例

以下示例演示了结构声明：

```
struct employee   /* Defines a structure variable named temp */
{
    char name[20];
    int id;
    long class;
} temp;
```

`employee` 结构有三个成员：`name`、`id` 和 `class`。`name` 成员是一个包含 20 个元素的数组，`id` 和 `class` 是类型分别为 `int` 和 `long` 的简单成员。标识符 `employee` 是结构标识符。

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

本示例定义了三个结构变量：`student`、`faculty` 和 `staff`。每个结构都有相同的包含三个成员的列表。成员被声明为具有在前面的示例中定义的结构类型 `employee`。

```
struct           /* Defines an anonymous struct and a */
{                /* structure variable named complex  */
    float x, y;
} complex;
```

`complex` 结构有两个类型为 `float` 的成员，即 `x` 和 `y`。结构类型没有标记，因此是未命名或匿名的。

```
struct sample   /* Defines a structure named x */
{
    char c;
    float *pf;
    struct sample *next;
} x;
```

结构的前两个成员是 `char` 变量和指向 `float` 值的指针。第三个成员 (`next`) 被声明为指向正在定义的结构类型 (`sample`) 的指针。

当不需要命名的标记时，匿名结构很有用。当一个声明定义了所有结构实例时，就是这种情况。例如：

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

嵌入结构通常是匿名的。

```
struct somestruct
{
    struct    /* Anonymous structure */
    {
        int x, y;
    } point;
    int type;
} w;
```

**Microsoft 专用**

编译器允许未确定大小或零大小的数组作为结构的最后一个成员。如果常量数组在不同的情况下大小不同，这可能很有用。此类结构的声明类似于以下形式：

`struct` identifier { set-of-declarations type array-name[]; };

未确定大小的数组仅在作为结构的最后一个成员时才出现。只要所有封闭结构中都不再进一步声明成员，那么包含未确定大小的数组声明的结构就可以嵌入其他结构中。不允许有此类结构的数组。当将 `sizeof` 运算符应用于此类型的变量或应用于此类型本身时，假定数组的大小为 0。

如果结构声明是另一结构或联合的成员，则可以在没有声明符的情况下指定该结构声明。字段名将提升到该封闭结构中。例如，无名称结构类似于以下形式：

```
struct s
{
    float y;
    struct
    {
        int a, b, c;
    };
    char str[10];
} *p_s;
.
.
.
p_s->b = 100;  /* A reference to a field in the s structure */
```

有关结构引用的信息，请参阅结构和联合成员。

**结束 Microsoft 专用**

# 请参阅

声明符和变量声明

# C 位域

2021/8/16 •

除了结构或联合的成员的声明符外，结构声明符也可以是指定数目的位，称为"位域"。其长度从字段名称的声明符到冒号。位域被解释为整型类型。

## 语法

*struct-declarator*:
  *declarator*
  *type-specifier declarator*$_{opt}$ : *constant-expression*

constant-expression 指定域的宽度（以位为单位）。`declarator` 的 type-specifier 必须为 `unsigned int`、`signed int` 或 `int`，而且 constant-expression 必须为非负整数值。如果值为零，则声明没有任何 `declarator`。不允许位域的数组、指向位域的指针和返回位域的函数。可选的 `declarator` 命名位域。位域只能声明为结构的一部分。Address-of 运算符 (&) 不能应用于位域组件。

未命名位域不可引用，且其内容在运行时是不可预测的。它们可以出于对齐目的用作"虚拟"字段。宽度指定为 0 的未命名位域可以保证 struct-declaration-list 中它后面的成员的存储从 `int` 边界开始。

位域还必须足够长以包含该位模式。例如，以下两个语句均不合法：

```
short a:17;         /* Illegal! */
int long y:33;      /* Illegal! */
```

以下示例定义了一个名为 `screen` 的二维结构数组。

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

该数组包含 2,000 个元素。每个元素都是包含以下四个位域成员的单个结构：`icon`、`color`、`underline` 和 `blink`。每个结构的大小是两个字节。

位域与整数类型具有相同的语义。这意味着位域在表达式中的使用方式与同样基类型使用变量的方式完全相同，无论位域中有多少位。

### Microsoft 专用

定义为 `int` 的位域被视为 `signed`。ANSI C 标准的 Microsoft 扩展允许对位域使用 `char` 和 `long` 类型（`signed` 和 `unsigned`）。带基类型 `long`、`short` 或 `char`（`signed` 或 `unsigned`）的未命名位域强制与适合基类型的边界对齐。

在整数中按照从最高有效位到最低有效位的顺序来分配位域。在以下代码中

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

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

这些位将按如下所示排列：

```
00000001 11110010
ccccccccb bbbbaaaa
```

由于 8086 系列处理器将整数值的低字节存储在高字节之前，因此上面的整数 `0x01F2` 将按 `0xF2` 后跟 `0x01` 的形式存储在物理内存中。

结束 Microsoft 专用

## 请参阅

结构声明

# 结构的存储和对齐

2021/8/17 •

**Microsoft 专用**

结构成员按其声明顺序进行存储：第一个成员的内存地址最低，最后一个成员的内存地址最高。

每个数据对象均具有一个 alignment-requirement 。对于结构，需求是其成员中的最大者。为每个对象分配一个 offset ，以便

*offset* `%` *alignment-requirement* `==` 0

如果整型的大小相同，并且下一个位域适合当前分配单元而未跨位域的常见对齐需求所强加的边界，则将相邻位域打包到相同的 1 字节、2 字节或 4 字节分配单元中。

为了节省空间或遵循现有数据结构，您可能需要或多或少的简洁存储结构。/Zp[n] 编译器选项和 #pragma pack 控制将结构数据"打包"到内存中的方式 。使用 /Zp[n] 选项（其中，n 为 1、2、4、8 或 16）时，第一个结构成员后的每个结构成员将存储在字节边界上，这些字节边界是字段或包装大小 (n) 的对齐需求（以较小者为准）。表示为公式，字节边界为

```
min( n, sizeof( item ) )
```

其中，n 是使用 /Zp[n] 选项表示的包装大小，而 item 是结构成员 。默认包装大小为 /Zp8。

若要使用 `pack` 杂注为特定结构指定命令行上指定的包装以外的包装，请在结构的前面提供 `pack` 杂注，其中包装大小为 1、2、4、8 或 16。若要恢复命令行上提供的包装，请指定不带参数的 `pack` 杂注。

对于 Microsoft C 编译器，位域默认为大小 `long` 。基于类型大小或 /Zp[n] 大小（以较小者为准）对齐结构成员 。默认大小为 4。

**结束 Microsoft 专用**

## 请参阅

结构声明

# 联合声明

2021/8/17 •

"联合声明"指定一组变量值和(可选)一个命名联合的标记。变量值称为联合的"成员",并且可以具有不同的类型。联合类似于其他语言中的"变体记录"。

## 语法

*struct-or-union-specifier*:
  *struct-or-union identifier*$_{opt}$ { *struct-declaration-list* }
  *struct-or-union identifier*

*struct-or-union*:
  `struct`
  `union`

*struct-declaration-list*:
  *struct-declaration*
  *struct-declaration-list struct-declaration*

联合内容定义为

*struct-declaration*:
  *specifier-qualifier-list struct-declarator-list* ;

*specifier-qualifier-list*:
  *type-specifier specifier-qualifier-list*$_{opt}$
  *type-qualifier specifier-qualifier-list*$_{opt}$

*struct-declarator-list*:
  *struct-declarator*
  *struct-declarator-list* , *struct-declarator*

具有 `union` 类型的变量存储此类型所定义的值之一。相同的规则可控制结构和联合声明。联合还可以具有位域。

联合成员不能具有不完整类型、类型 `void` 或函数类型。因此,成员不能是联合的实例,但可以是指向将声明的联合类型的指针。

联合类型声明只是一个模板。不保留内存,直到声明变量。

> **NOTE**
> 如果声明两个类型的联合并存储一个值,但使用其它类型访问该联合,则结果是不可靠的。例如,声明了 `float` 和 `int` 的联合。存储 `float` 值,但程序稍后会将此值作为 `int` 进行访问。在这种情况下,此值取决于 `float` 值的内部存储。整数值是不可靠的。

## 示例

下面是联合的示例:

```
union sign   /* A definition and a declaration */
{
    int svar;
    unsigned uvar;
} number;
```

此示例定义了带 `sign` 类型的联合变量，并声明了一个名为 `number` 的变量，该变量包含两个成员：`svar`（一个带符号整数）和 `uvar`（一个无符号整数）。此声明允许将 `number` 的当前值存储为带符号的值或无符号的值。与此联合类型关联的标记为 `sign`。

```
union               /* Defines a two-dimensional */
{                   /*  array named screen */
    struct
    {
      unsigned int icon : 8;
      unsigned color : 4;
    } window1;
    int screenval;
} screen[25][80];
```

`screen` 数组包含 2,000 个元素。数组的每个元素均为一个包含以下两个成员的联合：`window1` 和 `screenval`。`window1` 成员是带两个位域成员（`icon` 和 `color`）的结构。`screenval` 成员是 `int`。在任意给定时间，每个联合元素都保留由 `screenval` 表示的 `int` 或由 `window1` 表示的结构。

### Microsoft 专用

如果嵌套的联合是另一个结构或联合的成员，则可以匿名声明嵌套的联合。这是一个无名称联合的示例：

```
struct str
{
    int a, b;
    union            / * Unnamed union */
    {
      char c[4];
      long l;
      float f;
   };
   char c_array[10];
} my_str;
.
.
.
my_str.l == 0L;  /* A reference to a field in the my_str union */
```

联合通常嵌套在一个结构中，该结构包含一个在任意特定时间给定联合中包含的数据类型的字段。这是此类联合的声明的示例：

```
struct x
{
    int type_tag;
    union
    {
      int x;
      float y;
    }
}
```

有关引用联合的信息，请参阅结构和联合成员。

# 请参阅

声明符和变量声明

# 联合的存储

2021/8/16 •

与联合变量关联的存储是联合的最大成员所需的存储。在存储较小的成员时，联合变量可以包含未使用的内存空间。所有成员都存储在同一内存空间中并以相同的地址开始。每次将值赋给不同的成员时，都会重写存储的值。例如：

```
union         /* Defines a union named x */
{
    char *a, b;
    float f[20];
} x;
```

按照声明的顺序，`x` 联合的成员是指向 `char` 值的指针、`char` 值和包含 `float` 值的数组。由于 `x` 是联合的最长成员，因此为 `f` 分配的存储是 20 个元素数组 `f` 所需的存储。由于没有与联合关联的标记，因此其类型是未命名的或“匿名的”。

## 请参阅

联合声明

# 数组声明

2021/8/14 •

"数组声明"将命名数组并指定其元素的类型。它还可定义数组中的元素数。带数组类型的变量被视为指向数组元素的类型的指针。

## 语法

*declaration*:
  *declaration-specifiers init-declarator-list*$_{opt}$ ;

*init-declarator-list*:
  *init-declarator*
  *init-declarator-list* , *init-declarator*

*init-declarator*:
  *declarator*
  *declarator* = *initializer*

*declarator*:
  *pointer*$_{opt}$ *direct-declarator*

direct-declarator : /* 函数声明符 */
  *direct-declarator* [ *constant-expression*$_{opt}$ ]

由于 constant-expression 是可选的，因此该语法有两种形式：

- 第一种形式定义一个数组变量。括号内的 constant-expression 参数指定数组中的元素数量。constant-expression（如果有）必须具有整型类型和大于零的值。每个元素都有 type-specifier 给出的类型，它可以是除 `void` 之外的任何类型。数组元素不能是函数类型。
- 第二种形式声明已在其他位置定义了变量。它省略括号中的 constant-expression 参数而不是括号。仅在之前已初始化数组、将其声明为参数或声明为对在程序中的其他位置显式定义的某个数组的引用的情况下才能使用此形式。

在两种形式中，direct-declarator 都会命名变量并且可以修改变量的类型。紧跟 direct-declarator 的方括号 ([ ]) 会将声明符修改为数组类型。

类型限定符可以出现在数组类型对象的声明中，但限定符应用于元素而不是数组本身。

您可以声明一系列数组（"多维"数组），方法是遵循以下形式的带括号的常量表达式的列表的数组声明符：

> *type-specifier declarator* [ *constant-expression* ] [ *constant-expression* ] ...

方括号中的每个 constant-expression 均定义给定维度中的元素数量：二维数组具有两个带括号的表达式，三维数组具有三个带括号的表达式，依此类推。如果您已初始化数组、将其声明为参数或声明为对在程序中的其他位置显式定义的某个数组的引用，则可以忽略第一个常量表达式。

可使用复杂的声明符定义指向各种类型的对象的指针的数组，如解释更复杂的声明符中所述。

按行存储数组。例如，下面的数组包含两个行，每个行具有三个列：

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

首先存储第一行的三个列，然后存储第二行的三个列。这意味着最后一个下标的变化速度最快。

若要引用数组的单个元素，请使用下标表达式，如后缀运算符中所述。

## 示例

这些示例阐释了数组声明：

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

名为 `matrix` 的二维数组有 150 个元素，每个元素的类型都是 `float` 。

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

这是结构数组的声明。此数组有 100 个元素；每个元素均为一个包含两个成员的结构。

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

此语句声明指向 `char` 的指针数组的类型和名称。 `name` 的实际定义会在其他位置出现。

**Microsoft 专用**

保存数组的最大大小所需的整数类型为 size_t 的大小。头文件 STDDEF.H 中定义的 size_t 是介于 0x00000000 和 0x7CFFFFFF 范围之间的 `unsigned int` 。

**结束 Microsoft 专用**

## 请参阅

声明符和变量声明

# 数组的存储

2021/8/16 •

与数组类型关联的存储是其所有元素所需的存储。数组的元素存储在连续且增加的内存位置(从第一个元素到最后一个元素)。

## 请参阅

数组声明

# 指针声明

2021/8/11 •

"指针声明"可命名指针变量并指定该变量所指向的对象的类型。声明为指针的变量保留了一个内存地址。

## 语法

*declarator*:
  *pointer*$_{opt}$ *direct-declarator*

*direct-declarator*:
  *identifier*
  **(** *declarator* **)**
  *direct-declarator* **[** *constant-expression*$_{opt}$ **]**
  *direct-declarator* **(** *parameter-type-list* **)**
  *direct-declarator* **(** *identifier-list*$_{opt}$ **)**

*pointer*:
  ***** *type-qualifier-list*$_{opt}$
  ***** *type-qualifier-list*$_{opt}$ *pointer*

*type-qualifier-list*:
  *type-qualifier*
  *type-qualifier-list type-qualifier*

*type-specifier* 用于指定对象的类型，可以是任何基本、结构或联合类型。指针变量也可以指向函数、数组和其他指针。（有关声明和解释更复杂的指针类型的信息，请参阅解释更复杂的声明符。）

通过将 type-specifier 设为 `void`，可以延迟指定指针所引用的类型。这样的项被称为"指向 `void` 的指针"，并被写成 `void *`。声明为指向 *void* 的指针的变量可用于指向任何类型的对象。但是，若要对指针或指针指向的对象执行大多数操作，则必须为每个操作显式指定指针指向的类型。（类型为 `char *` 和 `void *` 的变量是赋值兼容的，不需要强制转换类型。此类转换可使用类型强制转换完成（有关详细信息，请参阅类型强制转换）。

type-qualifier 可以是 `const` 和/或 `volatile`。它们分别指定了指针不能被程序本身修改（`const`），或指针可以被超出程序的控制范围的某进程以合法方式修改（`volatile`）。（若要详细了解 `const` 和 `volatile`，请参阅类型限定符。）

*declarator* 可为变量命名，并可包含类型修饰符。例如，如果 *declarator* 表示数组，则将指针的类型修改为指向数组的指针。

在定义结构、联合或枚举类型之前，您可以声明指向结构、联合或枚举类型的指针。您可使用结构或联合标记声明指针，如下面的示例所示。此类声明是允许的，因为编译器不需要知道要为指针变量分配空间的结构或联合的大小。

## 示例

以下示例演示了指针声明。

```
char *message; /* Declares a pointer variable named message */
```

message 指针指向 `char` 类型的变量。

```
int *pointers[10];  /* Declares an array of pointers */
```

pointer 数组有 10 个元素；每个元素都是指向 `int` 类型的变量的指针。

```
int (*pointer)[10]; /* Declares a pointer to an array of 10 elements */
```

指针 变量指向具有 10 个元素的数组。此数组中的每个元素都是 `int` 类型。

```
int const *x;      /* Declares a pointer variable, x,
                      to a constant value */
```

可以将指针 x 修改为指向不同的 `int` 值，但无法修改它所指向的值。

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

这些声明中的变量 y 被声明为指向 `int` 值的常数指针。可以修改该指针指向的值，但指针本身必须始终指向同一位置：*fixed_object* 的地址。同样，z 也是常数指针，而它也被声明为指向值不能被程序修改的 `int`。附加说明符 `volatile` 指明，尽管 z 所指向的 const int 的值不能被程序修改，但可以被当前与程序并发运行的进程以合法方式修改。*w* 的声明指定，程序无法更改指向的值，并且程序无法修改指针。

```
struct list *next, *previous; /* Uses the tag for list */
```

本示例声明了指向结构类型 *list* 的两个指针变量 *next* 和 *previous*。只要 *list* 类型定义与声明具有相同的可见性，此声明就可以出现在 *list* 结构类型的定义前面（请参阅下一个示例）。

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

变量 *line* 具有名为 *list* 的结构类型。list 结构类型有三个成员：第一个成员是指向 `char` 值的指针，第二个成员是 `int` 值，第三个成员是指向另一个 list 结构的指针。

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

变量 *record* 具有结构类型 *id*。请注意，*pname* 被声明为指向名为 *name* 的另一个结构类型的指针。此声明可在定义 *name* 类型之前出现。

## 请参阅

声明符和变量声明

# 地址存储

2021/8/14 •

地址所需的存储量和该地址的含义取决于编译器的实现。指向不同类型的指针不能保证具有相同的长度。因此，sizeof(char *) 不必与 sizeof(int *) 相等。

**Microsoft 专用**

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

**结束 Microsoft 专用**

## 请参阅

指针声明

# 基指针 (C)

2021/8/16 •

**Microsoft 专用**

__based(C++ 参考)

对于 Microsoft 32 位和 64 位 C 编译器，基指针是相对于 32 位或 64 位指针基的 32 位或 64 位偏移量。基寻址对于控制分配对象的部分很有用，这可减少可执行文件的大小并加快执行速度。通常，用于指定基指针的形式为

> *type* __based( *base* ) *declarator*

基寻址的"based on pointer"变体支持作为基的指针的规范。该基指针是内存部分的偏移量，它从所基于的指针开始。基于指针地址的指针是 `__based` 关键字的唯一有效的 32 位和 64 位编译 形式。在这些编译中，它们是来自 32 位或 64 位基的 32 位或 64 位置换。

基于指针的指针的用途之一是用于包含指针的永久标识符。可将包含基于指针的指针的链接列表保存到磁盘，然后重新加载到内存中的另一个位置，并且指针保持有效。

以下示例演示基于指针的指针。

```
void *vpBuffer;

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

将指针 `vpBuffer` 分配给程序中后面某个时间点分配的内存地址。相对于 `vpBuffer` 的值重新定位链接的列表。

**结束 Microsoft 专用**

## 请参阅

声明符和变量声明

# C 抽象声明符

2021/8/14 •

抽象声明符是没有标识符的声明符，由一个或多个指针、数组或函数修饰符组成。指针修饰符 (*) 始终在声明符中的标识符前面；数组 ([ ]) 和函数 ( ( ) ) 修饰符紧跟在标识符后面。了解这种情况后，您可以确定标识符将在抽象声明符中显示的位置并相应地解释声明符。有关复杂声明符的其他信息和示例，请参阅解释更复杂的声明符。 `typedef` 通常可用于简化声明符。请参阅 Typedef 声明。

抽象声明符可能很复杂。复杂的抽象声明符中的括号指定一个特定的解释，正如它们为声明中的复杂声明符所做的一样。

以下示例阐释了抽象声明符：

```
int *         // The type name for a pointer to type int:

int *[3]      // An array of three pointers to int

int (*) [5]   // A pointer to an array of five int

int *()       // A function with no parameter specification
              // returning a pointer to int

// A pointer to a function taking no arguments and
// returning an int

int (*) ( void )

// An array of an unspecified number of constant pointers to
// functions each with one parameter that has type unsigned int
// and an unspecified number of other parameters returning an int

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

> **NOTE**
> 不允许使用由一组空括号 ( ) 组成的抽象声明符，因为它的意义不明确。无法确定隐含标识符是位于括号内(此时它是未修改类型)还是括号前(此时它是函数类型)。

## 请参阅

声明符和变量声明

# 解释复杂声明符

2021/8/12 •

您可以将任何声明符括在圆括号中以指定“复杂声明符”的特殊解释。复杂声明符是由多个数组、指针或函数修饰符限定的标识符。您可以将数组、指针和函数修饰符的各种组合应用于单个标识符。`typedef` 通常可用于简化声明。请参阅 Typedef 声明。

在解释复杂声明符时，方括号和圆括号（即，标识符右侧的修饰符）优先于星号（即，标识符左侧的修饰符）。方括号和圆括号具有相同的优先级并且都是从左到右关联。在完全解释声明符之后，将应用类型说明符以作为最后一步。通过使用圆括号，您可以重写默认关联顺序和强制实施特定解释。但是，绝不要单独在标识符名称两边使用圆括号。这可能会被错误解释为参数列表。

解释复杂声明符的一个简单方法是通过下列 4 个步骤“从里到外”地读取它们：

1. 从标识符开始并直接查找方括号或圆括号（如果有）的右侧。
2. 解释这些方括号或圆括号，然后查找星号的左侧。
3. 如果在任何阶段遇到一个右圆括号，请返回并将规则 1 和 2 应用于圆括号内的所有内容。
4. 应用类型说明符。

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

在此示例中，步骤是按顺序编号的，并且可以按如下方式解释：

1. 标识符 `var` 声明为
2. 指向以下内容的指针
3. 返回以下内容的函数
4. 指向以下内容的指针
5. 包含 10 个元素的数组，这些元素分别为
6. 指向以下内容的指针
7. `char` 值。

## 示例

以下示例阐释了其他复杂声明并演示了圆括号如何影响声明的含义。

```
int *var[5]; /* Array of pointers to int values */
```

数组修饰符的优先级高于指针修饰符，因此，`var` 将声明为数组。指针修饰符应用于数组元素的类型；因此，数组元素是指向 `int` 值的指针。

```
int (*var)[5]; /* Pointer to array of int values */
```

在对 `var` 的此声明中，括号赋予指针修饰符比数组修饰符更高的优先级，并且 `var` 被声明为指向包含 5 个 `int` 值的数组的指针。

```
long *var( long, long ); /* Function returning pointer to long */
```

函数修饰符的优先级也高于指针修饰符，因此对 `var` 的此声明将 `var` 声明为返回指向 `long` 值的指针的函数。此函数被声明为接受两个 `long` 值作为参数。

```
long (*var)( long, long ); /* Pointer to function returning long */
```

此示例与前一个示例类似。括号赋予指针修饰符比函数修饰符更高的优先级，并且 `var` 被声明为指向返回 `long` 值的函数的指针。同样，此函数接受两个 `long` 参数。

```
struct both       /* Array of pointers to functions */
{                 /*   returning structures         */
    int a;
    char b;
} ( *var[5] )( struct both, struct both );
```

数组的元素不能是函数，但此声明演示了如何将指针数组声明为函数。在此示例中，`var` 被声明为包含 5 个指向函数(返回具有两个成员的结构)的指针的数组。函数的参数被声明为具有同一结构类型 `both` 的两个结构。请注意，`*var[5]` 两边需要圆括号。如果没有它们，声明将是对声明函数数组的非法尝试，如下所示：

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

以下语句声明指针的数组。

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

`name` 数组具有组织在一个多维数组中的 50 个元素。这些元素是指向常量指针的指针。此常量指针指向没有参数并返回指向无符号类型的指针的函数。

下一个示例是函数，此函数返回指向包含三个 `double` 值的数组的指针。

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

在此声明中，函数将返回指向数组的指针，因为返回数组的函数是非法的。在此处，`var` 被声明为函数，此函数返回指向包含三个 `double` 值的数组的指针。函数 `var` 将采用一个参数。与返回值一样，此参数是指向包含三个 `double` 值的数组的指针。参数类型由一个复杂 abstract-declarator 给定。参数类型中星号两边的括号是必需的；如果没有括号，参数类型就是包含三个指向 `double` 值的指针的数组。有关抽象声明符的讨论和示例，请参阅抽象声明符。

```
union sign         /* Array of arrays of pointers */
{                  /* to pointers to unions       */
     int x;
     unsigned y;
} **var[5][5];
```

如上面的示例所示，指针可以指向另一个指针，数组可包含数组作为元素。这里的 `var` 是一个包含五个元素的数组。每个元素都是一个五元素指针数组，这些指针指向指向具有两个成员的联合的指针。

```
union sign *(*var[5])[5]; /* Array of pointers to arrays
                             of pointers to unions        */
```

此示例演示圆括号的放置如何更改声明的含义。在此示例中，`var` 是一个五元素指针数组，这些指针指向联合的五元素指针数组。有关如何使用 `typedef` 避免复杂声明的示例，请参阅 Typedef 声明。

## 请参阅

声明和类型

# 初始化

2021/8/12 •

“初始值设定项”是要赋给正在声明的变量的一个值或一系列值。通过在变量声明中将初始值设定项应用于声明符，可以将变量设置为初始值。初始值设定项的值将赋给变量。

以下各节介绍如何初始化标量、聚合和字符串类型的变量。“标量类型”包括所有算术类型，还包括指针。“聚合类型”包括数组、结构和联合。

## 请参阅

声明和类型

# 初始化标量类型

2021/8/13 •

初始化标量类型时，`assignment-expression` 的值被赋给变量。赋值的转换规则适用。（有关转换规则的信息，请参阅类型转换。）

## 语法

*declaration*：
  declaration-specifiers init-declarator-list opt ;

*declaration-specifiers*：
  storage-class-specifier declaration-specifiers opt
  type-specifier declaration-specifiers opt
  type-qualifier declaration-specifiers opt

*init-declarator-list*：
  *init-declarator*
  *init-declarator-list* , *init-declarator*

*init-declarator*：
  *declarator*
  declarator = initializer /* 用于标量初始化 */

*initializer*：
  *assignment-expression*

您可初始化任何类型的变量，前提是遵循下列规则：

- 在文件范围级别声明的变量可初始化。如果未显式初始化外部级别的变量，则默认情况下它将初始化为 0。
- 常数表达式可用于初始化使用 `static` `storage-class-specifier` 声明的任何全局变量。声明为 `static` 的变量在程序开始执行时初始化。如果没有显式初始化全局 `static` 变量，则默认情况下它初始化为 0，并且为每个具有指针类型的成员分配 null 指针。
- 使用 `auto` 或 `register` 存储类说明符声明的变量在每次执行控制权传递给声明它们的块时初始化。如果在 `auto` 或 `register` 变量的声明中省略初始值设定项，则变量的初始值是未定义的。对于自动值和寄存器值，初始值设定项未限制为常量；它可以是包含之前定义的值的任何表达式，甚至是函数调用。
- 外部变量声明和所有 `static` 变量（无论是外部还是内部变量）的初始值必须是常数表达式。（有关详细信息，请参阅常量表达式。）由于任何外部声明的变量或静态变量的地址为常数，因此可以使用它初始化内部声明的 `static` 指针变量。不过，`auto` 变量的地址不能用作静态初始值设定项，因为它可能在每次执行块时都不同。可使用常数或变量值来初始化 `auto` 和 `register` 变量。
- 如果标识符的声明具有块范围，并且标识符具有外部链接，则该声明不能具有初始化。

## 示例

下列示例阐释了初始化：

```
int x = 10;
```

整数变量 `x` 将初始化为常量表达式 `10`。

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

指针 `px` 将初始化为 0，从而生成“null”指针。

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

此示例使用常数表达式 `(3 * 1024)` 将 `c` 初始化为由于 `const` 关键字而无法修改的常数值。

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

此语句使用另一变量 `b` 的地址初始化指针 `x`。

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

指针 `a` 是使用名为 `z` 的变量的地址初始化的。不过，由于它被指定为 `const`，因此变量 `a` 只能被初始化，决不能被修改。该指针始终指向同一位置。

```
int GLOBAL ;

int function( void )
{
    int LOCAL ;
    static int *lp = &LOCAL;   /* Illegal initialization */
    static int *gp = &GLOBAL;  /* Legal initialization   */
    register int *rp = &LOCAL; /* Legal initialization   */
}
```

全局变量 `GLOBAL` 是在外部级别声明的，因此它具有全局生存期。局部变量 `LOCAL` 有 `auto` 存储类，并且在执行声明它的函数期间只有一个地址。因此，不允许尝试使用 `LOCAL` 的地址来初始化 `static` 指针变量 `lp`。`static` 指针变量 `gp` 可初始化为 `GLOBAL` 的地址，因为此地址始终都是相同的。同样，`*rp` 可以被初始化，因为 `rp` 是局部变量，并且可以有非常数初始值设定项。每次输入块时，`LOCAL` 都具有新地址，该地址之后将赋给 `rp`。

## 请参阅

初始化

# 初始化聚合类型

2021/8/16 •

聚合 类型是结构、联合或数组类型。如果聚合类型包含聚合类型的成员，则初始化规则将以递归方式应用。

## 语法

*initializer*:
  { *initializer-list* } /* For aggregate initialization */
  { *initializer-list* , }

*initializer-list*:
  *initializer*
  *initializer-list* , *initializer*

initializer-list 是用逗号分隔的初始值设定项的列表。列表中的每个初始值设定项是常量表达式或初始值设定项列表。因此，可以嵌入初始值设定项列表。此形式对于初始化聚合类型的聚合成员很有用，如本节中的示例所示。但是，如果自动标识符的初始值设定项是一个表达式，则它无需是常量表达式；它只需要针对标识符的适当类型的赋值。

对于每个初始值设定项列表，常量表达式的值将按顺序赋给聚合变量的相应成员。

如果 initializer-list 具有的值少于聚合类型，则聚合类型的其余成员或元素将初始化为 0。未显式初始化的自动标识符的初始值是不确定的。如果 initializer-list 具有的值多于聚合类型，则会导致错误。这些规则适用于每个嵌入的初始值设定项列表以及整个聚合。

结构的初始值设定项要么是同一类型的表达式，要么是其成员包含在大括号 ({ } ) 中的初始值设定项的列表。未命名的位域成员是未初始化的。

在初始化联合时，initializer-list 必须是单个常量表达式。常量表达式的值将赋给联合的第一个成员。

如果数组的大小未知，则初始值设定项的数目将确定数组的大小，并且其类型将变为已完成。无法在 C 中指定初始值设定项的重复，也无法在数组中间初始化元素而不提供前面的所有值。如果您的程序中需要此操作，请使用汇编语言编写该例程。

请注意，初始值设定项的数目可以设置数组的大小：

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

但是，如果您指定大小并提供错误的初始值设定项数目，则编译器将生成错误。

**Microsoft 专用**

数组的最大大小由 size_t 定义。头文件 STDDEF.H 中定义的 size_t 是介于 0x00000000 和 0x7CFFFFFF 范围之间的 `unsigned int` 。

**结束 Microsoft 专用**

## 示例

此示例演示数组的初始值设定项。

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

此语句将 `P` 声明为一个四列三行的数组，并将其第一行的元素初始化为 1，将其第二行的元素初始化为 2，将其第三行的元素初始化为 3，将其第四行的元素初始化为 4。请注意，第三行和第四行的初始值设定项列表在最后一个常量表达式后包含逗号。最后一个初始值设定项列表 ( `{4, 4, 4,}` ) 的后面也跟有一个逗号。这些额外的逗号是允许的，但不是必需的；只需要将常量表达式彼此分隔开的逗号以及将初始值设定项列表彼此分隔开的逗号。

如果聚合成员没有嵌入的初始值设定项列表，则只会按顺序将值赋给子聚合的每个成员。因此，前面的示例中的初始化与以下项是等效的：

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

大括号还可以显示在列表中的各个初始值设定项的两边，并有助于阐明上面的示例。

在初始化聚合变量时，您必须谨慎正确地使用大括号和初始值设定项列表。下面的示例更详细地阐释了编译器的大括号的说明：

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

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

在此示例中，`nlist` 声明为结构的 2x3 数组，每个结构具有三个成员。初始化的第 1 行将值赋给 `nlist` 的第一行，如下所示：

1. 第 1 行上的第一个左大括号告知编译器 `nlist` 的第一个聚合成员（即 `nlist[0]` ）的初始化已开始。

2. 第二个左大括号指示 `nlist[0]` 的第一个聚合成员（即 `nlist[0][0]` 处的结构）的初始化已开始。

3. 第一个右大括号结束 `nlist[0][0]` 结构的初始化；下一个左大括号开始 `nlist[0][1]` 的初始化。

4. 此过程将继续，直到右大括号结束 `nlist[0]` 的初始化的行的末尾。

第 2 行采用类似的方法将值赋给 `nlist` 的第二行。请注意，包含第 1 行和第 2 行上的初始值设定项的大括号的外部集是必需的。忽略外部大括号的下列构造将会导致错误：

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

在此构造中，第 1 行上的第一个左大括号开始 `nlist[0]`（它是三个结构的数组）的初始化。值 1、2 和 3 将赋给第一个结构的三个成员。当遇到下一个右大括号（值 3 的后面）时，`nlist[0]` 的初始化已完成，并且三个结构数组中的其余两个结构将自动初始化为 0。同样，`{ 4,5,6 }` 会初始化 `nlist` 的第二行中的第一个结构。`nlist[1]` 的其余两个结构将设置为 0。当编译器遇到下一个初始值设定项列表（`{ 7,8,9 }`）时，它会尝试初始化 `nlist[2]`。由于 `nlist` 只具有两个行，因此该尝试会导致错误。

在下一个示例中，`x` 的三个 `int` 成员分别初始化为 1、2 和 3。

```
struct list
{
    int i, j, k;
    float m[2][3];
} x = {
        1,
        2,
        3,
       {4.0, 4.0, 4.0}
      };
```

在上面的 `list` 结构中，`m` 的第一行中的三个元素将初始化为 4.0；`m` 的剩余行的元素将初始化为 0.0（默认值）。

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

在此示例中，将初始化联合变量 `y`。该联合的第一个元素是数组，因此该初始值设定项是聚合初始值设定项。初始值设定项列表 `{'1'}` 将值赋给数组的第一行。由于列表中仅显示一个值，因此，第一列中的元素将初始化为字符 `1`，而该行中的其余两个元素将初始化为值 0（默认值）。同样，`x` 的第二行的第一个元素将初始化为字符 `4`，而该行中的其余两个元素将初始化为值 0。

## 请参阅

初始化

# 初始化字符串

2021/8/16 •

您可以使用字符串文本(或宽字符串文本)初始化字符(或宽字符)的数组。例如:

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

将 `code` 初始化为一个四元素字符数组。第四个元素为 null 字符,用于终止所有字符串文本。

标识符列表的长度值只能与要初始化的标识符的数量相同。如果指定短于字符串的数组大小,则会忽略多余字符。例如,以下声明将 `code` 初始化为一个三元素字符数组:

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

只有初始值设定项的前三个字符将分配给 `code`。字符 `d` 和字符串终止 null 字符将被丢弃。请注意,这将创建一个非终止字符串(即,没有 0 值标记其结束的字符串)并生成指示这种情况的诊断消息。

声明

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

等同于

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

如果字符串短于指定的数组大小,数组中的剩余元素将初始化为 0。

**Microsoft 专用**

在 Microsoft C 中,字符串文本的长度最大为 2048 个字节。

**结束 Microsoft 专用**

## 请参阅

初始化

# 基本类型的存储

2021/8/14 •

下表汇总了与每个基本类型关联的存储。

## 基础类型的大小

| " | " |
|---|---|
| `char` , `unsigned char` , `signed char` | 1 个字节 |
| `short` , `unsigned short` | 2 个字节 |
| `int` , `unsigned int` | 4 个字节 |
| `long` , `unsigned long` | 4 个字节 |
| `long long` , `unsigned long long` | 8 个字节 |
| `float` | 4 个字节 |
| `double` | 8 个字节 |
| `long double` | 8 个字节 |

C 数据类型属于常规类别。整型类型包括 `int` 、 `char` 、 `short` 、 `long` 和 `long long` 。这些类型可使用 `signed` 或 `unsigned` 进行限定， `unsigned` 本身可以用作 `unsigned int` 的简写。枚举类型 ( `enum` ) 在大多数情况下也被视为整型类型。浮点类型包括 `float` 、 `double` 和 `long double` 。"算术类型"包括所有浮点型和整型类型 。

## 请参阅

声明和类型

# char 类型

2021/8/12 •

`char` 类型用于存储可表示的字符集的成员的整数值。该整数值是与指定字符对应的 ASCII 代码。

**Microsoft 专用**

类型 `unsigned char` 的字符值介于 0 和 0xFF(十六进制)范围之间。`signed char` 介于 0x80 和 0x7F 范围之间。这些范围分别转换为 0 到 255(十进制)以及 -128 到 +127(十进制)。/J 编译器选项将默认值从 `signed` 更改为 `unsigned`。

**结束 Microsoft 专用**

## 请参阅

基本类型的存储

# int 类型

2021/8/16 •

`signed int` 或 `unsigned int` 项的大小是特定计算机上的标准整数大小。例如，在 16 位操作系统中，`int` 类型通常是 16 位（或 2 字节）。在 32 位操作系统中，`int` 类型通常是 32 位（或 4 字节）。因此，`int` 类型与 `short int` 或 `long int` 类型等效，`unsigned int` 类型与 `unsigned short` 或 `unsigned long` 类型等效，具体视目标环境而定。除非另有规定，否则 `int` 类型都表示带符号值。

类型说明符 `int` 和 `unsigned int`（或简写为 `unsigned`）定义了 C 语言的某些功能（例如，`enum` 类型）。在这种情况下，特定实现的 `int` 和 `unsigned int` 的定义决定了实际存储。

**Microsoft 专用**

带符号整数以 2 的补数的形式表示。最高有效位保留符号：1 表示负数，0 表示正数和零。值的范围在 C 和 C++ 整数限制中给定（摘自 LIMITS.H 头文件）。

**结束 Microsoft 专用**

> **NOTE**
>
> `int` 和 `unsigned int` 类型说明符在 C 程序中广泛使用，因为它们可便于特定计算机以对自己最高效的方式处理整数值。不过，由于 `int` 和 `unsigned int` 类型的大小不同，因此依赖特定 `int` 大小的程序可能无法移植到其他计算机中。为了提高程序的可移植性，可以使用带 `sizeof` 运算符（如 `sizeof` 运算符中所述）的表达式，而不是硬编码的数据大小。

## 请参阅

基本类型的存储

# C 调整了大小的整型

2021/8/16 •

**Microsoft 专用**

Microsoft C 支持固定大小整数类型。可以使用 `__intN` 类型说明符声明 8 位、16 位、32 位或 64 位整型变量，其中 `N` 是整型变量的大小（以位为单位）。n 的值可以是 8、16、32 或 64 。以下示例为四种类型的固定大小整数各声明了一个变量：

```
__int8  nSmall;     // Declares 8-bit integer
__int16 nMedium;    // Declares 16-bit integer
__int32 nLarge;     // Declares 32-bit integer
__int64 nHuge;      // Declares 64-bit integer
```

前三种类型的大小整数与有相同大小的 ANSI 类型同义。它们对于编写跨多个平台具有完全相同行为的可移植代码非常有用。`__int8` 数据类型与 `char` 类型同义，`__int16` 与 `short` 类型同义，`__int32` 与 `int` 类型同义，`__int64` 与 `long long` 类型同义。

**结束 Microsoft 专用**

## 请参阅

基本类型的存储

# float 类型

2021/8/16 •

浮点数使用 IEEE(电气和电子工程师协会)格式。浮点类型的单精度值具有 4 个字节，包括一个符号位、一个 8 位 excess-127 二进制指数和一个 23 位尾数。尾数表示一个介于 1.0 和 2.0 之间的数。由于尾数的高顺序位始终为 1，因此它不是以数字形式存储的。此表示形式为 float 类型提供了一个大约在 3.4E-38 和 3.4E+38 之间的范围。

您可根据应用程序的需求将变量声明为 float 或 double。这两种类型之间的主要差异在于它们可表示的基数、它们需要的存储以及它们的范围。下表显示了基数与存储需求之间的关系。

**浮点类型**

| | | |
|---|---|---|
| float | 6 - 7 | 4 |
| double | 15 - 16 | 8 |

浮点变量由尾数(包含数字的值)和指数(包含数字的数量级)表示。

下表显示了分配给每个浮点类型的尾数和指数的位数。任何 float 或 double 的最高有效位始终是符号位。如果符号位为 1，则将数字视为负数；否则，将数字视为正数。

**指数和尾数的长度**

| | | |
|---|---|---|
| float | 8 位 | 23 位 |
| double | 11 位 | 52 位 |

由于指数是以无符号形式存储的，因此指数的偏差为其可能值的一半。对于 float 类型，偏差为 127；对于 double 类型，偏差为 1023。您可以通过将指数值减去偏差值来计算实际指数值。

存储为二进制分数的尾数大于或等于 1 且小于 2。对于 float 和 double 类型，最高有效位位置的尾数中有一个隐含的前导 1，这样，尾数实际上分别为 24 和 53 位长，即使最高有效位从未存储在内存中也是如此。

浮点包可以将二进制浮点数存储为非标准化数，而不使用刚刚介绍的存储方法。“非标准化数”是带有保留指数值的非零浮点数，其中尾数的最高有效位为 0。通过使用非标准化格式，浮点数的范围可以扩展，但会失去精度。您无法控制浮点数以标准化形式还是非标准化形式表示；浮点包决定了表示形式。浮点包从不使用非标准化形式，除非指数变为小于可以标准化形式表示的最小值。

下表显示了可在每种浮点类型的变量中存储的最小值和最大值。此表中所列的值仅适用于标准化浮点数；非标准化浮点数的最小值更小。请注意，在 80 *x*87 寄存器中保留的数字始终以 80 位标准化形式表示；数字存储在 32 位或 64 位浮点变量(float 类型和 long 类型的变量)中时只能以非标准化形式表示。

**浮点类型的范围**

| | | |
|---|---|---|
| 浮动 | 1.175494351 E - 38 | 3.402823466 E + 38 |
| double | 2.2250738585072014 E - 308 | 1.7976931348623158 E + 308 |

如果存储比精度更重要，请考虑对浮点变量使用 float 类型。相反，如果精度是最重要的条件，则使用 double 类型。

浮点变量可以提升为更大基数的类型（从 float 类型到 double 类型）。当您对浮点变量执行算术时，通常会出现提升。此算术始终以与具有最高精度的变量一样高的精度执行。例如，请考虑下列类型声明：

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

f_short = f_short * f_long;
```

在前面的示例中，变量 `f_short` 提升到类型 double 并且与 `f_long` 相乘；然后，结果舍入到类型 float，然后赋给 `f_short`。

在以下示例中（使用前面示例中的声明），将以浮点（32 位）精度对变量执行算术；结果随后将提升到 double 类型：

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

## 请参阅

基本类型的存储

# 类型 double

2021/8/17 •

双精度类型的双精度值具有 8 个字节。此格式类似于浮点格式，只不过该格式具有一个 11 位 excess-1023 指数和一个 52 位尾数以及隐含的高顺序 1 位。此格式为双精度类型提供的范围大约介于 1.7E-308 和 1.7E+308 之间。

**Microsoft 专用**

double 类型包含 64 位：1 位用于符号、11 位用于指数、52 表位用于尾数。其范围为精度至少为 15 个数字的 +/-1.7E308。

**结束 Microsoft 专用**

## 请参阅

基本类型的存储

# 长双精度类型

2021/8/16 •

`long double` 类型与 double 类型完全相同。

## 请参阅

基本类型的存储

# 不完整类型

2021/8/16 •

不完整类型 是一种用于描述标识符但缺少确定该标识符的大小所需的信息的类型。"不完整类型"可以是：

- 您尚未指定其成员的结构类型。
- 您尚未指定其成员的联合类型。
- 您尚未指定其维度的数组类型。

`void` 类型是无法完成的不完整类型。若要完成不完整类型，请指定缺少的信息。以下示例演示如何创建和完成不完整类型。

- 若要创建不完整的结构类型，请声明结构类型而不指定其成员。在本例中，`ps` 指针指向称为 `student` 的不完整的结构类型。

```
struct student *ps;
```

- 若要完成不完整的结构类型，请在稍后在指定其成员的同一范围中声明相同的结构类型，如下所示：

```
struct student
{
    int num;
}                   /* student structure now completed */
```

- 若要创建不完整的数组类型，请声明数组类型而不指定其重复计数。例如：

```
char a[];  /* a has incomplete type */
```

- 若要完成不完整的数组类型，请在稍后在指定其重复计数的同一范围中声明相同的名称，如下所示：

```
char a[25]; /* a now has complete type */
```

## 请参阅

声明和类型

# Typedef 声明

2021/8/16 •

typedef 声明是具有作为存储类的 typedef 的声明。声明符将成为新类型。您可以使用 typedef 声明为已由 C 定义的类型或您已声明的类型构造更短和更有意义的名称。利用 Typedef 名称，您可以封装可能会发生更改的实现详细信息。

typedef 声明的解释方式与变量或函数声明的解释方式相同，只不过标识符没有假定由声明指定的类型，而是成为了该类型的同义词。

## 语法

*declaration*:
  *declaration-specifiers init-declarator-list*$_{opt}$ ;

*declaration-specifiers*:
  *storage-class-specifier declaration-specifiers*$_{opt}$
  *type-specifier declaration-specifiers*$_{opt}$
  *type-qualifier declaration-specifiers*$_{opt}$

*storage-class-specifier*:
  `typedef`

*type-specifier*:
  `void`
  `char`
  `short`
  `int`
  `long`
  `float`
  `double`
  `signed`
  `unsigned`
  *struct-or-union-specifier*
  *enum-specifier*
  *typedef-name*

*typedef-name*:
  *identifier*

请注意，typedef 声明不会创建类型，而是创建现有类型的同义词或可通过其他方式指定的类型的名称。当使用 typedef 名称作为类型说明符时，可以将其与特定的类型说明符组合，但不可以将其与其他类型说明符组合。可接受的修饰符包括 `const` 和 `volatile`。

Typedef 名称与普通标识符共享命名空间（有关详细信息，请参阅命名空间）。因此，程序可以有一个 typedef 名称和一个具有相同名称的本地范围标识符。例如：

```
typedef char FlagType;

int main()
{
}

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

当通过与 typedef 相同的名称声明本地范围标识符时，或者在同一范围内或内部范围内声明结构或联合的成员时，必须指定类型说明符。以下示例演示了此约束：

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

若要对标识符、结构成员或联合成员重用 `FlagType` 名称，则必须提供类型：

```
const int FlagType;  /* Type specifier required */
```

仅仅编写以下语句是不够的

```
const FlagType;      /* Incomplete specification */
```

由于 `FlagType` 被当做该类型的一部分，因此没有要重新声明的标识符。此声明被视为非法声明，例如

```
int;  /* Illegal declaration */
```

可以使用 typedef 声明任何类型，包括指针、函数和数组类型。只要定义具有与声明相同的可见性，那么在定义结构或联合类型之前，您就可以为指向结构或联合类型的指针声明 typedef 名称。

Typedef 名称可用于提高代码可读性。`signal` 的所有以下三个声明指定了完全相同的类型，第一个声明没有使用任何 typedef 名称。

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

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

## 示例

以下示例演示了 typedef 声明：

```
typedef int WHOLE; /* Declares WHOLE to be a synonym for int */
```

请注意，`WHOLE` 现在可用于变量声明，如 `WHOLE i;` 或 `const WHOLE i;`。但是，声明 `long WHOLE i;` 是非法的。

```
typedef struct club
{
    char name[30];
    int size, year;
} GROUP;
```

此语句将 `GROUP` 声明为具有三个成员的结构类型。由于也指定了结构标记 `club`，因此 typedef 名称 (`GROUP`) 或结构标记可用于声明。您必须使用带标记的 struct 关键字，并且不能使用带 typedef 名称的 struct 关键字。

```
typedef GROUP *PG; /* Uses the previous typedef name
                      to declare a pointer           */
```

类型 `PG` 被声明为指向 `GROUP` 类型的指针，而 类型又被定义为结构类型。

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

此示例为不返回值并采用两个 int 参数的函数提供了类型 `DRAWF`。例如，这意味着声明

```
DRAWF box;
```

等效于声明

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

## 请参阅

声明和类型

# C 扩展的存储类特性

2021/8/16 •

**Microsoft 专用**

有关存储类特性的更多最新信息，可查看 `__declspec`（C++ 参考）。

扩展的特性语法简化并标准化了特定于 Microsoft 的 C 语言扩展。使用扩展的特性语法的存储类特性包括 `thread`、`naked`、`dllimport` 和 `dllexport`。

用于指定存储类信息的扩展特性语法使用 `__declspec` 关键字，该关键字指定给定类型的实例将与 Microsoft 专用存储类特性（`thread`、`naked`、`dllimport` 或 `dllexport`）一起存储。其他存储类修饰符的示例包括 `static` 和 `extern` 关键字。但是，这些关键字是 ISO C 标准的一部分，未涵盖在扩展的特性语法中。

## 语法

*storage-class-specifier*:
    `__declspec (` *extended-decl-modifier-seq* `)` /* Microsoft 专用*/

extended-decl-modifier-seq: /* Microsoft 专用*/
    *extended-decl-modifier* opt
    *extended-decl-modifier-seq* *extended-decl-modifier*

extended-decl-modifier: /* Microsoft 专用*/
    `thread`
    `naked`
    `dllimport`
    `dllexport`

空格可分隔声明修饰符。`extended-decl-modifier-seq` 可为空；在此情况下，`__declspec` 无效。

`thread`、`naked`、`dllimport` 和 `dllexport` 存储类特性只是要将其应用到的数据或函数的声明的属性。它们不重新定义函数自身的类型特性。`thread` 特性只影响数据。`naked` 特性仅影响函数。`dllimport` 和 `dllexport` 特性仅影响函数。

**结束 Microsoft 专用**

## 请参阅

声明和类型

# DLL 导入和导出

2021/8/15 •

**Microsoft 专用**

dllimport 和 dllexport 存储类修饰符是 C 语言的 Microsoft 专用扩展。这些修饰符定义其客户端的 DLL 的接口（可执行文件或另一个 DLL）。有关使用这些修饰符的具体信息，请参阅 dllexport、dllimport。

**结束 Microsoft 专用**

## 请参阅

C 扩展的存储类特性

# Naked (C)

2021/8/15 •

**Microsoft 专用**

裸存储类特性是特定于 Microsoft 的 C 语言扩展。编译器生成代码，而没有使用裸存储类特性声明的函数的 prolog 和 epilog 代码。当您需要使用内联汇编代码编写自己的 prolog/epilog 代码序列时，裸函数很有用。裸函数对于编写虚拟设备驱动程序很有用。

有关使用 naked 特性的特定信息，请参阅 Naked 函数。

**结束 Microsoft 专用**

## 请参阅

C 扩展的存储类特性

# 线程本地存储

2021/8/11 •

**Microsoft 专用**

线程本地存储 (TLS) 是给定的多线程进程中的每个线程为线程特定的数据分配存储时所采用的机制。在标准多线程程序中，数据在给定进程的所有线程间共享，而线程本地存储是用于分配每个线程数据的机制。有关线程的完整讨论，请参阅 Windows SDK 中的进程和线程。

Microsoft C 语言包括扩展的存储类特性 thread，可将它与 __declspec 关键字一起使用来声明线程本地变量。例如，以下代码声明了一个整数线程局部变量，并用一个值对其进行初始化：

```
__declspec( thread ) int tls_i = 1;
```

在声明静态绑定线程本地变量时，必须遵守这些准则：

- 仅在引发 DLL 加载的线程上和已在进程中运行的线程上初始化具有动态初始化的线程局部变量。有关详细信息，请参阅线程。
- 您只能将 thread 特性应用于数据声明和定义。它不能用于函数声明或定义。例如，下面的代码生成一个编译器错误：

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

- 只能在具有静态存储持续时间的数据项上指定 thread 特性。这包括全局数据（静态的和外部的）和本地静态数据。不能使用 thread 特性声明自动数据。例如，下面的代码将生成编译器错误：

  ```
  #define Thread   __declspec( thread )
  void func1()
  {
      Thread int tls_i;            /* Error */
  }

  int func2( Thread int tls_i )    /* Error */
  {
     return tls_i;
  }
  ```

- 必须将 thread 特性用于线程本地数据的声明和定义，无论声明和定义是出现在同一文件中还是单独的文件中。例如，下面的代码将生成错误：

  ```
  #define Thread   __declspec( thread )
  extern int tls_i;     /* This generates an error, because the   */
  int Thread tls_i;     /* declaration and the definition differ. */
  ```

- 无法将 thread 特性用作类型修饰符。例如，下面的代码生成一个编译器错误：

  ```
  char *ch __declspec( thread );      /* Error */
  ```

- 不将线程局部变量的地址视为常数，并且涉及此类地址的任何表达式不会被视为常量表达式。这意味着，

无法将线程局部变量的地址用作指针的初始值设定项。例如，编译器会将下面的代码标记为错误：

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

- C 允许使用涉及对自身的引用的表达式来初始化变量，但只适用于非静态范围的对象。例如：

```
#define Thread   __declspec( thread )
Thread int tls_i = tls_i;             /* Error */
int j = j;                            /* Error */
Thread int tls_i = sizeof( tls_i )    /* Okay  */
```

请注意，包含正在初始化的变量的 sizeof 表达式不构成对自身的引用，并且允许使用该表达式。

- 使用 __declspec(thread) 可能会干扰 DLL 导入的延迟加载。

有关使用 thread 特性的详细信息，请参阅多线程主题。

结束 Microsoft 专用

## 请参阅

C 扩展的存储类特性

# 表达式和赋值

2021/8/13 •

本节描述如何构建表达式以及如何在 C 语言中赋值。常量、标识符、字符串和函数调用是表达式中操作的所有操作数。C 语言具有所有常见语言运算符。本节包括这些运算符以及对 C 或 Microsoft C 唯一的运算符。讨论的主题包括：

- 左值和右值表达式
- 常量表达式
- 副作用
- 序列点
- 运算符
- 运算符优先级
- 类型转换
- 类型强制转换

## 请参阅

C 语言参考

# 操作数和表达式

2021/8/16 •

“操作数”是运算符作用于的实体。“表达式”是用于执行以下操作的任意组合的一系列运算符和操作数：

- 计算值
- 指定对象或函数
- 生成副作用

C 中的操作数包括常量、标识符、字符串、函数调用、下标表达式、成员选择表达式以及通过将操作数与运算符组合或将操作数括在括号中而形成的复杂表达式。主表达式中提供了这些操作数的语法。

## 请参阅

表达式和赋值

# C 主要表达式

2021/8/15 •

主表达式是更复杂的表达式的构造块。它们可以是常数、标识符、泛型选择，或者是括号中的表达式。

## 语法

*primary-expression*:

*identifier*

*constant*

*string-literal*

( *expression* )

*generic-selection*

*expression*:

*assignment-expression*

*expression* , *assignment-expression*

## 另请参阅

泛型表达式 操作数和表达式

# 主要表达式中的标识符

2021/8/14 •

标识符可以具有整型、`float`、`enum`、`struct`、`union`、数组、指针或函数类型。如果已将标识符声明为指定对象(此时为左值)或声明为函数(此时为函数指示符),则它是主函数。有关左值的定义,请参阅左值和右值表达式。

数组标识符表示的指针值不是一个变量,因此数组标识符不能构成赋值运算的左操作数,因而不是一个可修改的左值。

声明为函数的标识符表示其值是函数的地址的指针。该指针为返回指定类型的值的函数寻址。因此,函数标识符也不能是赋值运算中的左值。有关详细信息,请参阅标识符。

## 请参阅

C 主要表达式

# 主要表达式中的常量

2021/8/13 •

常量操作数具有它表示的常量的值和类型。字符常数的类型为 `int`。整型常数的类型为 `int`、`long`、`unsigned int` 或 `unsigned long`，具体视整型常数大小和指定值的方式而定。有关详细信息，请参阅常量。

## 请参阅

C 主要表达式

# 主要表达式中的字符串文本

2021/8/16 •

“字符串”是字符、宽字符或包含在双引号内的相邻字符序列。由于它们不是变量，因此字符串和其任一元素都不能作为赋值运算中的左操作数。字符串文本的类型为 `char` 的数组（或对于宽字符串文本，类型为 `wchar_t` 的数组）。表达式中的数组将转换为指针。有关字符串的详细信息，请参阅字符串文本。

## 请参阅

C 主要表达式

# 括号中的表达式

2021/8/13 •

可以在不更改封闭表达式的类型或值的情况下，将任何操作数包含在括号中。例如，在下面的表达式中：

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

`10 + 5` 两边的括号表示先计算 `10 + 5` 的值，然后它会成为除法 (/) 运算符的左操作数。`( 10 + 5 ) / 5` 的结果为 3。如果没有括号，则 `10 + 5 / 5` 的计算结果为 11。

尽管括号会影响操作数在表达式中的分组方式，但它们不能在所有情况下确保按照某个特定顺序进行计算。例如，下列表达式的括号和从左至右的分组不能确保 `i` 的值将位于下列任一子表达式中：

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

编译器可以按任意顺序随意计算乘法的两边内容。如果 `i` 的初始值为零，则整个表达式可能会计算为两个语句之一：

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

因副作用产生的异常将在副作用中进行讨论。

## 请参阅

C 主要表达式

# 一般选择 (C11)

2021/8/25 •

使用 `_Generic` 关键字编写代码，该代码根据参数的类型在编译时选择表达式。这类似于 C++ 中的重载，其中参数的类型确定要调用的函数。在此示例中，参数的类型确定要计算的表达式。

例如，表达式 `_Generic(42, int: "integer", char: "character", default: "unknown");` 计算 `42` 的类型，并在列表中查找匹配类型 `int`。它找到该匹配类型并返回 `"integer"`。

## 语法

*`generic-selection`*:
  `_Generic` ( *`assignment-expression`* , *`assoc-list`* )

*`assoc-list`*:
  *`association`*
  *`assoc-list`* , *`association`*

*`association`*:
  *`type-name`* : *`assignment-expression`*
  `default` : *`assignment-expression`*

第一个 `assignment-expression` 称为控制表达式。控制表达式的类型在编译时确定，并与 `assoc-list` 匹配，以查找要计算和返回的表达式。不会计算控制表达式。例如，
`_Generic(intFunc(), int: "integer", default: "error");` 不会在运行时调用 `intFunc`。

确定控制表达式的类型后，会在与 `assoc-list` 匹配之前删除 `const`、`volatile` 和 `restrict`。

不会计算 `assoc-list` 中未选择的条目。

## 约束

- `assoc-list` 不能多次指定同一类型。
- `assoc-list` 不能指定彼此兼容的类型，例如枚举和该枚举的基础类型。
- 如果一般选择没有默认值，则控制表达式在通用关联列表中必须只有一个兼容的类型名称。

## 示例

使用 `_Generic` 的一种方法是在宏中。<tgmath.h> 标头文件使用 `_Generic` 来根据参数的类型调用正确的数学函数。例如，`cos` 的宏将带有浮点的调用映射到 `cosf`，同时将带有复杂双精度的调用映射到 `ccos`。

下面的示例演示如何编写可标识传递给它的参数的类型的宏。如果 `assoc-list` 中没有匹配控制表达式的条目，则会生成 `"unknown"`：

```
// Compile with /std:c11

#include <stdio.h>

/* Get a type name string for the argument x */
#define TYPE_NAME(X) _Generic((X), \
      int: "int", \
      char: "char", \
      double: "double", \
      default: "unknown")

int main()
{
    printf("Type name: %s\n", TYPE_NAME(42.42));

    // The following would result in a compile error because
    // 42.4 is a double, doesn't match anything in the list,
    // and there is no default.
    // _Generic(42.4, int: "integer", char: "character"));
}

/* Output:
Type name: double
*/
```

## 要求

使用 `/std:c11` 进行编译。

Windows SDK 10.0.20348.0（版本 2104）或更高版本。请参阅 Windows 10 SDK 以下载最新 SDK。有关安装和使用 SDK 进行 C11 和 C17 开发的说明，请参阅在 Visual Studio 中安装 C11 和 C17 支持。

## 另请参阅

`/std`（指定语言标准版本）
泛型类型数学

# 左值和右值表达式

2021/8/13 •

引用内存位置的表达式称为"左值"表达式。左值表示存储区域的"locator"值或"left"值，并暗示它可以出现在等号 ( = ) 的左侧。左值通常是标识符。

引用可修改的位置的表达式称为"可修改的左值"。可修改的左值不能具有数组类型、不完整类型或包含 `const` 特性的类型。要使结构和联合成为可修改的左值，它们必须没有任何包含 `const` 特性的成员。标识符的名称表示存储位置，而变量的值是存储在该位置的值。

如果标识符引用内存位置且如果其类型为算术、结构、联合或指针，则该标识符是可修改的左值。例如，如果 `ptr` 是指向存储区域的指针，则 `*ptr` 是指定 `ptr` 所指向的存储区域的可修改的左值。

以下任一 C 表达式可为左值表达式：

- 整型、浮点、指针、结构或联合类型的标识符
- 计算结果不为数组的下标 ( [ ] ) 表达式
- 成员选择表达式 ( -> 或 . )
- 不引用数组的一元间接寻址 (*) 表达式
- 包含在括号内的左值表达式
- `const` 对象（不可修改的左值）

术语"右值"有时用于描述表达式的值以及将其与左值区分开来。所有左值都是右值，但并不是所有右值都是左值。

**Microsoft 专用**

Microsoft C 包括对 ANSI C 标准的扩展，该扩展允许将左值的转换用作左值，只要对象的大小不通过转换来扩展即可。（有关详细信息，请参阅类型强制转换。）下面的示例阐释了此功能：

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

(long *) p = &l ;       /* Legal cast   */
(long) i = l ;          /* Illegal cast */
```

Microsoft C 的默认设置是启用 Microsoft 扩展。使用 /Za 编译器选项禁用这些扩展。

**结束 Microsoft 专用**

## 请参阅

操作数和表达式

# C 常量表达式

2021/8/17 •

常量表达式将在编译时而不是运行时计算，并且可在可使用常量的任何位置使用。常量表达式的计算结果必须是位于该类型的可表示值范围内的常量。常数表达式的操作数可以是整型常数、字符常数、浮点常数、枚举常数、类型强制转换、`sizeof` 表达式和其他常数表达式。

## 语法

*constant-expression*:
*conditional-expression*

*conditional-expression*:
*logical-OR-expression*
*logical-OR-expression* `?` *expression* `:` *conditional-expression*

*expression*:
*assignment-expression*
*expression* `,` *assignment-expression*

*assignment-expression*:
*conditional-expression*
*unary-expression* *assignment-operator* *assignment-expression*

*assignment-operator*：以下之一
`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`

结构声明符、枚举数、直接声明符、直接抽象声明符和标记语句的非终止符包含 `constant-expression` 非终止符。

整型常数表达式必须用于指定结构的位域成员的大小、枚举常数的值、数组的大小或 `case` 常数的值。

预处理器指令中使用的常量表达式受多项限制的约束。它们被称为受限制的常量表达式。受限制的常量表达式不能包含 `sizeof` 表达式、枚举常量、到任何类型的类型强制转换或浮点类型常量。但它可包含特殊常量表达式 `defined (` identifier `)`。

## 请参阅

操作数和表达式

# 表达式计算 (C)

2021/8/16 •

涉及赋值、一元递增、一元递减或调用函数的表达式可能具有其计算附带的结果(副作用)。当达到"序列点"时,确保对序列点后面的任何内容执行计算之前已计算序列点前面的所有内容(包括任何副作用)。

"副作用"是由表达式的计算引起的更改。只要表达式计算更改变量的值,就会出现副作用。所有赋值运算都具有副作用。如果函数调用通过直接赋值或使用指针进行间接赋值来更改外部可见项的值,则函数调用还会产生副作用。

## 请参阅

操作数和表达式

# 副作用

2021/8/13 •

表达式的计算顺序由特定实现定义，但语言保证特定的计算顺序时除外（如计算的优先级和顺序中概述）。例如，下列函数调用中将出现副作用：

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

函数调用的参数可按任意顺序进行计算。表达式 `i + 1` 可以在 `i = j + 2` 前计算，或者 `i = j + 2` 可以在 `i + 1` 前计算。每种情况下的结果都不同。同样，无法保证将哪些字符会实际传递到 `myproc`。由于一元递增和递减运算涉及赋值，因此此类运算可能导致副作用，如以下示例中所示：

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

在此示例中，修改后的 `x` 值不可预知。下标的值可以是 `i` 的新值或旧值。结果因编译器或优化级别而异。

由于 C 未定义副作用的计算顺序，因此上面讨论的两种计算方式都是正确的，并且其中一个是可以实现的。若要确保你的代码可移植且清晰明了，请避免使用依赖于副作用的特定计算顺序的语句。

## 请参阅

表达式计算

# C 序列点

2021/8/17 •

在连续的"序列点"之间，仅能通过表达式修改一次对象的值。C 语言定义以下序列点：

- 逻辑"与"运算符 (&& ) 的左操作数。完全计算逻辑"与"运算符的左操作数，并在继续之前完成所有副作用。如果左操作数的计算结果为 false (0)，则不计算另一个操作数。
- 逻辑"或"运算符 ( `||` ) 的左操作数。完全计算逻辑"或"运算符的左操作数，并在继续之前完成所有副作用。如果左操作数的计算结果为 true（非零），则不计算另一个操作数。
- 逗号运算符的左操作数。完全计算逗号运算符的左操作数，并在继续之前完成所有副作用。始终计算逗号运算符的两个操作数。请注意，函数调用中的逗号运算符不保证计算顺序。
- 函数调用运算符。计算函数的所有参数，并在输入函数前完成所有副作用。未指定参数之间的计算顺序。
- 条件运算符的第一个操作数。完全计算条件运算符的第一个操作数，并在继续之前完成所有副作用。
- 完全初始化表达式的末尾（即，不是一个表达式的一部分的另一个表达式，如声明语句中的初始化的末尾）。
- 表达式语句中的表达式。表达式语句由可选表达式后跟分号 (; ) 组成。为其副作用计算该表达式，并且此计算后面有一个序列点。
- 选择语句（ `if` 或 `switch` ）中的控制表达式。完全计算该表达式，并在执行依赖于选择的代码之前完成所有副作用。
- `while` 或 `do` 语句的控制表达式。在执行 `while` 或 `do` 循环的下一个迭代中的任何语句前，完全计算表达式，并且完成所有副作用。
- `for` 语句的所有三个表达式。在执行 `for` 循环的下一个迭代中的任何语句前，完全计算表达式，并且完成所有副作用。
- `return` 语句中的表达式。完全计算该表达式，并在控制返回调用函数之前完成所有副作用。

## 请参阅

表达式计算

# C 运算符

2021/8/16 •

C 运算符是 C++ 的内置运算符的子集。

有三种类型的运算符。一元表达式由在操作数最前面插入的一元运算符或后面跟表达式的 `sizeof` 关键字组成。该表达式可以是变量的名称，也可以是强制转换表达式。如果表达式是强制转换表达式，则它必须括在括号中。二进制表达式包括通过二元运算符联接的两个操作数。三元表达式包括通过条件表达式运算符联接的三个操作数。

C 包含以下一元运算符：

| | |
|---|---|
| ** | "**" |
| - ~ ! | 求反和补数运算符 |
| * & | 间接寻址运算符和 address-of 运算符 |
| `sizeof` | Size 运算符 |
| + | 一元加运算符 |
| ++ -- | 一元递增和减量运算符 |

二元运算符从左至右关联。C 提供了以下二进制运算符：

| | |
|---|---|
| ** | "**" |
| * / % | 乘法运算符 |
| + - | 相加运算符 |
| << >> | 移位运算符 |
| <** > <= **>= == != | 关系运算符 |
| & \| ^ | 位运算符 |
| && \|\| | 逻辑运算符 |
| , | 有序评估运算符 |

Microsoft 16 位 C 编译器的早期版本所支持的基本运算符 (:> ) 在 C 语言语法摘要中进行了介绍。

条件表达式运算符的优先级低于二进制表达式的优先级并与其在右关联上存在差异。

使用运算符的表达式还包括赋值表达式，该表达式使用一元或二元赋值运算符。一元赋值运算符是增量 (++ ) 和减量 (-- ) 运算符；二元赋值运算符是简单赋值运算符 (= ) 和复合赋值运算符。每个复合赋值运算符是另一个二元运算符与简单赋值运算符的组合。

## 请参阅

- 表达式和赋值

# 计算的优先级和顺序

2021/8/13 •

C 运算符的优先级和结合性将影响表达式中操作数的分组和计算。仅当存在优先级较高或较低的其他运算符时，运算符的优先级才有意义。首先计算带优先级较高的运算符的表达式。也可以通过"绑定"一词描述优先级。优先级较高的运算符被认为具有更严格的绑定。

下表总结了 C 运算符的优先级和结合性（计算操作数的顺序），并按照从最高优先级到最低优先级的顺序将其列出。如果几个运算符一起出现，则其具有相同的优先级并且将根据其结合性对其进行计算。以后缀运算符开头的部分描述了表中的运算符。此部分的其余部分提供了有关优先级和结合性的常规信息。

## C 运算符的优先级和关联性

| 运算符[1] | 说明 | 关联性 |
|---|---|---|
| `[` `]` `(` `)` `.` `->` `++` `--`（后缀） | 表达式 | 从左到右 |
| `sizeof` `&` `*` `+` `-` `~` `!` `++` `--`（前缀） | 一元 | 从右到左 |
| *typecasts* | 一元 | 从右到左 |
| `*` `/` `%` | 乘法 | 从左到右 |
| `+` `-` | 加法 | 从左到右 |
| `<<` `>>` | 按位移动 | 从左到右 |
| `<` `>` `<=` `>=` | 关系 | 从左到右 |
| `==` `!=` | 相等 | 从左到右 |
| `&` | 按位"与" | 从左到右 |
| `^` | 按位"异或" | 从左到右 |
| `\|` | 按位"与或" | 从左到右 |
| `&&` | 逻辑"与" | 从左到右 |
| `\|\|` | 逻辑"或" | 从左到右 |
| `? :` | 条件表达式 | 从右到左 |
| `=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `\|=` | 简单和复合赋值[2] | 从右到左 |

| [1] | [2] | [3] |
| --- | --- | --- |
| `,` | 顺序计算 | 从左到右 |

[1] 运算符按优先级的降序顺序列出。如果多个运算符出现在同一行或一个组中，则它们具有相同的优先级。

[2] 所有简单的和复合的赋值运算符都有相同的优先级。

表达式可以包含优先级相同的多个运算符。当多个具有相同级别的这类运算符出现在表达式中时，计算将根据该运算符的结合性按从右到左或从左至右的顺序来执行。计算的方向不影响在相同级别包括多个乘法 (`*`)、加法 (`+`) 或二进制按位 (`&`、`|` 或 `^`) 运算符的表达式的结果。语言未定义运算的顺序。如果编译器可以保证一致的结果，则编译器可以按任意顺序随意计算此类表达式。

只有顺序计算 (`,`)、逻辑“与”(`&&`)、逻辑“或” (`||`)、条件表达式 (`? :`) 和函数调用运算符构成序列点，因此，确保对其操作数的计算采用特定顺序。函数调用运算符是一组紧跟函数标识符的圆括号。确保顺序计算运算符 (`,`) 按从左到右的顺序计算其操作数。（函数调用中的逗号运算符与顺序计算运算符不同，不提供任何此类保证。）有关详细信息，请参阅序列点。

逻辑运算符还确保按从左至右的顺序计算操作数。但是，它们会计算确定表达式结果所需的最小数目的操作数。这称作“短路”计算。因此，无法计算表达式的一些操作数。例如，在下面的表达式中

```
x && y++
```

仅当 `y++` 为 true（非零）时，才计算第二操作数 (`x`)。因此，如果 `y` 为 false (0)，则 `x` 不增加。

## 示例

以下列表显示编译器如何自动绑定多个示例表达式：

| [1] | [2] |
| --- | --- |
| `a & b \|\| c` | `(a & b) \|\| c` |
| `a = b \|\| c` | `a = (b \|\| c)` |
| `q && r \|\| s--` | `(q && r) \|\| s--` |

在第一个表达式中，按位“与”运算符 (`&`) 的优先级高于逻辑“或”运算符 (`||`) 的优先级，因此，`a & b` 构成了逻辑“或”运算的第一操作数。

在第二个表达式中，逻辑“或”运算符 (`||`) 的优先级高于简单赋值运算符 (`=`) 的优先级，因此，`b || c` 在赋值中分组为右操作数。请注意，赋给 `a` 的值为 0 或 1。

第三个表达式显示可能会生成意外结果的格式正确的表达式。逻辑“与”运算符 (`&&`) 的优先级高于逻辑“或”运算符 (`||`) 的优先级，因此，将 `q && r` 分组为操作数。由于逻辑运算符确保按从左到右的顺序计算操作数，因此 `q && r` 先于 `s--` 被计算。但是，如果 `q && r` 计算的结果为非零值，则不计算 `s--`，并且 `s` 不会减少。如果 `s` 未减少会导致程序出现问题，则 `s--` 应显示为表达式的第一操作数，或者在单独的运算中应减少 `s`。

以下表达式是非法的并会在编译时生成诊断消息：

| [1] | [2] |
| --- | --- |
| `p == 0 ? p += 1: p += 2` | `( p == 0 ? p += 1 : p ) += 2` |

在此表达式中，相等运算符 (`==`) 的优先级最高，因此，将 `p == 0` 分组为操作数。条件表达式运算符 (`? :`) 具

有下一个最高级别的优先级。其第一操作数是 `p == 0`，第二操作数是 `p += 1`。但是，条件表达式运算符的最后一个操作数被视为 `p` 而不是 `p += 2`，因为与复合赋值运算符相比，`p` 的匹配项将更紧密地绑定到条件表达式运算符。由于 `+= 2` 没有左操作数，因此发生语法错误。您应使用括号以防止此类错误发生并生成可读性更高的代码。例如，可以按如下所示使用括号来更正和阐明前面的示例：

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

## 请参阅

C 运算符

# 常用算术转换

2021/8/16 •

大多数 C 运算符执行类型转换以将表达式的操作数引入常见类型或将较短的值扩展到计算机运算中使用的整数大小。C 运算符执行的转换取决于特定的运算符和操作数的类型。但是，许多运算符对整型和浮点型的操作数执行相似的转换。这些转换称为"算术转换"。从操作数值到兼容类型的转换会导致不改变其值。

以下汇总的算术转换称为"常用算术转换"。这些步骤仅应用于需要算术类型的二元运算符。目的是为了产生常见类型（它也是结果的类型）。若要确定实际执行的转换，编译器可将以下算法应用于表达式中的二元运算。下面的步骤不具有优先级。

1. 如果其中一个操作数的类型为 `long double`，则另一个操作数被转换为 `long double` 类型。
2. 如果上述一个条件不满足，且其中一个操作数的类型为 `double`，则另一个操作数被转换为 `double` 类型。
3. 如果上述两个条件不满足，且其中一个操作数的类型为 `float` 类型，则另一个操作数被转换为 `float` 类型。
4. 如果未满足上述三个条件（所有操作数都不是浮点型），则对操作数执行整型转换，如下所示：
   - 如果其中一个操作数的类型为 `unsigned long`，则另一个操作数被转换为 `unsigned long` 类型。
   - 如果上述一个条件不满足，且其中一个操作数的类型为 `long` 类型，另一个操作数的类型为 `unsigned int`，则这两个操作数都被转换为 `unsigned long` 类型。
   - 如果上述两个条件不满足，且其中一个操作数的类型为 `long`，则另一个操作数被转换为 `long` 类型。
   - 如果上述三个条件不满足，且其中一个操作数的类型为 `unsigned int`，则另一个操作数被转换为 `unsigned int` 类型。
   - 如果上述任何条件都不满足，则这两个操作数都被转换为 `int` 类型。

以下代码阐释了这些转换规则：

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

dVal = iVal * ulVal; /* iVal converted to unsigned long
                      * Uses step 4.
                      * Result of multiplication converted to double
                      */
dVal = ulVal + fVal; /* ulVal converted to float
                      * Uses step 3.
                      * Result of addition converted to double
                      */
```

## 请参阅

C 运算符

# 后缀运算符

2021/8/11 •

后缀运算符在表达式计算中具有最高优先级（最紧密的绑定）。

## 语法

*postfix-expression*:
  *primary-expression*
  *postfix-expression* **[** *expression* **]**
  *postfix-expression* **(** *argument-expression-list*$_{opt}$ **)**
  *postfix-expression* **.** *identifier*
  *postfix-expression* **->** *identifier*
  *postfix-expression* **++**
  *postfix-expression* **--**

具有此优先级别的运算符为数组下标、函数调用、结构和联合成员以及后缀递增和递减运算符。

## 请参阅

C 运算符

# 一维数组

2021/8/16 •

后跟用方括号 ([ ] ) 括起的表达式的后缀表达式是数组对象元素的下标表示形式。下标表达式表示在表示为以下形式时位于超出 postfix-expression 的 expression 位置的地址的值

```
postfix-expression [ expression ]
```

通常，postfix-expression 表示的值是一个指针值（如数组标识符），而 expression 是一个整数值。但是，从语法上来说，只需要一个表达式是指针类型，另一个表达式是整型。因此整数值可以位于 postfix-expression 位置，指针值可以位于 expression 的方括号中或"下标"位置。例如，以下代码是合法的：

```
// one_dimensional_arrays.c
int sum, *ptr, a[10];
int main() {
   ptr = a;
   sum = 4[ptr];
}
```

下标表达式通常用于引用数组元素，但您可以将下标应用于任何指针。无论值的顺序如何，expression 必须用方括号 ([ ] ) 括起来。

通过将整数值添加到指针值，然后将间接寻址运算符 (*) 应用于结果，可以计算下标表达式。（有关间接寻址运算符的讨论，请参阅间接寻址运算符和 Address-of 运算符。）实际上，对于一维数组，假定 `a` 是指针，`b` 是整数，则以下四个表达式等效：

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

根据加法运算符的转换规则（在相加运算符中提供），可通过将整数值乘以指针寻址的类型的长度来将整数值转换为地址偏移量。

例如，假设标识符 `line` 引用包含 `int` 值的数组。以下过程用于计算下标表达式 `line[ i ]`：

1. 整数值 `i` 与定义为 `int` 项的长度的字节数相乘。`i` 的转换后的值表示 `i` `int` 位置。

2. 此转换后的值与原始指针值 ( `line` ) 相加，以生成与 `line` 之间偏移 `i` `int` 位置的地址。

3. 间接寻址运算符将应用于此新地址。结果是位于该位置（直观地说，就是 `line [ i ]` ）的数组元素的值。

下标表达式 `line[0]` 表示行的第一个元素的值，因为 `line` 表示的地址的偏移量为 0。同样，表达式（如 `line[5]` ）引用了从行偏移 5 个位置的元素，或数组的第 6 个元素。

## 请参阅

下标运算符：

# 多维数组 (C)

2021/8/12 •

下标表达式还可以有多个下标，如下所示：

```
expression1 [ expression2 ] [ expression3 ] ...
```

下标表达式从左至右关联。首先计算最左侧的下标表达式 expression1 [expression2] 。通过添加 expression1 和 expression2 得到的地址构成一个指针表达式；然后 expression3 将添加到此指针表达式，从而构成一个新的指针表达式，依此类推，直到添加最后一个下标表达式。在计算最后一个下标表达式之后应用间接寻址运算符 (*)，除非最终指针值寻址数组类型（请参阅以下示例）。

具有多个下标的表达式引用"多维数组"的元素。多维数组是其元素为数组的数组。例如，三维数组的第一个元素是一个具有两个维度的数组。

## 示例

对于下面的示例，名为 `prop` 的数组声明为包含 3 个元素，其中每个元素都是由 `int` 值组成的 4x6 数组。

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

对 `prop` 数组的引用如下所示：

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

上面的示例展示了如何引用 `prop` 的第二个单独的 `int` 元素。数组将按行存储，因此最后一个下标变化最快；表达式 `prop[0][0][2]` 引用数组的下一个（第三个）元素，依此类推。

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

此语句是对 `prop` 的单个元素的更复杂的引用。此表达式的计算方式如下：

1. 第一个下标 `2` 先与 4x6 `int` 数组的大小相乘，再与指针值 `prop` 相加。结果将指向 `prop` 的第三个 4 x 6 数组。
2. 第二个下标 `1` 先与 6 元素 `int` 数组的大小相乘，再与 `prop[2]` 表示的地址相加。
3. 由于 6 元素数组的每个元素都是 `int` 值，因此最后一个下标 `3` 先与 `int` 的大小相乘，再与 `prop[2][1]` 相加。生成的指针将寻址到 6 元素数组的第四个元素。
4. 将对指针值应用间接寻址运算符。结果是此地址处的 `int` 元素。

下面两个示例演示未应用间接寻址运算符的情况。

```
ip = prop[2][1];

ipp = prop[2];
```

在上面的第一个语句中，表达式 `prop[2][1]` 是对三维数组 `prop` 的有效引用；它引用一个 6 元素数组（上面已声明）。由于指针值将寻址到一个数组，因此不会应用间接寻址运算符。

同样，第二个语句 `prop[2]` 中的表达式 `ipp = prop[2];` 的结果是一个寻址到一个二维数组的指针值。

## 请参阅

下标运算符：

# 函数调用 (C)

2021/8/16 •

函数调用 是包含被调用函数的名称或函数指针的值以及（可选）传递给函数的自变量的表达式。

## 语法

*postfix-expression*:
  *postfix-expression* ( *argument-expression-list*$_{opt}$ )

*argument-expression-list*:
  *assignment-expression*
  *argument-expression-list* , *assignment-expression*

postfix-expression 的计算结果必须为函数地址（例如，函数标识符或函数指针值），argument-expression-list 是其值（“参数”）传递到函数的表达式的列表（用逗号分隔）。argument-expression-list 参数可以为空。

function-call 表达式具有函数的返回值的值和类型。函数不能返回数组类型的对象。如果函数的返回类型是 `void`（即函数已被声明为永不返回值），那么函数调用表达式的类型也是 `void`。（有关详细信息，请参阅函数调用。）

## 请参阅

函数调用运算符 : ()

# 结构和联合成员

2021/8/16 •

“成员选择表达式”是指结构和联合的成员。此类表达式具有选定成员的值和类型。

```
postfix-expression . identifier
postfix-expression -> identifier
```

此列表描述了成员选择表达式的两种形式：

1. 在第一种形式中，postfix-expression 表示类型为 `struct` 或 `union` 的值，而 identifier 为指定的结构或联合的成员命名。运算的值是 *identifier* 的值且是一个左值（如果 *postfix-expression* 是左值）。有关详细信息，请参阅左值和右值表达式。
2. 在第二种形式中，*postfix-expression* 表示指向结构或联合的指针，而 *identifier* 命名指定的结构或联合的成员。该值是 *identifier* 的值且是一个左值。

成员选择表达式的两种形式具有类似的效果。

实际上，如果句点前面的表达式由适用于指针值的间接寻址运算符 (*) 构成，则包含成员选择运算符 ( -> ) 的表达式是使用句点 ( . ) 的表达式的速记版。因此，

```
expression->identifier
```

等效于

```
(*expression).identifier
```

当 *expression* 为指针值时。

## 示例

以下示例引用此结构声明。有关这些示例中使用的间接寻址运算符 (*) 的信息，请参阅间接寻址运算符和 Address-of 运算符。

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

`item` 结构的成员选择表达式与下面类似：

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

在上面的示例中，将 `item` 结构的地址分配给结构的 `sp` 成员。这意味着，`item` 包含一个指向自身的指针。

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

在此示例中，将指针表达式 `item.sp` 与成员选择运算符 ( `->` ) 一起使用以便将值赋给成员 `a` 。

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

此语句演示如何从结构数组中选择单个结构成员。

## 请参阅

成员访问运算符 `.` 和 `->`

# C 后缀增量和减量运算符

2021/8/12 •

后缀递增和递减运算符的操作数是可修改的左值的标量类型。

## 语法

*postfix-expression*:
  *postfix-expression* ++
  *postfix-expression* --

后缀递增或递减运算的结果是操作数的值。获取结果后，操作数的值将增加（或减少）。以下代码演示了后缀递增运算符。

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

在本示例中，变量 `var` 先与 0 进行比较，然后增加。如果 `var` 在增加之前为正数，则执行下一条语句。首先，`q` 所指向的对象的值赋给 `p` 所指向的对象。然后，`q` 和 `p` 增加。

## 请参阅

后缀增量和减量运算符：++ 和 --

# C 一元运算符

2021/8/12 •

一元运算符出现在其操作数前，并按照从右到左的顺序关联。

## 语法

*unary-expression*: *postfix-expression*

**++** *unary-expression*

`--` *unary-expression*

*unary-operator cast-expression*

`sizeof` *unary-expression*

**sizeof (** *type-name* **)**

*unary-operator*: one of **&** ***** **+** **-** `~` **!**

## 请参阅

C 运算符

# 前缀增量和减量运算符

2021/8/12 •

当增量和减量运算符出现在操作数的前面时，一元运算符（`++` 和 `--`）称作“前缀”增量和减量运算符。与前缀递增和递减相比，后缀递增和递减的优先级更高。操作数必须具有整型、浮点型或指针类型，且必须是可修改的左值表达式（不含 `const` 特性的表达式）。结果为一个左值。

当运算符出现在其操作数的前面时，操作数会递增或递减，并且其新值为表达式的结果。

整型或浮动类型的操作数将按整数值 1 递增或递减。结果的类型与操作数类型相同。指针类型的操作数将按其所寻址对象的大小递增或递减。递增的指针将指向下一个对象；递减的指针将指向上一个对象。

## 示例

此示例阐释一元前缀递减运算符：

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

在此示例中，变量 `i` 在用作 `line` 的下标之前是递减的。

## 请参阅

C 一元运算符

# 间接寻址和 Address-of 运算符

2021/8/16 •

一元间接寻址运算符 (*) 通过指针间接访问一个值 。操作数必须是指针类型。操作的结果是操作数所寻址的值；即其操作数指向的地址处的值。结果的类型是操作数寻址的类型。

如果操作数的类型为指向类型的指针，则间接寻址运算符的结果为类型 。如果操作数指向函数，则结果是函数指示符。如果指向对象，则结果为指定对象的左值。

如果指针值无效，则间接寻址运算符的结果不确定。以下是使指针值无效的一些最常见条件：

- 该指针为 null 指针。
- 在引用过程中，该指针在对象生存期结束后（例如，对象已超出范围或已解除分配）指定其地址。
- 该指针指定未针对所指向的对象类型正确对齐的地址。
- 该指针指定执行程序未使用的地址。

一元 address-of 运算符 (&) 给出其操作数的地址 。操作数必须是用于指定对象的左值，该对象不得声明为“register”，也不得为位域、一元 * 运算符或数组取消引用 ([]) 运算符的结果或函数指示符 。操作数类型为类型时，结果的类型为指向类型的指针 。

如果操作数是一元 * 运算符的结果，则不对两个运算符进行运算，并且结果像是同时省略了这两个运算符 。结果不为左值，运算符约束仍适用。如果操作数是 [] 运算符的结果，则不会对 & 运算符进行运算，也不会对 [] 运算符暗含的一元 * 进行运算 。其结果与删除 & 运算符并将 [] 运算符更改为 + 运算符的效果相同 。否则，结果为指向对象或操作数指定的函数的指针。

## 示例

下面的示例使用这些常用声明：

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

此语句使用 address-of 运算符 (&) 来获取数组 `a` 的第六个元素的地址 。结果存储在指针变量 `pa` 中：

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

在此示例中，使用间接寻址运算符 (*) 来访问存储在 `pa` 中的地址处的 `int` 值。将此值分配给整数变量 `x`：

```
x = *pa;
```

此示例表明对 `x` 的地址应用间接运算符的结果与 `x` 相同：

```
assert( x == *&x );
```

此示例演示用于声明指向函数的指针的等效方法：

```
int roundup( void );     /* Function declaration */

int  *proundup  = roundup;
int  *pround  = &roundup;
assert( pround == proundup );
```

一旦声明函数 `roundup`，将声明并初始化指向 `roundup` 的两个指针。第一个指针为 `proundup`，它仅通过函数名称进行初始化；第二个指针为 `pround`，它在初始化中使用 address-of 运算符。初始化是等效的。

## 请参阅

间接寻址运算符：*
Address-of 运算符：&

# 一元算术运算符

2021/8/16 •

下面的列表中讨论了 C 一元加、算术求反、求补和逻辑求反运算符：

| ||
|---|---|
| + | 括号内的表达式前面的一元加运算符强制对包含的运算进行分组。它用于涉及多个结合的或可交换的二元运算符的表达式。操作数必须具有算法类型。结果为操作数的值。整型操作数将进行整型提升。结果的类型为提升后的操作数的类型。 |
| - | 算术求反运算符生成其操作数的负值(2 的补数)。操作数必须是整型值或浮点值。此运算符执行常用算术转换。 |
| `~` | 按位求补(或按位"非")运算符将产生其操作数的按位补数。操作数必须为整型。此运算符执行常用算术转换;结果具有转换后的操作数的类型。 |
| ! | 如果其操作数为 true(非零)，则逻辑求反(逻辑"非")运算符将生成值 0;如果其操作数为 false (0)，则生成值 1。结果的类型为 `int`。操作数必须是整型值、浮点值或指针值。 |

指针上的一元算术运算是非法的。

## 示例

下面的示例演示了一元算术运算符：

```
short x = 987;
    x = -x;
```

在上面的示例中，`x` 的新值为 987 的负数或 -987。

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

在此示例中，赋给 `y` 的新值是无符号值 0xAAAA 或 0x5555 的二进制反码。

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

如果 `x` 大于或等于 `y`，则该表达式的结果为 1 (true)。如果 `x` 小于 `y`，则结果为 0 (false)。

## 请参阅

使用一元运算符的表达式

# sizeof 运算符 (C)

2021/8/15 •

`sizeof` 运算符提供了存储操作数的类型的对象所需的存储量(以字节为单位)。利用此运算符，你可以避免在程序中指定依赖于计算机的数据大小。

## 语法

```
sizeof unary-expression
sizeof ( type-name )
```

## 备注

操作数是作为 *unary-expression* 或 type-cast 表达式的标识符(即，用括号括起的类型说明符)。*unary-expression* 不能表示位域对象、不完整类型或函数指示符。结果是一个无符号整数常量。标准标头 STDDEF.H 将此类型定义为 **size_t**。

当你将 `sizeof` 运算符应用于数组标识符时，结果是整个数组的大小，而不是由数组标识符表示的指针的大小。

当你将 `sizeof` 运算符应用于结构或联合类型名称，或应用于结构或联合类型的标识符时，结果是结构或联合中的字节数(包括内部填充和尾部填充)。此大小可能包括用于在内存边界上对齐结构成员或联合成员的内部和尾部填充。因此，结果可能不对应于通过将各个成员的存储需求相加计算出的大小。

如果未调整大小的数组是结构的最后一个元素，则 `sizeof` 运算符返回没有此数组的结构的大小。

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

此示例使用 `sizeof` 运算符来传递 `int` 的大小(因计算机而异)，以作为名为 `calloc` 的运行时函数的参数。该函数返回的值存储在 `buffer` 中。

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

在此示例中，`strings` 是包含指向 `char` 的指针的数组。指针的数目是数组中元素的数目，但是未指定。使用 `sizeof` 运算符来计算数组中的元素数量，可以很容易地确定指针的数量。`const` 整数值 `string_no` 初始化为此数字。由于它是 `const` 值，因此无法修改 `string_no`。

## 请参阅

C 运算符
C++ 内置运算符、优先级和关联性

# 强制转换运算符

2021/8/13 •

在特定情况下，类型强制转换提供了用于显式转换对象类型的方法。

## 语法

*cast-expression*: *unary-expression*

( *type-name* ) *cast-expression*

在进行类型强制转换后，编译器将 cast-expression 视为类型 type-name 。强制转换可用于在任意标量类型的对象与任何其他标量类型之间进行来回转换。显式类型强制转换受到确定隐式转换效果的相同规则的约束，如赋值转换中所述。有关强制转换的其他约束可能来源于特定类型的实际大小或表示形式。有关整型类型的实际大小的信息，请参阅基本类型的存储 。有关类型强制转换的详细信息，请参阅类型强制转换。

## 请参阅

强制转换运算符 : ()

# C 乘法运算符

2021/8/13 •

乘法运算符执行乘法 (*)、除法 (/) 和余数 (%) 运算 。

## 语法

*multiplicative-expression*: *cast-expression* *multiplicative-expression* ***** *cast-expression* *multiplicative-expression* **/** *cast-expression* *multiplicative-expression* **%** *cast-expression*

余数操作符 (%) 的操作数必须是整数 。乘法 (*) 和除法 (/) 运算符可采用整型或浮点类型操作数;操作数的类型可以是不同的。

乘法运算符对操作数执行常用算术转换。结果的类型是转换后操作数的类型。

> **NOTE**
> 由于在溢出或下溢条件不提供由乘法运算符执行的转换，因此，如果乘法操作的结果在转换后不能用操作数类型表示，则信息可能丢失。

C 乘法运算符的描述如下:

| | |
|---|---|
| | |
| * | 乘法运算符使其两个操作数相乘。 |
| / | 除法运算符使第一个操作数除以第二个操作数。如果两个整数操作数相除，结果不是整数，则根据下列规则截断它:<br><br>- 根据 ANSI C 标准，被 0 除的结果是不确定的。Microsoft C 编译器将在编译时或运行时生成错误。<br><br>- 如果两个操作数都为正或无符号，则结果将截断到 0。<br><br>- 如果其中一个操作数为负，则不管操作结果是小于或等于代数商的最大整数还是大于或等于代数商的最小整数，结果均为定义的实现。(请参阅下面的 Microsoft 专用部分。) |
| % | 第一个操作数除以第二个操作数时，余数运算符的结果是余数。如果除法不精确，则结果将由下列规则确定:<br><br>- 如果右操作数为零，则结果是不确定的。<br><br>- 如果两个操作数均为正或无符号，则结果为正。<br><br>- 如果其中一个操作数为负，并且结果不精确，则结果将是定义的实现。(请参阅下面的 Microsoft 专用部分。) |

**Microsoft 专用**

在其中一个操作数为负的除法中，截断的方向将是朝向 0。

如果使用余数运算符的除法中任一操作数为负，则结果与被除数(表达式中的第一个操作数)有相同的符号。

## 示例

如下所示的声明将用于下列示例：

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

此语句使用乘法运算符：

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

在此示例中，`x` 乘以 `i` 将得到值 20.0。结果的类型为 `double`。

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

在此示例中，10 除以 3。结果将被截断到 0，同时产生整数值 3。

```
n = i % j;
```

当 10 除以 3 时，此语句为 `n` 分配整数余数 1。

**Microsoft 专用**

余数的符号与被除数的符号相同。例如：

```
50 % -6 = 2
-50 % 6 = -2
```

在所有情况下，`50` 和 `2` 具有相同的符号。

**结束 Microsoft 专用**

## 请参阅

[乘法运算符和取模运算符]

# C 加法运算符

2021/8/17 •

加法运算符执行加法 ( + ) 和减法 ( - ) 运算。

## 语法

*additive-expression*:
  *multiplicative-expression*
  *additive-expression* + *multiplicative-expression*
  *additive-expression* - *multiplicative-expression*

> **NOTE**
> 虽然 *additive-expression* 的语法包括 *multiplicative-expression*，但这并不表示需要使用乘法表达式。对于 *multiplicative-expression*、*cast-expression* 和 *unary-expression*，请参阅 C 语言语法摘要中的语法。

操作数可以是整型值或浮点值。还可以对指针值执行一些加法运算，如针对每个运算符的讨论中所述。

相加运算符对整型和浮点型操作数执行常用算术转换。结果的类型是转换后操作数的类型。由于相加运算符执行的转换不提供溢出或下溢条件，因此，如果在转换后，无法用操作数类型表示加法运算的结果，则信息可能会丢失。

## 请参阅

加法运算符：+ 和 -

# 加 (+)

2021/8/13 •

加法运算符 (+) 导致添加其两个操作数。两个操作数可同时为整型或浮点型，或者一个操作数为指针，另一个操作数为整数。

在将整数添加到指针时，会通过将整数值 (i) 乘以指针寻址的值的大小来转换该整数值。转换后，整数值表示 *i* 个内存位置，其中每个位置均具有指针类型所指定的长度。在将转换的整数值添加到指针值时，结果为表示原始地址中的地址 i 位置的新指针值。新指针值对类型与原始指针值的类型相同的值进行寻址，因此它与数组索引相同(请参阅一维数组和多维数组)。如果 sum 指针指向数组的外部，除非位于在高端外的第一个位置，否则结果是不确定的。有关详细信息，请参阅指针算法。

## 请参阅

C 加法运算符

# 减法 (–)

2021/8/12 •

减法运算符 ( - ) 从第一个操作数中减去第二个操作数。两个操作数可同时为整型或浮点型，或者一个操作数为指针，另一个操作数为整数。

在将两个指针相减时，会通过将差值除以指针寻址的类型值的大小来将该差值转换为带符号的整数值。整数值的大小由标准包含文件 STDDEF.H 中的类型 **ptrdiff_t** 定义。结果表示两个地址间的类型的内存位置数。只能保证结果对于同一数组的两个元素有意义，如指针算法中讨论的那样。

在用指针值减去整数值时，减法运算符会通过将整数值乘以指针寻址的值的大小来转换整数值 (*i*)。转换后，整数值表示 *i* 个内存位置，其中每个位置均具有指针类型所指定的长度。当用指针值减去转换后的整数值时，结果将为原始地址前的内存地址 *i* 位置数。新指针指向通过原始指针值寻址的类型的值。

## 请参阅

C 加法运算符

# 使用加法运算符

2021/8/13 •

以下示例阐释了加法和减法运算符，它使用这些声明：

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

这些语句是等效的：

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

`i` 的值先与 `float` 的长度相乘，再与 `&x[4]` 相加。结果指针值是 `x[8]` 的地址。

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

在此示例中，用 `x` 的第五个元素的地址（由 `x[i-2]` 给定）减去 `x` 的第三个元素的地址（由 `x[i]` 给定）。用差值除以 `float` 的长度；结果为整数值 2。

## 请参阅

C 加法运算符

# 指针算术

2021/8/12 •

仅当指针操作数寻址数组成员且整数值在同一数组的边界中产生偏移时，涉及指针和整数的加法运算才会提供有意义的结果。当整数值转换为地址偏移量时，编译器将假定只有大小相同的内存位置位于原始地址和该地址加上偏移量之间。

此假设对数组成员有效。按照定义，数组是一系列相同类型的值；数组元素位于连续内存位置。但是，任何类型（数组元素除外）的存储不一定由相同类型的标识符填充。即，在内存位置（即使是相同类型的位置）之间可能会出现空白。因此，对任何值（数组元素除外）的地址进行加法或减法的结果是不确定的。

同样，当两个指针值相减时，转换将假定只有相同类型的值（没有空白）位于操作数给定的地址之间。

## 请参阅

C 加法运算符

# 按位移位运算符

2021/8/16 •

移位运算符按第二个操作数指定的位置数量向左 (<<) 或向右 (>>) 移动第一个操作数 。

## 语法

*shift-expression*:
  *additive-expression*
  *shift-expression* << *additive-expression*
  *shift-expression* >> *additive-expression*

两个操作数都必须是整数值。这些运算符执行常用算术转换;结果的类型是转换后左操作数的类型。

对于左移，留空的右位将设置为 0。对于右移，将根据转换后第一个操作数的类型填充留空的左位。如果类型是 `unsigned`，则将留空的左位设置为 0。否则，将使用符号位的副本填充它们。对于没有溢出的左移运算符，语句

```
expr1 << expr2
```

等效于乘以 $2^{expr2}$。对于右移运算符，

```
expr1 >> expr2
```

等效于除以 $2^{expr2}$（如果 `expr1` 为无符号或具有非负值）。

如果第二个操作数为负，或者右操作数大于或等于提升后的左操作数的宽度（以位为单位），则移位运算的结果不确定的。

由于没有为溢出或下溢情况提供移位运算符执行的转换，因此当移位运算的结果不能用转换后第一个操作数的类型表示时，信息可能丢失。

```
unsigned int x, y, z;

x = 0x00AA;
y = 0x5500;

z = ( x << 8 ) + ( y >> 8 );
```

在此示例中，`x` 将向左移位 8 个位置，`y` 将向右移位 8 个位置。移位值（假定 0xAA55）将相加并赋给 `z`。

将负值向右移位可生成原始值一半的值（向下舍入）。例如，-253（二进制 11111111 00000011）向右移动 1 位会生成 -127（二进制 11111111 10000001）。将 + 253 向右移位生成 +126。

右移保留符号位。当带符号的整数向右移位时，最高有效位将保留。当无符号的整数右移位时，将清除最高有效位。

## 请参阅

左移和右移运算符（>> 和 <<）

# C 关系和相等运算符

2021/8/17 •

二元关系运算符和相等运算符将其第一个操作数与其第二个操作数进行比较以测试指定关系的有效性。如果测试的关系为 true，则关系表达式的结果为 1；如果测试的关系为 false，则关系表达式的结果为 0。结果的类型为 `int`。

**语法**

*relational-expression*:
  *shift-expression*
  *relational-expression* < *shift-expression*
  *relational-expression* > *shift-expression*
  *relational-expression* <= *shift-expression*
  *relational-expression* >= *shift-expression*

*equality-expression*:
  *relational-expression*
  *equality-expression* == *relational-expression*
  *equality-expression* != *relational-expression*

关系运算符和相等运算符测试以下关系：

| ttt | ttttt |
| --- | --- |
| < | 第一个操作数小于第二个操作数 |
| > | 第一个操作数大于第二个操作数 |
| <= | 第一个操作数小于或等于第二个操作数 |
| >= | 第一个操作数大于或等于第二个操作数 |
| == | 第一个操作数等于第二个操作数 |
| != | 第一个操作数不等于第二个操作数 |

上面的列表中的前四个运算符的优先级高于相等运算符（`==` 和 `!=`）的优先级。请参阅表 C 运算符的优先级和关联性中的优先级信息。

操作数可以具有整型、浮点型或指针类型。操作数的类型可以不同。关系运算符对整型和浮点型的操作数执行常用算术转换。此外，您可以使用以下操作数类型与关系运算符和相等运算符的组合：

- 任意关系运算符或相等运算符的操作数可以是指向同一类型的指针。对于相等（`==`）运算符和不相等（`!=`）运算符，比较的结果指示两个指针是否对相同的内存位置寻址。对于其他关系运算符（<**、>、<=、and **>=），比较的结果指明了所指向的对象的两个内存地址的相对位置。关系运算符仅比较偏移量。

  仅为同一个对象的部分定义指针比较。如果指针引用数组的成员，则比较与相应下标的比较是等效的。第一个数组元素的地址"少于"最后一个元素的地址。对于结构，指向稍后声明的结构成员的指针"大于"指向之前在结构中声明的成员的指针。指向同一联合的成员的指针是相等的。

- 指针值可以与常量值 0 进行比较以确定相等（`==`）或不相等（`!=`）。值为 0 的指针称为"null"指针；即，它

不指向有效的内存位置。

- 相等运算符遵循与关系运算符相同的规则，但允许执行更多可能的运算，即可以将指针与值为 0 的常数整型表达式或指向 `void` 的指针进行比较。如果两个指针都是 null 指针，则它们的比较结果相等。相等运算符比较段和偏移量。

## 示例

下面的示例阐释了关系运算符和相等运算符。

```
int x = 0, y = 0;
if ( x < y )
```

由于 `x` 和 `y` 相等，因此该示例中的表达式会生成值 0。

```
char array[10];
char *p;

for ( p = array; p < &array[10]; p++ )
    *p = '\0';
```

此示例中的片段将 `array` 的每个元素设置为 null 字符常量。

```
enum color { red, white, green } col;
   .
   .
   .
   if ( col == red )
   .
   .
   .
```

这些语句声明带标记 `col` 的名为 `color` 的枚举变量。在任何时候，变量可以包含整数值 0、1 或 2，这表示枚举集 `color` 的某个元素：分别为红色、白色或绿色。如果在执行 `if` 语句时 `col` 包含 0，则任何依赖 `if` 的语句都将被执行。

## 请参阅

关系运算符：<, >、<=, and >=
相等运算符：== 和 !=

# C 按位运算符

2021/8/11 •

按位运算符执行按位“与”(& )、按位“异或”(^ ) 和按位“与或”(| ) 运算。

## 语法

*AND-expression*: *equality-expression* *AND-expression* **&** *equality-expression*

*exclusive-OR-expression*: *AND-expression* *exclusive-OR-expression* **^** *AND-expression*

*inclusive-OR-expression*: *exclusive-OR-expression* *inclusive-OR-expression* **|** *exclusive-OR-expression*

按位运算符的操作数必须具有整数类型，但其类型会不同。这些运算符执行常用算术转换；结果的类型是转换后操作数的类型。

C 按位运算符如下所述：

| | |
|---|---|
| | |
| & | 按位“与”运算符将其第一操作数的每个位与其第二操作数的相应位进行比较。如果两个位均为 1，则对应的结果位将设置为 1。否则，将对应的结果位设置为 0。 |
| ^ | 按位“异或”运算符将其第一操作数的每个位与其第二操作数的相应位进行比较。如果一个位是 0，另一个位是 1，则相应的结果位将设置为 1。否则，将对应的结果位设置为 0。 |
| \| | 按位“与或”运算符将其第一操作数的每个位与第二操作数的相应位进行比较。如果其中一个位是 1，则将对应的结果位设置为 1。否则，将对应的结果位设置为 0。 |

## 示例

这些声明用于以下三个示例：

```
short i = 0xAB00;
short j = 0xABCD;
short n;

n = i & j;
```

第一个示例中的分配给 `n` 的结果与 `i` 相同(0xAB00 十六进制)。

```
n = i | j;

n = i ^ j;
```

第二个示例中的按位“与或”生成值 0xABCD(十六进制)，而第三个示例中的按位“异或”生成 0xCD(十六进制)。

### Microsoft 专用

根据 ANSI C 标准，对有符号整数进行的按位运算的结果是实现定义的。对于 Microsoft C 编译器，对有符号整数

进行的按位运算与对无符号整数进行的按位运算的工作原理相同。例如，`-16 & 99` 可用二进制格式表示

```
  11111111 11110000
& 00000000 01100011
  ________________
  00000000 01100000
```

按位 AND 的结果为 96（十进制）。

结束 Microsoft 专用

## 请参阅

按位 AND 运算符：(&)
按位异或运算符：^
按位“与或”运算符：|

# C 逻辑运算符

2021/8/16 •

逻辑运算符执行 logical-AND (&& ) 和 logical-OR ||( ) 运算。

## 语法

*logical-AND-expression*:
  inclusive-OR-expression
  *logical-AND-expression* **&&** *inclusive-OR-expression*

*logical-OR-expression*:
  logical-AND-expression
  *logical-OR-expression* **||** *logical-AND-expression*

## 备注

逻辑运算符不执行常用算术转换。相反，它们根据其等效性为 0 计算每个操作数。逻辑运算的结果不是 0 就是 1。结果的类型为 `int`。

C 逻辑运算符如下所述：

| ||| | ||
|---|---|
| && | 如果两个操作数具有非零值，则逻辑“与”运算符产生值 1。如果其中一个操作数等于 0，则结果为 0。如果逻辑“与”运算的第一个操作数等于 0，则不会计算第二个操作数。 |
| \|\| | 逻辑“或”运算符对其操作数执行“与或”运算。如果两个操作数的值均为 0，则结果为 0。如果其中一个操作数具有非零值，则结果为 1。如果逻辑“或”运算的第一个操作数具有非零值，则不会计算第二个操作数。 |

逻辑“与”和逻辑“或”表达式的操作数从左到右进行计算。如果第一个操作数的值足以确定运算的结果，则不会计算第二个操作数。这称作“短路计算”。第一个操作数后有一个序列点。有关详细信息，请参阅序列点。

## 示例

下面的示例演示了逻辑运算符：

```
int w, x, y, z;

if ( x < y && y < z )
    printf( "x is less than z\n" );
```

在此示例中，如果 `x` 小于 `y` 且 `y` 小于 `z`，则调用 printf 函数以输出消息。如果 `x` 大于 `y`，则不会计算第二个操作数 ( `y < z` ) 且不会输出任何内容。请注意，如果第二个操作数具有由于某个其他原因而产生的副作用，这可能会导致出现问题。

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

在此示例中，如果 `x` 与 `w`、`y` 或 `z` 相等，则 printf 函数的第二个参数的计算结果将为 true，并输出值 1。否则，它的计算结果将为 false，并打印值 0。只要其中一个条件的计算结果为 true，计算便会停止。

## 请参阅

- 逻辑 AND 运算符：&&
- 逻辑“或”运算符：||

# 条件表达式运算符

2021/8/12 •

C 具有一个三元运算符：conditional-expression 运算符 (? :)。

## 语法

*conditional-expression*:
  *logical-OR-expression*
  *logical-OR expression* ? *expression* : *conditional-expression*

logical-OR-expression 必须具有整型类型、浮点型或指针类型。根据其等效性，其计算结果为 0。序列点紧跟 logical-OR-expression 。操作数的计算将继续，如下所示：

- 如果 logical-OR-expression 不等于 0，则计算 expression 。由非终止符 expression 给定的表达式的计算结果。（这意味着，仅当 logical-OR-expression 为 true 时计算 expression 。）
- 如果 logical-OR-expression 等于 0，则计算 conditional-expression 。该表达式的结果是 conditional-expression 的值。（这意味着，仅当 logical-OR-expression 为 false 时计算 conditional-expression 。）

请注意，计算 expression 或 conditional-expression ，但不同时计算二者。

条件运算的结果类型取决于 expression 或 conditional-expression 操作数的类型，如下所示：

- 如果 expression 或 conditional-expression 具有整型类型或浮点型（其类型可不同），则运算符执行常用算术转换。结果的类型是转换后操作数的类型。
- 如果 expression 和 conditional-expression 都具有相同的结构、联合或指针类型，则结果的类型为相同的结构、联合或指针类型。
- 如果两个操作数的类型都是 `void` ，则结果的类型为 `void` 。
- 如果其中一个操作数是指向任何类型的对象的指针，而另一个操作数是指向 `void` 的指针，则指向对象的指针会被转换为指向 `void` 的指针，结果是指向 `void` 的指针。
- 如果 expression 或 conditional-expression 是指针，且另一个操作数是具有值 0 的常量表达式，则结果的类型为指针类型。

在指针的类型比较中，指针所指向的类型中的任何类型限定符（ `const` 或 `volatile` ）都是不重要的，但结果类型从条件的两个组件继承了限定符。

## 示例

下面的示例演示条件运算符的用法：

```
j = ( i < 0 ) ? ( -i ) : ( i );
```

此示例将 `i` 的绝对值赋给 `j` 。如果 `i` 小于 0，则将 `-i` 赋给 `j` 。如果 `i` 大于或等于 0，则将 `i` 赋给 `j` 。

```
void f1( void );
void f2( void );
int x;
int y;
    .
    .
    .
( x == y ) ? ( f1() ) : ( f2() );
```

在此示例中，声明两个函数（`f1` 和 `f2`）和两个变量（`x` 和 `y`）。如果两个变量具有相同的值，则稍后在程序中将调用函数 `f1`。否则，将调用 `f2`。

## 请参阅

条件运算符 :?:

# C 赋值运算符

2021/8/16 •

赋值操作将右侧操作数的值分配给左侧操作数命名的存储位置。因此，赋值操作的左侧操作数必须是一个可修改的左值。在赋值后，赋值表达式具有左操作数的值，但不是左值。

## 语法

`assignment-expression`：

`conditional-expression`

`unary-expression` `assignment-operator` `assignment-expression`

`assignment-operator`：以下之一

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

C 中的赋值运算符可以在单个操作中转换值和赋值。C 提供了以下赋值运算符：

| | |
|---|---|
| `=` | 简单赋值 |
| `*=` | 乘法赋值 |
| `/=` | 除法赋值 |
| `%=` | 余数赋值 |
| `+=` | 加法赋值 |
| `-=` | 减法赋值 |
| `<<=` | 左移赋值 |
| `>>=` | 右移赋值 |
| `&=` | 按位“与”赋值 |
| `^=` | 按位“异或”赋值 |
| `\|=` | 按位“与或”赋值 |

在赋值中，右侧值的类型将转换为左侧值的类型，在完成赋值后，该值将存储在左操作数中。左操作数不得为数组、函数或常量。类型转换中详细介绍了依赖两个类型的特定转换路径。

## 请参阅

- 赋值运算符

# 简单赋值 (C)

2021/8/15 •

简单赋值运算符可将其右操作数赋给其左操作数。右操作数的值将转换为赋值表达式的类型，并替换存储在左侧操作数指定的对象中的值。用于赋值的转换规则适用(请参阅赋值转换)。

```
double x;
int y;

x = y;
```

在此示例中，`y` 的值被转换为类型 `double` 并赋给 `x`。

## 请参阅

C 赋值运算符

# C 复合赋值

2021/8/14 •

复合赋值运算符将简单赋值运算符与另一个二元运算符相结合。复合赋值运算符执行其他运算符指定的运算，然后将结果赋给左操作数。例如，一个复合赋值表达式，如

> *expression1* += *expression2*

可以理解为

> *expression1* = *expression1* + *expression2*

但是，复合赋值表达式不等于扩展版本，因为复合赋值表达式只计算 *expression1* 一次，而扩展版本将计算 *expression1* 两次：在加法运算和赋值运算中。

复合赋值运算符的操作数必须为整型或浮点型。每个复合赋值运算符都将执行对应的二元运算符所执行的转换并相应地限制其操作数的类型。加法赋值 ( `+=` ) 和减法赋值 ( `-=` ) 运算符还可以具有指针类型的左操作数，在此情况下，右操作数必须为整型类型。复合赋值运算的结果具有左操作数的值和类型。

```
#define MASK 0xff00

n &= MASK;
```

在此示例中，对 `n` 和 `MASK` 执行了按位“与”运算，并将结果赋给了 `n` 。使用 #define 预处理器指令定义了清单常量 `MASK` 。

## 请参阅

C 赋值运算符

# 顺序评估运算符

2021/8/13 •

顺序计算运算符(也称为"逗号运算符")按从左到右的顺序计算其两个操作数。

## 语法

*expression*:
  *assignment-expression*
  *expression* , *assignment-expression*

顺序求值运算符的左操作数作为 `void` 表达式进行求值。该运算的结果与右操作数具有相同的值和类型。所有操作数都可以是任意类型。顺序计算运算符在其操作数之间不执行类型转换,也不产生左值。第一个操作数后有一个序列点,这意味着来自左操作数的计算的所有副作用在开始右操作数的计算前已完成。有关详细信息,请参阅序列点。

顺序计算运算符通常用于计算只允许有一个表达式的上下文中的两个或多个表达式。

逗号可以在上下文中用作分隔符。但是,您务必小心,不要混淆将逗号用作分隔符与将逗号用作运算符;这两种用法完全不同。

## 示例

以下示例演示顺序计算运算符:

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

在此示例中, `for` 语句的第三个表达式的每个操作数都是独立进行求值的。首先计算左操作数 `i += i` ,然后计算右操作数 `j--` 。

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

在对 `func_one` 的函数调用中,将传递以逗号分隔的三个参数: `x` 、 `y + 2` 和 `z` 。在对 `func_two` 的函数调用中,圆括号强制编译器将第一个逗号解释为顺序计算运算符。此函数调用将两个参数传递给 `func_two` 。第一个参数是顺序计算运算 `(x--, y + 2)` 的结果,具有表达式 `y + 2` 的值和类型;第二个参数为 `z` 。

## 请参阅

逗号运算符: ,

# 类型转换 (C)

2021/8/12 •

类型转换取决于指定的运算符以及操作数或运算符的类型。下列情况下将执行类型转换：

- 当将一个类型的值赋给其他类型的变量或运算符在执行运算前转换了其一个或多个操作数的类型时
- 当一个类型的值显式强制转换为其他类型时
- 当值作为参数传递给函数时，或当类型从函数返回时

字符、短整数或整数位域（无论带符号还是无符号）或枚举类型的对象均可在可使用整数的表达式中使用。如果 `int` 可表示原始类型的所有值，则值转换为 `int`；否则，值转换为 `unsigned int`。此过程称为“整型提升”。整型提升将保留值。即，保证提升后的值与提升前一样。有关详细信息，请参阅常用算术转换。

## 请参阅

表达式和赋值

# 赋值转换

2021/8/15 •

在赋值操作中，赋值的类型将转换为接受赋值的变量的类型。即使转换过程中会丢失信息，C 仍然允许在整型和浮点类型之间进行赋值转换。使用的转换方法取决于赋值涉及的类型，如常用算术转换和下列各节中所述：

- 从带符号整型的转换
- 从无符号整型的转换
- 从浮点类型的转换
- 指针类型之间的转换
- 从其他类型的转换

虽然不能在赋值的左侧使用 `const` 左值，但类型限定符不会影响转换的可允许性。

## 请参阅

类型转换

# 从带符号整型的转换

2021/8/16 •

当有符号整数转换为整数或浮点类型时，如果原始值可以在结果类型中表示，则该值不变。

当有符号整数转换为更大的整数时，该值将被符号扩展。当转换成较小的整数时，高序位将被截断。使用结果类型解释结果，如本例所示：

```
int i = -3;
unsigned short u;

u = i;
printf_s( "%hu\n", u );  // Prints 65533
```

将带符号整数转换为浮点类型时，如果原始值不能在结果类型中完全表示，则结果是下一个更高或更低的可表示值。

有关整型和浮点类型大小的信息，请参阅基本类型的存储。

下表汇总了来自带符号整型的转换。它假定 `char` 类型在默认情况下是带符号的。如果你使用编译时选项将 `char` 类型更改为在默认情况下是不带符号的，则应用的是 `unsigned char` 类型的从不带符号的整型类型转换表中给定的转换，而不是此表中的转换。

**Microsoft 专用**

在 Microsoft 编译器中，`int` 和 `long` 是不同但等效的类型。`int` 值与 `long` 的转换方式是一样的。

**结束 Microsoft 专用**

## 从带符号整型转换的表

| FROM | " | " |
|---|---|---|
| `char` [1] | `short` | 符号扩展 |
| `char` | `long` | 符号扩展 |
| `char` | `long long` | 符号扩展 |
| `char` | `unsigned char` | 保留模式；高序位失去符号位的函数 |
| `char` | `unsigned short` | 符号扩展为 `short`；将 `short` 转换为 `unsigned short` |
| `char` | `unsigned long` | 符号扩展为 `long`；将 `long` 转换为 `unsigned long` |
| `char` | `unsigned long long` | 符号扩展为 `long long`；将 `long long` 转换为 `unsigned long long` |

| FROM | 〃 | 〃 |
| --- | --- | --- |
| `char` | `float` | 符号扩展为 `long` ;将 `long` 转换为 `float` |
| `char` | `double` | 符号扩展为 `long` ;将 `long` 转换为 `double` |
| `char` | `long double` | 符号扩展为 `long` ;将 `long` 转换为 `double` |
| `short` | `char` | 保留低位字节 |
| `short` | `long` | 符号扩展 |
| `short` | `long long` | 符号扩展 |
| `short` | `unsigned char` | 保留低位字节 |
| `short` | `unsigned short` | 保留位模式;高序位丢失符号位的函数 |
| `short` | `unsigned long` | 符号扩展为 `long` ;将 `long` 转换为 `unsigned long` |
| `short` | `unsigned long long` | 符号扩展为 `long long` ;将 `long long` 转换为 `unsigned long long` |
| `short` | `float` | 符号扩展为 `long` ;将 `long` 转换为 `float` |
| `short` | `double` | 符号扩展为 `long` ;将 `long` 转换为 `double` |
| `short` | `long double` | 符号扩展为 `long` ;将 `long` 转换为 `double` |
| `long` | `char` | 保留低位字节 |
| `long` | `short` | 保留低位字 |
| `long` | `long long` | 符号扩展 |
| `long` | `unsigned char` | 保留低位字节 |
| `long` | `unsigned short` | 保留低位字 |
| `long` | `unsigned long` | 保留位模式;高序位丢失符号位的函数 |
| `long` | `unsigned long long` | 符号扩展为 `long long` ;将 `long long` 转换为 `unsigned long long` |

| FROM | 〃 | 〃 |
| --- | --- | --- |
| `long` | `float` | 表示为 `float`。如果无法精确表示 `long`，就会丢失一些精度。 |
| `long` | `double` | 表示为 `double`。如果无法将 `long` 精确表示为 `double`，就会丢失一些精度。 |
| `long` | `long double` | 表示为 `double`。如果无法将 `long` 精确表示为 `double`，就会丢失一些精度。 |
| `long long` | `char` | 保留低位字节 |
| `long long` | `short` | 保留低位字 |
| `long long` | `long` | 保留低位双字节 |
| `long long` | `unsigned char` | 保留低位字节 |
| `long long` | `unsigned short` | 保留低位字 |
| `long long` | `unsigned long` | 保留低位双字节 |
| `long long` | `unsigned long long` | 保留位模式；高序位丢失符号位的函数 |
| `long long` | `float` | 表示为 `float`。如果无法精确表示 `long long`，就会丢失一些精度。 |
| `long long` | `double` | 表示为 `double`。如果无法将 `long long` 精确表示为 `double`，就会丢失一些精度。 |
| `long long` | `long double` | 表示为 `double`。如果无法将 `long long` 精确表示为 `double`，就会丢失一些精度。 |

[1]所有 `char` 条目假定 `char` 类型在默认情况下是带符号的。

## 请参阅

赋值转换

# 从无符号整型的转换

2021/8/14 •

当无符号整数转换为整数或浮点类型时，如果原始值可以在结果类型中表示，则该值不变。

将无符号整数转换为更大的整数时，该值扩展为零。当转换成较小的整数时，高序位将被截断。使用结果类型解释结果，如本例所示。

```
unsigned k = 65533;
short j;

j = k;
printf_s( "%hd\n", j );   // Prints -3
```

将无符号整数转换为浮点类型时，如果原始值不能在结果类型中完全表示，则结果是下一个更高或更低的可表示值。

有关整型和浮点类型大小的信息，请参阅基本类型的存储。

### Microsoft 专用

在 Microsoft 编译器中，`unsigned`（或 `unsigned int`）和 `unsigned long` 是不同但等效的类型。`unsigned int` 值与 `unsigned long` 的转换方式是一样的。

### 结束 Microsoft 专用

下表汇总了来自无符号整型的转换。

## 从无符号整型转换的表

| FROM | “ | “ |
|---|---|---|
| `unsigned char` | `char` | 保留位模式；高序位将成为符号位 |
| `unsigned char` | `short` | 零扩展 |
| `unsigned char` | `long` | 零扩展 |
| `unsigned char` | `long long` | 零扩展 |
| `unsigned char` | `unsigned short` | 零扩展 |
| `unsigned char` | `unsigned long` | 零扩展 |
| `unsigned char` | `unsigned long long` | 零扩展 |
| `unsigned char` | `float` | 转换为 `long`；将 `long` 转换为 `float` |
| `unsigned char` | `double` | 转换为 `long`；将 `long` 转换为 `double` |

| FROM | 以 | 以 |
| --- | --- | --- |
| `unsigned char` | `long double` | 转换为 `long`；将 `long` 转换为 `double` |
| `unsigned short` | `char` | 保留低位字节 |
| `unsigned short` | `short` | 保留位模式；高序位将成为符号位 |
| `unsigned short` | `long` | 零扩展 |
| `unsigned short` | `long long` | 零扩展 |
| `unsigned short` | `unsigned char` | 保留低位字节 |
| `unsigned short` | `unsigned long` | 零扩展 |
| `unsigned short` | `unsigned long long` | 零扩展 |
| `unsigned short` | `float` | 转换为 `long`；将 `long` 转换为 `float` |
| `unsigned short` | `double` | 转换为 `long`；将 `long` 转换为 `double` |
| `unsigned short` | `long double` | 转换为 `long`；将 `long` 转换为 `double` |
| `unsigned long` | `char` | 保留低位字节 |
| `unsigned long` | `short` | 保留低位字 |
| `unsigned long` | `long` | 保留位模式；高序位将成为符号位 |
| `unsigned long` | `long long` | 零扩展 |
| `unsigned long` | `unsigned char` | 保留低位字节 |
| `unsigned long` | `unsigned short` | 保留低位字 |
| `unsigned long` | `unsigned long long` | 零扩展 |
| `unsigned long` | `float` | 转换为 `long`；将 `long` 转换为 `float` |
| `unsigned long` | `double` | 直接转换为 `double` |
| `unsigned long` | `long double` | 转换为 `long`；将 `long` 转换为 `double` |
| `unsigned long long` | `char` | 保留低位字节 |

| FROM | 至 | 至 |
| --- | --- | --- |
| `unsigned long long` | `short` | **保留低位字** |
| `unsigned long long` | `long` | **保留低位双字**节 |
| `unsigned long long` | `long long` | **保留位模式**;**高序位将成**为**符号位** |
| `unsigned long long` | `unsigned char` | **保留低位字**节 |
| `unsigned long long` | `unsigned short` | **保留低位字** |
| `unsigned long long` | `unsigned long` | **保留低位双字**节 |
| `unsigned long long` | `float` | 转换为 `long` ;**将** `long` 转换为 `float` |
| `unsigned long long` | `double` | **直接**转换为 `double` |
| `unsigned long long` | `long double` | 转换为 `long` ;**将** `long` 转换为 `double` |

## 请参阅

赋值转换

# 从浮点类型的转换

2021/8/13 •

如果原始值在结果类型中是可精确表示的，则转换为另一种浮点类型的浮点值的值不会发生任何变化。如果原始值是数值，但不能精确地表示，则结果是下一个更大的可表示值或下一个更小的可表示值。有关浮点类型的范围，请参阅浮点常量的限制。

首先通过丢弃任何小数值来截断转换为整型类型的浮点值。如果这个截断的值在结果类型中是可表示的，那么结果必须是该值。如果结果值不可表示，结果值是未定义的。

### Microsoft 专用

Microsoft 编译器对 `float` 值使用 IEEE-754 binary32 表示，并对 `long double` 和 `double` 使用 binary64 表示。由于 `long double` 和 `double` 使用相同的表示，因此它们有相同的范围和精度。

当编译器将 `double` 或 `long double` 浮点数转换为 `float` 时，它根据浮点环境控制（默认为"舍入到最近值，但绑定到偶数"）对结果进行舍入。如果数值太大或太小而无法表示为数值 `float`，转换结果为正无穷或负无穷（具体视原始值的符号而定），并抛出溢出异常（如果启用的话）。

在转换为整型类型时，转换为小于 `long` 的类型的结果是将值转换为 `long`，然后转换为结果类型的结果。

对于转换为至少与 `long` 一样大的整型类型，如果转换的值太大或太小而无法在结果类型中表示，则可能会返回以下任何值：

- 结果可能是一个标记值，它是离零最远的可表示值 。对于有符号类型，它是最小的可表示值 (0x800...0)。对于无符号类型，它是最大的可表示值 (0xFF...F)。
- 结果可能是饱和值，其中过大的值被转换为最大的可表示值，过小的值被转换为最小的可表示值 。这两个值中的一个也用作标记值。
- 对于转换为 `unsigned long` 或 `unsigned long long`，转换超出范围的值的结果可能是除最高或最低可表示值之外的其他值。结果是标记值还是饱和值取决于编译器选项和目标体系结构。将来的编译器版本可能会返回一个饱和值或标记值。

### 结束 Microsoft 专用

下表汇总了来自浮点型的转换。

## 浮点类型转换表

| FROM | “ | “ |
|---|---|---|
| `float` | `char` | 转换为 `long`；将 `long` 转换为 `char` |
| `float` | `short` | 转换为 `long`；将 `long` 转换为 `short` |
| `float` | `int` | 在小数点处截断。如果结果太大而无法表示为 `int`，则结果是未定义的。 |
| `float` | `long` | 在小数点处截断。如果结果太大而无法表示为 `long`，则结果是未定义的。 |

| FROM | 〃 | 〃 |
| --- | --- | --- |
| `float` | `long long` | 在小数点处截断。如果结果太大而无法表示为 `long long`，则结果是未定义的。 |
| `float` | `unsigned char` | 转换为 `long`；将 `long` 转换为 `unsigned char` |
| `float` | `unsigned short` | 转换为 `long`；将 `long` 转换为 `unsigned short` |
| `float` | `unsigned` | 在小数点处截断。如果结果太大而无法表示为 `unsigned`，则结果是未定义的。 |
| `float` | `unsigned long` | 在小数点处截断。如果结果太大而无法表示为 `unsigned long`，则结果是未定义的。 |
| `float` | `unsigned long long` | 在小数点处截断。如果结果太大而无法表示为 `unsigned long long`，则结果是未定义的。 |
| `float` | `double` | 表示为 `double`。 |
| `float` | `long double` | 表示为 `long double`。 |
| `double` | `char` | 转换为 `float`；将 `float` 转换为 `char` |
| `double` | `short` | 转换为 `float`；将 `float` 转换为 `short` |
| `double` | `int` | 在小数点处截断。如果结果太大而无法表示为 `int`，则结果是未定义的。 |
| `double` | `long` | 在小数点处截断。如果结果太大而无法表示为 `long`，则结果是未定义的。 |
| `double` | `unsigned char` | 转换为 `long`；将 `long` 转换为 `unsigned char` |
| `double` | `unsigned short` | 转换为 `long`；将 `long` 转换为 `unsigned short` |
| `double` | `unsigned` | 在小数点处截断。如果结果太大而无法表示为 `unsigned`，则结果是未定义的。 |
| `double` | `unsigned long` | 在小数点处截断。如果结果太大而无法表示为 `unsigned long`，则结果是未定义的。 |

| FROM | TO | TO |
| --- | --- | --- |
| `double` | `unsigned long long` | 在小数点处截断。如果结果太大而无法表示为 `unsigned long long`，则结果是未定义的。 |
| `double` | `float` | 表示为 `float`。如果 `double` 值无法精确表示为 `float`，则会丢失精度。如果值太大而无法表示为 `float`，则结果是未定义的。 |
| `double` | `long double` | 将 `long double` 值视为 `double`。 |

从 `long double` 的转换与从 `double` 的转换遵循相同的方法。

## 请参阅

赋值转换

# 指针类型之间的转换

2021/8/13 •

指向值的一个类型的指针可以转换为指向另一类型的指针。但是，由于对齐需求和存储中不同类型的大小，结果可能是未定义的。指向对象的指针可转换为指向其类型要求小于或等于严格存储对齐的对象的指针，然后再次返回而不做更改。

指向 `void` 的指针可以与指向任何类型的指针之间来回转换，既不受限制，也不会丢失信息。如果结果转换回原始类型，则将恢复原始指针。

如果指针转换为另一个类型相同但具有不同的或其它限定符的指针，则新指针与旧指针相同（新限定符强加的限制除外）。

指针值也可以转换为整数值。根据以下规则，转换路径取决于指针的大小和整型的大小：

- 如果指针的大小大于或等于整型的大小，则指针的行为类似于转换中的无符号值，除非它无法转换为浮点值。
- 如果指针小于整型，则指针首先转换为与整型大小相同的指针，然后转换为整型。

相反，整型可以基于以下规则转换为指针类型：

- 如果整型与指针类型的大小相同，则转换只会促使整数值被视为指针（无符号整数）。
- 如果整型类型的大小与指针类型的大小不同，则使用表从带符号整型类型转换和从无符号整型类型转换中给定的转换路径，首先将整型转换为指针的大小。然后将其视为一个指针值。

值为 0 的整型常数表达式或强制转换为类型 `void` * 的此类表达式可以通过类型强制转换、赋值或与任何类型的指针进行比较来进行转换。这将产生与同一类型的另一个 null 指针相等的 null 指针，但此 null 指针与指向函数或对象的任何指针不相等。常数 0 以外的整数可以转换为指针类型，但结果是不可移植的。

## 请参阅

赋值转换

# 从其他类型的转换

2021/8/13 •

由于根据定义 `enum` 值是 `int` 值，因此与 `enum` 值之间的来回转换和 `int` 类型的转换是相同的。对于 Microsoft C 编译器，整数与 `long` 相同。

**Microsoft 专用**

结构或联合类型之间不允许转换。

任何值都可以转换为类型 `void`，但此类转换的结果只能在放弃表达式值的上下文中（如在表达式语句中）使用。

根据定义，`void` 类型没有值。因此，它不能转换为其他任何类型，而其他类型也不能通过赋值转换为 `void`。不过，可以显式地将值强制转换为类型 `void`，如类型强制转换中所述。

**结束 Microsoft 专用**

## 请参阅

赋值转换

# 类型强制转换的转换

2021/8/11 •

可以使用类型强制转换来显式转换类型。

**语法**

*cast-expression*:
一元表达式
( *type-name* ) *cast-expression*

*type-name*:
*specifier-qualifier-list abstract-declarator*$_{opt}$

type-name 是类型，cast-expression 是要转换为该类型的值。具有类型强制转换的表达式不是左值。cast-expression 也会被转换，就好像它已分配到 type-name 类型的变量一样。赋值的转换规则(在赋值转换中进行了概述)也适用于类型强制转换。下表显示了可强制转换为任何给定类型的类型。

**合法类型强制转换**

| | |
|---|---|
| | |
| 整型 | 整数类型或浮点类型，或者指向对象的指针 |
| 浮点 | 任何算术类型 |
| 指向对象的指针或 ( `void` *) | 任何整型类型、( `void` *)、指向对象的指针或函数指针 |
| 函数指针 | 任何整数类型、指向对象的指针或函数指针 |
| 结构、联合或数组 | None |
| Void 类型 | 任何类型 |

任何标识符都可以强制转换为 `void` 类型。不过，如果在类型强制转换表达式中指定的类型不是 `void`，那么要强制转换为此类型的标识符就不能是 `void` 表达式。任何表达式都可以强制转换为 `void`，但类型为 `void` 的表达式无法强制转换为其他任何类型。例如，包含 `void` 返回类型的函数无法将其返回值强制转换为另一类型。

请注意，`void` * 表达式有指向 `void` 的类型指针，而没有类型 `void`。如果对象被强制转换为 `void` 类型，则无法将生成的表达式分配给任何项。同样，type-cast 对象是不可接受的左值，因此不能对 type-cast 对象进行任何分配。

**Microsoft 专用**

只要标识符的大小不变，类型强制转换就可以是左值表达式。有关左值表达式的信息，请参阅左值和右值表达式。

**结束 Microsoft 专用**

可以通过强制转换将表达式转换为类型 `void`，但生成的表达式只能在不需要值的情况下使用。转换为 `void` * 再转换回原始类型的对象指针将恢复其原始值。

## 请参阅

类型转换

# 函数调用转换

2021/8/12 •

对函数调用中的自变量执行的转换的类型取决于是否存在具有调用的函数的声明自变量类型的函数原型(前向声明)。

如果函数原型存在并包含声明的参数类型，编译器将执行类型检查(请参阅函数)。

如果函数原型不存在，则只对函数调用中的自变量执行常用算术转换。这些转换独立于调用中的每个自变量执行。这意味着，`float` 值被转换为 `double`；`char` 或 `short` 值被转换为 `int`；且 `unsigned char` 或 `unsigned short` 被转换为 `unsigned int`。

## 请参阅

类型转换

# 语句 (C)

2021/8/12 •

C 程序的语句控制程序执行流。在 C 中就像在其他编程语言中一样，有多种语句可用于执行循环、选择要执行的其他语句和传输控制。在简短的语句语法概述之后，本节将按字母顺序介绍 C 语句：

break 语句
复合语句
continue 语句
do-while 语句
表达式语句

for 语句
goto 和标记语句
if 语句
null 语句
return 语句

switch 语句
try-except 语句
try-finally 语句
while 语句

## 请参阅

C 语言参考

# C 语句概述

2021/8/13 •

C 语句由标记、表达式和其他语句组成。构成另一个语句的组成部分的语句称为封闭语句的“体”。本节中将讨论以下语法给定的每个语句类型。

## 语法

*statement*: labeled-statement

compound-statement

expression-statement

selection-statement

iteration-statement

jump-statement

try-except-statement /* Microsoft-specific */

try-finally-statement /* Microsoft-specific */

通常，语句体为“复合语句”。复合语句由可包含关键字的其他语句组成。复合语句由大括号 ({ } ) 分隔。所有其他 C 语句以分号 (; ) 结尾。分号是语句结束符。

表达式语句包含可包含表达式和赋值中介绍的算术或逻辑运算符的 C 表达式。null 语句是空语句。

所有 C 语句都可以以由名称和冒号组成的标识标签开头。由于只有 `goto` 语句才能识别语句标签，因此将用 `goto` 介绍语句标签。有关详细信息，请参阅 goto 和标记语句。

## 请参阅

语句

# break 语句 (C)

2021/8/13 •

`break` 语句终止执行出现它的最近的封闭 `do`、`for`、`switch` 或 `while` 语句。控制权将传递给已终止语句后面的语句。

## 语法

*jump-statement*:
  **break ;**

`break` 语句经常用于终止 `switch` 语句中对特定情况的处理。缺少封闭的迭代或 `switch` 语句会生成错误。

在嵌套语句中，`break` 语句只终止直接围住它的 `do`、`for`、`switch` 或 `while` 语句。可以使用 `return` 或 `goto` 语句将控制权从嵌套结构转移出到其他地方。

下面的示例展示了 `break` 语句：

```
#include <stdio.h>
int main() {
   char c;
   for(;;) {
      printf_s( "\nPress any key, Q to quit: " );

      // Convert to character value
      scanf_s("%c", &c);
      if (c == 'Q')
          break;
   }
} // Loop exits only when 'Q' is pressed
```

## 请参阅

break 语句

# 复合语句 (C)

2021/8/15 •

复合语句(亦称为"块")通常作为另一条语句(如 `if` 语句)的主体出现。声明和类型描述可在复合语句的头部出现的声明的格式和含义。

## 语法

compound-statement：
　{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list：
　*declaration*
　*declaration-list declaration*

statement-list：
　*statement*
　*statement-list statement*

如果有声明，则它们必须在任何语句之前出现。在复合语句开头声明的每个标识符的范围从其声明点扩展到块的末尾。它在整个块中都可见，除非内部块中存在对同一标识符的声明。

复合语句中的标识符被假定为 `auto`，除非另外使用 `register`、`static` 或 `extern`(只能是 `extern` 的函数除外)显式声明。可以在函数声明中去掉 `extern` 说明符，但函数仍然是 `extern`。

如果在包含存储类 `extern` 的复合语句中声明变量或函数，则不会分配存储，也不允许初始化。声明引用在其他位置定义的外部变量或函数。

使用 `auto` 或 `register` 关键字在块中声明的变量在每次进入复合语句时重新分配并初始化(如果需要)。在退出复合语句后将不再定义这些变量。如果在块内声明的变量有 `static` 特性，则此变量在程序开始执行时初始化，并在整个程序中保持值不变。若要了解 `static`，请参阅存储类。

此示例演示了一个复合语句：

```
if ( i > 0 )
{
    line[i] = x;
    x++;
    i--;
}
```

在此示例中，如果 `i` 大于 0，则复合语句内的所有语句将按顺序执行。

## 请参阅

语句

# `continue` 语句 (C)

2021/8/13 •

`continue` 语句将控制权传递给出现它的最近的封闭 `do`、`for` 或 `while` 语句的下一个迭代，并绕过 `do`、`for` 或 `while` 语句主体中的任何剩余语句。

## 语法

*`jump-statement`*:
  `continue ;`

`do`、`for` 或 `while` 语句的下一个迭代是按如下方式确定的：

- 在 `do` 或 `while` 语句中，下一个迭代会先重新计算 `do` 或 `while` 语句的表达式。
- `for` 语句中的 `continue` 语句会导致计算 `for` 语句的循环表达式 。然后，代码重新计算条件表达式。根据结果，它会终止或循环访问语句体。若要详细了解 `for` 语句及其非终止符，请查看 `for` 语句。

下面是 `continue` 语句示例：

```
while ( i-- > 0 )
{
    x = f( i );
    if ( x == 1 )
        continue;
    y += x * x;
}
```

在此示例中，当 `i` 大于 0 时，将执行语句主体。首先将 `f(i)` 赋给 `x`；然后，如果 `x` 等于 1，则执行 `continue` 语句。正文中的其余语句会被忽略。在循环的顶部继续执行，并评估循环的其余部分。

## 另请参阅

`continue` 语句 (C++)

# do-while 语句 (C)

2021/8/13 •

利用 do-while 语句，可以重复语句或复合语句，直到指定的表达式的计算结果为 false。

## 语法

iteration-statement: `do` statement while ( expression ) ;

在执行循环体后，将计算 do-while 语句中的 expression 。因此，总是至少执行一次循环体。

expression 必须具有算法或指针类型。执行过程如下所示：

1. 执行语句体。
2. 接着，计算 expression 。如果 expression 为 false，则 do-while 语句将终止，控制权将传递到程序中的下一条语句。如果 expression 为 true（非零），则将从第 1 步开始重复此过程。

当 `break` 、`goto` 或 `return` 语句在语句主体中执行时，do-while 语句也可以终止。

以下是 do-while 语句的示例：

```
do
{
    y = f( x );
    x--;
} while ( x > 0 );
```

在此 do-while 语句中，无论 `x` 的初始值是什么，`y = f( x );` 和 `x--;` 这两个语句都会执行。然后将计算 `x > 0` 。如果 `x` 大于 0，则会再次执行语句体并重新计算 `x > 0` 。只要 `x` 保持大于 0，语句主体就会重复执行。当 `x` 变为 0 或负值时，do-while 语句的执行将终止。将至少执行一次循环体。

## 请参阅

do-while 语句 (C++)

# 表达式语句 (C)

2021/8/13 •

在执行表达式语句时，将根据表达式和赋值中概述的规则来计算表达式。

## 语法

expression-statement：

*expression*$_{opt}$ ;

在执行下一个语句前，完成表达式计算的所有副作用。空表达式语句被称为 null 语句。有关详细信息，请参阅 Null 语句。

这些示例演示了表达式语句。

```
x = ( y + 3 );           /* x is assigned the value of y + 3  */
x++;                     /* x is incremented                  */
x = y = 0;               /* Both x and y are initialized to 0 */
proc( arg1, arg2 );      /* Function call returning void      */
y = z = ( f( x ) + 3 );  /* A function-call expression        */
```

在最后一个语句中，函数调用表达式的值（包括函数返回的任何值）增加 3，然后被赋给变量 `y` 和 `z`。

## 请参阅

语句

# for 语句 (C)

2021/8/12 •

利用 `for` 语句，可以将语句或复合语句重复执行指定的次数。执行 `for` 语句的主体零次或多次，直到可选条件变成 false。可以在 `for` 语句中使用可选表达式，以便在 `for` 语句的执行期间初始化和更改值。

## 语法

*iteration-statement*:
`for` ( init-expression$_{opt}$ ; cond-expression$_{opt}$ ; loop-expression$_{opt}$ ) statement

`for` 语句的执行过程如下：

1. 将计算 *init-expression*（如果有）。这将为循环指定初始化。对 *init-expression* 的类型没有限制。
2. 将计算 *cond-expression*（如果有）。此表达式必须具有算法或指针类型。它在每次迭代前计算。可能有三个结果：
   - 如果 cond-expression 为 `true`（非零），则执行语句，然后计算 loop-expression（若有）。在每次迭代之后，将计算 *loop-expression*。对其类型没有限制。副作用将按顺序执行。该过程随后从计算 *cond-expression* 重新开始。
   - 如果省略了 *cond-expression*，则 *cond-expression* 被视为 true，执行将完全按上一段中所述方式继续。只有在执行了语句主体中的 `break` 或 `return` 语句时，或只有在执行了 `goto`（转到 `for` 语句主体外的带标签的语句）时，没有 cond-expression 参数的 `for` 语句才会终止。
   - 如果 cond-expression 为 `false` (0)，则 `for` 语句的执行终止，并将控制权传递给程序中的下一个语句。

在执行了语句主体中的 `break`、`goto` 或 `return` 语句时，`for` 语句也会终止。`for` 循环中的 `continue` 语句会导致计算 loop-expression。当 `break` 语句在 `for` 循环中执行时，不会计算或执行 loop-expression。以下语句

```
for( ; ; )
```

是生成只能通过 `break`、`goto` 或 `return` 语句退出的无限循环的惯用方法。

## 示例

下面的示例展示了 `for` 语句：

```
// c_for.c
int main()
{
   char* line = "H e  \tl\tlo World\0";
   int space = 0;
   int tab = 0;
   int i;
   int max = strlen(line);
   for (i = 0; i < max; i++ )
   {
      if ( line[i] == ' ' )
      {
          space++;
      }
      if ( line[i] == '\t' )
      {
          tab++;
      }
   }

   printf("Number of spaces: %i\n", space);
   printf("Number of tabs: %i\n", tab);
   return 0;
}
```

## Output

```
Number of spaces: 4
Number of tabs: 2
```

## 请参阅

语句

# goto 和标记语句 (C)

2021/8/16 •

`goto` 语句将控制权转移给标签。给定标签必须位于同一函数中，并且只可以出现在同一函数中的一个语句前面。

## 语法

*statement*:
  *labeled-statement*
  *jump-statement*

*jump-statement*:
  `goto` identifier ;

*labeled-statement*:
  *identifier* : *statement*

语句标签只对 `goto` 语句有意义；在其他任何上下文中，执行带标签的语句时不考虑标签。

jump-statement 必须位于同一函数中，并且只能出现在同一函数中的一个语句前面。`goto` 后面的 identifier 名称集有自己的命名空间，因此名称不会干扰其他标识符。不能重新声明标签。有关详细信息，请参阅命名空间。

尽可能优先使用 `break`、`continue` 和 `return` 语句，而不是 `goto`，这是很好的编程风格。由于 `break` 语句只从循环的一个级别退出，因此从深度嵌套的循环中退出循环可能需要使用 `goto`。

下面的示例展示了 `goto` 语句：

```
// goto.c
#include <stdio.h>

int main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf_s( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 3; j++ )
        {
            printf_s( " Inner loop executing. j = %d\n", j );
            if ( i == 5 )
                goto stop;
        }
    }

    /* This message does not print: */
    printf_s( "Loop exited. i = %d\n", i );

    stop: printf_s( "Jumped to stop. i = %d\n", i );
}
```

在此示例中，当 `i` 等于 5 时，`goto` 语句将控制权转移给标记为 `stop` 的点。

## 请参阅

语句

# if 语句 (C)

2021/8/16 •

`if` 语句控制条件分支。如果表达式的值不为零，则执行 `if` 语句的主体。`if` 语句的语法有两种形式。

## 语法

*selection-statement*: **if (** *expression* **)** *statement*

if ( expression ) statement `else` statement

在 `if` 语句的两种形式中，计算除了结构之外可以有任何值的表达式，包括所有副作用。

在第一种形式的语法中，如果 expression 为 true（非零），则执行 statement 。如果 expression 为 false，则忽略 statement 。在使用 `else` 的第二种语法形式中，如果 expression 为 false，则执行第二个 statement。对于这两种形式，控制权随后从 `if` 语句传递给程序中的下一个语句，除非其中一个语句包含 `break`、`continue` 或 `goto`。

下面的几个示例展示了 `if` 语句：

```
if ( i > 0 )
    y = x / i;
else
{
    x = i;
    y = f( x );
}
```

在此示例中，如果 `y = x/i;` 大于 0，则执行 `i` 语句。如果 `i` 小于或等于 0，则将 `i` 赋给 `x`，并将 `f( x )` 赋给 `y`。请注意，构成 `if` 子句的语句以分号结尾。

嵌套 `if` 语句和 `else` 子句时，请使用大括号将语句和子句组合成复合语句，以阐明你的意图。如果没有大括号，编译器会将每个 `else` 与缺少 `else` 的最近 `if` 关联，从而解决二义性。

```
if ( i > 0 )          /* Without braces */
    if ( j > i )
        x = j;
    else
        x = i;
```

在此示例中，`else` 子句与内部 `if` 语句关联。如果 `i` 小于或等于 0，则不会将任何值赋给 `x`。

```
if ( i > 0 )
{                     /* With braces */
    if ( j > i )
        x = j;
}
else
    x = i;
```

此示例中的内部 `if` 语句两边的大括号让 `else` 子句成为外部 `if` 语句的一部分。如果 `i` 小于或等于 0，则将 `i` 赋给 `x`。

## 请参阅

if-else 语句 (C++)

# Null 语句 (C)

2021/8/15 •

"null 语句"是仅包含分号的语句;它可在需要语句时显示。执行 null 语句时不会发生任何事件。编码 null 语句的正确方式是:

## 语法

```
;
```

## 备注

`do`、`for`、`if` 和 `while` 等语句要求可执行语句作为语句主体出现。在无需实质性语句体的情况下,null 语句可满足语法要求。

与任何其他 C 语句一起使用时,您可在 null 语句前包含一个标签。若要标记某个不是语句的项(如复合语句的右大括号),您可标记一个 null 语句并紧靠该项的前面插入该语句以取得相同的效果。

以下示例阐释了 null 语句:

```
for ( i = 0; i < 10; line[i++] = 0 )
     ;
```

在此示例中,`for` 语句 `line[i++] = 0` 的循环表达式将 `line` 的前 10 个元素初始化为 0。由于无需任何其他语句,因此语句体为 null 语句。

## 请参阅

语句

# return 语句 (C)

2021/8/13 •

`return` 语句会结束函数的执行并返回对调用函数的控制。紧接在调用之后在调用函数中恢复执行。`return` 语句可将值返回给调用函数。有关详细信息，请参阅返回类型。

## 语法

*jump-statement*:
    `return` *expression*$_{opt}$ `;`

如果表达式存在的话，*expression* 的值将返回到调用函数。如果 *expression* 省略，该函数返回值未定义。先计算表达式（如果存在），然后转换为函数返回的类型。如果 `return` 语句在具有 `void` 返回类型的函数中包含表达式，则编译器会生成一个警告，并且不计算该表达式 。

如果函数定义中未出现 `return` 语句，则在执行被调用函数的最后一个语句后，控件自动返回到调用函数。在这种情况下，当调用该函数时，返回值将未定义。如果函数具有 `void` 以外的返回类型，则这是一个严重的 bug，编译器会打印一条警告诊断消息。如果函数具有 `void` 返回类型，则此行为正常，但可能被视为不良样式。请使用纯文本 `return` 语句阐明意图。

一个好的工程实践是始终为函数指定一个返回类型。如果不需要返回值，请将函数声明为具有 `void` 返回类型。如果未指定返回类型，则 C 编译器会假定默认返回类型 `int`。

许多程序员使用括号将 `return` 语句的表达式参数括起来。但是，C 不需要括号。

如果该编译器发现 `return` 语句后放置了任何语句，则它可能会发出一条警告诊断消息，指出代码无法访问。

在 `main` 函数中，`return` 语句和表达式是可选的 。返回的值（若指定了返回值）发生的情况取决于实现。**Microsoft 专用**：Microsoft C 实现会将表达式值返回给调用程序的进程，例如 `cmd.exe`。如果未提供 `return` 表达式，则 Microsoft C 运行时会返回一个值来指示成功 (0) 还是失败（非零值）。

## 示例

以下示例是一个程序，很多部分都用到了它。它演示了 `return` 语句，还演示了如何使用它来结束函数执行和根据需要返回值。

```
// C_return_statement.c
// Compile using: cl /W4 C_return_statement.c
#include <limits.h>      // for INT_MAX
#include <stdio.h>       // for printf

long long square( int value )
{
    // Cast one operand to long long to force the
    // expression to be evaluated as type long long.
    // Note that parentheses around the return expression
    // are allowed, but not required here.
    return ( value * (long long) value );
}
```

`square` 函数在更大范围的类型中返回其参数的平方，以防止出现算术错误。**Microsoft 专用**：在 Microsoft C 实现中，`long long` 类型足够大，可容纳两个 `int` 值的乘积而不出现溢出 。

`square` 中 `return` 表达式两侧的括号在计算时被看做是表达式的一部分，`return` 语句不需要使用括号 。

```
double ratio( int numerator, int denominator )
{
    // Cast one operand to double to force floating-point
    // division. Otherwise, integer division is used,
    // then the result is converted to the return type.
    return numerator / (double) denominator;
}
```

`ratio` 函数会以浮点 `double` 值的形式返回其两个 `int` 参数之比 。`return` 表达式被强制使用浮点运算，方式是将其中一个操作数强制转换为 `double` 。否则，会使用整除运算符，而小数部分将丢失。

```
void report_square( void )
{
    int value = INT_MAX;
    long long squared = 0LL;
    squared = square( value );
    printf( "value = %d, squared = %lld\n", value, squared );
    return; // Use an empty expression to return void.
}
```

`report_square` 函数使用 `INT_MAX` 的参数值调用 `square`，INT_MAX 是适合 `int` 的最大带符号整数值。`long long` 结果存储在 `squared` 中，然后打印出来。`report_square` 函数具有 `void` 返回类型，因此它的 `return` 语句中没有表达式 。

```
void report_ratio( int top, int bottom )
{
    double fraction = ratio( top, bottom );
    printf( "%d / %d = %.16f\n", top, bottom, fraction );
    // It's okay to have no return statement for functions
    // that have void return types.
}
```

`report_ratio` 函数使用 `1` 和 `INT_MAX` 的参数值调用 `ratio`。`double` 结果存储在 `fraction` 中，然后打印出来。`report_ratio` 函数具有 `void` 返回类型，因此无需显式返回值。`report_ratio` 的执行被放弃，不会向调用方返回任何值。

```
int main()
{
    int n = 1;
    int x = INT_MAX;

    report_square();
    report_ratio( n, x );

    return 0;
}
```

`main` 函数会调用两个值：`report_square` 和 `report_ratio`。由于 `report_square` 不采用任何参数且返回 `void` 因此我们不会将其结果分配给变量。同样地，`report_ratio` 会返回 `void`，因此我们不保存它的返回值。在上述每个函数调用后，都继续在下一条语句执行。然后，`main` 会返回值 `0`（通常用于报告成功）来结束程序。

要编译该示例，请创建一个名为 `C_return_statement.c` 的源代码文件。然后，按照所示顺序复制所有示例代码。保存文件，再使用以下命令在开发人员命令提示窗口中编译它：

```
cl /W4 C_return_statement.c
```

然后，在命令提示符处输入 `C_return_statement.exe` 来运行示例代码。该示例的输出与以下内容类似：

```
value = 2147483647, squared = 4611686014132420609
1 / 2147483647 = 0.0000000004656613
```

## 请参阅

语句

# `_Static_assert` 关键字和 `static_assert` 宏 (C11)

2021/8/11 •

在编译时测试断言。如果指定的常数表达式为 `false`，则编译器显示指定的消息，并且编译失败，错误为 C2338；否则，不会产生任何影响。C11 中的新增功能。

`_Static_assert` 是 C11 中引入的关键字。`static_assert` 是 C11 中引入的宏，它映射到 `_Static_assert` 关键字。

## 语法

```
_Static_assert(constant-expression, string-literal);
static_assert(constant-expression, string-literal);
```

**parameters**

*`constant-expression`*
可在编译时计算的整型常数表达式。如果表达式为零 (false)，则显示 `string-literal` 参数，并且编译因出错而失败。如果表达式不为零 (true)，则不会产生任何影响。

*`string-literal`*
如果 `constant-expression` 计算结果为零 (false)，则显示此消息。此消息必须使用编译器的基本字符集来生成。字符不能为多字节字符或宽字符。

## 备注

`_Static_assert` 关键字和 `static_assert` 宏均在编译时测试软件断言 。它们可用于全局或函数范围。

相反，`assert` 宏、`_assert` 和 `_wassert` 函数在运行时测试软件断言，并产生运行时成本。

Microsoft 特定行为

在 C 中，如果不包含 `<assert.h>`，Microsoft 编译器会将 `static_assert` 视为映射到 `_Static_assert` 的关键字。首选使用 `static_assert`，因为相关代码在 C 和 C++ 中均适用。

## 编译时断言示例

在下面的示例中，`static_assert` 和 `_Static_assert` 用于验证枚举中有多少个元素以及整数的宽度是否为 32 位。

```
// requires /std:c11 or higher
#include <assert.h>

enum Items
{
    A,
    B,
    C,
    LENGTH
};

int main()
{
    // _Static_assert is a C11 keyword
    _Static_assert(LENGTH == 3, "Expected Items enum to have three elements");

    // Preferred: static_assert maps to _Static_assert and is compatible with C++
    static_assert(sizeof(int) == 4, "Expecting 32 bit integers");

    return 0;
}
```

## 要求

| [illegible] | [illegible] |
| --- | --- |
| `static_assert` | <assert.h> |

使用 `/std:c11` 进行编译。

Windows SDK 10.0.20348.0（版本 2104）或更高版本。请参阅 Windows 10 SDK 以下载最新 SDK。有关安装和使用 SDK 进行 C11 和 C17 开发的说明，请参阅在 Visual Studio 中安装 C11 和 C17 支持。

## 另请参阅

`_STATIC_ASSERT` 宏
`assert` 宏、`_assert` 和 `_wassert` 函数 `/std`（指定语言标准版本）

# `switch` 语句 (C)

2021/8/11 •

`switch` 和 `case` 语句帮助控制复杂条件和分支运算。 `switch` 语句将控制权转交给其正文中的语句。

## 语法

```
selection-statement:
    switch ( expression ) statement
```

```
labeled-statement:
    case constant-expression : statement
    default : statement
```

## 备注

`switch` 语句使控件根据 `expression` 的值转移到其语句正文中的一个 `labeled-statement` 。

`expression` 和每个 `constant-expression` 的值必须有一个整型类型 。在编译时， `constant-expression` 必须有一个明确的常数整型值 。

控件传递给 `case` 语句，该语句的 `constant-expression` 值与 `expression` 值匹配 。 `switch` 语句可以包含任意数量的 `case` 实例。但是，同一个 `switch` 语句中的两个 `constant-expression` 值不能具有相同的值 。 `switch` 语句正文的执行从匹配的 `labeled-statement` 中或之后的第一个语句开始 。执行一直持续到正文的末尾，或者直到 `break` 语句将控制权从主体中传出。

`switch` 语句的使用通常类似于：

```
switch ( expression )
{
    // declarations
    // . . .
    case constant_expression:
        // statements executed if the expression equals the
        // value of this constant_expression
        break;
    default:
        // statements executed if expression does not equal
        // any case constant_expression
}
```

可以使用 `break` 语句结束 `switch` 语句中特定标记语句的处理。它分支到 `switch` 语句的结尾。如果不使用 `break` ，则程序会继续到下一标记语句，并执行语句，直到达到 `break` 或该语句的末尾。在某些情况下，可能需要此继续符。

如果没有 `case` `constant-expression` 值等于 `expression` ，则执行 `default` 语句 。如果没有 `default` 语句，并且找不到 `case` 匹配，则 `switch` 正文中的任何语句都不会执行。最多可以有一个 `default` 语句。 `default` 语句不必在末尾出现。它可能出现在 `switch` 语句正文中的任何位置。 `case` 或 `default` 标签只能显示在 `switch` 语句内部 。

`switch` `expression` 和 `case` `constant-expression` 的类型必须为整型 。每个 `case``constant-expression` 的值

在语句正文中必须是唯一的 。

`switch` 语句正文的 `case` 和 `default` 标签只在初始测试中有意义，该测试将确定语句体中开始执行的位置 。 可以嵌套 `switch` 语句。在执行到任何 `switch` 语句中之前，初始化任何静态变量。

> **NOTE**
>
> 声明可以出现在构成 `switch` 正文的复合语句的前面，但不执行包含在声明中的初始化。`switch` 语句将控制权直接转交给正文中的一个可执行语句，并绕过包含初始化的行。

以下示例演示了 `switch` 语句：

```
switch( c )
{
    case 'A':
        capital_a++;
    case 'a':
        letter_a++;
    default :
        total++;
}
```

如果 `c` 等于 `'A'`，则会执行此示例中的 `switch` 正文的所有三个语句，因为不会在以下 `case` 前显示 `break` 语句。将执行控制转交给第一个语句 ( `capital_a++;` ) 并继续按顺序转交给主体的其余部分。如果 `c` 等于 `'a'` ，则 `letter_a` 和 `total` 将增加。当 `c` 不等于 `'A'` 或 `'a'` 时，仅递增 `total` 。

```
switch( i )
{
    case -1:
        n++;
        break;
    case 0 :
        z++;
        break;
    case 1 :
        p++;
        break;
}
```

在此示例中，`break` 语句跟在 `switch` 正文的每个语句的后面 。在执行一个语句后，`break` 语句将强制从语句正文中退出 。如果 `i` 等于 -1，则仅 `n` 将增加。`n++;` 语句后面的 `break` 会导致执行控制传递出语句正文，并绕过剩余语句 。同样，如果 `i` 等于 0，则仅 `z` 将增加；如果 `i` 等于 1，则仅 `p` 将增加。从严格意义上讲，最后的 `break` 语句不是必需的，因为控制权将在复合语句的结尾传递出语句正文 。包含此语句是为了获得一致性。

如下面的示例所示，一个语句可以包含多个 `case` 标签：

```
switch( c )
{
    case 'a' :
    case 'b' :
    case 'c' :
    case 'd' :
    case 'e' :
    case 'f' :  convert_hex(c);
}
```

在此示例中，如果 *constant-expression* 等于 `'a'` 和 `'f'` 之间的任何字母，则调用 `convert_hex` 函数。

**Microsoft 专用**

Microsoft C 未限制 `switch` 语句中 `case` 值的数量 。该数量仅受可用内存的限制。ANSI C 要求 `switch` 语句内至少允许使用 257 个 `case` 标签 。

Microsoft C 的 default 是启用 Microsoft 扩展。使用 /Za 编译器选项禁用这些扩展。

## 请参阅

switch语句 (C++)

# try-except 语句 (C)

2021/8/14 •

Microsoft 专用

`try-except` 语句是一项 Microsoft C++ 语言扩展，它使应用程序能够在正常终止执行的事件发生时获取对程序的控制权。此类事件称为异常，处理异常的机制称为结构化异常处理。

异常可能基于硬件或软件。即使应用程序无法从硬件或软件异常中完全恢复，结构化异常处理也可以记录和显示错误信息。这有助于捕获应用程序的内部状态，从而帮助诊断问题。特别是，这对于不容易重现的间歇性问题很有用。

## 语法

```
try-except-statement:
    __try compound-statement __except ( expression ) compound-statement
```

`__try` 子句后的复合语句是受保护节。`__except` 子句后的复合语句是异常处理程序。如果在受保护节的执行过程中引发了异常，处理程序将指定要采取的一系列措施。执行过程如下所示：

1. 执行受保护节。
2. 如果在执行受保护的部分期间没有发生异常，则在 `__except` 子句之后的语句处继续执行。
3. 如果在受保护节的执行过程中或受保护节调用的任何例程中发生异常，则会计算 `__except` 表达式。返回的值将确定该异常的处理方式。有三种可能的值：
   - `EXCEPTION_CONTINUE_SEARCH`：无法识别异常。继续向上搜索堆栈查找处理程序，首先是所在的 `try-except` 语句，然后是具有下一个最高优先级的处理程序。
   - `EXCEPTION_CONTINUE_EXECUTION`：异常可识别，但被关闭。从出现异常的点继续执行。
   - `EXCEPTION_EXECUTE_HANDLER` 异常可识别。通过执行 `__except` 复合语句来将控制权转移给异常处理程序，然后在发生异常的语句处继续执行。

由于 `__except` 表达式是作为 C 表达式计算的，因此它被限制为单个值、条件表达式运算符或逗号运算符。如果需要更大量的处理，表达式可调用返回上面列出的三个值之一的例程。

> **NOTE**
> 结构化异常处理适用于 C 和 C++ 源文件。但是，这不是专门为 C++ 设计的。对于可移植 C++ 程序，应使用 C++ 异常处理，而不是结构化异常处理。此外，C++ 异常处理机制灵活得多，因为它可以处理任何类型的异常。有关详细信息，请参阅《C++ 语言参考》中的异常处理。

应用程序中的每个例程可以有自己的异常处理程序。`__except` 表达式在 `__try` 主体的范围内执行。它可以访问在该处声明的任何局部变量。

`__leave` 关键字在 `try-except` 语句块中有效。`__leave` 的效果是跳转到 `try-except` 块的末尾。执行将在异常处理程序结束后恢复。尽管可以使用 `goto` 语句来完成相同的结果，但 `goto` 语句会导致堆栈展开。由于 `__leave` 语句不涉及堆栈展开，因此更有效。

使用 `longjmp` 运行时函数退出 `try-except` 语句被视为异常终止。跳转到 `__try` 语句是非法的，但跳出该语句

是合法的。如果有进程在执行 `try-except` 语句的过程中终止，则不会调用异常处理程序。

## 示例

下面是异常处理程序和终止处理程序的示例。有关终止处理程序的详细信息，请参阅 `try-finally` 语句 (C)。

```
.
.
.
puts("hello");
__try {
   puts("in try");
   __try {
      puts("in try");
      RAISE_AN_EXCEPTION();
   } __finally {
      puts("in finally");
   }
} __except( puts("in filter"), EXCEPTION_EXECUTE_HANDLER ) {
   puts("in except");
}
puts("world");
```

这是上面的示例的输出，右侧还添加了注释：

```
hello
in try              /* fall into try                        */
in try              /* fall into nested try                 */
in filter           /* execute filter; returns 1 so accept  */
in finally          /* unwind nested finally                */
in except           /* transfer control to selected handler */
world               /* flow out of handler                  */
```

结束 Microsoft 专用

## 另请参阅

`try-except` 语句 (C++)

# try-finally 语句 (C)

2021/8/11 •

**Microsoft 专用**

`try-finally` 语句是 C 语言的 Microsoft 扩展，用于使应用程序能够在代码块的执行被中断时保证清理代码的执行。清理包括多个任务，如释放内存、关闭文件和释放文件句柄。`try-finally` 语句对此类例程特别有用：具有几个位置，在这些位置上执行了检查以找出可能导致例程提前返回内容的错误。

```
try-finally-statement:
    __try compound-statement __finally compound-statement
```

`__try` 子句后的复合语句是受保护节。`__finally` 子句后的复合语句是终止处理程序。处理程序将指定在退出受保护节时要执行的一系列操作。无论受保护节是由异常（异常终止）还是由标准贯穿（正常终止）退出，都没有问题。

控制权通过简单的顺序执行（贯穿）传递到 `__try` 语句。当控制权交给 `__try` 语句时，其关联的处理程序将变为活动状态。执行过程如下所示：

1. 执行受保护节。
2. 调用终止处理程序。
3. 当终止处理程序完成时，在 `__finally` 语句后继续执行。无论受保护节如何结束（例如，通过受保护的主体外部的 `goto` 语句或通过 `return` 语句），终止处理程序都在控制流移出受保护节之前执行。

`__leave` 关键字在 `try-finally` 语句块中有效。`__leave` 的效果是跳转到 `try-finally` 块的末尾。终止处理程序将立即执行。尽管可以使用 `goto` 语句来完成相同的结果，但 `goto` 语句会导致堆栈展开。由于 `__leave` 语句不涉及堆栈展开，因此更有效。

使用 `return` 语句或 `longjmp` 运行时函数退出 `try-finally` 语句被视为异常终止。跳转到 `__try` 语句是非法的，但跳出该语句是合法的。必须运行在起点和终点之间处于活动状态的所有 `__finally` 语句。这称为“局部展开”。

如果在执行 `try-finally` 语句时取消了进程，则不会调用终止处理程序。

> **NOTE**
> 结构化异常处理适用于 C 和 C++ 源文件。但是，这不是专门为 C++ 设计的。对于可移植 C++ 程序，应使用 C++ 异常处理，而不是结构化异常处理。此外，C++ 异常处理机制灵活得多，因为它可以处理任何类型的异常。有关详细信息，请参阅《C++ 语言参考》中的异常处理。

请参阅 `try-except` 语句的示例以了解 `try-finally` 语句如何运行。

**结束 Microsoft 专用**

## 另请参阅

`try-finally` 语句 (C++)

# While 语句 (C)

2021/8/12 •

利用 `while` 语句，可以重复执行语句，直到指定的 expression 变成 false。

## 语法

*iteration-statement*:
  **while (** *expression* **)** *statement*

expression 必须具有算法或指针类型。执行过程如下所示：

1. 计算 expression 。
2. 如果 expression 最初为 false，则 `while` 语句的主体永远不会执行，并且控制权从 `while` 语句传递给程序中的下一个语句。

   如果 expression 为 true（非零），则执行语句体，并且此过程从第 1 步开始重复。

当语句主体中的 `break`、`goto` 或 `return` 执行时，`while` 语句也可以终止。使用 `continue` 语句可以在不退出 `while` 循环的情况下终止迭代。`continue` 语句将控制权传递给 `while` 语句的下一个迭代。

下面的示例展示了 `while` 语句：

```
while ( i >= 0 )
{
    string1[i] = string2[i];
    i--;
}
```

此示例将 `string2` 中的字符复制到 `string1`。如果 `i` 大于或等于 0，则 `string2[i]` 将赋给 `string1[i]`，并且 `i` 将递减。当 `i` 达到或小于 0 时，`while` 语句的执行终止。

## 请参阅

While 语句 (C++)

# 函数 (C)

2021/8/15 •

函数是 C 中的基本模块单元。函数通常设计为执行一项特定任务且其名称通常会反映该任务。函数包含声明和语句。本节描述如何声明、定义和调用 C 函数。讨论的其他主题包括：

- 函数概述
- 函数特性
- 指定调用约定
- 内联函数
- DLL 导出和导入函数
- 函数
- 存储类
- 返回类型
- 参数
- 参数

## 请参阅

C 语言参考

# 函数概述

2021/8/13 •

函数必须具有定义且应具有声明，尽管定义可用作声明（如果声明在调用函数前出现）。函数定义包含函数主体（调用函数时执行的代码）。

函数声明为程序中其他位置定义的函数建立名称、返回类型和特性。函数声明必须在对函数的调用之前。这就是为什么在调用运行时函数之前将包含运行时函数的声明的头文件包含在您的代码中的原因。如果声明包含有关参数的类型和数目的信息，则该声明为原型。有关详细信息，请参阅函数原型。

编译器使用原型来比较随后通过函数的参数对函数进行的调用中的自变量类型，并在需要时，将自变量类型转换为参数类型。

函数调用将执行控制从正在调用的函数传递到已调用函数。通过值将参数（如果有）传递给已调用函数。在被调用的函数中执行 `return` 语句会向正在调用的函数返回控制权和可能的值。

## 请参阅

函数

# 函数声明和定义的过时形式

2021/8/16 •

旧式函数声明和定义使用与 ANSI C 标准建议的语法略微不同的规则来声明参数。首先，旧式声明不具有参数列表。第二，在函数定义中，列出了参数，但未在参数列表中声明其类型。类型声明在构成函数主体的复合语句之前。该旧式语法已过时，不应在新代码中使用。但仍支持使用旧式语法的代码。此示例阐释声明和定义的过时形式：

```
double old_style();           /* Obsolete function declaration */

double alt_style( a , real )  /* Obsolete function definition */
    double *real;
    int a;
{
    return ( *real + a ) ;
}
```

返回与 `int` 大小相同的整数或指针的函数不需要有声明，尽管建议有声明。

为了遵循 ANSI C 标准，使用省略号的旧式函数声明现在会在使用 /Za 选项进行编译是生成错误，并在使用 /Ze 进行编译时生成 4 级别警告。例如：

```
void funct1( a, ... )  /* Generates a warning under /Ze or */
int a;                 /* an error when compiling with /Za */
{
}
```

您应将此声明重写为原型：

```
void funct1( int a, ... )
{
}
```

旧式函数声明也会生成警告（如果您随后声明或定义具有省略号或具有与其提升的类型不同的类型的参数的相同函数）。

下一节（C 函数定义）显示函数定义的语法（包括旧式语法）。旧式语法中的参数列表的非终止符是 identifier-list 。

## 请参阅

函数概述

# C 函数定义

2021/8/12 •

函数定义指定函数的名称、函数期望接收的参数的类型和数量以及函数的返回类型。函数定义还包括带有局部变量的声明的函数体和确定函数行为的语句。

## 语法

translation-unit：
*external-declaration*
*translation-unit external-declaration*

external-declaration : /* 只允许在外部（文件）范围内 */
*function-definition*
*declaration*

function-definition：
*declaration-specifiers*$_{opt}$ *attribute-seq*$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

/* *attribute-seq* is Microsoft-specific */

原型参数为：

*declaration-specifiers*:
*storage-class-specifier declaration-specifiers*$_{opt}$
*type-specifier declaration-specifiers*$_{opt}$
*type-qualifier declaration-specifiers*$_{opt}$

declaration-list：
*declaration*
*declaration-list declaration*

*declarator*:
*pointer*$_{opt}$ *direct-declarator*

direct-declarator : /* 函数声明符 */
*direct-declarator* **(** *parameter-type-list* **)** /* New-style declarator */
*direct-declarator* **(** *identifier-list*$_{opt}$ **)** /* Obsolete-style declarator */

定义中的参数列表使用以下语法：

parameter-type-list：/* 参数列表 */
*parameter-list*
*parameter-list*, ...

*parameter-list*:
*parameter-declaration*
*parameter-list*, *parameter-declaration*

*parameter-declaration*:
*declaration-specifiers declarator*
*declaration-specifiers abstract-declarator*$_{opt}$

旧式函数定义中的参数列表使用以下语法：

identifier-list :/* 在旧式函数定义和声明中使用 */
*identifier*
*identifier-list* , *identifier*

函数体的语法为:

compound-statement :
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

唯一可以修改函数声明的存储类说明符是 `extern` 和 `static`。`extern` 说明符表示可以从其他文件引用函数;也就是说,将函数名称导出到链接器。`static` 说明符表示不能从其他文件引用函数;也就是说,链接器不会导出名称。如果函数定义中没有出现存储类,则假定为 `extern`。在任何情况下,从定义点到文件的末尾函数始终可见。

可选的 declaration-specifiers 和必需的 declarator 共同指定函数的返回类型和名称。declarator 是用来命名函数的标识符与函数名后面的括号的组合。可选的 attribute-seq 非终止符是在函数特性中定义的 Microsoft 专用功能。

direct-declarator (在 declarator 语法中)指定要定义的函数的名称及其参数的标识符。如果 direct-declarator 包括 parameter-type-list ,则该列表将指定所有参数的类型。此类声明符还用作以后对函数进行调用时的函数原型。

函数定义中的 declaration-list 内的声明不能包含除 `register` 之外的 storage-class-specifier。只有当为 `int` 类型的值指定 `register` 存储类时,才能省略 declaration-specifiers 语法中的 type-specifier。

compound-statement 是包含局部变量声明、对在外部声明的项的引用和语句的函数体。

函数特性、存储类、返回类型、参数和函数体节详细地描述了函数定义的组成部分。

## 请参阅

函数

# 函数特性

2021/8/17 •

**Microsoft 专用**

可选的 attribute-seq 非终止符允许你逐个函数地选择调用约定。也可以将函数指定为 `__fastcall` 或 `__inline` 。

**结束 Microsoft 专用**

## 请参阅

C 函数定义

# 指定调用转换

2021/8/13 •

**Microsoft 专用**

**有关调用约定的信息**，请参阅调用约定主题。

**结束 Microsoft 专用**

## 请参阅

**函数特性**

# 内联函数

2021/8/13 •

**Microsoft 专用**

`__inline` 关键字指示编译器用函数定义中的代码替换函数调用的每个实例。但是，替换操作完全由编译器自行决定。例如，如果某个函数的地址被采用或者由于过大而无法内联，则编译器不会内联该函数。

对要作为内联候选项加以考虑的函数，它必须使用新式的函数定义。

请使用以下形式指定内联函数：

```
__inline type_opt function-definition
```

使用内联函数将生成更快的代码，有时可能生成比等效函数调用所生成的更小的代码，原因如下：

- 它节省了执行函数调用所需的时间。
- 小的内联函数（可能是三行或更少）创建的代码比等效函数调用创建的代码更少，因为编译器不会生成处理参数和返回值的代码。
- 函数生成的内联需要进行普通函数不可用的代码优化，因为编译器不执行过程间优化。

不应该将使用 `__inline` 的函数与内联汇编程序代码混淆。有关详细信息，请参阅内联汇编程序。

**结束 Microsoft 专用**

## 请参阅

inline、__inline、__forceinline

# 内联汇编程序 (C)

2021/8/12 •

**Microsoft 专用**

利用内联汇编程序，您可以直接在 C 源程序中嵌入汇编语言指令，而无需额外的程序集和链接步骤。内联汇编程序生成到该编译器中，因此您不需要一个单独的汇编程序，例如 Microsoft Macro Assembler (MASM)。

由于内联汇编程序不需要单独的程序集和链接步骤，因此它比单独的汇编程序更方便。内联程序集代码可以使用任何 C 变量或范围中的函数名，因此，将其与程序的 C 代码集成非常容易。由于程序集代码可以与 C 语句混合，因此它可以完成在单独的 C 中难以完成或无法完成的任务。

`__asm` 关键字调用内联汇编程序，并且可以在 C 语句合法的任何位置出现。它不能单独出现。它必须后跟一个程序集指令、一组括在大括号中的指令或者至少一对空大括号。此处的“`__asm` 块”一词是指任何指令或指令组（无论是否在大括号中）。

下面的代码是一个用大括号括起来的简单 `__asm` 块。（此代码是一个自定义函数 prolog 序列。）

```
__asm
{
   push ebp
   mov  ebp, esp
   sub  esp, __LOCAL_SIZE
}
```

或者，也可以将 `__asm` 放在每个汇编指令前面：

```
__asm push ebp
__asm mov  ebp, esp
__asm sub  esp, __LOCAL_SIZE
```

由于 `__asm` 关键字是语句分隔符，因此还可以将汇编指令放在同一行中：

```
__asm push ebp   __asm mov  ebp, esp   __asm sub  esp, __LOCAL_SIZE
```

**结束 Microsoft 专用**

## 请参阅

函数特性

# `_Noreturn` 关键字和 `noreturn` 宏 (C11)

2021/8/25 •

`_Noreturn` 关键字在 C11 中引入。它告知编译器，应用编译器的函数不返回调用方。编译器知晓调用 `_Noreturn` 函数后的代码是不可访问的。不返回的函数的一个示例为 abort。如果控制流有可能返回调用方，则函数不能具有 `_Noreturn` 属性。

该关键字通常通过 <stdnoreturn.h> 中提供的便捷宏 `noreturn` 来进行使用，该宏映射到 `_Noreturn` 关键字。

使用 `_Noreturn`（或等效的 `noreturn`）的主要好处是在代码中明确函数的意向，便于将来的读者了解，以及检测意外产生的无法访问的代码。

标记为 `noreturn` 的函数不应包括返回类型，因为它不会将值返回给调用方。它应为 `void`。

## 使用 `noreturn` 宏和 `_Noreturn` 关键字的示例

下面的示例演示了 `_Noreturn` 关键字和等效的 `noreturn` 宏。

如果使用可以忽略的宏 `noreturn`，则 IntelliSense 可能会生成虚假错误 `E0065`。它不会阻止你运行该示例。

```
// Compile with Warning Level4 (/W4) and /std:c11
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void fatal_error(void)
{
    exit(3);
}

_Noreturn void not_coming_back(void)
{
    puts("There's no coming back");
    fatal_error();
    return; // warning C4645 - function declared with noreturn has a return statement
}

void done(void)
{
    puts("We'll never get here");
}

int main(void)
{
    not_coming_back();
    done(); // warning c4702 - unreachable code

    return 0;
}
```

## 要求

| [illegible] | [illegible] |
| --- | --- |
| `noreturn` | <stdnoreturn.h> |

## 另请参阅

/std（指定语言标准版本）
/W4（指定警告等级）
C4702 警告
__declspec(noreturn)

# DLL 导入和导出函数

2021/8/12 •

**Microsoft 专用**

有关本主题的最完整且最新的信息可在 dllexport、dllimport 中找到。

`dllimport` 和 `dllexport` 存储类修饰符是 Microsoft 对 C 语言的特定扩展。这些修饰符显式定义了 DLL 与其客户端(可执行文件或另一个 DLL)的接口。如果将函数声明为 `dllexport`，则不再需要模块定义 (.DEF) 文件。还可以对数据和对象使用 `dllimport` 和 `dllexport` 修饰符。

`dllimport` 和 `dllexport` 存储类修饰符必须与扩展的特性语法关键字 `__declspec` 一起使用，如下面的示例所示：

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllExport void func();
DllExport int i = 10;
DllExport int j;
DllExport int n;
```

有关扩展的存储类修饰符的语法的特定信息，请参阅扩展的存储类特性。

**结束 Microsoft 专用**

## 请参阅

C 函数定义

# 定义和声明 (C)

2021/8/16 •

**Microsoft 专用**

DLL 接口引用已知由系统中的某程序导出的所有项(函数和数据);即所有被声明为 `dllimport` 或 `dllexport` 的项。DLL 接口中包含的所有声明都必须指定 `dllimport` 或 `dllexport` 特性。但是，定义仅可指定 `dllexport` 特性。例如，以下函数定义产生了一个编译器错误：

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllImport int func()    /* Error; dllimport prohibited in */
                        /* definition. */
{
   return 1;
}
```

以下代码也会产生错误：

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllImport int i = 10;      /* Error; this is a definition. */
```

但是，这是正确的语法：

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

DllExport int i = 10;      /* Okay: this is an export definition. */
```

使用 `dllexport` 意味着定义，而使用 `dllimport` 则意味着声明。必须结合使用 `extern` 关键字和 `dllexport` 来强制声明;否则意味着定义。

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

extern DllImport int k;   /* These are correct and imply */
Dllimport int j;          /* a declaration. */
```

**结束 Microsoft 专用**

## 请参阅

DLL 导入和导出函数

# 定义带有 dllexport 和 dllimport 的内联 C 函数

2021/8/16 •

**Microsoft 专用**

可以定义为将函数与 `dllexport` 特性内联。在这种情况下，将始终实例化并导出该函数，无论程序中是否有模块引用该函数。假定该函数由另一个程序导入。

还可以定义为内联使用 `dllimport` 特性声明的函数。在这种情况下，函数可以展开（遵从 /Ob（内联）编译器选项规范），但决不实例化。具体而言，如果采用内联导入函数的地址，则返回驻留在 DLL 中的函数地址。此行为与采用非内联导入函数的地址相同。

内联函数中的静态本地数据和字符串在 DLL 和客户端之间保持的标识与它们在单一程序（即，没有 DLL 接口的可执行文件）中保持的一样。

在提供导入的内联函数时谨慎操作。例如，如果更新 DLL，请不要假定该客户端将使用更改后的 DLL 版本。若要确保加载 DLL 的适当版本，请重新生成 DLL 的客户端。

**结束 Microsoft 专用**

## 请参阅

DLL 导入和导出函数

# dllimport/dllexport 的规则和限制

2021/8/11 •

### Microsoft 专用

- 如果你没有使用 `dllimport` 或 `dllexport` 特性声明函数，则此函数被视为不是 DLL 接口的一部分。因此，函数的定义必须存在于该模块或同一程序的另一个模块中。若要使函数成为 DLL 接口的一部分，您必须将其他模块中函数的定义声明为 `dllexport`。否则，在构建客户端时，将生成链接器错误。

- 如果程序中的单个模块包含对同一函数的 `dllimport` 和 `dllexport` 声明，则 `dllexport` 特性优先于 `dllimport` 特性。但是，会生成编译器警告。例如：

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport void func1( void );
   DllExport void func1( void );   /* Warning; dllexport */
                                   /* takes precedence. */
```

- 无法利用使用 `dllimport` 特性声明的数据对象的地址来初始化静态数据指针。例如，下面的代码将生成错误：

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport int i;
   .
   .
   .
   int *pi = &i;                          /* Error */

   void func2()
   {
      static int *pi = &i;                  /* Error */
   }
```

- 如果利用使用 `dllimport` 声明的函数的地址来初始化静态函数指针，会将指针设置为 DLL 导入 thunk（将控制权转移给函数的代码存根）的地址，而不是函数的地址。此赋值不生成错误消息：

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport void func1( void
   .
   .
   .
   static void ( *pf )( void ) = &func1;   /* No Error */

   void func2()
   {
      static void ( *pf )( void ) = &func1;  /* No Error */
   }
```

- 由于包含对象声明中的 `dllexport` 特性的程序必须为该对象提供定义，因此您可以利用 `dllexport` 函数的地址初始化全局或局部静态函数指针。同样，您可以利用 `dllexport` 数据对象的地址初始化全局或局部静态数据指针。例如：

```
#define DllImport   __declspec( dllimport )
#define DllExport   __declspec( dllexport )

   DllImport void func1( void );
   DllImport int i;

   DllExport void func1( void );
   DllExport int i;
   .
   .
   .
   int *pi = &i;                              /* Okay */
   static void ( *pf )( void ) = &func1;    /* Okay */

   void func2()
   {
      static int *pi = i;                      /* Okay */
      static void ( *pf )( void ) = &func1;   /* Okay */
   }
```

结束 Microsoft 专用

## 请参阅

DLL 导入和导出函数

# Naked 函数

2021/8/12 •

**Microsoft 专用**

`naked` 存储类特性是特定于 Microsoft 的 C 语言扩展。对于使用 `naked` 存储类特性声明的函数，编译器生成不带 prolog 和 epilog 代码的代码。利用此功能，可以使用内联汇编程序代码编写您自己的 prolog/epilog 代码序列。裸函数对于编写虚拟设备驱动程序特别有用。

由于 `naked` 特性仅与函数定义相关且不是类型修饰符，因此 naked 函数使用扩展的特性语法，如扩展的存储类特性中所述。

下面的示例利用 `naked` 特性定义了一个函数：

```
__declspec( naked ) int func( formal_parameters )
{
   /* Function body */
}
```

或者：

```
#define Naked   __declspec( naked )

Naked int func( formal_parameters )
{
   /* Function body */
}
```

`naked` 特性仅影响函数的 prolog 和 epilog 序列的编译器代码生成的性质。它不影响为调用这些函数而生成的代码。因此，`naked` 特性不被视为函数的类型的一部分，并且函数指针不能具有 `naked` 特性。此外，`naked` 特性不能应用于数据定义。例如，下面的代码将生成错误：

```
__declspec( naked ) int i;  /* Error--naked attribute not */
                            /* permitted on data declarations. */
```

`naked` 特性仅与函数的定义相关，且无法在函数原型中指定。下面的声明生成编译器错误：

```
__declspec( naked ) int func();   /* Error--naked attribute not */
                    /* permitted on function declarations.    */   \
```

**结束 Microsoft 专用**

## 请参阅

C 函数定义

# 针对使用 Naked 函数的规则和限制

2021/8/13 •

若要了解有关使用裸函数的规则和限制，请参阅“C++ 语言参考”中的相应主题：裸函数的规则和限制。

## 请参阅

naked 函数

# 编写 Prolog/Epilog 代码时的注意事项

2021/8/13 •

**Microsoft 专用**

在编写你自己的 prolog 和 epilog 代码序列之前，请务必了解堆栈帧的布局方式。了解如何使用 __LOCAL_SIZE 预定义的常量也很有用。

## CStack 帧布局

此示例显示了可能出现在 32 位函数中的标准 prolog 代码：

```
push    ebp                 ; Save ebp
mov     ebp, esp            ; Set stack frame pointer
sub     esp, localbytes     ; Allocate space for locals
push    <registers>         ; Save registers
```

`localbytes` 变量表示局部变量堆栈上所需的字节数，`registers` 变量是表示要保存在堆栈上的寄存器列表的占位符。推入寄存器后，您可以将任何其他适当的数据放置在堆栈上。下面是相应的 epilog 代码：

```
pop     <registers>         ; Restore registers
mov     esp, ebp            ; Restore stack pointer
pop     ebp                 ; Restore ebp
ret                         ; Return from function
```

堆栈始终向下增长(从高内存地址到低内存地址)。基指针 (`ebp`) 指向 `ebp` 的推入值。局部变量区域从 `ebp-2` 开始。若要访问局部变量，可通过从 `ebp` 中减去适当的值来计算 `ebp` 的偏移量。

## __LOCAL_SIZE 常量

编译器提供常量 __LOCAL_SIZE 以用于函数 prolog 代码的内联汇编程序块。此常数用于在自定义 prolog 代码中的堆栈帧上为局部变量分配空间。

编译器确定 __LOCAL_SIZE 的值。该值是所有用户定义的局部变量和编译器生成的临时变量的总字节数。__LOCAL_SIZE 只能用作即时操作数；它不能在表达式中使用。您不得更改或重新定义此常量的值。例如：

```
mov     eax, __LOCAL_SIZE           ;Immediate operand--Okay
mov     eax, [ebp - __LOCAL_SIZE]   ;Error
```

包含自定义 prolog 和 epilog 序列的 naked 函数的以下示例在 prolog 序列中使用 __LOCAL_SIZE：

```
__declspec ( naked ) func()
{
   int i;
   int j;

   __asm      /* prolog */
      {
      push   ebp
      mov      ebp, esp
      sub      esp, __LOCAL_SIZE
      }

   /* Function body */

   __asm      /* epilog */
      {
      mov      esp, ebp
      pop      ebp
      ret
      }
}
```

**结束 Microsoft 专用**

## 请参阅

naked 函数

# 存储类

2021/8/11 •

函数定义中的存储类说明符为函数提供 `extern` 或 `static` 存储类。

## 语法

function-definition：
*declaration-specifiers*$_{opt}$ *attribute-seq*$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

/* *attribute-seq* is Microsoft-specific */

*declaration-specifiers*:
*storage-class-specifier declaration-specifiers*$_{opt}$
*type-specifier declaration-specifiers*$_{opt}$
*type-qualifier declaration-specifiers*$_{opt}$

storage-class-specifier : /* 针对函数定义 */
`extern`
`static`

如果函数定义不包括 storage-class-specifier，则存储类默认为 `extern`。可以将函数显式声明为 `extern`，但这不是必需的。

如果函数的声明包含 storage-class-specifier `extern`，则标识符的链接与带文件范围的标识符的任何可见声明相同。如果没有带文件范围的可见声明，则标识符具有外部链接。如果标识符具有文件范围但没有 storage-class-specifier，则标识符具有外部链接。外部链接意味着，标识符的每个实例表示相同的对象或函数。有关链接和文件范围的详细信息，请参阅生存期、范围、可见性和链接。

包含除了 `extern` 之外的存储类说明符的块范围函数声明会生成错误。

包含 `static` 存储类的函数只在定义它的源文件中可见。其他所有函数(无论是显式还是隐式地为它们提供 `extern` 存储类)在程序中的所有源文件中都可见。如果需要 `static` 存储类，则必须在函数的第一个声明(若有)和函数定义中声明它。

**Microsoft 专用**

在 Microsoft 扩展启用后，如果函数定义在同一源文件中，且定义显式指定 `static` 存储类，则会为最初没有使用存储类(或使用 `extern` 存储类)声明的函数提供 `static` 存储类。

当使用 /Ze 编译器选项进行编译时，在块内使用 `extern` 关键字声明的函数具有全局可见性。在使用 /Za 编译时，并非如此。如果需要考虑源代码的可移植性，则不应依赖此功能。

**结束 Microsoft 专用**

## 请参阅

C 函数定义

# 返回类型

2021/8/13 •

函数的返回类型建立由该函数返回的值的大小和类型，并与以下语法中的 type-specifier 相对应：

## 语法

function-definition：
*declaration-specifiers*$_{opt}$ *attribute-seq*$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

/* *attribute-seq* is Microsoft-specific */

*declaration-specifiers*:
*storage-class-specifier declaration-specifiers*$_{opt}$
*type-specifier declaration-specifiers*$_{opt}$
*type-qualifier declaration-specifiers*$_{opt}$

*type-specifier*:
`void`
`char`
`short`
`int`
`__int8` /* 特定于 Microsoft */
`__int16` /* 特定于 Microsoft */
`__int32` /* 特定于 Microsoft */
`__int64` /* 特定于 Microsoft */
`long`
`float`
`double`
`signed`
`unsigned`
*struct-or-union-specifier*
*enum-specifier*
*typedef-name*

type-specifier 可以指定任何基本、结构或联合类型。如果不包括 type-specifier，则假定返回类型为 `int`。

函数定义中给定的返回类型必须与程序中其他位置的函数声明中的返回类型相匹配。当执行包含表达式的 `return` 语句时，函数返回值。计算该表达式，转换为返回值类型（如果需要）并返回到调用函数的点。如果函数是使用返回类型 `void` 进行声明，则包含表达式的返回语句会生成警告，并且表达式不会被计算。

以下示例阐释函数返回值。

```
typedef struct
{
    char name[20];
    int id;
    long class;
} STUDENT;

/* Return type is STUDENT: */

STUDENT sortstu( STUDENT a, STUDENT b )
{
    return ( (a.id < b.id) ? a : b );
}
```

此示例使用 `typedef` 声明定义了 `STUDENT` 类型，并将函数 `sortstu` 定义为包含 `STUDENT` 返回类型。函数选择并返回其两个结构参数之一。在对函数的后续调用中，编译器会检查以确保参数类型是 `STUDENT`。

> **NOTE**
> 通过传递指向结构的指针而不是整个结构来提高效率。

```
char *smallstr( char s1[], char s2[] )
{
    int i;

    i = 0;
    while ( s1[i] != '\0' && s2[i] != '\0' )
        i++;
    if ( s1[i] == '\0' )
        return ( s1 );
    else
        return ( s2 );
}
```

此示例定义了一个返回指向字符数组的指针的函数。该函数采用两个字符数组（字符串）作为参数，并返回指向两个字符串中较短的一个字符串的指针。指向数组的指针指向数组中的第一个元素，并具有其类型；因此，函数的返回类型是指向类型 `char` 的指针。

在调用函数之前，不需要使用 `int` 返回类型来声明函数，但建议使用原型，以便启用对参数和返回值的正确类型检查。

## 请参阅

C 函数定义

# 参数

2021/8/13 •

自变量是通过函数调用传递到函数的值的名称。形参是函数期望接收的值。在函数原型中，函数名称后的括号包含函数的参数及其类型的完整列表。参数声明指定参数中存储的值的类型、大小和标识符。

## 语法

*function-definition*:
  declaration-specifiers opt attribute-seq opt declarator declaration-list opt compound-statement

/* attribute-seq 是特定于 Microsoft 的 */

*declarator*:
  pointer opt direct-declarator

direct-declarator: /* 函数声明符 */
  direct-declarator ( parameter-type-list ) /* 新样式声明符 */
  direct-declarator ( identifier-list opt ) /* 已过时样式声明符 */

parameter-type-list: /* 参数列表 */
  *parameter-list*
  *parameter-list* , ...

*parameter-list*:
  *parameter-declaration*
  *parameter-list* , *parameter-declaration*

*parameter-declaration*:
  *declaration-specifiers* *declarator*
  declaration-specifiers abstract-declarator opt

parameter-type-list 是以逗号分隔的参数声明序列。参数列表中的每个参数的格式如下所示：

```
register opt type-specifier declarator opt
```

使用 auto 特性声明的函数参数生成错误。参数的标识符在函数体中使用以引用传递给函数的值。您可以在原型中命名参数，但名称会超出声明的末尾的范围。也就是说，可以在函数定义中以相同或不同的方式分配参数名称。这些标识符不能在函数主体的最外层块中重新定义，但它们可在内部的嵌套块中重新定义，就像参数列表是封闭块一样。

parameter-type-list 中的每个标识符前面都必须有适当的类型说明符，如下面的示例所示：

```
void new( double x, double y, double z )
{
    /* Function body here */
}
```

如果至少有一个参数出现在参数列表中，此列表的结尾可以是一个逗号后跟三个句点 ( , ... )。此构造称为"省略号表示法"，表示函数的可变数量的自变量。（有关详细信息，请参阅参数数量可变的调用。）但是，对函数进行调用时，自变量的数量必须至少与最后一个逗号前面的参数的数量相同。

如果没有参数要传递给函数，那么参数列表将被关键字 `void` 替换。`void` 的这种用法与作为类型说明符的用法不同。

参数的顺序和类型（包括省略号表示法的任何用法）在所有函数声明（如果有）和函数定义中都必须相同。进行常用算术转换后，自变量的类型与对应参数的类型必须是赋值兼容的。（有关算术转换的信息，请参阅常用算术转换。）不检查省略号后面的参数。参数可以具有任何基础、结构、联合、指针或数组类型。

如果需要，编译器将独立于每个参数和每个自变量执行常用算术转换。转换后，没有参数短于 `int`，且没有参数的类型为 `float`，除非参数类型在原型中被显式指定为 `float`。也就是说，例如，将参数声明为 `char` 与声明为 `int` 的效果相同。

## 请参阅

C 函数定义

# 函数体

2021/8/16 •

**函数体** 是包含指定函数行为的语句的复合语句。

## 语法

function-definition：

*declaration-specifiers*$_{opt}$ *attribute-seq*$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

/* *attribute-seq* is Microsoft-specific */

compound-statement：/* 函数体 */

{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

除非另有说明，否则在函数主体中声明的变量（称为“局部变量”）包含 `auto` 存储类。调用函数时，将为局部变量创建存储并执行本地初始化。执行控制权传递给 compound-statement 中的第一个语句，并继续传递，直到执行了 `return` 语句或到达函数主体的末尾。控制权随后返回到调用功能的点。

如果函数要返回值，则必须执行包含表达式的 `return` 语句。如果没有执行 `return` 语句或 `return` 语句不包含表达式，则函数的返回值是未定义的。

## 请参阅

C 函数定义

# 函数原型

2021/8/14 •

函数声明位于函数定义之前，用来指定函数的名称、返回类型、存储类和其他特性。若要作为原型，函数声明还必须为函数的参数确定类型和标识符。

## 语法

*declaration*:
  *declaration-specifiers attribute-seq*$_{opt}$ *init-declarator-list*$_{opt}$ ;

/* *attribute-seq*$_{opt}$ is Microsoft-specific */

*declaration-specifiers*:
  *storage-class-specifier declaration-specifiers*$_{opt}$
  *type-specifier declaration-specifiers*$_{opt}$
  *type-qualifier declaration-specifiers*$_{opt}$

*init-declarator-list*:
  *init-declarator*
  *init-declarator-list* , *init-declarator*

*init-declarator*:
  *declarator*
  *declarator* = *initializer*

*declarator*:
  *pointer*$_{opt}$ *direct-declarator*

direct-declarator : /* 函数声明符 */
  *direct-declarator* ( *parameter-type-list* ) /* New-style declarator */
  *direct-declarator* ( *identifier-list*$_{opt}$ ) /* Obsolete-style declarator */

原型与函数定义具有相同的形式，只不过前者由紧跟在右括号后的分号结尾，因此没有主体。在任一情况下，返回类型都必须与函数定义中指定的返回类型一致。

函数原型有下列重要用途：

- 它们为返回除 `int` 以外的类型的函数建立返回类型。尽管返回 `int` 值的函数不需要原型，但仍建议使用原型。
- 如果没有完整原型，将进行标准转换，但不会尝试使用形参的数量检查实参的类型或数量。
- 原型用于在定义函数之前初始化指向函数的指针。
- 形参列表用于通过函数定义中的形参来检查函数调用中的实参的对应性。

每个形参的转换类型决定函数调用在堆栈上放置的实参的解释。自变量和参数的类型不匹配可能导致堆栈上的自变量被错误解释。例如，在 16 位计算机上，如果 16 位指针先作为实参传递，再声明为 `long` 形参，那么堆栈上的前 32 位将解释为 `long` 形参。此错误不仅会导致 `long` 参数出现问题，而且还会导致其后的所有参数都出现问题。您可通过声明所有函数的完整函数原型来检测此类错误。

原型将确定函数的特性，以便能检查位于函数定义前面（或者出现在其他源文件中）的函数调用是否存在参数类型和返回类型不匹配。例如，如果在原型中指定 `static` 存储类说明符，还必须在函数定义中指定 `static` 存储类。

完整参数声明 ( `int a` ) 可以与抽象声明符 ( `int` ) 在同一个声明中混合使用。例如，以下声明是合法的：

```
int add( int a, int );
```

原型可同时包含作为参数传递的每个表达式的类型和标识符。但是，此类标识符的范围只到该声明的末尾。原型也可以反映参数的数量是变量或未传递参数的事实。如果没有此类列表，则不能显示不匹配项，从而使编译器无法生成有关它们的诊断消息。有关类型检查的详细信息，请参阅参数。

使用 /Za 编译器选项进行编译时，Microsoft C 编译器中的原型范围现在符合 ANSI。也就是说，如果你在原型中声明 `struct` 或 `union` 标记，那么标记是在此范围（而不是全局范围）输入。例如，为符合 ANSI 而使用 /Za 进行编译时，绝不可能调用此函数而不遇到类型不匹配错误：

```
void func1( struct S * );
```

若要更正代码，请在函数原型之前在全局范围定义或声明 `struct` 或 `union` ：

```
struct S;
void func1( struct S * );
```

在 /Ze 下，仍将在全局范围内输入标记。

## 请参阅

函数

# 函数调用

2021/8/13 •

函数调用 是一个用于将控制权和参数(如果有)传递给函数的表达式，格式如下：

*expression* (*expression-list*$_{opt}$)

其中 expression 是一个函数名称或者其计算结果为函数地址，而 expression-list 是表达式的列表(以逗号分隔)。后面这些表达式的值是传递给函数的自变量。如果函数没有返回值，则将它声明为返回 `void` 的函数。

如果在函数调用之前存在声明，但未提供任何与参数有关的信息，则任何未声明的参数只需进行常用算术转换。

> **NOTE**
>
> 函数参数列表中的表达式可以任何顺序进行计算，因此其值可能受其他参数的副作用而更改的参数具有未定义的值。函数调用运算符定义的序列点仅保证在将控制权传递给所调用函数之前计算参数列表中的所有副作用。(请注意，在堆栈上推送自变量的顺序是另一回事。)有关详细信息，请参阅**序列点**。

所有函数调用中的唯一要求是，括号前的表达式的计算结果必须是函数地址。这意味着可通过任何函数指针表达式调用函数。

## 示例

此示例展示了通过 `switch` 语句调用的函数调用：

```
int main()
{
    /* Function prototypes */

    long lift( int ), step( int ), drop( int );
    void work( int number, long (*function)(int i) );

    int select, count;
    .
    .
    .
    select = 1;
    switch( select )
    {
        case 1: work( count, lift );
                break;

        case 2: work( count, step );
                break;

        case 3: work( count, drop );
                /* Fall through to next case */
        default:
                break;
    }
}

/* Function definition */

void work( int number, long (*function)(int i) )
{
    int i;
    long j;

    for ( i = j = 0; i < number; i++ )
            j += ( *function )( i );
}
```

在此示例中，`main` 中的函数调用

```
work( count, lift );
```

将整数变量 `count` 和函数 `lift` 的地址传递给函数 `work`。请注意，由于函数标识符的计算结果是一个针表达式，因此通过提供函数标识符即可传入函数地址。若要通过此方式使用函数标识符，必须在使用标识符前先声明或定义函数；否则，将不会识别标识符。在此示例中，`work` 的原型在 `main` 函数的开头处提供。

`work` 中的参数 `function` 被声明为指向函数的指针，此函数接受一个 `int` 参数，并返回 `long` 值。参数名称两边的括号是必需的；如果没有括号，声明会指定函数，此函数返回指向 `long` 值的指针。

函数 `work` 使用以下函数调用从 `for` 循环内部调用所选函数：

```
( *function )( i );
```

一个参数 `i` 将传递给所调用的参数。

## 请参阅

函数

# 自变量

2021/8/14 •

函数调用中的自变量具有此形式：

```
expression ( expression-list opt ) /* Function call */
```

在函数调用中，expression-list 是表达式的列表（用逗号分隔）。后面这些表达式的值是传递给函数的自变量。如果函数不接受参数，expression-list 应包含关键字 `void`。

自变量可以是具有基本、结构、联合或指针类型的任何值。通过值传递所有参数。这意味着，参数的副本将分配给对应的参数。该函数不了解已传递的参数的实际内存位置。该函数使用此副本，而不影响其最初派生自的变量。

虽然您无法将数组或函数作为参数传递，但可以将指针传递给这些项。利用指针，函数可以通过引用访问值。由于指向变量的指针包含该变量的地址，因此该函数可以使用此地址访问该变量的值。指针参数允许函数访问数组和函数，即使数组和函数不能作为参数传递。

计算参数的顺序因不同的编译器和不同的优化级别而异。但是，在输入函数之前，将完全计算自变量和任何副作用。有关副作用的信息，请参阅副作用。

计算函数调用中的 expression-list，并在函数调用中对每个参数执行常用算术转换。如果原型可用，则生成的自变量类型将与原型的对应参数进行比较。如果这些参数不匹配，则执行转换或发布诊断消息。参数还执行常用算术转换。

除非函数的原型或定义显式指定实参的变量数目，否则 expression-list 中的表达式的数目必须与形参的数目匹配。在这种情况下，编译器将检查与参数列表中的类型名称一样多的自变量，并根据需要转换这些自变量，如上所述。有关详细信息，请参阅使用可变数量的参数进行调用。

如果原型的形参列表只包含关键字 `void`，则编译器需要函数调用中没有实参，且定义中没有形参。如果编译器找到任何参数，则会发出该诊断消息。

## 示例

此示例将指针用作参数：

```
int main()
{
    /* Function prototype */

    void swap( int *num1, int *num2 );
    int x, y;
    .
    .
    .
    swap( &x, &y );  /* Function call */
}

/* Function definition */

void swap( int *num1, int *num2 )
{
    int t;

    t = *num1;
    *num1 = *num2;
    *num2 = t;
}
```

在此示例中，`swap` 函数在 `main` 中声明为包含两个分别由标识符 `num1` 和 `num2` 表示的参数，这两个参数都是指向 `int` 值的指针。原型样式定义中的参数 `num1` 和 `num2` 也被声明为指向 `int` 类型值的指针。

在函数调用中

```
swap( &x, &y )
```

`x` 的地址存储在 `num1` 中，而 `y` 的地址存储在 `num2` 中。现在，这两个名称或"别名"存在于同一个位置。对 `*num1` 中的 `*num2` 和 `swap` 的引用是对 `x` 中的 `y` 和 `main` 的有效引用。`swap` 中的赋值实际交换了 `x` 和 `y` 的内容。因此，不需要 `return` 语句。

编译器对 `swap` 的参数执行类型检查，因为 `swap` 的原型包含每个参数的参数类型。原型和定义的括号内的标识符可能相同，也可能不同。重要的一点是，自变量的类型与原型和定义中的这些参数列表相匹配。

## 请参阅

函数调用

# 使用数目可变的自变量调用

2021/8/13 •

部分参数列表可由省略号表示法(一个逗号后跟三个句点 (, ...) 终止，以指示可能有多个自变量传递给函数，但没有有关这些自变量的详细信息。对此类自变量不执行类型检查。省略号表示法前面必须至少有一个参数，并且省略号表示法必须是参数列表中的最后一个标记。如果没有省略号表示法，当函数收到除参数列表中声明的参数以外的参数时，该函数的行为是不确定的。

若要调用具有可变数量的参数的函数，只需在函数调用中指定任意数量的参数即可。一个示例是 C 运行库中的 `printf` 函数。函数调用必须包含参数列表或参数类型列表中声明的每个类型名称的一个参数。

除非指定了 `__fastcall` 调用约定，否则函数调用中指定的所有参数都被放在堆栈上。为函数声明的形参的数量决定了从堆栈获取和分配给形参的实参的数量。您负责从堆栈中检索任何其他参数和确定应存在的参数数量。STDARG.H 文件包含 ANSI 样式宏，该宏用于访问采用可变数量的参数的函数的参数。此外，VARARGS.H 中的 XENIX 样式宏仍受支持。

以下示例声明用于调用可变数量的自变量的函数：

```
int average( int first, ...);
```

## 请参阅

函数调用

# 递归函数

2021/8/16 •

C 程序中的任何函数都可以以递归方式调用；也就是说，函数可以调用自己。递归调用的数量受堆栈的大小的限制。若要了解设置堆栈大小的链接器选项，请参阅 `/STACK`（堆栈分配）链接器选项。每次调用函数时，都会为参数以及 `auto` 和 `register` 变量分配新存储，这样它们在以前未完成的调用中的值就不会被覆盖。只有从中创建参数的函数的实例才能直接访问该参数。前面的参数对函数的后续实例不可直接访问。

请注意，使用 `static` 存储声明的变量在每次递归调用时不需要新存储。它们的存储在程序的生存期内存在。每次引用此类变量时都将访问相同的存储区域。

## 示例

本示例演示了递归调用：

```
int factorial( int num );      /* Function prototype */

int main()
{
    int result, number;
    .
    .
    .
    result = factorial( number );
}

int factorial( int num )      /* Function definition */
{
    .
    .
    .
    if ( ( num > 0 ) || ( num <= 10 ) )
        return( num * factorial( num - 1 ) );
}
```

## 请参阅

函数调用

# C 语言语法摘要

2021/8/11 •

本节提供 C 语言和 Microsoft 特定的 C 语言功能的完整说明。可以使用本节中的语法表示法，确定任意语言组件的确切语法。该语法的说明将显示在本手册讨论主题的节中。

> **NOTE**
> 此语法摘要不是 ANSI C 标准的一部分，但包括在此仅用作信息参考。Microsoft 特定的语法在该语法后面的注释中进行了注明。

## 请参阅

C 语言参考

# 定义和约定

2021/8/16 •

终止符是语法定义中的终结点。不提供其他解决方法。终止符包括保留字和用户定义的标识符的集。

非终止符是语法中的占位符并在此语法摘要中的其他位置进行定义。定义可是递归的。

可选组件由带下标的 opt 指示。例如，应用于对象的

> { *expression*$_{opt}$ }

指示包含在大括号中的可选表达式。

语法约定对语法的不同组件使用不同的字体特性。符号和字体如下所示：

| " | " |
|---|---|
| nonterminal | 斜体类型指示非终止符。 |
| `const` | 粗体类型的终止符是必须按所示方式输入的文本保留字和符号。此上下文中的字符始终区分大小写。 |
| opt | 后跟 opt 的非终止符始终是可选的。 |
| default typeface | 用此字样描述或列出的集中的字符可在 C 语句中用作终止符。 |

跟在非终止符之后的冒号 (:) 引入其定义。替代定义将在单独的行中列出（以单词"one of"开头的情况除外）。

## 请参阅

C 语言语法摘要

# C 词法语法

2021/8/12 •

## 令牌

*token*:

- *keyword*
- *identifier*
- *constant*
- *string-literal*
- *punctuator*

*preprocessing-token*:

- *header-name*
- *identifier*
- *pp-number*
- *character-constant*
- *string-literal*
- *punctuator*

不能成为上述项目之一的每个非空白字符

## 关键字

*keyword*：以下之一

auto break case char const continue
default do double else enum extern
float for goto if inline int long
register restrict return short signed
sizeof static struct switch typedef union
unsigned void volatile while _Alignas
_Alignof _Atomic _Bool _Complex _Generic
_Imaginary _Noreturn _Static_assert
_Thread_local

有关其他 Microsoft 专用的关键字的列表，请参阅 C 关键字。

## 标识符

*identifier*:

- *identifier-nondigit*
- *identifier* *identifier-nondigit*
- *identifier* *digit*

*identifier-nondigit*:

- *nondigit*
- *universal-character-name*

其他实现定义的字符

*nondigit*：以下之一

`_` `a` `b` `c` `d` `e` `f` `g` `h` `i` `j` `k` `l` `m`
`n` `o` `p` `q` `r` `s` `t` `u` `v` `w` `x` `y` `z`
`A` `B` `C` `D` `E` `F` `G` `H` `I` `J` `K` `L` `M`
`N` `O` `P` `Q` `R` `S` `T` `U` `V` `W` `X` `Y` `Z`

*digit* : 以下之一
`0` `1` `2` `3` `4` `5` `6` `7` `8` `9`

*universal-character-name*:
`\u` *hex-quad*
`\U` *hex-quad* *hex-quad*

*hex-quad*:
*hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit*

# 常量

*constant*:
*integer-constant*
*floating-constant*
*enumeration-constant*
*character-constant*

*integer-constant*:
*decimal-constant* *integer-suffix* opt\ *binary-constant* [1] *integer-suffix* opt
*octal-constant* *integer-suffix* opt\ *hexadecimal-constant* *integer-suffix* opt

*decimal-constant*:
*nonzero-digit*
*decimal-constant* *digit*

*binary-constant* :[1]
*binary-prefix* *binary-digit*
*binary-constant* *binary-digit*

*binary-prefix* [1] : 以下之一
`0b` `0B`

*binary-digit* [1] : 以下之一
`0` `1`

*octal-constant*:
`0`
*octal-constant* *octal-digit*

*hexadecimal-constant*:
*hexadecimal-prefix* *hexadecimal-digit*
*hexadecimal-constant* *hexadecimal-digit*

*hexadecimal-prefix* : 以下之一
`0x` `0X`

*nonzero-digit* : 以下之一
`1` `2` `3` `4` `5` `6` `7` `8` `9`

*octal-digit* : 以下之一

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

*hexadecimal-digit* : 以下之一

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

`a` `b` `c` `d` `e` `f`

`A` `B` `C` `D` `E` `F`

*integer-suffix* :

*unsigned-suffix* *Long-suffix* opt\
*unsigned-suffix* *Long-Long-suffix* opt\
*Long-suffix* *unsigned-suffix* opt\
*Long-Long-suffix* *unsigned-suffix* opt

*unsigned-suffix* : 以下之一

`u` `U`

*Long-suffix* : 以下之一

`l` `L`

*Long-Long-suffix* : 以下之一

`ll` `LL`

*floating-constant* :

*decimal-floating-constant*

*hexadecimal-floating-constant*

*decimal-floating-constant* :

*fractional-constant* *exponent-part* opt *floating-suffix* opt

*digit-sequence* *exponent-part* *floating-suffix* opt

*hexadecimal-floating-constant* :

*hexadecimal-prefix* *hexadecimal-fractional-constant* *binary-exponent-part* opt *floating-suffix* opt

*hexadecimal-prefix* *hexadecimal-digit-sequence* *binary-exponent-part* *floating-suffix* opt

*fractional-constant* :

*digit-sequence* opt `.` *digit-sequence*

*digit-sequence* `.`

*exponent-part* :

`e` *sign* opt *digit-sequence*\
`E` *sign* opt *digit-sequence*

*sign* : 以下之一

`+` `-`

*digit-sequence* :

*digit*

*digit-sequence* *digit*

*hexadecimal-fractional-constant* :

*hexadecimal-digit-sequence* opt `.` *hexadecimal-digit-sequence*

*hexadecimal-digit-sequence* `.`

*binary-exponent-part* :

`p` *sign* opt *digit-sequence*\
`P` *sign* opt *digit-sequence*

*hexadecimal-digit-sequence* :

*hexadecimal-digit*

*hexadecimal-digit-sequence* *hexadecimal-digit*

*floating-suffix* : 以下之一
  f l F L

*enumeration-constant*:
  *identifier*

*character-constant*:
  ' *c-char-sequence* '
  L' *c-char-sequence* '

*c-char-sequence*:
  *c-char*
  *c-char-sequence* *c-char*

*c-char*:
  除单引号 ( ' )、反斜杠 ( \ ) 或者换行符 \ 以外的所有源字符集成员 *escape-sequence*

*escape-sequence*:
  *simple-escape-sequence*
  *octal-escape-sequence*
  *hexadecimal-escape-sequence*
  *universal-character-name*

*simple-escape-sequence* : 以下之一
  \a \b \f \n \r \t \v
  \' \" \\ \?

*octal-escape-sequence*:
  \ *octal-digit*
  \ *octal-digit* *octal-digit*
  \ *octal-digit* *octal-digit* *octal-digit*

*hexadecimal-escape-sequence*:
  \x *hexadecimal-digit*
  *hexadecimal-escape-sequence* *hexadecimal-digit*

## 字符串文本

*string-literal*:
  *encoding-prefix* *s-char-sequence* opt "

*encoding-prefix*:
  u8
  u
  U
  L

*s-char-sequence*:
  *s-char*
  *s-char-sequence* *s-char*

*s-char*:
  除双引号 ( " )、反斜杠 ( \ ) 或者换行符 \ 以外的任何源字符集成员 *escape-sequence*

## 标点符号

*punctuator* : 以下之一

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

## 标头名称

*header-name*:

< *h-char-sequence* >
" *q-char-sequence* "

*h-char-sequence*:

*h-char*
*h-char-sequence* *h-char*

*h-char*:

除换行符和 > 以外的任何源字符集成员

*q-char-sequence*:

*q-char*
*q-char-sequence* *q-char*

*q-char*:

除换行符和 " 以外的任何源字符集成员

## 预处理数字

*pp-number*:

*digit*
. *digit*
*pp-number* *digit*
*pp-number* *identifier-nondigit*
*pp-number* e *sign*
*pp-number* E *sign*
*pp-number* p *sign*
*pp-number* P *sign*
*pp-number* .

[1] `binary-constant`、`binary-prefix` 和 `binary-digit` 是 Microsoft 专用的扩展。

## 请参阅

C 语言语法摘要

# 短语结构语法

2021/8/13 •

- 表达式
- 声明
- 语句
- 外部定义

## 请参阅

C 语言语法摘要

# 表达式摘要

2021/8/16 •

*primary-expression*:
- *identifier*
- *constant*
- *string-literal*
- ( *expression* )
- *generic-selection*

*generic-selection*:
- _Generic ( *assignment-expression* , *generic-assoc-list* )

*generic-assoc-list*:
- *generic-association*
- *generic-assoc-list* , *generic-association*

*generic-association*:
- *type-name* : *assignment-expression*
- default : *assignment-expression*

*postfix-expression*:
- *primary-expression*
- *postfix-expression* [ *expression* ]
- *postfix-expression* ( *argument-expression-list* opt )
- *postfix-expression* . *identifier*
- *postfix-expression* -> *identifier*
- *postfix-expression* ++
- *postfix-expression* --
- ( *type-name* ) { *initializer-list* }
- ( *type-name* ) { *initializer-list* , }

*argument-expression-list*:
- *assignment-expression*
- *argument-expression-list* , *assignment-expression*

*unary-expression*:
- *postfix-expression*
- ++ *unary-expression*
- -- *unary-expression*
- *unary-operator* *cast-expression*
- sizeof *unary-expression*
- sizeof ( *type-name* ) _Alignof ( *type-name* )

*unary-operator* : 以下之一
- & * + - ~ !

*cast-expression*:
- *unary-expression*
- ( *type-name* ) *cast-expression*

`multiplicative-expression`:

- `cast-expression`
- `multiplicative-expression` `*` `cast-expression`
- `multiplicative-expression` `/` `cast-expression`
- `multiplicative-expression` `%` `cast-expression`

`additive-expression`:

- `multiplicative-expression`
- `additive-expression` `+` `multiplicative-expression`
- `additive-expression` `-` `multiplicative-expression`

`shift-expression`:

- `additive-expression`
- `shift-expression` `<<` `additive-expression`
- `shift-expression` `>>` `additive-expression`

`relational-expression`:

- `shift-expression`
- `relational-expression` `<` `shift-expression`
- `relational-expression` `>` `shift-expression`
- `relational-expression` `<=` `shift-expression`
- `relational-expression` `>=` `shift-expression`

`equality-expression`:

- `relational-expression`
- `equality-expression` `==` `relational-expression`
- `equality-expression` `!=` `relational-expression`

`AND-expression`:

- `equality-expression`
- `AND-expression` `&` `equality-expression`

`exclusive-OR-expression`:

- `AND-expression`
- `exclusive-OR-expression` `^` `AND-expression`

`inclusive-OR-expression`:

- `exclusive-OR-expression`
- `inclusive-OR-expression` `|` `exclusive-OR-expression`

`logical-AND-expression`:

- `inclusive-OR-expression`
- `logical-AND-expression` `&&` `inclusive-OR-expression`

`logical-OR-expression`:

- `logical-AND-expression`
- `logical-OR-expression` `||` `logical-AND-expression`

`conditional-expression`:

- `logical-OR-expression`
- `logical-OR-expression` `?` `expression` `:` `conditional-expression`

`assignment-expression`:

- `conditional-expression`
- `unary-expression` `assignment-operator` `assignment-expression`

*assignment-operator* ：以下之一

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

*expression*：

*assignment-expression*

*expression* `,` *assignment-expression*

*constant-expression*：

*conditional-expression*

## 请参阅

- 短语结构语法

# 声明摘要

2021/8/16 •

*declaration*:

*declaration-specifiers* *attribute-seq* opt[1] *init-declarator-list* opt `;`

*static_assert-declaration*

*declaration-specifiers*:

*storage-class-specifier* *declaration-specifiers* opt\
*type-specifier* *declaration-specifiers* opt\
*type-qualifier* *declaration-specifiers* opt\
*function-specifier* *declaration-specifiers* opt\
*alignment-specifier* *declaration-specifiers* opt

*attribute-seq*[1]:

*attribute*[1] *attribute-seq* opt[1]

*attribute*[1, 2]: one of

`__asm` `__based` `__cdecl` `__clrcall` `__fastcall` `__inline` `__stdcall` `__thiscall` `__vectorcall`

*init-declarator-list*:

*init-declarator*

*init-declarator-list* `,` *init-declarator*

*init-declarator*:

*declarator*

*declarator* `=` *initializer*

*storage-class-specifier*:

`auto`

`extern`

`register`

`static`

`_Thread_local`

`typedef`

`__declspec` `(` *extended-decl-modifier-seq* `)`[1]

*extended-decl-modifier-seq*[1]:

*extended-decl-modifier* opt

*extended-decl-modifier-seq* *extended-decl-modifier*

*extended-decl-modifier*[1]:

`thread`

`naked`

`dllimport`

`dllexport`

*type-specifier*:

`void`

`char`

`short`

`int`

`__int8`[1]

`__int16` [1]
`__int32` [1]
`__int64` [1]
`long`
`float`
`double`
`signed`
`unsigned`
`_Bool`
`_Complex`
*`atomic-type-specifier`*
*`struct-or-union-specifier`*
*`enum-specifier`*
*`typedef-name`*

*`struct-or-union-specifier`*:
*`struct-or-union`* *`identifier`*opt `{` *`struct-declaration-list`* `}`
*`struct-or-union`* *`identifier`*

*`struct-or-union`*:
`struct`
`union`

*`struct-declaration-list`*:
*`struct-declaration`*
*`struct-declaration-list`* *`struct-declaration`*

*`struct-declaration`*:
*`specifier-qualifier-list`* *`struct-declarator-list`*opt `;` \ *`static_assert-declaration`*

*`specifier-qualifier-list`*:
*`type-specifier`* *`specifier-qualifier-list`*opt\ *`type-qualifier`* *`specifier-qualifier-list`*opt\
*`alignment-specifier`* *`specifier-qualifier-list`*opt

*`struct-declarator-list`*:
*`struct-declarator`*
*`struct-declarator-list`* `,` *`struct-declarator`*

*`struct-declarator`*:
*`declarator`*
*`declarator`*opt `:` *`constant-expression`*

*`enum-specifier`*:
`enum` *`identifier`*opt `{` *`enumerator-list`* `}`
`enum` *`identifier`*opt `{` *`enumerator-list`* `,` `}`
`enum` *`identifier`*

*`enumerator-list`*:
*`enumerator`*
*`enumerator-list`* `,` *`enumerator`*

*`enumerator`*:
*`enumeration-constant`*
*`enumeration-constant`* `=` *`constant-expression`*

*atomic-type-specifier*:

_Atomic ( *type-name* )

*type-qualifier*:

const

restrict

volatile

_Atomic

*function-specifier*:

inline

_Noreturn

*alignment-specifier*:

_Alignas ( *type-name* )

_Alignas ( *constant-expression* )

*declarator*:

*pointer* opt *direct-declarator*

*direct-declarator*:

*identifier*

( *declarator* )

*direct-declarator* [ *type-qualifier-list* opt *assignment-expression* opt ]

*direct-declarator* [ static *type-qualifier-list* opt *assignment-expression* ]

*direct-declarator* [ *type-qualifier-list* static *assignment-expression* ]

*direct-declarator* [ *type-qualifier-list* opt * ]

*direct-declarator* ( *parameter-type-list* )

*direct-declarator* ( *identifier-list* opt ) [3]

*pointer*:

* *type-qualifier-list* opt\ * *type-qualifier-list* opt *pointer*

*type-qualifier-list*:

*type-qualifier*

*type-qualifier-list* *type-qualifier*

*parameter-type-list*:

*parameter-list*

*parameter-list* , ...

*parameter-list*:

*parameter-declaration*

*parameter-list* , *parameter-declaration*

*parameter-declaration*:

*declaration-specifiers* *declarator*

*declaration-specifiers* *abstract-declarator* opt

identifier-list: /* 用于旧样式声明符 */

*identifier*

*identifier-list* , *identifier*

*type-name*:

*specifier-qualifier-list* *abstract-declarator* opt

*abstract-declarator*:
  *pointer*
  *pointer* opt *direct-abstract-declarator*

*direct-abstract-declarator*:
  ( *abstract-declarator* )
  *direct-abstract-declarator* [ *type-qualifier-list* opt *assignment-expression* opt ]
  *direct-abstract-declarator* [ static *type-qualifier-list* opt *assignment-expression* ]
  *direct-abstract-declarator* [ *type-qualifier-list* static *assignment-expression* ]
  *direct-abstract-declarator* [ *type-qualifier-list* opt * ]
  *direct-abstract-declarator* opt ( *parameter-type-list* opt )

*typedef-name*:
  *identifier*

*initializer*:
  *assignment-expression*
  { *initializer-list* }
  { *initializer-list* , }

*initializer-list*:
  *designation* opt *initializer*
  *initializer-list* , *designation* opt *initializer*

*designation*:
  *designator-list* =

*designator-list*:
  *designator*
  *designator-list* *designator*

*designator*:
  [ *constant-expression* ]
  . *identifier*

*static_assert-declaration*:
  _Static_assert ( *constant-expression* , *string-literal* ) ;

[1] 此语法元素是 Microsoft 专用的。
[2] 有关这些元素的详细信息，请参阅 `__asm`、`__clrcall`、`__stdcall`、`__based`、`__fastcall`、`__thiscall`、`__cdecl`、`__inline` 和 `__vectorcall`。[3] 此样式已过时。

# 另请参阅

调用约定
短语结构语法
已过时调用约定

# C 语句摘要

2021/8/11 •

*statement*:

- *labeled-statement*
- *compound-statement*
- *expression-statement*
- *selection-statement*
- *iteration-statement*
- *jump-statement*
- *try-except-statement* /* Microsoft 专用 */
- *try-finally-statement* /* Microsoft 专用 */

*jump-statement*:

- goto *identifier* ;
- continue ;
- break ;
- return *expression* opt ;
- __leave ; /* Microsoft 专用 [1] */

*compound-statement*:

- { *declaration-list* opt *statement-list* opt }

*declaration-list*:

- *declaration*
- *declaration-list* *declaration*

*statement-list*:

- *statement*
- *statement-list* *statement*

*expression-statement*:

- *expression* opt ;

*iteration-statement*:

- while ( *expression* ) *statement*
- do *statement* while ( *expression* ) ;
- for ( *expression* opt ; *expression* opt ; *expression* opt ) *statement*

*selection-statement*:

- if ( *expression* ) *statement*
- if ( *expression* ) *statement* else *statement*
- switch ( *expression* ) *statement*

*labeled-statement*:

- *identifier* : *statement*
- case *constant-expression* : *statement*
- default : *statement*

*try-except-statement* :/* Microsoft 专用 */

- __try *compound-statement* __except ( *expression* ) *compound-statement*

*try-finally-statement* :/* Microsoft 专用 */
    __try *compound-statement* __finally *compound-statement*

1 __leave 关键字仅在 try-except-statement 或 try-finally-statement 的 __try 块中有效。

## 请参阅

短语结构语法

# 外部定义

2021/8/16 •

translation-unit：
*external-declaration*
*translation-unit external-declaration*

external-declaration : /* 只允许在外部（文件）范围内 */
*function-definition*
*declaration*

function-definition : /* 此处的声明符是函数声明符 */
*declaration-specifiers*$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

## 请参阅

短语结构语法

# 实现定义的行为

2021/8/16 •

ANSI X3.159-1989，*American National Standard for Information Systems - 编程语言 - C*，包含"可移植性问题"节。ANSI 部分列出了 ANSI 针对每个特定实现开放的 C 语言区域。本节描述 Microsoft C 如何处理 C 语言的这些实现定义的区域。

本节遵循 ANSI 节所采用的同一顺序。所涵盖的每个项均包括对 ANSI 的引用，该引用说明实现定义的行为。

> **NOTE**
> 本节仅描述 C 编译器的美国英语版本。其他语言的 Microsoft C 的实现可能略有不同。

## 请参阅

C 语言参考

# 翻译：诊断

2021/8/13 •

**ANSI 2.1.1.3 如何标识诊断**

Microsoft C 生成以下形式的错误消息：

> *filename*(*line-number*)：*diagnostic* C *number message*

其中，filename 是出现错误的源文件的名称；line-number 是编译器在其中检测到错误的行号；diagnostic 是"错误"或"警告"；number 是标识错误或警告的唯一的四位数（前面带有 C，如语法中所注明的）；message 是解释性消息。

## 请参阅

实现定义的行为

# 环境

2021/8/12 •

- 要保留的参数
- 交互式设备

## 请参阅

实现定义的行为

# 要保留的自变量

2021/8/15 •

**ANSI 2.1.2.2.1**：main 参数的语义

在 Microsoft C 中，程序启动时调用的函数称为 **main**。没有针对 **main** 声明的原型，可以用零个、两个或三个参数对其进行定义：

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

上面第三行（其中，**main** 接受三个参数）是 Microsoft ANSI C 标准扩展。第三个参数 (**envp**) 是指向环境变量的指针的数组。**envp** 数组以 null 指针终止。有关 **main** 和 **envp** 的详细信息，请参阅 main 函数和程序执行。

变量 **argc** 从不保留负值。

字符串数组以包含 null 指针的 **argv[argc]** 结束。

**argv** 数组的所有元素都是指向字符串的指针。

未使用命令行参数调用的程序将接收 **argc** 的值 1，因为可执行文件的名称将置于 **argv[0]** 内。（在 3.0 之前的 MS-DOS 版本中，可执行文件名不可用。字母“C”将置于 **argv[0]** 中。）**argv[1]** 通过 **argv[argc – 1]** 指向的字符串表示程序参数。

参数 **arg** 和 **argv** 是可修改的，并在程序启动与程序终止之间保留最后存储的值。

## 请参阅

环境

# 交互式设备

2021/8/12 •

**ANSI 2.1.2.3 构成交互式设备的组件**

Microsoft C 将键盘和显示屏定义为交互式设备。

## 请参阅

环境

# 标识符的行为

2021/8/17 •

- 没有外部链接的重要字符
- 带有外部链接的重要字符
- 大写和小写

## 请参阅

使用 extern 指定链接

# 没有外部链接的重要字符

2021/8/12 •

**ANSI 3.1.2 不具有外部链接的重要字符的数量**

标识符对 247 个字符是有意义的。编译器不限制可以在标识符中使用的字符数；它只会忽略限制之外的任何字符。

## 请参阅

使用 extern 指定链接

# 带有外部链接的重要字符

2021/8/17 •

**ANSI 3.1.2 具有外部链接的重要字符的数量**

在使用 Microsoft C 编译的程序中，声明为 `extern` 的标识符对 247 个字符很重要。您可使用 /H（限制外部名称的长度）选项将此默认值修改为较小的数字。

## 请参阅

使用 extern 指定链接

# 大写和小写

2021/8/13 •

**ANSI 3.1.2 大小写区别是否重要**

Microsoft C 将编译单元中的标识符视为区分大小写。

Microsoft 链接器区分大小写。您必须根据情况一致地指定所有标识符。

## 请参阅

标识符的行为

# 字符

2021/8/16 •

- ASCII 字符集
- 多字节字符
- 每字符的位数
- 字符集
- 无代表的字符常量
- 宽字符
- 转换多字节字符
- 字符值的范围

## 请参阅

实现定义的行为

# ASCII 字符集

2021/8/17 •

**ANSI 2.2.1 源和执行字符集的成员**

**源字符集是可以出现在源文件中的合法字符集。对于 Microsoft C，源字符集是标准 ASCII 字符集。**

> **NOTE**
>
> ■ 由于键盘和控制台驱动程序可以重新映射字符集，因此针对国际分发的程序应检查国家/地区代码。

## 请参阅

字符

# 多字节字符

2021/8/13 •

**ANSI 2.2.1.2 多字节字符的移位状态**

多字节字符由某些实现(包括 Microsoft C)用来表示基字符集中未表示的外语字符。但是，Microsoft C 不支持任何依赖于状态的编码。因此，没有移位状态。有关详细信息，请参阅多字节字符和宽字符。

## 请参阅

字符

# 每字符的位数

2021/8/15 •

**ANSI 2.2.4.2.1 字符中的位数**

**字符中的位数由清单常量 CHAR_BIT 表示。LIMITS.H 文件将 CHAR_BIT 定义为 8。**

## 请参阅

字符

# 字符集

2021/8/12 •

**ANSI 3.1.3.4 源字符集的映射成员**

**源字符集和执行字符集包含下表中所列的 ASCII 字符。该表中也显示了转义序列。**

## 转义序列

| [illegible] | [illegible] | ASCII [illegible] |
|---|---|---|
| \a | 提醒/响铃 | 7 |
| \b | Backspace | 8 |
| \f | 换页 | 12 |
| \n | 换行符 | 10 |
| \r | 回车 | 13 |
| \t | 水平制表符 | 9 |
| \v | 垂直制表符 | 11 |
| \" | 双引号 | 34 |
| \' | 单引号 | 39 |
| \\ | 反斜杠 | 92 |

## 请参阅

字符

# 无代表的字符常量

2021/8/17 •

**ANSI 3.1.3.4 包含没有以宽字符常量的基本执行字符集或扩展字符集表示的字符或转义序列的整数字符常量的值**

**所有字符常量或转义序列均可以用扩展字符集表示。**

## 请参阅

字符

# 宽字符

2021/8/11 •

**ANSI 3.1.3.4 包含多个字符的整数字符常量的值或包含多个多字节字符的宽字符常量的值**

常规字符常量"ab"具有整数值 (int)0x6162。当存在多个字节时，以前读取的字节将按 CHAR_BIT 的值向左移位，并使用具有低 CHAR_BIT 位的按位"或"运算符比较下一个字节。多字节字符常量中的字节数不能超过 sizeof(int)（对于 32 位目标代码，为 4）。

如上所述读取多字节字符常量，并使用 `mbtowc` 运行时函数将其转换为宽字符常量。如果结果不是有效的宽字符常量，则将发出错误。在任何情况下，`mbtowc` 函数检查的字节数限制为 `MB_CUR_MAX` 的值。

## 请参阅

字符

# 转换多字节字符

2021/8/15 •

**ANSI 3.1.3.4 用于将多字节字符转换为宽字符常量的相应宽字符(代码)的当前区域设置**

**默认情况下，当前区域设置为"C"区域设置。可以使用 #pragma setlocale 更改该设置。**

## 请参阅

字符

# 字符值的范围

2021/8/15 •

ANSI 3.2.1.1“普通” `char` 的值范围是否与 `signed char` 或 `unsigned char` 相同

所有带符号的字符值的范围都介于 -128 和 127 之间。所有无符号的字符值的范围介于 0 和 255 之间。

`/J` 编译器选项将 `char` 的默认类型从 `signed char` 更改为 `unsigned char`。

## 请参阅

字符

# 整数

2021/8/15 •

- 整数值的范围
- 整数的降级
- 带符号的按位运算
- 余数
- 右移

## 请参阅

实现定义的行为

# 整数值的范围

2021/8/13 •

**ANSI 3.1.2.5：各种类型的整数值的表示形式和集**

**整数包含 32 位（4 个字节）。带符号整数以 2 的补数的形式表示。最高有效位保留符号：1 表示负数，0 表示正数和零。下面列出了这些值：**

| tt | ttttttt |
| --- | --- |
| `unsigned short` | 0 到 65535 |
| `signed short` | -32768 到 32767 |
| `unsigned long` | 0 到 4294967295 |
| `signed long` | -2147483648 到 2147483647 |

## 请参阅

整数

# 整数的降级

2021/8/13 •

**ANSI 3.2.1.2**：在值无法表示的情况下，将整数转换为较短的带符号整数的结果，或者将无符号整数转换为同等长度的带符号整数的结果

如果 `long` 整数被强制转换为 `short`，或 `short` 被强制转换为 `char`，则保留最低有效字节。

例如，此行

```
short x = (short)0x12345678L;
```

将值 0x5678 赋给 `x`，此行

```
char y = (char)0x1234;
```

将值 0x34 赋给 `y`。

如果 `signed` 变量转换为 `unsigned` 变量（反之亦然），位模式保持不变。例如，将 -2 (0xFE) 强制转换为 `unsigned` 值将得到 254（也是 0xFE）。

## 请参阅

整数

# 带符号的按位运算

2021/8/17 •

**ANSI 3.3**：对带符号整数进行的按位运算的结果

对带符号整数进行的按位运算的工作方式与对无符号整数进行的按位运算的工作方式相同。例如，`-16 & 99` 可用二进制格式表示

```
  11111111 11110000
& 00000000 01100011
  ________________
  00000000 01100000
```

按位“与”的结果为 96。

## 请参阅

整数

# 余数

2021/8/16 •

**ANSI 3.3.5 整除的余数的符号**

**余数的符号与被除数的符号相同。例如，应用于对象的**

```
50 / -6 == -8
50 % -6 ==  2
-50 /  6 == -8
-50 %  6 == -2
```

## 请参阅

整数

# 右移

2021/8/16 •

带符号的整型负值右移位的结果

将负值向右移位可生成绝对值的一半(向下舍入)。例如，`signed short` 值 -253(十六进制数是 0xFF03，二进制数是 11111111 00000011)右移一位会生成 -127(十六进制数是 0xFF81，二进制数是 11111111 10000001)。正 253 向右移位生成 +126。

右移保留带符号的整数类型的符号位。当带符号的整数向右移位时，最高有效位将保留。例如，如果 0xF0000000 是带符号 `int`，则右移会生成 0xF8000000。将负数 `int` 右移 32 次会生成 0xFFFFFFFF。

当无符号的整数右移位时，将清除最高有效位。例如，如果 0xF000 是无符号的，则结果为 0x7800。将 `unsigned` 或正数 `int` 右移 32 次会生成 0x00000000。

## 请参阅

整数

# 浮点数学

2021/8/16 •

- 值
- 将整数转换为浮点值
- 浮点值的截断

## 请参阅

实现定义的行为

# 值

2021/8/11 •

**ANSI 3.1.2.5 各种类型的浮点数的表示形式和值集**

`float` 类型包含 32 位：1 位用于符号，8 位用于指数，23 位用于尾数。其范围为精度至少为 7 个数字的 +/- 3.4E38。

`double` 类型包含 64 位：1 位用于符号，11 位用于指数，52 位用于尾数。其范围为精度至少为 15 个数字的 +/- 1.7E308。

`long double` 类型是不同的，但其表示形式与 Microsoft C 编译器中的 `double` 类型相同。

## 请参阅

**浮点数学**

# 将整数转换为浮点值

2021/8/13 •

**ANSI 3.2.1.3 当一个整数转换为无法确切表示原始值的浮点数时的截断方向**

**当一个整数转换为无法确切表示值的浮点值时，值将被舍入(向上或向下)到最接近的适当值。**

**例如，将** `unsigned long` **(精度为 32 位)强制转换为** `float` **(尾数的精度为 23 位)会将数字舍入为 256 的最接近倍数。介于 4,294,966,913 和 4,294,967,167 之间的所有** `long` **值都舍入为** `float` **值 4,294,967,040。**

## 请参阅

**浮点数学**

# 浮点值的截断

2021/8/11 •

**ANSI 3.2.1.4 在某个浮点数转换为较小的浮点数时截断或舍入的方向**

**当出现下溢时，浮点变量的值将舍入为 0。溢出可能导致运行时错误，也可能生成不可预知的值，具体取决于指定的优化。**

## 请参阅

**浮点数学**

# 数组和指针

2021/8/15 •

- 最大数组大小
- 指针减法

## 请参阅

实现定义的行为

# 最大数组大小

2021/8/13 •

**ANSI 3.3.3.4、4.1.1 保存数组的最大大小所需的整数类型，即 size_t 的大小**

在 32 位 x86 平台上，`size_t` typedef 是 `unsigned int`。在 64 位平台上，`size_t` typedef 是 `unsigned __int64`。

## 请参阅

**数组和指针**

# 指针减法

2021/8/11 •

**ANSI 3.3.6, 4.1.1 保留两个指向同一数组 ptrdiff_t 的元素的指针之间的差所需的整数类型**

在 32 位 x86 平台上，`ptrdiff_t` typedef 是 `int`。在 64 位平台上，`ptrdiff_t` typedef 是 `__int64`。

## 请参阅

**数组和指针**

# 寄存器：寄存器的可用性

2021/8/12 •

**ANSI 3.5.1** 通过使用寄存器存储类说明符可将对象实际放置在寄存器中的范围

编译器不接受对寄存器变量的用户请求。相反，在进行优化时，它将自行做出选择。

## 请参阅

实现定义的行为

# 结构、联合、枚举和位域

2021/8/15 •

- 对联合的不正确的访问
- 结构成员的填充和对齐
- 位域的符号
- 位域的存储
- 枚举类型

## 请参阅

实现定义的行为

# 对联合的不正确的访问

2021/8/13 •

**ANSI 3.3.2.3 使用不同类型的成员访问联合对象的成员**

**如果声明两个类型的联合并存储一个值，但使用其它类型访问该联合，则结果是不可靠的。**

**例如，声明了** `float` **和** `int` **的联合。存储** `float` **值，但程序稍后会将此值作为** `int` **进行访问。在这种情况下，此值取决于** `float` **值的内部存储。整数值是不可靠的。**

## 请参阅

结构、联合、枚举和位域

# 结构成员的填充和对齐

2021/8/13 •

**ANSI 3.5.2.1** 结构的成员的填充和对齐方式，以及位域是否可以跨存储单元边界

结构成员按其声明顺序进行存储：第一个成员的内存地址最低，最后一个成员的内存地址最高。

每个数据对象具有 alignment-requirement。所有数据（结构、联合和数组除外）的对齐要求是对象的大小或当前打包大小（使用 /Zp 或 `pack` 杂注指定，以较小者为准）。对于结构、联合和数组，对齐要求是其成员的最大对齐要求。为每个对象分配一个 offset，以便

*offset* % *alignment-requirement* == 0

如果整型的大小相同，并且下一个位域适合当前分配单元而未跨位域的常见对齐需求所强加的边界，则将相邻位域打包到相同的 1 字节、2 字节或 4 字节分配单元中。

## 请参阅

结构、联合、枚举和位域

# 位域的符号

2021/8/16 •

ANSI 3.5.2.1“普通”`int` 域被视为 `signed int` 位域还是 unsigned int 位域

位域可以带符号也可以不带符号。纯位域被视为带符号的。

## 请参阅

结构、联合、枚举和位域

# 位域的存储

2021/8/16 •

**ANSI 3.5.2.1 int 内的位域的分配顺序**

**在整数中按照从最高有效位到最低有效位的顺序来分配位域。在以下代码中**

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

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

**这些位将按如下所示排列**:

```
00000001 11110010
ccccccccb bbbbaaaa
```

**由于 80x86 处理器将整数值的低字节存储在高字节之前，因此上面的整数 0x01F2 将按 0xF2 后跟 0x01 的形式存储在物理内存中。**

## 请参阅

**结构、联合、枚举和位域**

# 枚举类型

2021/8/12 •

**ANSI 3.5.2.2** 选择**用于表示枚**举类**型的**值**的整数**类型

**声明**为 `enum` **的**变**量是** `int` 。

## 请参阅

结**构**、联**合**、**枚**举**和位域**

# 限定符：访问易失对象

2021/8/16 •

**ANSI 3.5.5.3 构成对具有可变限定类型的对象的访问的内容**

对可变限定类型的任何引用均为访问。

## 请参阅

实现定义的行为

# 声明符：最大数量

2021/8/16 •

**ANSI 3.5.4 可以修改算术、结构或联合类型的声明符的最大数量**

Microsoft C 不限制声明符的数目。该数字仅受可用内存限制。

## 请参阅

实现定义的行为

# 语句：对 Switch 语句的限制

2021/8/17 •

ANSI 3.6.4.2 `switch` 语句中的 `case` 值数量上限

Microsoft C 没有限制 `switch` 语句中的 `case` 值数量。该数字仅受可用内存限制。

## 请参阅

实现定义的行为

# 预处理指令

2021/8/17 •

- 字符常量和条件包含
- 包含用括号括起来的文件名
- 包含带引号的文件名
- 字符序列
- 杂注
- 默认日期和时间

## 请参阅

实现定义的行为

# 字符常量和条件包含

2021/8/16 •

**ANSI 3.8.1 控制条件包含的常量表达式中的**单**个字符的字符常量的**值**是否与**执**行字符集中的相同字符常量的**值**匹配。此**类**字符常量是否可具有**负值

预处**理器**语**句中使用的字符集与**执**行字符集相同。**预处**理器**识别负**字符**值。

## 请参阅

预处**理指令**

# 包含用括号括起来的文件名

2021/8/16 •

**ANSI 3.8.2 查找可包含源文件的方法**

对于用尖括号括起的文件规范，预处理器不会搜索父文件的目录。“父级”文件是其中包含 #include 指令的文件。相反，它首先会通过 /I 后面的编译器命令行上指定的目录中搜索文件。如果 /I 选项不存在或失败，预处理器会使用 INCLUDE 环境变量在尖括号中查找包含文件。INCLUDE 环境变量可以包含使用分号 (;) 分隔的多个路径。如果多个目录显示为 /I 选项的一部分或在 INCLUDE 环境变量中，预处理器会按它们的出现顺序搜索它们。

## 请参阅

预处理指令

# 包含用引号引起来的文件名

2021/8/13 •

**ANSI 3.8.2** 对可包含的源文件的带引号名称的支持

如果在两组双引号 (" ") 之间为包含文件指定完整明确的路径说明，则预处理器只搜索该路径说明并会忽略标准目录。

对于指定为 #include"path-spec"的包含文件，目录搜索从父文件的目录开始，然后在任何祖父级文件的目录中继续进行。因此，搜索将相对于包含当前正在处理的源文件的目录开始。如果没有祖父文件且未找到该文件，则搜索会像文件名括在尖括号中一样继续进行。

## 请参阅

预处理指令

# 字符序列

2021/8/14 •

**ANSI 3.8.2 源文件字符序列的映射**

预处理器语句使用的字符集和源文件语句相同，只不过转义序列不受支持。

因此，若要指定包含文件的路径，请仅使用一个反斜杠：

```
#include "path1\path2\myfile"
```

在源代码中，需要两个反斜杠：

```
fil = fopen( "path1\\path2\\myfile", "rt" );
```

## 请参阅

预处理指令

# 杂注

2021/8/15 •

ANSI 3.8.6 每个识别的 #pragma 指令的行为。

以下 C 杂注是为 Microsoft C 编译器定义的：

alloc_text
auto_inline
check_stack
code_seg
comment

data_seg
function
hdrstop
include_alias

inline_depth
inline_recursion
intrinsic
message

optimize
pack
setlocale
warning

## 请参阅

预处理指令

# 默认日期和时间

2021/8/16 •

ANSI 3.8.8 转换的日期和时间不可用时应遵循的 `__DATE__` 和 `__TIME__` 的定义

当操作系统不提供转换的时间和日期时，`__DATE__` 和 `__TIME__` 的默认值为 `May 03 1957` 和 `17:00:00`。

## 请参阅

预处理指令

# 库函数

2021/8/16 •

- NULL 宏
- assert 函数输出的诊断
- 字符测试
- 域错误
- 浮点值的下溢
- fmod 函数
- signal 函数
- 默认信号
- 终止换行符
- 空行
- Null 字符
- 追加模式中的文件位置
- 文本文件的截断
- 文件缓冲
- 零长度文件
- 文件名
- 文件访问限制
- 删除打开的文件
- 使用已存在的名称进行重命名
- 读取指针值
- 读取范围
- 文件位置错误
- 由 perror 函数生成的消息
- 分配零内存
- abort 函数
- atexit 函数
- 环境名称
- system 函数
- strerror 函数

- 时区
- clock 函数

## 请参阅

实现定义的行为

# NULL 宏

2021/8/16 •

**ANSI 4.1.5 宏 NULL 将扩展到的 null 指针常量**

一些包含文件将 NULL 宏定义为 `((void *)0)`。

## 请参阅

库函数

# assert 函数输出的诊断

2021/8/13 •

**ANSI 4.2** assert 函数输出的诊断和终止行为

如果表达式为 false (0)，则 assert 函数将输出诊断消息并调用 abort 例程。诊断消息具有以下形式

> **Assertion failed**: *expression*, **file** *filename*, **line** *linenumber*

其中，filename 是源文件的名称，linenumber 是源文件中失败的断言的行号 。如果表达式为 true（非零），则不执行任何操作 。

## 请参阅

库函数

# 字符测试

2021/8/17 •

**ANSI 4.3.1**：由 `isalnum`、`isalpha`、`iscntrl`、`islower`、`isprint` 和 `isupper` 函数测试的字符集

以下列表描述了由 Microsoft C 编译器实现的这些函数。

| “ | “ |
| --- | --- |
| `isalnum` | 字符 0 - 9、A-Z、a–z ASCII 48-57、65-90、97-122 |
| `isalpha` | 字符 A-Z、a–z ASCII 65-90、97-122 |
| `iscntrl` | ASCII 0 -31、127 |
| `islower` | 字符 a-z ASCII 97-122 |
| `isprint` | 字符 A–Z、a–z、0 - 9、标点、空格 ASCII 32–126 |
| `isupper` | 字符 A-Z ASCII 65-90 |

## 请参阅

库函数

# 域错误

2021/8/11 •

**ANSI 4.5.1** 发生域错误时数学函数返回的值

ERRNO.H 文件将域错误常量 `EDOM` 定义为 33。有关返回值的信息，请参阅引发错误的特定函数的帮助主题。

## 请参阅

库函数

# 浮点值的下溢

2021/8/17 •

**ANSI 4.5.1** 对于下溢范围错误，数学函数是否将整数表达式 `errno` 设置为宏 `ERANGE` 的值

浮点下溢不会将表达式 `errno` 设置为 `ERANGE`。当值接近零且最终下溢时，该值将设置为零。

## 请参阅

库函数

# fmod 函数

2021/8/16 •

**ANSI 4.5.6.4 当 `fmod` 函数的第二个参数为零时，是发生域错误还是返回零**

**当 `fmod` 函数的第二个参数为零时，该函数将返回零。**

## 请参阅

**库函数**

# signal 函数 (C)

2021/8/16 •

**ANSI 4.7.1.1** signal 函数的信号集

传递给 signal 的第一个参数必须为《运行时库参考》中描述的 signal 函数的符号常量之一。《运行时库参考》中的信息还列出了每个信号的操作模式支持。SIGNAL.H 中也定义了这些常量。

## 请参阅

库函数

# 默认信号

2021/8/16 •

ANSI 4.7.1.1 调用信号处理程序前未执行 `signal(sig, SIG_DFL)` 的等效项时，对所执行信号的阻止程序开始运行时，信号将设置为其默认状态。

## 请参阅

库函数

# 终止换行符

2021/8/13 •

**ANSI 4.9.2 文本流的最后一行是否需要一个终止换行符**

**流函数将新行或文件结尾识别为一行的终止符。**

## 请参阅

库函数

# 空行

2021/8/14 •

**ANSI 4.9.2 读入时出现换行符之前，空格字符是否可以立即被写出到文本流**

**保留空格符。**

## 请参阅

库函数

# Null 字符

2021/8/13 •

**ANSI 4.9.2 可附加到写入二进制流的数据的 null 字符的数量**

**可以将任意数量的 null 字符附加到二进制流中。**

## 请参阅

库函数

# 追加模式中的文件位置

2021/8/16 •

**ANSI 4.9.3 追加模式流的文件位置指示器最初是位于文件的开头还是结尾**

**在追加模式中打开文件时，文件位置指示器最初指向文件的末尾。**

## 请参阅

库函数

# 文本文件的截断

2021/8/11 •

**ANSI 4.9.3 文本流中的写入是否会导致关联的文件在该点以外被截断**

对**文本流**进行**写入不会在**该**点以外截断文件**。

## 请参阅

库**函数**

# 文件缓冲

2021/8/16 •

**ANSI 4.9.3 文件缓冲的特性**

**将完全缓冲通过标准 I/O 函数访问的磁盘文件。默认情况下，缓冲区包含 512 个字节。**

## 请参阅

库函数

# 零长度文件

2021/8/15 •

**ANSI 4.9.3 零长度文件是否确实存在**

**允许长度为零的文件。**

## 请参阅

库函数

# 文件名

2021/8/16 •

**ANSI 4.9.3 撰写有效文件名的规则**

**文件规范可能包含可选的驱动器号(始终后跟冒号)、一系列可选的目录名称(用斜杠分隔)和文件名。**

**有关详细信息，请参阅为文件命名。**

## 请参阅

**库函数**

# 文件访问限制

2021/8/16 •

**ANSI 4.9.3 是否可将同一个文件打开多次**

**不允许打开已打开的文件。**

## 请参阅

库函数

# 删除打开的文件

2021/8/13 •

**ANSI 4.9.4.1** remove 函数对打开的文件的效果

remove 函数用于删除文件。如果文件处于打开状态，此函数将失败并返回 -1。

## 请参阅

库函数

# 使用已存在的名称进行重命名

2021/8/15 •

**ANSI 4.9.4.2 使用新名称的文件在调用 rename 函数之前存在的效果**

**如果尝试使用现有的名称重命名文件，则 rename 函数将失败并返回错误代码。**

## 请参阅

库函数

# 读取指针值

2021/8/16 •

**ANSI 4.9.6.2** `fscanf` **函数中的** %p **转换的输入**

**指定** %p **格式字符时，** `fscanf` **函数指针从十六进制** ASCII **值转换为正确地址。**

## 请参阅

库函数

# 读取范围

2021/8/16 •

ANSI 4.9.6.2：短划线 (-) 字符的解释，该字符既不是 `fscanf` 函数的 % [ 转换的扫描表中的第一个字符，也不是该表中的最后一个字符

下面的行

```
fscanf( fileptr, "%[A-Z]", strptr);
```

将 A-Z 范围内的任意数目的字符读取到 `strptr` 指向的字符串中。

## 请参阅

库函数

# 文件位置错误

2021/8/16 •

**ANSI 4.9.9.1, 4.9.9.4** 失败时，`fgetpos` 或 `ftell` 函数设置的宏 `errno` 的值

当 `fgetpos` 或 `ftell` 失败时，`errno` 将设置为清单常量 `EINVAL`（如果位置无效）或 `EBADF`（如果文件数量错误）。常量是在 ERRNO.H 中定义的。

## 请参阅

库函数

# perror 函数生成的消息

2021/8/16 •

### ANSI 4.9.10.4 `perror` 函数生成的消息

`perror` 函数生成了以下消息：

```
0  Error 0
1
2  No such file or directory
3
4
5
6
7  Arg list too long
8  Exec format error
9  Bad file number
10
11
12 Not enough core
13 Permission denied
14
15
16
17 File exists
18 Cross-device link
19
20
21
22 Invalid argument
23
24 Too many open files
25
26
27
28 No space left on device
29
30
31
32
33 Math argument
34 Result too large
35
36 Resource deadlock would occur
```

## 请参阅

库函数

# 分配零内存

2021/8/12 •

**ANSI 4.10.3 请求的大小为零时，`calloc`、`malloc` 或 `realloc` 函数的行为**

`calloc`、`malloc` 和 `realloc` 函数接受零作为参数。不分配实际内存，但会返回有效的指针，并且之后可通过 realloc 修改内存块。

## 请参阅

库函数

# abort 函数 (C)

2021/8/16 •

**ANSI 4.10.4.1 有关打开的文件和临时文件的 abort 函数的行为**

abort 函数不关闭已打开的或临时的文件。它不刷新流缓冲区。有关详细信息，请参阅中止。

## 请参阅

库函数

# atexit 函数 (C)

2021/8/13 •

**ANSI 4.10.4.3 如果参数的值为零、EXIT_SUCCESS 或 EXIT_FAILURE 之外的其他值，则为 `atexit` 函数返回的状态**

**如果成功，则 `atexit` 函数返回零；否则返回非零值。**

## 请参阅

库函数

# 环境名称

2021/8/12 •

**ANSI 4.10.4.4** 环境名称集和用于更改 getenv 函数使用的环境列表的方法

环境名称集是不受限的。

若要从 C 程序中更改环境变量，请调用 _putenv 函数。若要从 Windows 命令行中更改环境变量，请使用 SET 命令（例如，SET LIB = D:\ LIBS）。

C 程序内的环境变量集仅在操作系统命令外壳（CMD.EXE 或 COMMAND.COM）的主机副本运行时存在。例如，行

```
system( SET LIB = D:\LIBS );
```

将运行命令外壳 (CMD.EXE) 的副本、设置环境变量 LIB 并返回 C 程序，同时退出 CMD.EXE 的辅助副本。退出 CMD.EXE 的副本将移除临时环境变量 LIB。

同样，仅保留 `_putenv` 函数所做的更改，直到该程序结束。

## 请参阅

库函数
_putenv、_wputenv
getenv、_wgetenv

# 系统函数

2021/8/16 •

**ANSI 4.10.4.5** system 函数执行字符串的内容和模式

system 函数执行内部操作系统命令，或者 C 程序中（而不是命令行中）的 .EXE、.COM（Windows NT 中的 .CMD）或 .BAT 文件。

系统函数查找命令解释器，它通常是 Windows NT 操作系统中的 CMD.EXE 或 Windows 中的 COMMAND.COM。系统函数随后将自变量字符串传递到命令解释器。

有关详细信息，请参阅 system、_wsystem。

## 请参阅

库函数

# strerror 函数

2021/8/11 •

**ANSI 4.11.6.2 `strerror` 函数返回的错误消息字符串的内容**

**`strerror` 函数生成了以下消息：**

```
0   Error 0
1
2   No such file or directory
3
4
5
6
7   Arg list too long
8   Exec format error
9   Bad file number
10
11
12  Not enough core
13  Permission denied
14
15
16
17  File exists
18  Cross-device link
19
20
21
22  Invalid argument
23
24  Too many open files
25
26
27
28  No space left on device
29
30
31
32
33  Math argument
34  Result too large
35
36  Resource deadlock would occur
```

## 请参阅

库函数

# 时区

2021/8/11 •

**ANSI 4.12.1 本地时区和夏令时**

**本地时区为太平洋标准时间。Microsoft C 支持夏令时。**

## 请参阅

库函数

# clock 函数 (C)

2021/8/14 •

**ANSI 4.12.2.1 `clock` 函数的纪元**

**当 C 程序开始执行时，`clock` 函数的纪元将以值 0 开始。它返回以 1/CLOCKS_PER_SEC（对于 Microsoft C 等于 1/1000）度量的时间。**

## 请参阅

**库函数**

# C/C++ 预处理器参考

2021/8/13 •

*C/c + + 预处理器参考* 说明了在 Microsoft c/c + + 中实现的预处理器。在将 C 和 C++ 文件传递到编译器之前，预处理器将对这些文件执行预先操作。可以使用预处理器有条件地编译代码、插入文件、指定编译时错误消息以及将计算机特定规则应用于代码节。

在 Visual Studio 2019 中，/zc:预处理器编译器选项提供完全符合 C11 和 C17 预处理器。这是使用编译器标志或时的默认值 `/std:c11` `/std:c17` 。

## 本节内容

**程序**
概述传统和新的符合预处理器。

预处理器指令
介绍通常用于使源程序易于在不同的执行环境中更改和编译的指令。

预处理器运算符
讨论在 `#define` 指令的上下文中使用的四个预处理器特定运算符。

预定义的宏
讨论 C 和 c + + 标准指定的预定义宏和 Microsoft c + +。

杂注
讨论杂注，杂注提供了一种方法来让每个编译器提供计算机和操作系统特定的功能，同时保持与 C 和 C++ 语言的整体兼容性。

## 相关章节

C++ 语言参考
提供有关 Microsoft 的 C++ 语言实现的参考材料。

C 语言参考
提供有关 Microsoft 的 C 语言实现的参考材料。

C/c + + 生成参考
提供指向讨论编译器和链接器选项的主题的链接。

Visual Studio 项目-c + +
描述 Visual Studio 中使您能够指定目录(项目系统将在其中进行搜索以找到 C++ 项目的文件)的用户界面。

# Microsoft C 运行时库 (CRT) 参考

2021/8/14 •

Microsoft 运行时库提供用于对 Microsoft Windows编程的例程。这些例程自动执行许多不采用 C 和 C++ 语言提供的常见编程任务。

库中大部分例程的示例程序包含在独立参考主题中。

## 本节内容

**按类别分类的通用 C 运行时例程**
按类别提供运行时库的链接。

**全局变量和标准类型**
提供指向运行时库提供的全局变量和标准类型的链接。

**全局常量**
提供指向运行时库定义的全局常量的链接。

**全局状态**
描述 C 运行时库中全局状态的范围。

**泛型文本映射**
提供指向在 Tchar.h 中定义的通用文本映射的链接。

**按字母顺序的函数引用**
提供指向 C 运行时库函数的链接，按字母顺序进行组织。

**函数系列概述**
提供指向 C 运行时库函数的链接，这些函数按函数系列进行组织。

**语言和国家/地区字符串**
介绍如何使用 `setlocale` 函数设置语言和国家/地区字符串。

**C 运行时 (CRT) 和 C++ 标准库 (STL) `.lib` 文件**
包含 `.lib` C 运行时库及其关联的编译器选项和预处理器指令的文件列表。

## 相关章节

**调试例程**
提供指向运行时库例程的调试版本的链接。

**运行时错误检查**
提供指向支持运行时错误检查的函数的链接。

**DLL 和Visual C++运行时库行为**
讨论用于 DLL 的入口点和启动代码。

**调试**
提供一些链接，所涉及内容为使用 Visual Studio 调试器纠正应用程序或存储过程中的逻辑错误。