

C 语言教程



C 语言是一种通用的、面向过程式的计算机程序设计语言。1972 年，为了移植与开发 UNIX 操作系统，丹尼斯·里奇在贝尔电话实验室设计开发了 C 语言。

C 语言是一种广泛使用的计算机语言，它与 Java 编程语言一样普及，二者在现代软件程序员之间都得到广泛使用。

当前最新的C语言标准为 C11，在它之前的C语言标准为 C99。

[现在开始学习 C 编程！](#)

[C 在线工具](#)

谁适合阅读本教程？

本教程是专门为需要从零开始了解 C 语言的软件程序员打造的。本教程将让您对 C 语言有足够的认识，从而提升您自己的专业知识水平。

阅读本教程前，您需要了解的知识：

在开始学习本教程之前，您需要对计算机编程术语有基本的了解。对任何一种编程语言有基本的了解将有助于您理解 C 语言编程概念，并有助于加快您的学习进度。

编译/执行 C 程序

实例

```
#include <stdio.h>
int main()
{
    /* 我的第一个 C 程序 */
    printf("Hello, World! \n");

    return 0;
}
```

运行实例 »

实例解析：

所有的 C 语言程序都需要包含 **main()** 函数。代码从 **main()** 函数开始执行。

/* ... */ 用于注释说明。

printf() 用于格式化输出到屏幕。**printf()** 函数在 **"stdio.h"** 头文件中声明。

stdio.h 是一个头文件 (标准输入输出头文件)，**#include** 是一个预处理命令，用来引入头文件。当编译器遇到 **printf()** 函数时，如果没有找到 **stdio.h** 头文件，会发生编译错误。

return 0; 语句用于表示退出程序。

C 简介

C 语言是一种通用的高级语言，最初是由丹尼斯·里奇在贝尔实验室为开发 UNIX 操作系统而设计的。C 语言最开始是于 1972 年在 DEC PDP-11 计算机上被首次实现。

在 1978 年，布莱恩·柯林汉（Brian Kernighan）和丹尼斯·里奇（Dennis Ritchie）制作了 C 的第一个公开可用的描述，现在被称为 K&R 标准。

UNIX 操作系统，C编译器，和几乎所有的 UNIX 应用程序都是用 C 语言编写的。由于各种原因，C 语言现在已经成为一种广泛使用的专业语言。

易于学习。

结构化语言。

它产生高效率的程序。

它可以处理底层的活动。

它可以在多种计算机平台上编译。

关于 C

C 语言是为了编写 UNIX 操作系统而被发明的。

C 语言是以 B 语言为基础的，B 语言大概是在 1970 年被引进的。

C 语言标准是于 1988 年由美国国家标准协会（ANSI，全称 American National Standard Institute）制定的。

截至 1973 年，UNIX 操作系统完全使用 C 语言编写。

目前，C 语言是最广泛使用的系统程序设计语言。

大多数先进的软件都是使用 C 语言实现的。

当今最流行的 Linux 操作系统和 RDBMS（Relational Database Management System：关系数据库管理系统） MySQL 都是使用 C 语言编写的。

为什么要使用 C?

C 语言最初是用于系统开发工作，特别是组成操作系统的程序。由于 C 语言所产生的代码运行速度与汇编语言编写的代码运行速度几乎一样，所以采用 C 语言作为系统开发语言。下面列举几个使用 C 的实例：

- 操作系统
- 语言编译器
- 汇编器
- 文本编辑器
- 打印机
- 网络驱动器
- 现代程序
- 数据库
- 语言解释器
- 实体工具

C 程序

一个 C 语言程序，可以是 3 行，也可以是数百万行，它可以写在一个或多个扩展名为 **".c"** 的文本文件中，例如，*hello.c*。您可以使用 **"vi"**、**"vim"** 或任何其他文本编辑器来编写您的 C 语言程序。

本教程假定您已经知道如何编辑一个文本文件，以及如何在程序文件中编写源代码。

C11

C11（也被称为C1X）指ISO标准ISO/IEC 9899:2011，是当前最新的C语言标准。在它之前的C语言标准为C99。

新特性

对齐处理（Alignment）的标准化（包括 `_Alignas` 标志符，`alignof` 运算符，`aligned_alloc` 函数以及 `<stdalign.h>` 头文件）。

`_Noreturn` 函数标记，类似于 gcc 的 `__attribute__((noreturn))`。

`_Generic` 关键字。

多线程（Multithreading）支持，包括：

`_Thread_local` 存储类型标识符，`<threads.h>` 头文件，里面包含了线程的创建和管理函数。

`_Atomic` 类型修饰符和 `<stdatomic.h>` 头文件。

增强的Unicode的支持。基于C Unicode技术报告ISO/IEC TR 19769:2004，增强了对Unicode的支持。包括为UTF-16/UTF-32编码增加了 `char16_t` 和 `char32_t` 数据类型，提供了包含unicode字符串转换函数的头文件 `<uchar.h>`。

删除了 `gets()` 函数，使用一个新的更安全的函数 `gets_s()` 替代。

增加了边界检查函数接口，定义了新的安全的函数，例如 `fopen_s()`，`strcat_s()` 等等。

增加了更多浮点处理宏(宏)。

匿名结构体/联合体支持。这个在gcc早已存在，C11将其引入标准。

静态断言（Static assertions），`_Static_assert()`，在解释 `#if` 和 `#error` 之后被处理。

新的 `fopen()` 模式，`("...x")`。类似 POSIX 中的 `O_CREAT|O_EXCL`，在文件锁中比较常用。

新增 `quick_exit()` 函数作为第三种终止程序的方式。当 `exit()` 失败时可以做最少的清理工作。

C 环境设置

本地环境设置

如果您想要设置 C 语言环境，您需要确保电脑上有以下两款可用的软件，文本编辑器和 C 编译器。

文本编辑器

这将用于输入您的程序。文本编辑器包括 Windows Notepad、OS Edit command、Brief、Epsilon、EMACS 和 vim/vi。

文本编辑器的名称和版本在不同的操作系统上可能会有所不同。例如，Notepad 通常用于 Windows 操作系统上，vim/vi 可用于 Windows 和 Linux/UNIX 操作系统上。

通过编辑器创建的文件通常称为源文件，源文件包含程序源代码。C 程序的源文件通常使用扩展名 **".c"**。

在开始编程之前，请确保您有一个文本编辑器，且有足够的经验来编写一个计算机程序，然后把它保存在一个文件中，编译并执行它。

C 编译器

写在源文件中的源代码是人类可读的源。它需要"编译"，转为机器语言，这样 CPU 可以按给定指令执行程序。

C 语言编译器用于把源代码编译成最终的可执行程序。这里假设您已经对编程语言编译器有基本的了解了。

最常用的免费可用的编译器是 GNU 的 C/C++ 编译器，如果您使用的是 HP 或 Solaris，则可以使用各自操作系统上的编译器。

以下部分将指导您如何在不同的操作系统上安装 GNU 的 C/C++ 编译器。这里同时提到 C/C++，主要是因为 GNU 的 gcc 编译器适合于 C 和 C++ 编程语言。

UNIX/Linux 上的安装

如果您使用的是 **Linux 或 UNIX**，请在命令行使用下面的命令来检查您的系统上是否安装了 GCC：

```
$ gcc -v
```

如果您的计算机上已经安装了 GNU 编译器，则会显示如下消息：

```
Using built-in specs.
Target: i386-redhat-linux
Configured with: ../configure --prefix=/usr .....
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)
```

如果未安装 GCC，那么请按照 <http://gcc.gnu.org/install/> 上的详细说明安装 GCC。

本教程是基于 Linux 编写的，所有给定的实例都已在 Cent OS Linux 系统上编译过。

Mac OS 上的安装

如果您使用的是 Mac OS X，最快捷的获取 GCC 的方法是从苹果的网站下载 Xcode 开发环境，并按照安装说明进行安装。一旦安装上 Xcode，您就能使用 GNU 编译器。

Xcode 目前可从 developer.apple.com/technologies/tools/ 上下载。

Windows 上的安装

为了在 Windows 上安装 GCC，您需要安装 MinGW。为了安装 MinGW，请访问 MinGW 的主页 www.mingw.org，进入 MinGW 下载页面，下载最新版本的 MinGW 安装程序，命名格式为 MinGW-<version>.exe。

当安装 MinWG 时，您至少要安装 gcc-core、gcc-g++、binutils 和 MinGW runtime，但是一般情况下都会安装更多其他的项。

添加您安装的 MinGW 的 bin 子目录到您的 **PATH** 环境变量中，这样您就可以在命令行中通过简单的名称来指定这些工具。

当完成安装时，您可以从 Windows 命令行上运行 gcc、g++、ar、ranlib、dlltool 和其他一些 GNU 工具。

C 程序结构

在我们学习 C 语言的基本构建块之前，让我们先来看看一个最小的 C 程序结构，在接下来的章节中可以以此作为参考。

C Hello World 实例

C 程序主要包括以下部分：

- 预处理器指令
- 函数
- 变量
- 语句 & 表达式
- 注释

让我们看一段简单的代码，可以输出单词 "Hello World"：

实例

```
#include <stdio.h>
int main()
{
    /* 我的第一个 c 程序 */
    printf("Hello, World! \n");

    return 0;
}
```

接下来我们讲解一下上面这段程序：

- 程序的第一行 `#include <stdio.h>` 是预处理器指令，告诉 C 编译器在实际编译之前要包含 `stdio.h` 文件。
- 下一行 `int main()` 是主函数，程序从这里开始执行。
- 下一行 `/*...*/` 将会被编译器忽略，这里放置程序的注释内容。它们被称为程序的注释。
- 下一行 `printf(...)` 是 C 中另一个可用的函数，会在屏幕上显示消息 "Hello, World!"。
- 下一行 `return 0;` 终止 `main()` 函数，并返回值 0。

编译 & 执行 C 程序

接下来让我们看看如何把源代码保存在一个文件中，以及如何编译并运行它。下面是简单的步骤：

- 打开一个文本编辑器，添加上述代码。
- 保存文件为 `hello.c`。
- 打开命令提示符，进入到保存文件所在的目录。
- 键入 `gcc hello.c`，输入回车，编译代码。
- 如果代码中没有错误，命令提示符会跳到下一行，并生成 `a.out` 可执行文件。
- 现在，键入 `a.out` 来执行程序。
- 您可以看到屏幕上显示 `"Hello World"`。

```
$ gcc hello.c
$ ./a.out
Hello, World!
```

请确保您的路径中已包含 gcc 编译器，并确保在包含源文件 `hello.c` 的目录中运行它。

C 基本语法

我们已经看过 C 程序的基本结构，这将有助于我们理解 C 语言的其他基本的构建块。

C 的令牌（Tokens）

C 程序由各种令牌组成，令牌可以是关键字、标识符、常量、字符串值，或者是一个符号。例如，下面的 C 语句包括五个令牌：

```
printf("Hello, World! \n");
```

这五个令牌分别是：

```
printf
(
"Hello, World! \n"
)
;
```

分号；

在 C 程序中，分号是语句结束符。也就是说，每个语句必须以分号结束。它表明一个逻辑实体的结束。

例如，下面是两个不同的语句：

```
printf("Hello, World! \n");
return 0;
```

注释

注释就像是 C 程序中的帮助文本，它们会被编译器忽略。它们以 /* 开始，以字符 */ 终止，如下所示：

```
/* 我的第一个 c 程序 */
```

您不能在注释内嵌套注释，注释也不能出现在字符串或字符值中。

标识符

C 标识符是用来标识变量、函数，或任何其他用户自定义项目的名称。一个标识符以字母 A-Z 或 a-z 或下划线 _ 开始，后跟零个或多个字母、下划线和数字（0-9）。

C 标识符内不允许出现标点字符，比如 @、\$ 和 %。C 是区分大小写的编程语言。因此，在 C 中，*Manpower* 和 *manpower* 是两个不同的标识符。下面列出几个有效的标识符：

```
mohd      zara    abc    move_name  a_123
myname50  _temp   j      a23b9     retVal
```

关键字

下表列出了 C 中的保留字。这些保留字不能作为常量名、变量名或其他标识符名称。

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile

default	if	static	while
do	int	struct	_Packed
double			

C99新增关键字

_Bool	_Complex	_Imaginary	inline	restrict
-------	----------	------------	--------	----------

C11新增关键字

_Alignas	_Alignof	_Atomic	_Generic	_Noreturn
_Static_assert	_Thread_local			

C 中的空格

只包含空格的行，被称为空白行，可能带有注释，C 编译器会完全忽略它。

在 C 中，空格用于描述空白符、制表符、换行符和注释。空格分隔语句的各个部分，让编译器能识别语句中的某个元素（比如 int）在哪里结束，下一个元素在哪里开始。因此，在下面的语句中：

```
int age;
```

在这里，int 和 age 之间必须至少有一个空格字符（通常是一个空白符），这样编译器才能够区分它们。另一方面，在下面的语句中：

```
fruit = apples + oranges;    // 获取水果的总数
```

fruit 和 =，或者 = 和 apples 之间的空格字符不是必需的，但是为了增强可读性，您可以根据需要适当增加一些空格。

C 数据类型

在 C 语言中，数据类型指的是用于声明不同类型的变量或函数的一个广泛的系统。变量的类型决定了变量存储占用的空间，以及如何解释存储的位模式。

C 中的类型可分为以下几种：

序号	类型与描述
1	基本类型： 它们是算术类型，包括两种类型：整数类型和浮点类型。
2	枚举类型： 它们也是算术类型，被用来定义在程序中只能赋予其一定的离散整数值 的变量。
3	void 类型： 类型说明符 <i>void</i> 表明没有可用的值。
4	派生类型： 它们包括：指针类型、数组类型、结构类型、共用体类型和函数类型。

数组类型和结构类型统称为聚合类型。函数的类型指的是函数返回值的类型。在本章节接下来的部分我们将介绍基本类型，其他几种类型会在后边几个章节中进行讲解。

整数类型

下表列出了关于标准整数类型的存储大小和值范围的细节：

类型	存储大小	值范围
char	1 字节	-128 到 127 或 0 到 255
unsigned char	1 字节	0 到 255
signed char	1 字节	-128 到 127
int	2 或 4 字节	-32,768 到 32,767 或 -2,147,483,648 到 2,147,483,647
unsigned int	2 或 4 字节	0 到 65,535 或 0 到 4,294,967,295
short	2 字节	-32,768 到 32,767
unsigned short	2 字节	0 到 65,535
long	4 字节	-2,147,483,648 到 2,147,483,647
unsigned long	4 字节	0 到 4,294,967,295

注意，各种类型的存储大小与系统位数有关，但目前通用的以64位系统为主。

以下列出了32位系统与64位系统的存储大小的差别（windows 相同）：

Windows vc12		Linux gcc-5.3.1		Compiler
win32	x64	i686	x86_64	Target
1		1	1	char
1		1	1	unsigned char
2		2	2	short
2		2	2	unsigned short
4		4	4	int
4		4	4	unsigned int
4		4	8	long
4		4	8	unsigned long
4		4	4	float
8		8	8	double
4		4	8	long int
8		8	8	long long
8		12	16	long double

为了得到某个类型或某个变量在特定平台上的准确大小，您可以使用 **sizeof** 运算符。表达式 *sizeof(type)* 得到对象或类型的存储字节大小。下面的实例演示了获取 int 类型的大小：

实例

```
#include <stdio.h>
#include <limits.h>
int main()
{
    printf("int 存储大小 : %lu \n", sizeof(int));

    return 0;
}
```

当您在 Linux 上编译并执行上面的程序时，它会产生下列结果：

int 存储大小 : 4

浮点类型

下表列出了关于标准浮点类型的存储大小、值范围和精度的细节：

类型	存储大小	值范围	精度
float	4 字节	1.2E-38 到 3.4E+38	6 位小数
double	8 字节	2.3E-308 到 1.7E+308	15 位小数
long double	16 字节	3.4E-4932 到 1.1E+4932	19 位小数

头文件 float.h 定义了宏，在程序中可以使用这些值和其他有关实数二进制表示的细节。下面的实例将输出浮点类型占用的存储空间以及它的范围值：

实例

```
#include <stdio.h>
#include <float.h>
int main()
{
    printf("float 存储最大字节数 : %lu \n", sizeof(float));
    printf("float 最小值: %E\n", FLT_MIN );
    printf("float 最大值: %E\n", FLT_MAX );
    printf("精度值: %d\n", FLT_DIG );
}
```

```
return 0;
}
```

当您在 Linux 上编译并执行上面的程序时，它会产生下列结果：

```
float 存储最大字节数 : 4
float 最小值: 1.175494E-38
float 最大值: 3.402823E+38
精度值: 6
```

void 类型

void 类型指定没有可用的值。它通常用于以下三种情况下：

序号	类型与描述
1	函数返回为空 C 中有各种函数都不返回值，或者您可以说它们返回空。不返回值的函数的返回类型为 void 。例如 void exit (int status);
2	函数参数为空 C 中有各种函数不接受任何参数。不带参数的函数可以接受一个 void 。例如 int rand(void);
3	指针指向 void 类型为 void * 的指针代表对象的地址，而不是类型。例如，内存分配函数 void *malloc(size_t size); 返回指向 void 的指针，可以转换为任何数据类型。

如果现在您还是无法完全理解 void 类型，不用太担心，在后续的章节中我们将会详细讲解这些概念。

C 变量

变量其实只不过是程序可操作的存储区的名称。C 中每个变量都有特定的类型，类型决定了变量存储的大小和布局，该范围内的值都可以存储在内存中，运算符可应用于变量上。

变量的名称可以由字母、数字和下划线字符组成。它必须以字母或下划线开头。大写字母和小写字母是不同的，因为 C 是大小写敏感的。基于前一章讲解的基本类型，有以下几种基本的变量类型：

类型	描述
char	通常是一个字节（八位）。这是一个整数类型。
int	对机器而言，整数的最自然的大小。
float	单精度浮点值。单精度是这样的格式，1位符号，8位指数，23位小数。 □
double	双精度浮点值。双精度是1位符号，11位指数，52位小数。 □
void	表示类型的缺失。

C 语言也允许定义各种其他类型的变量，比如枚举、指针、数组、结构、共用体等等，这将会在后续的章节中进行讲解，本章节我们先讲解基本变量类型。

C 中的变量定义

变量定义就是告诉编译器在何处创建变量的存储，以及如何创建变量的存储。变量定义指定一个数据类型，并包含了该类型的一个或多个变量的列表，如下所示：

```
type variable_list;
```

在这里，**type** 必须是一个有效的 C 数据类型，可以是 char、w_char、int、float、double、bool 或任何用户自定义的对象，**variable_list** 可以由一个或多个标识符名称组成，多个标识符之间用逗号分隔。下面列出几个有效的声明：

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

行 **int i, j, k;** 声明并定义了变量 i、j 和 k，这指示编译器创建类型为 int 的名为 i、j、k 的变量。变量可以在声明的时候被初始化（指定一个初始值）。初始化器由一个等号，后跟一个常量表达式组成，如下所示：

```
type variable_name = value;
```

下面列举几个实例：

```
extern int d = 3, f = 5;    // d 和 f 的声明与初始化
int d = 3, f = 5;          // 定义并初始化 d 和 f
byte z = 22;               // 定义并初始化 z
char x = 'x';              // 变量 x 的值为 'x'
```

不带初始化的定义：带有静态存储持续时间的变量会被隐式初始化为 NULL（所有字节的值都是 0），其他所有变量的初始值是无定义的。

C 中的变量声明

变量声明向编译器保证变量以指定的类型和名称存在，这样编译器在不需要知道变量完整细节的情况下也能继续进一步的编译。变量声明只在编译时有它的意义，在程序连接时编译器需要实际的变量声明。

变量的声明有两种情况：

- 1、一种是需要建立存储空间的。例如：int a 在声明的时候就已经建立了存储空间。
- 2、另一种是不需要建立存储空间的，通过使用extern关键字声明变量名而不定义它。 例如：extern int a 其中变量 a 可以在别的文件中定义的。除非有extern关键字，否则都是变量的定义。

```
extern int i; //声明，不是定义
int i; //声明，也是定义
```

实例

尝试下面的实例，其中，变量在头部就已经被声明，但是定义与初始化在主函数内：

实例

```
#include <stdio.h>
// 变量声明
extern int a, b;
extern int c;
extern float f;

int main ()
{
    /* 变量定义 */
    int a, b;
    int c;
    float f;

    /* 初始化 */
    a = 10;
    b = 20;

    c = a + b;
    printf("value of c : %d \n", c);

    f = 70.0/3.0;
    printf("value of f : %f \n", f);

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of c : 30
value of f : 23.333334
```

C 中的左值（Lvalues）和右值（Rvalues）

C 中有两种类型的表达式：

1. **左值（lvalue）**：指向内存位置的表达式被称为左值（lvalue）表达式。左值可以出现在赋值号的左边或右边。
2. **右值（rvalue）**：术语右值（rvalue）指的是存储在内存中某些地址的数值。右值是不能对其进行赋值的表达式，也就是说，右值可以出现在赋值号的右边，但不能出现在赋值号的左边。

变量是左值，因此可以出现在赋值号的左边。数值型的字面值是右值，因此不能被赋值，不能出现在赋值号的左边。下面是一个有效的语句：

```
int g = 20;
```

但是下面这个就不是一个有效的语句，会生成编译时错误：

```
10 = 20;
```

C 常量

常量是固定值，在程序执行期间不会改变。这些固定的值，又叫做字面量。

常量可以是任何的基本数据类型，比如整数常量、浮点常量、字符常量，或字符串面值，也有枚举常量。

常量就像是常规的变量，只不过常量的值在定义后不能进行修改。

整数常量

整数常量可以是十进制、八进制或十六进制的常量。前缀指定基数：0x 或 0X 表示十六进制，0 表示八进制，不带前缀则默认表示十进制。

整数常量也可以带一个后缀，后缀是 U 和 L 的组合，U 表示无符号整数（unsigned），L 表示长整数（long）。后缀可以是大写，也可以是小写，U 和 L 的顺序任意。

下面列举几个整数常量的实例：

```
212      /* 合法的 */
215u     /* 合法的 */
0xFeeL   /* 合法的 */
078      /* 非法的：8 不是八进制的数字 */
032UU    /* 非法的：不能重复后缀 */
```

以下是各种类型的整数常量的实例：

```
85       /* 十进制 */
0213     /* 八进制 */
0x4b     /* 十六进制 */
30       /* 整数 */
30u      /* 无符号整数 */
30l      /* 长整数 */
30ul     /* 无符号长整数 */
```

浮点常量

浮点常量由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。

当使用小数形式表示时，必须包含整数部分、小数部分，或同时包含两者。当使用指数形式表示时， 必须包含小数点、指数，或同时包含两者。带符号的指数是用 e 或 E 引入的。

下面列举几个浮点常量的实例：

```
3.14159   /* 合法的 */
314159E-5L /* 合法的 */
510E      /* 非法的：不完整的指数 */
210f      /* 非法的：没有小数或指数 */
.e55      /* 非法的：缺少整数或分数 */
```

字符常量

字符常量是括在单引号中，例如，'x' 可以存储在 char 类型的简单变量中。

字符常量可以是一个普通的字符（例如 'x'）、一个转义序列（例如 '\t'）， 或一个通用的字符（例如 '\u02C0'）。

在 C 中，有一些特定的字符，当它们前面有反斜杠时，它们就具有特殊的含义，被用来表示如换行符（\n）或制表符（\t）等。下表列出了一些这样的转义序列码：

转义序列	含义
\\	\ 字符
\'	' 字符
\"	" 字符
\?	? 字符

\a	警报铃声
\b	退格键
\f	换页符
\n	换行符
\r	回车
\t	水平制表符
\v	垂直制表符
\ooo	一到三位的八进制数
\xhh...	一个或多个数字的十六进制数

下面的实例显示了一些转义序列字符：

实例

```
#include <stdio.h>
int main()
{
    printf("Hello\tWorld\n\n");

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Hello    World
```

字符串常量

字符串面值或常量是括在双引号 "" 中的。一个字符串包含类似于字符常量的字符：普通的字符、转义序列和通用的字符。

您可以使用空格做分隔符，把一个很长的字符串常量进行分行。

下面的实例显示了一些字符串常量。下面这三种形式所显示的字符串是相同的。

```
"hello, dear"

"hello, \
dear"

"hello, " "d" "ear"
```

定义常量

在 C 中，有两种简单的定义常量的方式：

1. 使用 **#define** 预处理器。
2. 使用 **const** 关键字。

#define 预处理器

下面是使用 **#define** 预处理器定义常量的形式：

```
#define identifier value
```

具体请看下面的实例：

实例

```
#include <stdio.h>
#define LENGTH 10
#define WIDTH 5
#define NEWLINE '\n'
int main()
{
    int area;

    area = LENGTH * WIDTH;
    printf("value of area : %d", area);
    printf("%c", NEWLINE);

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of area : 50
```

const 关键字

您可以使用 **const** 前缀声明指定类型的常量，如下所示：

```
const type variable = value;
```

具体请看下面的实例：

实例

```
#include <stdio.h>
int main()
{
    const int LENGTH = 10;
    const int WIDTH = 5;
    const char NEWLINE = '\n';
    int area;

    area = LENGTH * WIDTH;
    printf("value of area : %d", area);
    printf("%c", NEWLINE);

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of area : 50
```

请注意，把常量定义为大写字母形式，是一个很好的编程实践。

C 存储类

存储类定义 C 程序中变量/函数的范围（可见性）和生命周期。这些说明符放置在它们所修饰的类型之前。下面列出 C 程序中可用的存储类：

auto
register
static
extern

auto 存储类

auto 存储类是所有局部变量默认的存储类。

```
{
    int mount;
    auto int month;
}
```

上面的实例定义了两个带有相同存储类的变量，**auto** 只能用在函数内，即 **auto** 只能修饰局部变量。

register 存储类

register 存储类用于定义存储在寄存器中而不是 RAM 中的局部变量。这意味着变量的最大尺寸等于寄存器的大小（通常是一个词），且不能对它应用一元的 '&' 运算符（因为它没有内存位置）。

```
{
    register int miles;
}
```

寄存器只用于需要快速访问的变量，比如计数器。还应注意的是，定义 '**register**' 并不意味着变量将被存储在寄存器中，它意味着变量可能存储在寄存器中，这取决于硬件和实现的限制。

static 存储类

static 存储类指示编译器在程序的生命周期内保持局部变量的存在，而不需要在每次它进入和离开作用域时进行创建和销毁。因此，使用 **static** 修饰局部变量可以在函数调用之间保持局部变量的值。

static 修饰符也可以应用于全局变量。当 **static** 修饰全局变量时，会使变量的作用域限制在声明它的文件内。

static 是全局变量的默认存储类，以下两个变量 (**count** 和 **road**) 都有一个 **static** 存储类。

```
static int Count;
int Road;

main()
{
    printf("%d\n", Count);
    printf("%d\n", Road);
}
```

以下实例演示了 **static** 修饰全局变量和局部变量的应用：

实例

```
#include <stdio.h>
/* 函数声明 */
void func1(void);

static int count=10;      /* 全局变量 - static 是默认的 */
int main()
{
    while (count--){
```



```
func1();
}
return 0;
}
void func1(void)
{
/* 'thingy' 是 'func1' 的局部变量 - 只初始化一次
 * 每次调用函数 'func1' 'thingy' 值不会被重置。
 */
static int thingy=5;
    thingy++;
    printf(" thingy 为 %d , count 为 %d\n", thingy, count);
}
```

实例中 count 作为全局变量可以在函数内使用，thingy 使用 static 修饰后，不会再每次调用时重置。

可能您现在还无法理解这个实例，因为我已经使用了函数和全局变量，这两个概念目前为止还没进行讲解。即使您现在不能完全理解，也没有关系，后续章节我们会详细讲解。当上面的代码被编译和执行时，它会产生下列结果：

```
thingy 为 6 , count 为 9
thingy 为 7 , count 为 8
thingy 为 8 , count 为 7
thingy 为 9 , count 为 6
thingy 为 10 , count 为 5
thingy 为 11 , count 为 4
thingy 为 12 , count 为 3
thingy 为 13 , count 为 2
thingy 为 14 , count 为 1
thingy 为 15 , count 为 0
```

extern 存储类

extern 存储类用于提供一个全局变量的引用，全局变量对所有的程序文件都是可见的。当您使用 'extern' 时，对于无法初始化的变量，会把变量名指向一个之前定义过的存储位置。

当您有多个文件且定义了一个可以在其他文件中使用的全局变量或函数时，可以在其他文件中使用 *extern* 来得到已定义的变量或函数的引用。可以这么理解，*extern* 是用来在另一个文件中声明一个全局变量或函数。

extern 修饰符通常用于当有两个或多个文件共享相同的全局变量或函数的时候，如下所示：

第一个文件：**main.c**

实例

```
#include <stdio.h>
int count ;
extern void write_extern();

int main()
{
    count = 5;
    write_extern();
}
```

第二个文件：**support.c**

实例

```
#include <stdio.h>
extern int count;

void write_extern(void)
{
    printf("count is %d\n", count);
}
```

在这里，第二个文件中的 *extern* 关键字用于声明已经在第一个文件 main.c 中定义的 *count*。现在，编译这两个文件，如下所示：

```
$ gcc main.c support.c
```

这会产生 **a.out** 可执行程序，当程序被执行时，它会产生下列结果：

```
count is 5
```

C 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C 语言内置了丰富的运算符，并提供了以下类型的运算符：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 杂项运算符

本章将逐一介绍算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符和其他运算符。

算术运算符

下表显示了 C 语言支持的所有算术运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
*	把两个操作数相乘	A * B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	自增运算符，整数值增加 1	A++ 将得到 11
--	自减运算符，整数值减少 1	A- 将得到 9

实例

请看下面的实例，了解 C 语言中所有可用的算术运算符：

实例

```
#include <stdio.h>
int main()
{
    int a = 21;
    int b = 10;
    int c ;

    c = a + b;
    printf("Line 1 - c 的值是 %d\n", c );
    c = a - b;
    printf("Line 2 - c 的值是 %d\n", c );
    c = a * b;
    printf("Line 3 - c 的值是 %d\n", c );
    c = a / b;
    printf("Line 4 - c 的值是 %d\n", c );
    c = a % b;
    printf("Line 5 - c 的值是 %d\n", c );
    c = a++; // 赋值后再加 1，c 为 21，a 为 22
    printf("Line 6 - c 的值是 %d\n", c );
    c = a--; // 赋值后再减 1，c 为 22，a 为 21
    printf("Line 7 - c 的值是 %d\n", c );

}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 31
```

Line 2 - c 的值是 11
Line 3 - c 的值是 210
Line 4 - c 的值是 2
Line 5 - c 的值是 1
Line 6 - c 的值是 21
Line 7 - c 的值是 22

以下实例演示了 `a++` 与 `++a` 的区别：

实例

```
#include <stdio.h>
int main()
{
    int c;
    int a = 10;
    c = a++;
    printf("先赋值后运算： \n");
    printf("Line 1 - c 的值是 %d\n", c );
    printf("Line 2 - a 的值是 %d\n", a );
    a = 10;
    c = a--;
    printf("Line 3 - c 的值是 %d\n", c );
    printf("Line 4 - a 的值是 %d\n", a );

    printf("先运算后赋值： \n");
    a = 10;
    c = ++a;
    printf("Line 5 - c 的值是 %d\n", c );
    printf("Line 6 - a 的值是 %d\n", a );
    a = 10;
    c = --a;
    printf("Line 7 - c 的值是 %d\n", c );
    printf("Line 8 - a 的值是 %d\n", a );

}
```

以上程序执行输出结果为：

先赋值后运算：
Line 1 - c 的值是 10
Line 2 - a 的值是 11
Line 3 - c 的值是 10
Line 4 - a 的值是 9
先运算后赋值：
Line 5 - c 的值是 11
Line 6 - a 的值是 11
Line 7 - c 的值是 9
Line 8 - a 的值是 9

关系运算符

下表显示了 C 语言支持的所有关系运算符。假设变量 **A** 的值为 10，变量 **B** 的值为 20，则：

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(A == B) 不为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(A != B) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(A > B) 不为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(A < B) 为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。	(A >= B) 不为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。	(A <= B) 为真。

实例

请看下面的实例，了解 C 语言中所有可用的关系运算符：

实例

```
#include <stdio.h>
int main()
{
    int a = 21;
        int b = 10;
        int c ;

        if( a == b )
    {
printf("Line 1 - a 等于 b\n" );
    }
    else
    {
printf("Line 1 - a 不等于 b\n" );
    }
    if ( a < b )
    {
printf("Line 2 - a 小于 b\n" );
    }
    else
    {
printf("Line 2 - a 不小于 b\n" );
    }
    if ( a > b )
    {
printf("Line 3 - a 大于 b\n" );
    }
    else
    {
printf("Line 3 - a 不大于 b\n" );
    }
    /* 改变 a 和 b 的值 */
    a = 5;
        b = 20;
        if ( a <= b )
    {
printf("Line 4 - a 小于或等于 b\n" );
    }
    if ( b >= a )
    {
printf("Line 5 - b 大于或等于 b\n" );
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - a 不等于 b
Line 2 - a 不小于 b
Line 3 - a 大于 b
Line 4 - a 小于或等于 b
Line 5 - b 大于或等于 b
```

逻辑运算符

下表显示了 C 语言支持的所有关系逻辑运算符。假设变量 **A** 的值为 1，变量 **B** 的值为 0，则：

运算符	描述	实例
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(A && B) 为假。
	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(A B) 为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符	!(A && B) 为真。

实例

请看下面的实例，了解 C 语言中所有可用的逻辑运算符：

实例

```
#include <stdio.h>
main()
{
    int a = 5;
    int b = 20;
    int c ;

    if ( a && b )
    {
        printf("Line 1 - 条件为真\n" );
    }
    if ( a || b )
    {
        printf("Line 2 - 条件为真\n" );
    }
    /* 改变 a 和 b 的值 */
    a = 0;
    b = 10;
    if ( a && b )
    {
        printf("Line 3 - 条件为真\n" );
    }
    else
    {
        printf("Line 3 - 条件不为真\n" );
    }
    if ( !(a && b) )
    {
        printf("Line 4 - 条件为真\n" );
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - 条件为真
Line 2 - 条件为真
Line 3 - 条件不为真
Line 4 - 条件为真
```

位运算符

位运算符作用于位，并逐位执行操作。&、| 和 ^ 的真值表如下所示：

p	q	p&q	p q	p^q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

假设如果 A = 60，且 B = 13，现在以二进制格式表示，它们如下所示：

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001
~A = 1100 0011

下表显示了 C 语言支持的位运算符。假设变量 **A** 的值为 60，变量 **B** 的值为 13，则：

运算符	描述	实例
&	如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。	(A & B) 将得到 12，即为 0000 1100
	如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	(A B) 将得到 61，即为 0011 1101
^	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	(A ^ B) 将得到 49，即为 0011 0001
~	二进制补码运算符是一元运算符，具有"翻转"位效果，即0变成1，1变成0。	(~A) 将得到 -61，即为 1100 0011，一个有符号二进制数的补码形式。
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	A << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	A >> 2 将得到 15，即为 0000 1111

实例

请看下面的实例，了解 C 语言中所有可用的位运算符：

实例

```
#include <stdio.h>
main()
{
    unsigned int a = 60;    /* 60 = 0011 1100 */
    unsigned int b = 13;    /* 13 = 0000 1101 */
    int c = 0;

    c = a & b;              /* 12 = 0000 1100 */
    printf("Line 1 - c 的值是 %d\n", c );

    c = a | b;              /* 61 = 0011 1101 */
    printf("Line 2 - c 的值是 %d\n", c );

    c = a ^ b;              /* 49 = 0011 0001 */
    printf("Line 3 - c 的值是 %d\n", c );

    c = ~a;                 /* -61 = 1100 0011 */
    printf("Line 4 - c 的值是 %d\n", c );

    c = a << 2;             /* 240 = 1111 0000 */
    printf("Line 5 - c 的值是 %d\n", c );

    c = a >> 2;             /* 15 = 0000 1111 */
    printf("Line 6 - c 的值是 %d\n", c );
}
```

当上面的代码被编译和执行时，它会产生下列结果：

Line 1 - c 的值是 12
Line 2 - c 的值是 61
Line 3 - c 的值是 49
Line 4 - c 的值是 -61
Line 5 - c 的值是 240
Line 6 - c 的值是 15

赋值运算符

下表列出了 C 语言支持的赋值运算符：

运算符	描述	实例
=	简单的赋值运算符，把右边操作数的值赋给左边操作数	C = A + B 将把 A + B 的值赋给 C

+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	C += A 相当于 C = C + A
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	C -= A 相当于 C = C - A
*=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	C *= A 相当于 C = C * A
/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	C /= A 相当于 C = C / A
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	C %= A 相当于 C = C % A
<<=	左移且赋值运算符	C <<= 2 等同于 C = C << 2
>>=	右移且赋值运算符	C >>= 2 等同于 C = C >> 2
&=	按位与且赋值运算符	C &= 2 等同于 C = C & 2
^=	按位异或且赋值运算符	C ^= 2 等同于 C = C ^ 2
=	按位或且赋值运算符	C = 2 等同于 C = C 2

实例

请看下面的实例，了解 C 语言中所有可用的赋值运算符：

实例

```
#include <stdio.h>
main()
{
    int a = 21;
    int c ;

    c = a;
    printf("Line 1 - = 运算符实例, c 的值 = %d\n", c );

    c += a;
    printf("Line 2 - += 运算符实例, c 的值 = %d\n", c );

    c -= a;
    printf("Line 3 - -= 运算符实例, c 的值 = %d\n", c );

    c *= a;
    printf("Line 4 - *= 运算符实例, c 的值 = %d\n", c );

    c /= a;
    printf("Line 5 - /= 运算符实例, c 的值 = %d\n", c );

    c = 200;
    c %= a;
    printf("Line 6 - %= 运算符实例, c 的值 = %d\n", c );

    c <<= 2;
    printf("Line 7 - <<= 运算符实例, c 的值 = %d\n", c );

    c >>= 2;
    printf("Line 8 - >>= 运算符实例, c 的值 = %d\n", c );

    c &= 2;
    printf("Line 9 - &= 运算符实例, c 的值 = %d\n", c );

    c ^= 2;
    printf("Line 10 - ^= 运算符实例, c 的值 = %d\n", c );

    c |= 2;
    printf("Line 11 - |= 运算符实例, c 的值 = %d\n", c );

}
```


当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - = 运算符实例, c 的值 = 21
Line 2 - += 运算符实例, c 的值 = 42
Line 3 - -= 运算符实例, c 的值 = 21
Line 4 - *= 运算符实例, c 的值 = 441
Line 5 - /= 运算符实例, c 的值 = 21
Line 6 - %= 运算符实例, c 的值 = 11
Line 7 - <<= 运算符实例, c 的值 = 44
Line 8 - >>= 运算符实例, c 的值 = 11
Line 9 - &= 运算符实例, c 的值 = 2
Line 10 - ^= 运算符实例, c 的值 = 0
Line 11 - |= 运算符实例, c 的值 = 2
```

杂项运算符 ↦ sizeof & 三元

下表列出了 C 语言支持的其他一些重要的运算符，包括 **sizeof** 和 **?:**。

运算符	描述	实例
sizeof()	返回变量的大小。	sizeof(a) 将返回 4，其中 a 是整数。
&	返回变量的地址。	&a; 将给出变量的实际地址。
*	指向一个变量。	*a; 将指向一个变量。
?:	条件表达式	如果条件为真？则值为 X: 否则值为 Y

实例

请看下面的实例，了解 C 语言中所有可用的杂项运算符：

实例

```
#include <stdio.h>
int main()
{
    int a = 4;
    short b;
    double c;
    int* ptr;

    /* sizeof 运算符实例 */
    printf("Line 1 - 变量 a 的大小 = %lu\n", sizeof(a) );
    printf("Line 2 - 变量 b 的大小 = %lu\n", sizeof(b) );
    printf("Line 3 - 变量 c 的大小 = %lu\n", sizeof(c) );

    /* & 和 * 运算符实例 */
    ptr = &a;    /* 'ptr' 现在包含 'a' 的地址 */
    printf("a 的值是 %d\n", a);
    printf("*ptr 是 %d\n", *ptr);

    /* 三元运算符实例 */
    a = 10;
    b = (a == 1) ? 20: 30;
    printf( "b 的值是 %d\n", b );

    b = (a == 10) ? 20: 30;
    printf( "b 的值是 %d\n", b );
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - 变量 a 的大小 = 4
Line 2 - 变量 b 的大小 = 2
Line 3 - 变量 c 的大小 = 8
a 的值是 4
*ptr 是 4
```

b 的值是 30
b 的值是 20

C 中的运算符优先级

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

例如 `x = 7 + 3 * 2`，在这里，`x` 被赋值为 13，而不是 20，因为运算符 `*` 具有比 `+` 更高的优先级，所以首先计算乘法 `3*2`，然后再加上 7。

下表将按运算符优先级从高到低列出各个运算符，具有较高优先级的运算符出现在表格的上面，具有较低优先级的运算符出现在表格的下面。在表达式中，较高优先级的运算符会优先被计算。

类别	运算符	结合性
后缀	<code>() [] -> . ++ --</code>	从左到右
一元	<code>+ - ! ~ ++ -- (type)* & sizeof</code>	从右到左
乘除	<code>* / %</code>	从左到右
加减	<code>+ -</code>	从左到右
移位	<code><< >></code>	从左到右
关系	<code>< <= > >=</code>	从左到右
相等	<code>== !=</code>	从左到右
位与 AND	<code>&</code>	从左到右
位异或 XOR	<code>^</code>	从左到右
位或 OR	<code> </code>	从左到右
逻辑与 AND	<code>&&</code>	从左到右
逻辑或 OR	<code> </code>	从左到右
条件	<code>?:</code>	从右到左
赋值	<code>= += -= *= /= %= >>= <<= &= ^= =</code>	从右到左
逗号	<code>,</code>	从左到右

实例

请看下面的实例，了解 C 语言中运算符的优先级：

实例

```
#include <stdio.h>
main()
{
    int a = 20;
    int b = 10;
    int c = 15;
    int d = 5;
    int e;

    e = (a + b) * c / d;      // ( 30 * 15 ) / 5
    printf("(a + b) * c / d 的值是 %d\n", e );

    e = ((a + b) * c) / d;    // (30 * 15 ) / 5
    printf("((a + b) * c) / d 的值是 %d\n", e );

    e = (a + b) * (c / d);    // (30) * (15/5)
    printf("(a + b) * (c / d) 的值是 %d\n", e );
}
```

```
e = a + (b * c) / d;    // 20 + (150/5)
printf("a + (b * c) / d 的值是 %d\n" , e );

return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

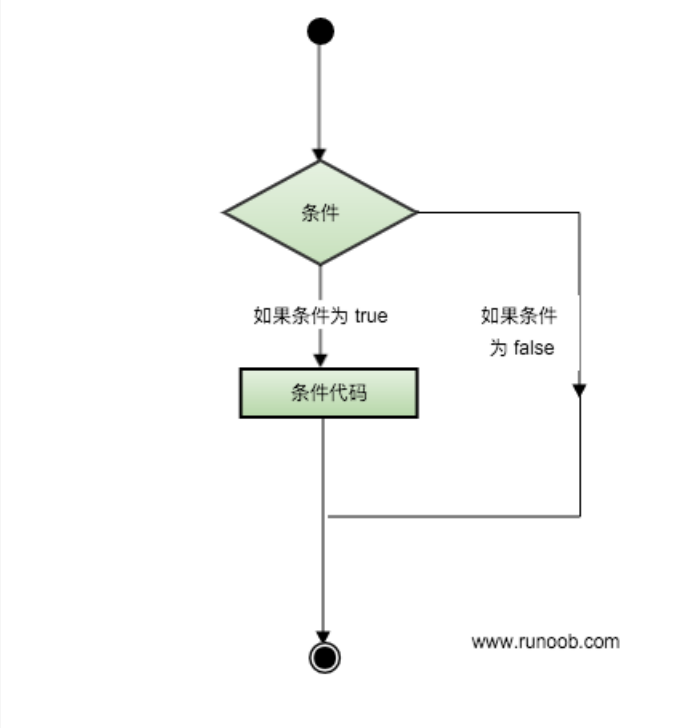
```
(a + b) * c / d 的值是 90
((a + b) * c) / d 的值是 90
(a + b) * (c / d) 的值是 90
a + (b * c) / d 的值是 50
```

C 判断

判断结构要求程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）。

C 语言把任何非零和非空的值假定为 **true**，把零或 **null** 假定为 **false**。

下面是大多数编程语言中典型的判断结构的一般形式：



判断语句

C 语言提供了以下类型的判断语句。点击链接查看每个语句的细节。

语句	描述
if 语句	一个 if 语句 由一个布尔表达式后跟一个或多个语句组成。
if...else 语句	一个 if 语句 后可跟一个可选的 else 语句 ，else 语句在布尔表达式为假时执行。
嵌套 if 语句	您可以在一个 if 或 else if 语句内使用另一个 if 或 else if 语句。
switch 语句	一个 switch 语句允许测试一个变量等于多个值时的情况。
嵌套 switch 语句	您可以在一个 switch 语句内使用另一个 switch 语句。

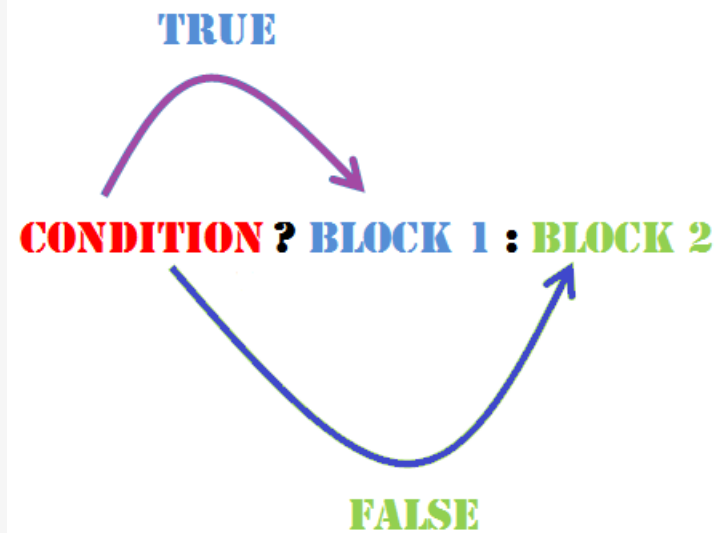
? : 运算符(三元运算符)

我们已经在前面的章节中讲解了 **条件运算符 ? :**，可以用来替代 **if...else** 语句。它的一般形式如下：

```
Exp1 ? Exp2 : Exp3;
```

其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和位置。

? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个 ? 表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个 ? 表达式的值。



实例

以下实例通过输入一个数字来判断它是否为奇数或偶数

实例

```
#include<stdio.h>
int main()
{
    int num;

    printf("输入一个数字 : ");
    scanf("%d",&num);

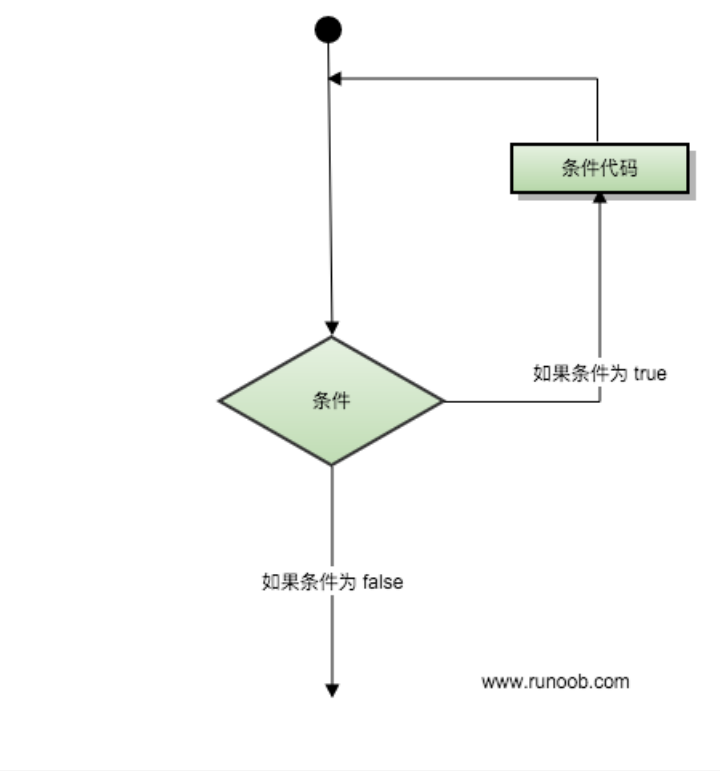
    (num%2==0)?printf("偶数"):printf("奇数");
}
```

C 循环

有的时候，我们可能需要多次执行同一块代码。一般情况下，语句是按顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。

编程语言提供了更为复杂执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的流程图：



循环类型

C 语言提供了以下几种循环类型。点击链接查看每个类型的细节。

循环类型	描述
while 循环	当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。
for 循环	多次执行一个语句序列，简化管理循环变量的代码。
do...while 循环	除了它是在循环主体结尾测试条件外，其他与 while 语句类似。
嵌套循环	您可以在 while、for 或 do..while 循环内使用一个或多个循环。

循环控制语句

循环控制语句改变你代码的执行顺序。通过它你可以实现代码的跳转。

C 提供了下列的循环控制语句。点击链接查看每个语句的细节。

控制语句	描述
break 语句	终止循环或 switch 语句，程序流将继续执行紧接着循环或 switch 的下一条语句。
continue 语句	告诉一个循环体立刻停止本次循环迭代，重新开始下次循环迭代。
goto 语句	将控制转移到被标记的语句。但是不建议在程序中使用 goto 语句。

无限循环

如果条件永远不为假，则循环将变成无限循环。**for** 循环在传统意义上可用于实现无限循环。由于构成循环的三个表达式中任何一个都不是必需的，您可以将某些条件表达式留空来构成一个无限循环。

实例

```
#include <stdio.h>
int main ()
{
    for( ; ; )
    {
        printf("该循环会永远执行下去! \n");
    }
    return 0;
}
```

当条件表达式不存在时，它被假设为真。您也可以设置一个初始值和增量表达式，但是一般情况下，C 程序员偏向于使用 for(;;) 结构来表示一个无限循环。

注意：您可以按 **Ctrl + C** 键终止一个无限循环。

C 函数

函数是一组一起执行一个任务的语句。每个 C 程序都至少有一个函数，即主函数 **main()**，所有简单的程序都可以定义其他额外的函数。您可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由您来决定的，但在逻辑上，划分通常是每个函数执行一个特定的任务来进行的。

函数声明告诉编译器函数的名称、返回类型和参数。函数定义提供了函数的实际主体。

C 标准库提供了大量的程序可以调用的内置函数。例如，函数 **strcat()** 用来连接两个字符串，函数 **memcpy()** 用来复制内存到另一个位置。

函数还有很多叫法，比如方法、子例程或程序，等等。

定义函数

C 语言中的函数定义的一般形式如下：

```
return_type function_name( parameter list )
{
    body of the function
}
```

在 C 语言中，函数由一个函数头和一个函数主体组成。下面列出一个函数的所有组成部分：

返回类型：一个函数可以返回一个值。**return_type** 是函数返回的值的数据类型。有些函数执行所需的操作而不返回值，在这种情况下，**return_type** 是关键字 **void**。

函数名称：这是函数的实际名称。函数名和参数列表一起构成了函数签名。

参数：参数就像是占位符。当函数被调用时，您向参数传递一个值，这个值被称为实际参数。参数列表包括函数参数的类型、顺序、数量。参数是可选的，也就是说，函数可能不包含参数。

函数主体：函数主体包含一组定义函数执行任务的语句。

实例

以下是 **max()** 函数的源代码。该函数有两个参数 **num1** 和 **num2**，会返回这两个数中较大的那个数：

```
/* 函数返回两个数中较大的那个数 */
int max(int num1, int num2)
{
    /* 局部变量声明 */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

函数声明

函数声明会告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。

函数声明包括以下几个部分：

```
return_type function_name( parameter list );
```

针对上面定义的函数 **max()**，以下是函数声明：

```
int max(int num1, int num2);
```


在函数声明中，参数的名称并不重要，只有参数的类型是必需的，因此下面也是有效的声明：

```
int max(int, int);
```

当您在源文件中定义函数且在另一个文件中调用函数时，函数声明是必需的。在这种情况下，您应该在调用函数的文件顶部声明函数。

调用函数

创建 C 函数时，会定义函数做什么，然后通过调用函数来完成已定义的任务。

当程序调用函数时，程序控制权会转移给被调用的函数。被调用的函数执行已定义的任务，当函数的返回语句被执行时，或到达函数的结束括号时，会把程序控制权交还给主程序。

调用函数时，传递所需参数，如果函数返回一个值，则可以存储返回值。例如：

```
#include <stdio.h>

/* 函数声明 */
int max(int num1, int num2);

int main ()
{
    /* 局部变量定义 */
    int a = 100;
    int b = 200;
    int ret;

    /* 调用函数来获取最大值 */
    ret = max(a, b);

    printf( "Max value is : %d\n", ret );

    return 0;
}

/* 函数返回两个数中较大的那个数 */
int max(int num1, int num2)
{
    /* 局部变量声明 */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

把 max() 函数和 main() 函数放一块，编译源代码。当运行最后的可执行文件时，会产生下列结果：

```
Max value is : 200
```

函数参数

如果函数要使用参数，则必须声明接受参数值的变量。这些变量称为函数的**形式参数**。

形式参数就像函数内的其他局部变量，在进入函数时被创建，退出函数时被销毁。

当调用函数时，有两种向函数传递参数的方式：

调用类型	描述
传值调用	该方法把参数的实际值复制给函数的形式参数。在这种情况下，修改函数内的形式参数不会影响实际参数。

<u>引用调用</u>	通过指针传递方式，形参为指向实参地址的指针，当对形参的指向操作时，就相当于对实参本身进行的操作。
-------------	--

默认情况下，C 使用**传值调用**来传递参数。一般来说，这意味着函数内的代码不能改变用于调用函数的实际参数。

C 作用域规则

任何一种编程中，作用域是程序中定义的变量所存在的区域，超过该区域变量就不能被访问。C 语言中有三个地方可以声明变量：

1. 在函数或块内部的**局部**变量
2. 在所有函数外部的**全局**变量
3. 在**形式**参数的函数参数定义中

让我们来看看什么是**局部**变量、**全局**变量和**形式**参数。

局部变量

在某个函数或块的内部声明的变量称为局部变量。它们只能被该函数或该代码块内部的语句使用。局部变量在函数外部是不可知的。下面是使用局部变量的实例。在这里，所有的变量 a、b 和 c 是 main() 函数的局部变量。

实例

```
#include <stdio.h>
int main ()
{
    /* 局部变量声明 */
    int a, b;
    int c;

    /* 实际初始化 */
    a = 10;
    b = 20;
    c = a + b;

    printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

    return 0;
}
```

全局变量

全局变量是定义在函数外部，通常是在程序的顶部。全局变量在整个程序生命周期内都是有效的，在任意的函数内部能访问全局变量。

全局变量可以被任何函数访问。也就是说，全局变量在声明后整个程序中都是可用的。下面是使用全局变量和局部变量的实例：

实例

```
#include <stdio.h>
/* 全局变量声明 */
int g;

int main ()
{
    /* 局部变量声明 */
    int a, b;

    /* 实际初始化 */
    a = 10;
    b = 20;
    g = a + b;

    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

    return 0;
}
```

在程序中，局部变量和全局变量的名称可以相同，但是在函数内，局部变量的值会覆盖全局变量的值。下面是一个实例：

实例

```
#include <stdio.h>
/* 全局变量声明 */
int g = 20;
```

```
int main ()
{
/* 局部变量声明 */
int g = 10;

printf ("value of g = %d\n", g);

return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of g = 10
```

形式参数

函数的参数，形式参数，被当作该函数内的局部变量，它们会优先覆盖全局变量。下面是一个实例：

实例

```
#include <stdio.h>
/* 全局变量声明 */
int a = 20;

int main ()
{
/* 在主函数中的局部变量声明 */
int a = 10;
int b = 20;
int c = 0;
int sum(int, int);

printf ("value of a in main() = %d\n", a);
c = sum( a, b);
printf ("value of c in main() = %d\n", c);

return 0;
}
/* 添加两个整数的函数 */
int sum(int a, int b)
{
printf ("value of a in sum() = %d\n", a);
printf ("value of b in sum() = %d\n", b);

return a + b;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

初始化局部变量和全局变量

当局部变量被定义时，系统不会对其初始化，您必须自行对其初始化。定义全局变量时，系统会自动对其初始化，如下所示：

数据类型	初始化默认值
int	0
char	'\0'
float	0
double	0

pointer

NULL

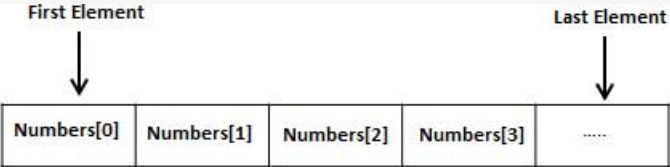
正确地初始化变量是一个良好的编程习惯，否则有时候程序可能会产生意想不到的结果，因为未初始化的变量会导致一些在内存位置中已经可用的垃圾值。

C 数组

C 语言支持**数组**数据结构，它可以存储一个固定大小的相同类型元素的顺序集合。数组是用来存储一系列数据，但它往往被认为是一系列相同类型的变量。

数组的声明并不是声明一个个单独的变量，比如 number0、number1、...、number99，而是声明一个数组变量，比如 numbers，然后使用 numbers[0]、numbers[1]、...、numbers[99] 来代表一个个单独的变量。数组中的特定元素可以通过索引访问。

所有的数组都是由连续的内存位置组成。最低的地址对应第一个元素，最高的地址对应最后一个元素。



声明数组

在 C 中要声明一个数组，需要指定元素的类型和元素的数量，如下所示：

```
type arrayName [ arraySize ];
```

这叫做一维数组。**arraySize** 必须是一个大于零的整数常量，**type** 可以是任意有效的 C 数据类型。例如，要声明一个类型为 double 的包含 10 个元素的数组 **balance**，声明语句如下：

```
double balance[10];
```

现在 *balance* 是一个可用的数组，可以容纳 10 个类型为 double 的数字。

初始化数组

在 C 中，您可以逐个初始化数组，也可以使用一个初始化语句，如下所示：

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

大括号 {} 之间的值的数目不能大于我们在数组声明时在方括号 [] 中指定的元素数目。

如果您省略掉了数组的大小，数组的大小则为初始化时元素的个数。因此，如果：

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

您将创建一个数组，它与前一个实例中所创建的数组是完全相同的。下面是一个为数组中某个元素赋值的实例：

```
balance[4] = 50.0;
```

上述的语句把数组中第五个元素的值赋为 50.0。所有的数组都是以 0 作为它们第一个元素的索引，也被称为基索引，数组的最后一个索引是数组的总大小减去 1。以下是上面所讨论的数组的图形表示：

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

访问数组元素

数组元素可以通过数组名称加索引进行访问。元素的索引是放在方括号内，跟在数组名称的后边。例如：

```
double salary = balance[9];
```

上面的语句将把数组中第 10 个元素的值赋给 salary 变量。下面的实例使用了上述的三个概念，即，声明数组、数组赋值、访问数组：

实例

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

```
int main ()
{
int n[ 10 ]; /* n 是一个包含 10 个整数的数组 */
int i,j;

/* 初始化数组元素 */
for ( i = 0; i < 10; i++ )
{
n[ i ] = i + 100; /* 设置元素 i 为 i + 100 */
}
/* 输出数组中每个元素的值 */
for (j = 0; j < 10; j++ )
{
printf("Element[%d] = %d\n", j, n[j] );
}
return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

C 中数组详解

在 C 中，数组是非常重要的，我们需要了解更多有关数组的细节。下面列出了 C 程序员必须清楚的一些与数组相关的重要概念：

概念	描述
多维数组	C 支持多维数组。多维数组最简单的形式是二维数组。
传递数组给函数	您可以通过指定不带索引的数组名称来给函数传递一个指向数组的指针。
从函数返回数组	C 允许从函数返回数组。
指向数组的指针	您可以通过指定不带索引的数组名称来生成一个指向数组中第一个元素的指针。

C 指针

学习 C 语言的指针既简单又有趣。通过指针，可以简化一些 C 编程任务的执行，还有一些任务，如动态内存分配，没有指针是无法执行的。所以，想要成为一名优秀的 C 程序员，学习指针是很有必要的。

正如您所知道的，每一个变量都有一个内存位置，每一个内存位置都定义了可使用连字号（&）运算符访问的地址，它表示了在内存中的一个地址。请看下面的实例，它将输出定义的变量地址：

实例

```
#include <stdio.h>
int main ()
{
    int var1;
    char var2[10];

    printf("var1 变量的地址:  %p\n", &var1 );
    printf("var2 变量的地址:  %p\n", &var2 );

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
var1 变量的地址:  0x7fff5cc109d4
var2 变量的地址:  0x7fff5cc109de
```

通过上面的实例，我们了解了什么是内存地址以及如何访问它。接下来让我们看看什么是指针。

什么是指针？

指针是一个变量，其值为另一个变量的地址，即，内存位置的直接地址。就像其他变量或常量一样，您必须在使用指针存储其他变量地址之前，对其进行声明。指针变量声明的一般形式为：

```
type *var-name;
```

在这里，**type** 是指针的基类型，它必须是一个有效的 C 数据类型，**var-name** 是指针变量的名称。用来声明指针的星号 * 与乘法中使用的星号是相同的。但是，在这个语句中，星号是用来指定一个变量是指针。以下是有效的指针声明：

```
int    *ip;    /* 一个整型的指针 */
double *dp;    /* 一个 double 型的指针 */
float  *fp;    /* 一个浮点型的指针 */
char   *ch;    /* 一个字符型的指针 */
```

所有指针的值的实际数据类型，不管是整型、浮点型、字符型，还是其他的数据类型，都是一样的，都是一个代表内存地址的长的十六进制数。不同数据类型的指针之间唯一的不同是，指针所指向的变量或常量的数据类型不同。

如何使用指针？

使用指针时会频繁进行以下几个操作：定义一个指针变量、把变量地址赋值给指针、访问指针变量中可用地址的值。这些是通过使用一元运算符 * 来返回位于操作数所指定地址的变量的值。下面的实例涉及到了这些操作：

实例

```
#include <stdio.h>
int main ()
{
    int var = 20;    /* 实际变量的声明 */
    int *ip;         /* 指针变量的声明 */
    ip = &var;       /* 在指针变量中存储 var 的地址 */
    printf("Address of var variable: %p\n", &var );

    /* 在指针变量中存储的地址 */
    printf("Address stored in ip variable: %p\n", ip );

    /* 使用指针访问值 */
}
```



```
printf("Value of *ip variable: %d\n", *ip );

return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

C 中的 NULL 指针

在变量声明的时候，如果没有确切的地址可以赋值，为指针变量赋一个 NULL 值是一个良好的编程习惯。赋为 NULL 值的指针被称为空指针。

NULL 指针是一个定义在标准库中的值为零的常量。请看下面的程序：

实例

```
#include <stdio.h>
int main ()
{
    int *ptr = NULL;

    printf("ptr 的值是 %p\n", ptr );

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
ptr 的值是 0
```

在大多数的操作系统上，程序不允许访问地址为 0 的内存，因为该内存是操作系统保留的。然而，内存地址 0 有特别重要的意义，它表明该指针不指向一个可访问的内存位置。但按照惯例，如果指针包含空值（零值），则假定它不指向任何东西。

如需检查一个空指针，您可以使用 if 语句，如下所示：

```
if(ptr) /* 如果 p 非空，则完成 */
if(!ptr) /* 如果 p 为空，则完成 */
```

C 指针详解

在 C 中，有很多指针相关的概念，这些概念都很简单，但是都很重要。下面列出了 C 程序员必须清楚的一些与指针相关的重要概念：

概念	描述
指针的算术运算	可以对指针进行四种算术运算：++、--、+、-
指针数组	可以定义用来存储指针的数组。
指向指针的指针	C 允许指向指针的指针。
传递指针给函数	通过引用或地址传递参数，使传递的参数在调用函数中被改变。
从函数返回指针	C 允许函数返回指针到局部变量、静态变量和动态内存分配。

函数指针

函数指针是指向函数的指针变量。

通常我们说的指针变量是指向一个整型、字符型或数组等变量，而函数指针是指向函数。

函数指针可以像一般函数一样，用于调用函数、传递参数。

函数指针变量的声明：

```
typedef int (*fun_ptr)(int,int); // 声明一个指向同样参数、返回值的函数指针类型
```

实例

以下实例声明了函数指针变量 p，指向函数 max：

实例

```
#include <stdio.h>
int max(int x, int y)
{
    return x > y ? x : y;
}
int main(void)
{
    /* p 是函数指针 */
    int (* p)(int, int) = & max; // &可以省略
    int a, b, c, d;

    printf("请输入三个数字:");
    scanf("%d %d %d", & a, & b, & c);

    /* 与直接调用函数等价，d = max(max(a, b), c) */
    d = p(p(a, b), c);

    printf("最大的数字是: %d\n", d);

    return 0;
}
```

编译执行，输出结果如下：

```
请输入三个数字:1 2 3
最大的数字是: 3
```

回调函数

函数指针作为某个函数的参数

函数指针变量可以作为某个函数的参数来使用的，回调函数就是一个通过函数指针调用的函数。

简单讲：回调函数是由别人的函数执行时调用你实现的函数。

以下是自知乎作者常溪玲的解说：

你到一个商店买东西，刚好你要的东西没有货，于是你在店员那里留下了你的电话，过了几天店里有货了，店员就打了你的电话，然后你接到电话后就到店里去取了货。在这个例子里，你的电话号码就叫回调函数，你把电话留给店员就叫登记回调函数，店里后来有货了叫做触发了回调关联的事件，店员给你打电话叫做调用回调函数，你到店里去取货叫做响应回调事件。

实例

实例中 populate_array 函数定义了三个参数，其中第三个参数是函数的指针，通过该函数来设置数组的值。

实例中我们定义了回调函数 getNextRandomValue，它返回一个随机值，它作为一个函数指针传递给 populate_array 函数。

populate_array 将调用 10 次回调函数，并将回调函数的返回值赋值给数组。

实例

```
#include <stdlib.h>
```

```
#include <stdio.h>
// 回调函数
void populate_array(int *array, size_t arraySize, int (*getNextValue)(void))
{
    for (size_t i=0; i<arraySize; i++)
        array[i] = getNextValue();
}
// 获取随机值
int getNextRandomValue(void)
{
    return rand();
}
int main(void)
{
    int myarray[10];
    populate_array(myarray, 10, getNextRandomValue);
    for(int i = 0; i < 10; i++) {
        printf("%d ", myarray[i]);
    }
    printf("\n");
    return 0;
}
```

编译执行，输出结果如下：

```
16807 282475249 1622650073 984943658 1144108930 470211272 101027544 1457850878 1458777923 2007237709
```

C 字符串

在 C 语言中，字符串实际上是使用 **null** 字符 `\0` 终止的一维字符数组。因此，一个以 `null` 结尾的字符串，包含了组成字符串的字符。

下面的声明和初始化创建了一个 "Hello" 字符串。由于在数组的末尾存储了空字符，所以字符数组的大小比单词 "Hello" 的字符数多一个。

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

依据数组初始化规则，您可以把上面的语句写成以下语句：

```
char greeting[] = "Hello";
```

以下是 C/C++ 中定义的字符串的内存表示：

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

其实，您不需要把 `null` 字符放在字符串常量的末尾。C 编译器会在初始化数组时，自动把 `\0` 放在字符串的末尾。让我们尝试输出上面的字符串：

```
#include <stdio.h>

int main ()
{
    char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

    printf("Greeting message: %s\n", greeting );

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Greeting message: Hello
```

C 中有大量操作字符串的函数：

序号	函数 & 目的
1	strcpy(s1, s2); 复制字符串 s2 到字符串 s1。
2	strcat(s1, s2); 连接字符串 s2 到字符串 s1 的末尾。
3	strlen(s1); 返回字符串 s1 的长度。
4	strcmp(s1, s2); 如果 s1 和 s2 是相同的，则返回 0；如果 s1<s2 则返回小于 0；如果 s1>s2 则返回大于 0。
5	strchr(s1, ch); 返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置。
6	strstr(s1, s2);

返回一个指针，指向字符串 s1 中字符串 s2 的第一次出现的位置。

下面的实例使用了上述的一些函数：

```
#include <stdio.h>
#include <string.h>

int main ()
{
    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    int len ;

    /* 复制 str1 到 str3 */
    strcpy(str3, str1);
    printf("strcpy( str3, str1) : %s\n", str3 );

    /* 连接 str1 和 str2 */
    strcat( str1, str2);
    printf("strcat( str1, str2):  %s\n", str1 );

    /* 连接后，str1 的总长度 */
    len = strlen(str1);
    printf("strlen(str1) : %d\n", len );

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
strcpy( str3, str1) : Hello
strcat( str1, str2):  HelloWorld
strlen(str1) : 10
```

您可以在 C 标准库中找到更多字符串相关的函数。

C 结构体

C 数组允许定义可存储相同类型数据项的变量，**结构**是 C 编程中另一种用户自定义的可用的数据类型，它允许您存储不同类型的数据项。结构用于表示一条记录，假设您想要跟踪图书馆中书本的动态，您可能需要跟踪每本书的下列属性：

Title
Author
Subject
Book ID

定义结构

为了定义结构，您必须使用 **struct** 语句。struct 语句定义了一个包含多个成员的新的数据类型，struct 语句的格式如下：

```
struct [structure tag]
{
    member definition;
    member definition;
    ...
    member definition;
} [one or more structure variables];
```

structure tag 是可选的，每个 member definition 是标准的变量定义，比如 int i; 或者 float f; 或者其他有效的变量定义。在结构定义的末尾，最后一个分号之前，您可以指定一个或多个结构变量，这是可选的。下面是声明 Book 结构的方式：

```
struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} book;
```

访问结构成员

为了访问结构的成员，我们使用**成员访问运算符**（.）。成员访问运算符是结构变量名称和我们要访问的结构成员之间的一个句号。您可以使用 **struct** 关键字来定义结构类型的变量。下面的实例演示了结构的用法：

实例

```
#include <stdio.h>
#include <string.h>
struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

int main( )
{
    struct Books Book1;          /* 声明 Book1, 类型为 Books */
    struct Books Book2;          /* 声明 Book2, 类型为 Books */
    /* Book1 详述 */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Nuha Ali");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;

    /* Book2 详述 */
    strcpy( Book2.title, "Telecom Billing");
    strcpy( Book2.author, "Zara Ali");
    strcpy( Book2.subject, "Telecom Billing Tutorial");
    Book2.book_id = 6495700;

    /* 输出 Book1 信息 */
```

```

printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);

/* 输出 Book2 信息 */
printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);

return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700

```

结构作为函数参数

您可以把结构作为函数参数，传参方式与其他类型的变量或指针类似。您可以使用上面实例中的方式来访问结构变量：

实例

```

#include <stdio.h>
#include <string.h>
struct Books
{
char title[50];
char author[50];
char subject[100];
int book_id;
};

/* 函数声明 */
void printBook( struct Books book );
int main( )
{
struct Books Book1;          /* 声明 Book1, 类型为 Books */
struct Books Book2;          /* 声明 Book2, 类型为 Books */
/* Book1 详述 */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;

/* Book2 详述 */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;

/* 输出 Book1 信息 */
printBook( Book1 );

/* 输出 Book2 信息 */
printBook( Book2 );

return 0;
}
void printBook( struct Books book )
{
printf( "Book title : %s\n", book.title);

```

```
printf( "Book author : %s\n", book.author);
printf( "Book subject : %s\n", book.subject);
printf( "Book book_id : %d\n", book.book_id);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

指向结构的指针

您可以定义指向结构的指针，方式与定义指向其他类型变量的指针相似，如下所示：

```
struct Books *struct_pointer;
```

现在，您可以在上述定义的指针变量中存储结构变量的地址。为了查找结构变量的地址，请把 & 运算符放在结构名称的前面，如下所示：

```
struct_pointer = &Book1;
```

为了使用指向该结构的指针访问结构的成员，您必须使用 -> 运算符，如下所示：

```
struct_pointer->title;
```

让我们使用结构指针来重写上面的实例，这将有助于您理解结构指针的概念：

实例

```
#include <stdio.h>
#include <string.h>
struct Books
{
char title[50];
char author[50];
char subject[100];
int book_id;
};

/* 函数声明 */
void printBook( struct Books *book );
int main( )
{
struct Books Book1;          /* 声明 Book1, 类型为 Books */
struct Books Book2;          /* 声明 Book2, 类型为 Books */
/* Book1 详述 */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;

/* Book2 详述 */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;

/* 通过传 Book1 的地址来输出 Book1 信息 */
printBook( &Book1 );
```



```
/* 通过传 Book2 的地址来输出 Book2 信息 */
printBook( &Book2 );

return 0;
}
void printBook( struct Books *book )
{
printf( "Book title : %s\n", book->title);
printf( "Book author : %s\n", book->author);
printf( "Book subject : %s\n", book->subject);
printf( "Book book_id : %d\n", book->book_id);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

位域

有些信息在存储时，并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态，用 1 位二进制位即可。为了节省存储空间，并使处理简便，C 语言又提供了一种数据结构，称为"位域"或"位段"。

所谓"位域"是把一个字节中的二进制位划分为几个不同的区域，并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。

典型的实例：

用 1 位二进制存放一个开关量时，只有 0 和 1 两种状态。

读取外部文件格式——可以读取非标准的文件格式。例如：9 位的整数。

位域的定义和位域变量的说明

位域定义与结构定义相仿，其形式为：

```
struct 位域结构名
{

    位域列表

};
```

其中位域列表的形式为：

类型说明符 位域名：位域长度

例如：

```
struct bs{
int a:8;
    int b:2;
    int c:6;
}data;
```

说明 data 为 bs 变量，共占两个字节。其中位域 a 占 8 位，位域 b 占 2 位，位域 c 占 6 位。

让我们再来看一个实例：

```
struct packed_struct {
unsigned int f1:1;
    unsigned int f2:1;
```

```
unsigned int f3:1;
unsigned int f4:1;
unsigned int type:4;
unsigned int my_int:9;
} pack;
```

在这里，packed_struct 包含了 6 个成员：四个 1 位的标识符 f1..f4、一个 4 位的 type 和一个 9 位的 my_int。

对于位域的定义尚有以下几点说明：

一个位域必须存储在同一个字节中，不能跨两个字节。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。例如：

```
struct bs{
unsigned a:4;
    unsigned :4;    /* 空域 */
unsigned b:4;    /* 从下一单元开始存放 */
unsigned c:4
}
```

在这个位域定义中，a 占第一字节的 4 位，后 4 位填 0 表示不使用，b 从第二字节开始，占用 4 位，c 占用 4 位。

由于位域不允许跨两个字节，因此位域的长度不能大于一个字节的长度，也就是说不能超过 8 位二进制。如果最大长度大于计算机的整数字长，一些编译器可能会允许域的内存重叠，另外一些编译器可能会把大于一个域的部分存储在下一个字中。

位域可以是无名位域，这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：

```
struct k{
int a:1;
    int :2;    /* 该 2 位不能使用 */
int b:3;
    int c:2;
};
```

从以上分析可以看出，位域在本质上就是一种结构类型，不过其成员是按二进制分配的。

位域的使用

位域的使用和结构成员的使用相同，其一般形式为：

```
位域变量名.位域名
位域变量名->位域名
```

位域允许用各种格式输出。

请看下面的实例：

实例

```
main(){
struct bs{
unsigned a:1;
    unsigned b:3;
    unsigned c:4;
} bit,*pbit;
bit.a=1;    /* 给位域赋值（应注意赋值不能超过该位域的允许范围） */
bit.b=7;    /* 给位域赋值（应注意赋值不能超过该位域的允许范围） */
bit.c=15;    /* 给位域赋值（应注意赋值不能超过该位域的允许范围） */
printf("%d,%d,%d\n",bit.a,bit.b,bit.c);    /* 以整型量格式输出三个域的内容 */
pbit=&bit;    /* 把位域变量 bit 的地址送给指针变量 pbit */
pbit->a=0;    /* 用指针方式给位域 a 重新赋值，赋为 0 */
pbit->b&=3;    /* 使用了复合的位运算符 "&="，相当于：pbit->b=pbit->b&3，位域 b 中原有值为 7，与 3 作按位与运算的结果为 3（111&011=011，十进制值为 3） */
pbit->c|=1;    /* 使用了复合位运算符 "|="，相当于：pbit->c=pbit->c|1，其结果为 15 */
printf("%d,%d,%d\n",pbit->a,pbit->b,pbit->c);    /* 用指针方式输出了这三个域的值 */
}
```

上例程序中定义了位域结构 bs，三个位域为 a、b、c。说明了 bs 类型的变量 bit 和指向 bs 类型的指针变量 pbit。这表示位域也是可以使用指针的。

C 共用体

共用体是一种特殊的数据类型，允许您在相同的内存位置存储不同的数据类型。您可以定义一个带有多成员的共用体，但是任何时候只能有一个成员带有值。共用体提供了一种使用相同的内存位置的有效方式。

定义共用体

为了定义共用体，您必须使用 **union** 语句，方式与定义结构类似。**union** 语句定义了一个新的数据类型，带有多成员。**union** 语句的格式如下：

```
union [union tag]
{
    member definition;
    member definition;
    ...
    member definition;
} [one or more union variables];
```

union tag 是可选的，每个 **member definition** 是标准的变量定义，比如 `int i;` 或者 `float f;` 或者其他有效的变量定义。在共用体定义的末尾，最后一个分号之前，您可以指定一个或多个共用体变量，这是可选的。下面定义一个名为 **Data** 的共用体类型，有三个成员 **i**、**f** 和 **str**：

```
union Data
{
    int i;
    float f;
    char str[20];
} data;
```

现在，**Data** 类型的变量可以存储一个整数、一个浮点数，或者一个字符串。这意味着一个变量（相同的内存位置）可以存储多个多种类型的数据。您可以根据需要在一个共用体内使用任何内置的或者用户自定义的数据类型。

共用体占用的内存应足够存储共用体中最大的成员。例如，在上面的实例中，**Data** 将占用 20 个字节的内存空间，因为在各个成员中，字符串所占用的空间是最大的。下面的实例将显示上面的共用体占用的总内存大小：

实例

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i;
    float f;
    char str[20];
};

int main( )
{
    union Data data;

    printf( "Memory size occupied by data : %d\n", sizeof(data));

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Memory size occupied by data : 20
```

访问共用体成员

为了访问共用体的成员，我们使用**成员访问运算符**（**.**）。成员访问运算符是共用体变量名称和我们要访问的共用体成员之间的一个句号。您可以使用 **union** 关键字来定义共用体类型的变量。下面的实例演示了共用体的用法：

实例

```
#include <stdio.h>
#include <string.h>
union Data
{
```

```

int i;
float f;
char str[20];
};

int main( )
{
union Data data;

    data.i = 10;
    data.f = 220.5;
    strcpy( data.str, "C Programming");

    printf( "data.i : %d\n", data.i);
    printf( "data.f : %f\n", data.f);
    printf( "data.str : %s\n", data.str);

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming

```

在这里，我们可以看到共用体的 **i** 和 **f** 成员的值有损坏，因为最后赋给变量的值占用了内存位置，这也是 **str** 成员能够完好输出的原因。现在让我们再来看一个相同的实例，这次我们在同一时间只使用一个变量，这也演示了使用共用体的主要目的：

实例

```

#include <stdio.h>
#include <string.h>
union Data
{
int i;
float f;
char str[20];
};

int main( )
{
union Data data;

    data.i = 10;
    printf( "data.i : %d\n", data.i);

    data.f = 220.5;
    printf( "data.f : %f\n", data.f);

    strcpy( data.str, "C Programming");
    printf( "data.str : %s\n", data.str);

    return 0;
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

data.i : 10
data.f : 220.500000
data.str : C Programming

```

在这里，所有的成员都能完好输出，因为同一时间只用到一个成员。

C 位域

如果程序的结构中包含多个开关量，只有 TRUE/FALSE 变量，如下：

```
struct
{
    unsigned int widthValidated;
    unsigned int heightValidated;
} status;
```

这种结构需要 8 字节的内存空间，但在实际上，在每个变量中，我们只存储 0 或 1。在这种情况下，C 语言提供了一种更好的利用内存空间的方式。如果您在结构内使用这样的变量，您可以定义变量的宽度来告诉编译器，您将只使用这些字节。例如，上面的结构可以重写成：

```
struct
{
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status;
```

现在，上面的结构中，**status** 变量将占用 4 个字节的内存空间，但是只有 2 位被用来存储值。如果您用了 32 个变量，每一个变量宽度为 1 位，那么 **status** 结构将使用 4 个字节，但只要您再多用一个变量，如果使用了 33 个变量，那么它将分配内存的下一段来存储第 33 个变量，这个时候就开始使用 8 个字节。让我们看看下面的实例来理解这个概念：

```
#include <stdio.h>
#include <string.h>

/* 定义简单的结构 */
struct
{
    unsigned int widthValidated;
    unsigned int heightValidated;
} status1;

/* 定义位域结构 */
struct
{
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status2;

int main( )
{
    printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
    printf( "Memory size occupied by status2 : %d\n", sizeof(status2));

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

位域声明

在结构内声明位域的形式如下：

```
struct
{
```

```
type [member_name] : width ;
};
```

下面是有关位域中变量元素的描述：

元素	描述
type	整数类型，决定了如何解释位域的值。类型可以是整型、有符号整型、无符号整型。
member_name	位域的名称。
width	位域中位的数量。宽度必须小于或等于指定类型的位宽度。

带有预定义宽度的变量被称为**位域**。位域可以存储多于 1 位的数，例如，需要一个变量来存储从 0 到 7 的值，您可以定义一个宽度为 3 位的位域，如下：

```
struct
{
    unsigned int age : 3;
} Age;
```

上面的结构定义指示 C 编译器，age 变量将只使用 3 位来存储这个值，如果您试图使用超过 3 位，则无法完成。让我们来看下面的实例：

```
#include <stdio.h>
#include <string.h>

struct
{
    unsigned int age : 3;
} Age;

int main( )
{
    Age.age = 4;
    printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
    printf( "Age.age : %d\n", Age.age );

    Age.age = 7;
    printf( "Age.age : %d\n", Age.age );

    Age.age = 8;
    printf( "Age.age : %d\n", Age.age );

    return 0;
}
```

当上面的代码被编译时，它会带有警告，当上面的代码被执行时，它会产生下列结果：

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

C typedef

C 语言提供了 **typedef** 关键字，您可以使用它来为类型取一个新的名字。下面的实例为单字节数字定义了一个术语 **BYTE**:

```
typedef unsigned char BYTE;
```

在这个类型定义之后，标识符 **BYTE** 可作为类型 **unsigned char** 的缩写，例如：

```
BYTE    b1, b2;
```

按照惯例，定义时会大写字母，以便提醒用户类型名称是一个象征性的缩写，但您也可以使用小写字母，如下：

```
typedef unsigned char byte;
```

您也可以使用 **typedef** 来为用户自定义的数据类型取一个新的名字。例如，您可以对结构体使用 **typedef** 来定义一个新的数据类型名字，然后使用这个新的数据类型来直接定义结构变量，如下：

实例

```
#include <stdio.h>
#include <string.h>
typedef struct Books
{
char title[50];
char author[50];
char subject[100];
int book_id;
} Book;

int main( )
{
Book book;

strcpy( book.title, "C 教程");
strcpy( book.author, "Runoob");
strcpy( book.subject, "编程语言");
book.book_id = 12345;

printf( "书标题 : %s\n", book.title);
printf( "书作者 : %s\n", book.author);
printf( "书类目 : %s\n", book.subject);
printf( "书 ID : %d\n", book.book_id);

return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
书标题 : C 教程
书作者 : Runoob
书类目 : 编程语言
书 ID : 12345
```

typedef vs #define

#define 是 C 指令，用于为各种数据类型定义别名，与 **typedef** 类似，但是它们有以下几点不同：

typedef 仅限于为类型定义符号名称，**#define** 不仅可以为类型定义别名，也能为数值定义别名，比如您可以定义 1 为 **ONE**。

typedef 是由编译器执行解释的，**#define** 语句是由预编译器进行处理的。

下面是 **#define** 的最简单的用法：

实例

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
int main( )
{
    printf( "TRUE 的值: %d\n", TRUE);
    printf( "FALSE 的值: %d\n", FALSE);

    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
TRUE 的值: 1
FALSE 的值: 0
```


C 输入 & 输出

当我们提到**输入**时，这意味着要向程序填充一些数据。输入可以是以文件的形式或从命令行中进行。C 语言提供了一系列内置的函数来读取给定的输入，并根据需要填充到程序中。

当我们提到**输出**时，这意味着要在屏幕上、打印机上或任意文件中显示一些数据。C 语言提供了一系列内置的函数来输出数据到计算机屏幕上和保存数据到文本文件或二进制文件中。

标准文件

C 语言把所有的设备都当作文件。所以设备（比如显示器）被处理的方式与文件相同。以下三个文件会在程序执行时自动打开，以便访问键盘和屏幕。

标准文件	文件指针	设备
标准输入	stdin	键盘
标准输出	stdout	屏幕
标准错误	stderr	您的屏幕

文件指针是访问文件的方式，本节将讲解如何从屏幕读取值以及如何把结果输出到屏幕上。

C 语言中的 I/O (输入/输出) 通常使用 `printf()` 和 `scanf()` 两个函数。

`scanf()` 函数用于从标准输入（键盘）读取并格式化，`printf()` 函数发送格式化输出到标准输出（屏幕）。

实例

```
#include <stdio.h>      // 执行 printf() 函数需要该库
int main()
{
    printf("菜鸟教程"); //显示引号中的内容
    return 0;
}
```

编译以上程序，输出结果为：

菜鸟教程

实例解析：

- 所有的 C 语言程序都需要包含 `main()` 函数。代码从 `main()` 函数开始执行。
- `printf()` 用于格式化输出到屏幕。`printf()` 函数在 "`stdio.h`" 头文件中声明。
- `stdio.h` 是一个头文件 (标准输入输出头文件) and `#include` 是一个预处理命令，用来引入头文件。当编译器遇到 `printf()` 函数时，如果没有找到 `stdio.h` 头文件，会发生编译错误。
- `return 0;` 语句用于表示退出程序。

%d 格式化输出整数

```
#include <stdio.h>
int main()
{
    int testInteger = 5;
    printf("Number = %d", testInteger);
    return 0;
}
```

编译以上程序，输出结果为：

Number = 5

在 `printf()` 函数的引号中使用 "`%d`" (整型) 来匹配整型变量 `testInteger` 并输出到屏幕。

%f 格式化输出浮点型数据

```
#include <stdio.h>
int main()
{
    float f;
    printf("Enter a number: ");
    // %f 匹配浮点型数据
    scanf("%f",&f);
    printf("Value = %f", f);
    return 0;
}
```

getchar() & putchar() 函数

int getchar(void) 函数从屏幕读取下一个可用的字符，并把它返回为一个整数。这个函数在同一个时间内只会读取一个单一的字符。您可以在循环内使用这个方法，以便从屏幕上读取多个字符。

int putchar(int c) 函数把字符输出到屏幕上，并返回相同的字符。这个函数在同一个时间内只会输出一个单一的字符。您可以在循环内使用这个方法，以便在屏幕上输出多个字符。

请看下面的实例：

实例

```
#include <stdio.h>
int main( )
{
    int c;

    printf( "Enter a value :");
    c = getchar( );

    printf( "\nYou entered: ");
    putchar( c );
    printf( "\n");
    return 0;
}
```

当上面的代码被编译和执行时，它会等待您输入一些文本，当您输入一个文本并按下回车键时，程序会继续并只会读取一个单一的字符，显示如下：

```
$/./a.out
Enter a value :runoob

You entered: r
```

gets() & puts() 函数

char *gets(char *s) 函数从 **stdin** 读取一行到 **s** 所指向的缓冲区，直到一个终止符或 EOF。

int puts(const char *s) 函数把字符串 **s** 和一个尾随的换行符写入到 **stdout**。

实例

```
#include <stdio.h>
int main( )
{
    char str[100];

    printf( "Enter a value :");
    gets( str );

    printf( "\nYou entered: ");
    puts( str );
    return 0;
}
```

当上面的代码被编译和执行时，它会等待您输入一些文本，当您输入一个文本并按下回车键时，程序会继续并读取一整行直到该行结束，显示如下：

```
$/./a.out
Enter a value :runoob
```

You entered: runoob

scanf() 和 printf() 函数

int scanf(const char *format, ...) 函数从标准输入流 **stdin** 读取输入，并根据提供的 **format** 来浏览输入。

int printf(const char *format, ...) 函数把输出写入到标准输出流 **stdout**，并根据提供的格式产生输出。

format 可以是一个简单的常量字符串，但是您可以分别指定 %s、%d、%c、%f 等来输出或读取字符串、整数、字符或浮点数。还有许多其他可用的格式选项，可以根据需要使用。如需了解完整的细节，可以查看这些函数的参考手册。现在让我们通过下面这个简单的实例来加深理解：

实例

```
#include <stdio.h>
int main( ) {
char str[100];
    int i;

    printf( "Enter a value :");
    scanf("%s %d", str, &i);

    printf( "\nYou entered: %s %d ", str, i);
    printf("\n");
    return 0;
}
```

当上面的代码被编译和执行时，它会等待您输入一些文本，当您输入一个文本并按下回车键时，程序会继续并读取输入，显示如下：

```
./a.out
Enter a value :runoob 123

You entered: runoob 123
```

在这里，应当指出的是，scanf() 期待输入的格式与您给出的 %s 和 %d 相同，这意味着您必须提供有效的输入，比如 "string integer"，如果您提供的是 "string string" 或 "integer integer"，它会被认为是错误的输入。另外，在读取字符串时，只要遇到一个空格，scanf() 就会停止读取，所以 "this is test" 对 scanf() 来说是三个字符串。

C 文件读写

上一章我们讲解了 C 语言处理的标准输入和输出设备。本章我们将介绍 C 程序员如何创建、打开、关闭文本文件或二进制文件。一个文件，无论它是文本文件还是二进制文件，都是代表了一系列的字节。C 语言不仅提供了访问顶层的函数，也提供了底层（OS）调用来处理存储设备上的文件。本章将讲解文件管理的重要调用。

打开文件

您可以使用 **fopen()** 函数来创建一个新的文件或者打开一个已有的文件，这个调用会初始化类型 **FILE** 的一个对象，类型 **FILE** 包含了所有用来控制流的必要的信息。下面是这个函数调用的原型：

```
FILE *fopen( const char * filename, const char * mode );
```

在这里，**filename** 是字符串，用来命名文件，访问模式 **mode** 的值可以是下列值中的一个：

模式	描述
r	打开一个已有的文本文件，允许读取文件。
w	打开一个文本文件，允许写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会从文件的开头写入内容。
a	打开一个文本文件，以追加模式写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会在已有的文件内容中追加内容。
r+	打开一个文本文件，允许读写文件。
w+	打开一个文本文件，允许读写文件。如果文件已存在，则文件会被截断为零长度，如果文件不存在，则会创建一个新文件。
a+	打开一个文本文件，允许读写文件。如果文件不存在，则会创建一个新文件。读取会从文件的开头开始，写入则只能是追加模式。

如果处理的是二进制文件，则需使用下面的访问模式来取代上面的访问模式：

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

关闭文件

；
为了关闭文件，请使用 **fclose()** 函数。函数的原型如下：

```
int fclose( FILE *fp );
```

如果成功关闭文件，**fclose()** 函数返回零，如果关闭文件时发生错误，函数返回 **EOF**。这个函数实际上，会清空缓冲区中的数据，关闭文件，并释放用于该文件的所有内存。EOF 是一个定义在头文件 **stdio.h** 中的常量。
C 标准库提供了各种函数来按字符或者以固定长度字符串的形式读写文件。

写入文件

下面是把字符写入到流中的最简单的函数：

```
int fputc( int c, FILE *fp );
```

函数 **fputc()** 把参数 **c** 的字符值写入到 **fp** 所指向的输出流中。如果写入成功，它会返回写入的字符，如果发生错误，则会返回 **EOF**。您可以使用下面的函数来把一个以 **null** 结尾的字符串写入到流中：

```
int fputs( const char *s, FILE *fp );
```

函数 **fputs()** 把字符串 **s** 写入到 **fp** 所指向的输出流中。如果写入成功，它会返回一个非负值，如果发生错误，则会返回 **EOF**。您也可以使用 **int fprintf(FILE *fp,const char *format, ...)** 函数来写把一个字符串写入到文件中。尝试下面的实例：

注意：请确保您有可用的 **/tmp** 目录，如果不存在该目录，则需要您的计算机上先创建该目录。

实例

```
#include <stdio.h>
int main()
{
    FILE *fp = NULL;

    fp = fopen("/tmp/test.txt", "w+");
    fprintf(fp, "This is testing for fprintf...\n");
    fputs("This is testing for fputs...\n", fp);
    fclose(fp);
}
```

当上面的代码被编译和执行时，它会在 `/tmp` 目录中创建一个新的文件 **test.txt**，并使用两个不同的函数写入两行。接下来让我们来读取这个文件。

读取文件

下面是从文件读取单个字符的最简单的函数：

```
int fgetc( FILE * fp );
```

fgetc() 函数从 **fp** 所指向的输入文件中读取一个字符。返回值是读取的字符，如果发生错误则返回 **EOF**。下面的函数允许您从流中读取一个字符串：

```
char *fgets( char *buf, int n, FILE *fp );
```

函数 **fgets()** 从 **fp** 所指向的输入流中读取 `n - 1` 个字符。它会把读取的字符串复制到缓冲区 **buf**，并在最后追加一个 **null** 字符来终止字符串。如果这个函数在读取最后一个字符之前就遇到一个换行符 `\n` 或文件的末尾 **EOF**，则只会返回读取到的字符，包括换行符。您也可以使用 **int fscanf(FILE *fp, const char *format, ...)** 函数来从文件中读取字符串，但是在遇到第一个空格字符时，它会停止读取。

实例

```
#include <stdio.h>
int main()
{
    FILE *fp = NULL;
    char buff[255];

    fp = fopen("/tmp/test.txt", "r");
    fscanf(fp, "%s", buff);
    printf("1: %s\n", buff );

    fgets(buff, 255, (FILE*)fp);
    printf("2: %s\n", buff );

    fgets(buff, 255, (FILE*)fp);
    printf("3: %s\n", buff );
    fclose(fp);
}
```

当上面的代码被编译和执行时，它会读取上一部分创建的文件，产生下列结果：

```
1: This
2: is testing for fprintf...

3: This is testing for fputs...
```

首先，**fscanf()** 方法只读取了 **This**，因为它在后边遇到了一个空格。其次，调用 **fgets()** 读取剩余的部分，直到行尾。最后，调用 **fgets()** 完整地读取第二行。

二进制 I/O 函数

下面两个函数用于二进制输入和输出：

```
size_t fread(void *ptr, size_t size_of_elements,
```

```
size_t number_of_elements, FILE *a_file);  
  
size_t fwrite(const void *ptr, size_t size_of_elements,  
size_t number_of_elements, FILE *a_file);
```

这两个函数都是用于存储块的读写 - 通常是数组或结构体。

C 预处理器

C 预处理器不是编译器的组成部分，但是它是编译过程中一个单独的步骤。简言之，C 预处理器只不过是一个文本替换工具而已，它们会指示编译器在实际编译之前完成所需的预处理。我们将把 C 预处理器（C Preprocessor）简写为 CPP。

所有的预处理器命令都是以井号（#）开头。它必须是第一个非空字符，为了增强可读性，预处理器指令应从第一列开始。下面列出了所有重要的预处理器指令：

指令	描述
#define	定义宏
#include	包含一个源代码文件
#undef	取消已定义的宏
#ifdef	如果宏已经定义，则返回真
#ifndef	如果宏没有定义，则返回真
#if	如果给定条件为真，则编译下面代码
#else	#if 的替代方案
#elif	如果前面的 #if 给定条件不为真，当前条件为真，则编译下面代码
#endif	结束一个 #if.....#else 条件编译块
#error	当遇到标准错误时，输出错误消息
#pragma	使用标准化方法，向编译器发布特殊的命令到编译器中

预处理器实例

分析下面的实例来理解不同的指令。

```
#define MAX_ARRAY_LENGTH 20
```

这个指令告诉 CPP 把所有的 MAX_ARRAY_LENGTH 替换为 20。使用 *#define* 定义常量来增强可读性。

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

这些指令告诉 CPP 从系统库中获取 stdio.h，并添加文本到当前的源文件中。下一行告诉 CPP 从本地目录中获取 **myheader.h**，并添加内容到当前的源文件中。

```
#undef FILE_SIZE
#define FILE_SIZE 42
```

这个指令告诉 CPP 取消已定义的 FILE_SIZE，并定义它为 42。

```
#ifndef MESSAGE
    #define MESSAGE "You wish!"
#endif
```

这个指令告诉 CPP 只有当 MESSAGE 未定义时，才定义 MESSAGE。

```
#ifdef DEBUG
    /* Your debugging statements here */
```

```
#endif
```

这个指令告诉 **CPP** 如果定义了 **DEBUG**，则执行处理语句。在编译时，如果您向 **gcc** 编译器传递了 *-DDEBUG* 开关量，这个指令就非常有用。它定义了 **DEBUG**，您可以在编译期间随时开启或关闭调试。

预定义宏

ANSI C 定义了许多宏。在编程中您可以使用这些宏，但是不能直接修改这些预定义的宏。

宏	描述
<code>__DATE__</code>	当前日期，一个以 "MMM DD YYYY" 格式表示的字符常量。
<code>__TIME__</code>	当前时间，一个以 "HH:MM:SS" 格式表示的字符常量。
<code>__FILE__</code>	这会包含当前文件名，一个字符串常量。
<code>__LINE__</code>	这会包含当前行号，一个十进制常量。
<code>__STDC__</code>	当编译器以 ANSI 标准编译时，则定义为 1。

让我们来尝试下面的实例：

```
#include <stdio.h>

main()
{
    printf("File :%s\n", __FILE__ );
    printf("Date :%s\n", __DATE__ );
    printf("Time :%s\n", __TIME__ );
    printf("Line :%d\n", __LINE__ );
    printf("ANSI :%d\n", __STDC__ );
}
```

当上面的代码（在文件 **test.c** 中）被编译和执行时，它会产生下列结果：

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

预处理器运算符

C 预处理器提供了下列的运算符来帮助您创建宏：

宏延续运算符（\）

一个宏通常写在一个单行上。但是如果宏太长，一个单行容纳不下，则使用宏延续运算符（\）。例如：

```
#define message_for(a, b) \
    printf("#a " and " #b ": We love you!\n")
```

字符串常量量化运算符（#）

在宏定义中，当需要把一个宏的参数转换为字符串常量时，则使用字符串常量量化运算符（#）。在宏中使用的该运算符有一个特定的参数或参数列表。

例如：

```
#include <stdio.h>

#define message_for(a, b) \
    printf("#a " and " #b ": We love you!\n")

int main(void)
```



```
{
    message_for(Carole, Debra);
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Carole and Debra: We love you!
```

标记粘贴运算符（##）

宏定义内的标记粘贴运算符（##）会合并两个参数。它允许在宏定义中两个独立的标记被合并为一个标记。例如：

```
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void)
{
    int token34 = 40;

    tokenpaster(34);
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
token34 = 40
```

这是怎么发生的，因为这个实例会从编译器产生下列的实际输出：

```
printf ("token34 = %d", token34);
```

这个实例演示了 `token##n` 会连接到 `token34` 中，在这里，我们使用了字符串常量化运算符（#）和标记粘贴运算符（##）。

defined() 运算符

预处理器 **defined** 运算符是在常量表达式中的，用来确定一个标识符是否已经使用 `#define` 定义过。如果指定的标识符已定义，则值为真（非零）。如果指定的标识符未定义，则值为假（零）。下面的实例演示了 `defined()` 运算符的用法：

```
#include <stdio.h>

#if !defined (MESSAGE)
    #define MESSAGE "You wish!"
#endif

int main(void)
{
    printf("Here is the message: %s\n", MESSAGE);
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Here is the message: You wish!
```

参数化的宏

CPP 一个强大的功能是可以使用参数化的宏来模拟函数。例如，下面的代码是计算一个数的平方：

```
int square(int x) {
    return x * x;
}
```

```
}
```

我们可以使用宏重写上面的代码，如下：

```
#define square(x) ((x) * (x))
```

在使用带有参数的宏之前，必须使用 **#define** 指令定义。参数列表是括在圆括号内，且必须紧跟在宏名称的后边。宏名称和左圆括号之间不允许有空格。例如：

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void)
{
    printf("Max between 20 and 10 is %d\n", MAX(10, 20));
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Max between 20 and 10 is 20
```

C 头文件

头文件是扩展名为 **.h** 的文件，包含了 C 函数声明和宏定义，被多个源文件中引用共享。有两种类型的头文件：程序员编写的头文件和编译器自带的头文件。

在程序中要使用头文件，需要使用 C 预处理指令 **#include** 来引用它。前面我们已经看过 **stdio.h** 头文件，它是编译器自带的头文件。

引用头文件相当于复制头文件的内容，但是我们不会直接在源文件中复制头文件的内容，因为这么做很容易出错，特别在程序是由多个源文件组成的时候。

A simple practice in C 或 C++ 程序中，建议把所有的常量、宏、系统全局变量和函数原型写在头文件中，在需要的时候随时引用这些头文件。

引用头文件的语法

使用预处理指令 **#include** 可以引用用户和系统头文件。它的形式有以下两种：

```
#include <file>
```

这种形式用于引用系统头文件。它在系统目录的标准列表中搜索名为 **file** 的文件。在编译源代码时，您可以通过 **-I** 选项把目录前置在该列表前。

```
#include "file"
```

这种形式用于引用用户头文件。它在包含当前文件的目录中搜索名为 **file** 的文件。在编译源代码时，您可以通过 **-I** 选项把目录前置在该列表前。

引用头文件的操作

#include 指令会指示 C 预处理器浏览指定的文件作为输入。预处理器的输出包含了已经生成的输出，被引用文件生成的输出以及 **#include** 指令之后的文本输出。例如，如果您有一个头文件 **header.h**，如下：

```
char *test (void);
```

和一个使用了头文件的主程序 *program.c*，如下：

```
int x;
#include "header.h"

int main (void)
{
    puts (test ());
}
```

编译器会看到如下的代码信息：

```
int x;
char *test (void);

int main (void)
{
    puts (test ());
}
```

只引用一次头文件

如果一个头文件被引用两次，编译器会处理两次头文件的内容，这将产生错误。为了防止这种情况，标准的做法是把文件的整个内容放在条件编译语句中，如下：

```
#ifndef HEADER_FILE
#define HEADER_FILE
```

```
the entire header file file
```

```
#endif
```

这种结构就是通常所说的包装器 **#ifndef**。当再次引用头文件时，条件为假，因为 `HEADER_FILE` 已定义。此时，预处理器会跳过文件的整个内容，编译器会忽略它。

有条件引用

有时需要从多个不同的头文件中选择一个引用到程序中。例如，需要指定在不同的操作系统上使用的配置参数。您可以通过一系列条件来实现这点，如下：

```
#if SYSTEM_1
    # include "system_1.h"
#elif SYSTEM_2
    # include "system_2.h"
#elif SYSTEM_3
    ...
#endif
```

但是如果头文件比较多的时候，这么做是很不妥当的，预处理器使用宏来定义头文件的名称。这就是所谓的**有条件引用**。它不是用头文件的名称作为 **#include** 的直接参数，您只需要使用宏名称代替即可：

```
#define SYSTEM_H "system_1.h"
...
#include SYSTEM_H
```

`SYSTEM_H` 会扩展，预处理器会查找 `system_1.h`，就像 **#include** 最初编写的那样。`SYSTEM_H` 可通过 `-D` 选项被您的 `Makefile` 定义。

C 强制类型转换

强制类型转换是把变量从一种类型转换为另一种数据类型。例如，如果您想存储一个 `long` 类型的值到一个简单的整型中，您需要把 `long` 类型强制转换为 `int` 类型。您可以使用**强制类型转换运算符**来把值显式地从一种类型转换为另一种类型，如下所示：

```
(type_name) expression
```

请看下面的实例，使用强制类型转换运算符把一个整数变量除以另一个整数变量，得到一个浮点数：

```
#include <stdio.h>

main()
{
    int sum = 17, count = 5;
    double mean;

    mean = (double) sum / count;
    printf("Value of mean : %f\n", mean );

}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of mean : 3.400000
```

这里要注意的是强制类型转换运算符的优先级大于除法，因此 **sum** 的值首先被转换为 **double** 型，然后除以 **count**，得到一个类型为 **double** 的值。类型转换可以是隐式的，由编译器自动执行，也可以是显式的，通过使用**强制类型转换运算符**来指定。在编程时，有需要类型转换的时候都用上强制类型转换运算符，是一种良好的编程习惯。

整数提升

整数提升是指把小于 **int** 或 **unsigned int** 的整数类型转换为 **int** 或 **unsigned int** 的过程。请看下面的实例，在 `int` 中添加一个字符：

```
#include <stdio.h>

main()
{
    int i = 17;
    char c = 'c'; /* ascii 值是 99 */
    int sum;

    sum = i + c;
    printf("Value of sum : %d\n", sum );

}
```

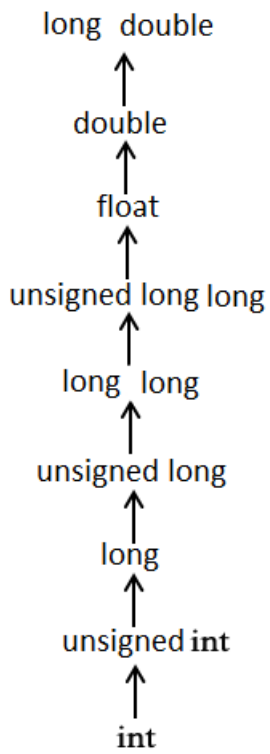
当上面的代码被编译和执行时，它会产生下列结果：

```
Value of sum : 116
```

在这里，`sum` 的值为 116，因为编译器进行了整数提升，在执行实际加法运算时，把 `'c'` 的值转换为对应的 `ascii` 值。

常用的算术转换

常用的算术转换是隐式地把值强制转换为相同的类型。编译器首先执行**整数提升**，如果操作数类型不同，则它们会被转换为下列层次中出现的最高层次的类型：



常用的算术转换不适用于赋值运算符、逻辑运算符 `&&` 和 `||`。让我们看看下面的实例来理解这个概念：

```
#include <stdio.h>

main()
{
    int i = 17;
    char c = 'c'; /* ascii 值是 99 */
    float sum;

    sum = i + c;
    printf("Value of sum : %f\n", sum );
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of sum : 116.000000
```

在这里，`c` 首先被转换为整数，但是由于最后的值是 `double` 型的，所以会应用常用的算术转换，编译器会把 `i` 和 `c` 转换为浮点型，并把它们相加得到一个浮点数。

C 错误处理

C 语言不提供对错误处理的直接支持，但是作为一种系统编程语言，它以返回值的形式允许您访问底层数据。在发生错误时，大多数的 C 或 UNIX 函数调用返回 1 或 NULL，同时会设置一个错误代码 **errno**，该错误代码是全局变量，表示在函数调用期间发生了错误。您可以在 `<error.h>` 头文件中找到各种各样的错误代码。

所以，C 程序员可以通过检查返回值，然后根据返回值决定采取哪种适当的动作。开发人员应该在程序初始化时，把 `errno` 设置为 0，这是一种良好的编程习惯。0 值表示程序中没有错误。

errno、perror() 和 strerror()

C 语言提供了 **perror()** 和 **strerror()** 函数来显示与 **errno** 相关的文本消息。

perror() 函数显示您传给它的字符串，后跟一个冒号、一个空格和当前 `errno` 值的文本表示形式。

strerror() 函数，返回一个指针，指针指向当前 `errno` 值的文本表示形式。

让我们来模拟一种错误情况，尝试打开一个不存在的文件。您可以使用多种方式来输出错误消息，在这里我们使用函数来演示用法。另外有一点需要注意，您应该使用 **stderr** 文件流来输出所有的错误。

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno ;

int main ()
{
    FILE * pf;
    int errnum;
    pf = fopen ("unexist.txt", "rb");
    if (pf == NULL)
    {
        errnum = errno;
        fprintf(stderr, "错误号: %d\n", errno);
        perror("通过 perror 输出错误");
        fprintf(stderr, "打开文件错误: %s\n", strerror( errnum ));
    }
    else
    {
        fclose (pf);
    }
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
错误号: 2
通过 perror 输出错误: No such file or directory
打开文件错误: No such file or directory
```

被零除的错误

在进行除法运算时，如果不检查除数是否为零，则会导致一个运行时错误。

为了避免这种情况发生，下面的代码在进行除法运算前会先检查除数是否为零：

```
#include <stdio.h>
#include <stdlib.h>

main()
{
    int dividend = 20;
```

```
int divisor = 0;
int quotient;

if( divisor == 0){
    fprintf(stderr, "除数为 0 退出运行...\n");
    exit(-1);
}
quotient = dividend / divisor;
fprintf(stderr, "quotient 变量的值为 : %d\n", quotient );

exit(0);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
除数为 0 退出运行...
```

程序退出状态

通常情况下，程序成功执行完一个操作正常退出的时候会带有值 `EXIT_SUCCESS`。在这里，`EXIT_SUCCESS` 是宏，它被定义为 `0`。

如果程序中存在一种错误情况，当您退出程序时，会带有状态值 `EXIT_FAILURE`，被定义为 `-1`。所以，上面的程序可以写成：

```
#include <stdio.h>
#include <stdlib.h>

main()
{
    int dividend = 20;
    int divisor = 5;
    int quotient;

    if( divisor == 0){
        fprintf(stderr, "除数为 0 退出运行...\n");
        exit(EXIT_FAILURE);
    }
    quotient = dividend / divisor;
    fprintf(stderr, "quotient 变量的值为: %d\n", quotient );

    exit(EXIT_SUCCESS);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
quotient 变量的值为 : 4
```


C 递归

递归指的是在函数的定义中使用函数自身的方法。

举个例子：

从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？"从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？'从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？.....'"

语法格式如下：

```
void recursion()
{
    recursion(); /* 函数调用自身 */
}
int main()
{
    recursion();
}
```

C 语言支持递归，即一个函数可以调用其自身。但在使用递归时，程序员需要注意定义一个从函数退出的条件，否则会进入死循环。

递归函数在解决许多数学问题上起了至关重要的作用，比如计算一个数的阶乘、生成斐波那契数列，等等。

数的阶乘

下面的实例使用递归函数计算一个给定的数的阶乘：

实例

```
#include <stdio.h>
double factorial(unsigned int i)
{
    if(i <= 1)
    {
        return 1;
    }
    return i * factorial(i - 1);
}
int main()
{
    int i = 15;
    printf("%d 的阶乘为 %f\n", i, factorial(i));
    return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
15 的阶乘为 1307674368000.000000
```

斐波那契数列

下面的实例使用递归函数生成一个给定的数的斐波那契数列：

实例

```
#include <stdio.h>
int fibonaci(int i)
{
    if(i == 0)
    {
        return 0;
    }
    if(i == 1)
    {
        return 1;
    }
    return fibonaci(i-1) + fibonaci(i-2);
}
```

```
}  
int main()  
{  
    int i;  
    for (i = 0; i < 10; i++)  
    {  
        printf("%d\t", fibonaci(i));  
    }  
    return 0;  
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
0  
1  
1  
2  
3  
5  
8  
13  
21  
34
```

C 可变参数

有时，您可能会碰到这样的情况，您希望函数带有可变数量的参数，而不是预定义数量的参数。C 语言为这种情况提供了一个解决方案，它允许您定义一个函数，能根据具体的需求接受可变数量的参数。下面的实例演示了这种函数的定义。

```
int func(int, ... )
{
    .
    .
    .
}

int main()
{
    func(2, 2, 3);
    func(3, 2, 3, 4);
}
```

请注意，函数 **func()** 最后一个参数写成省略号，即三个点号 (...)，省略号之前的那个参数是 **int**，代表了要传递的可变参数的总数。为了使用这个功能，您需要使用 **stdarg.h** 头文件，该文件提供了实现可变参数功能的函数和宏。具体步骤如下：

定义一个函数，最后一个参数为省略号，省略号前面可以设置自定义参数。

在函数定义中创建一个 **va_list** 类型变量，该类型是在 **stdarg.h** 头文件中定义的。

使用 **int** 参数和 **va_start** 宏来初始化 **va_list** 变量为一个参数列表。宏 **va_start** 是在 **stdarg.h** 头文件中定义的。

使用 **va_arg** 宏和 **va_list** 变量来访问参数列表中的每个项。

使用宏 **va_end** 来清理赋予 **va_list** 变量的内存。

现在让我们按照上面的步骤，来编写一个带有可变数量参数的函数，并返回它们的平均值：

```
#include <stdio.h>
#include <stdarg.h>

double average(int num,...)
{
    va_list valist;
    double sum = 0.0;
    int i;

    /* 为 num 个参数初始化 valist */
    va_start(valist, num);

    /* 访问所有赋给 valist 的参数 */
    for (i = 0; i < num; i++)
    {
        sum += va_arg(valist, int);
    }
    /* 清理为 valist 保留的内存 */
    va_end(valist);

    return sum/num;
}

int main()
{
    printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));
    printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));
}
```

当上面的代码被编译和执行时，它会产生下列结果。应该指出的是，函数 **average()** 被调用两次，每次第一个参数都是表示被传的可变参数的总数。

省略号被用来传递可变数量的参数。

```
Average of 2, 3, 4, 5 = 3.500000
```

```
Average of 5, 10, 15 = 10.000000
```

C 内存管理

本章将讲解 C 中的动态内存管理。C 语言为内存的分配和管理提供了几个函数。这些函数可以在 `<stdlib.h>` 头文件中找到。

序号	函数和描述
1	void *calloc(int num, int size); 在内存中动态地分配 num 个长度为 size 的连续空间，并将每一个字节都初始化为 0。所以它的结果是分配了 num*size 个字节长度的内存空间，并且每个字节的值都是0。
2	void free(void *address); 该函数释放 address 所指向的内存块,释放的是动态分配的内存空间。
3	void *malloc(int num); 在堆区分配一块指定大小的内存空间，用来存放数据。这块内存空间在函数执行完成后不会被初始化，它们的值是未知的。
4	void *realloc(void *address, int newsize); 该函数重新分配内存，把内存扩展到 newsize。

动态分配内存

编程时，如果您预先知道数组的大小，那么定义数组时就比较容易。例如，一个存储人名的数组，它最多容纳 100 个字符，所以您可以定义数组，如下所示：

```
char name[100];
```

但是，如果您预先不知道需要存储的文本长度，例如您向存储有关一个主题的详细描述。在这里，我们需要定义一个指针，该指针指向未定义所学内存大小的字符，后续再根据需求来分配内存，如下所示：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char name[100];
    char *description;

    strcpy(name, "Zara Ali");

    /* 动态分配内存 */
    description = malloc( 200 * sizeof(char) );
    if( description == NULL )
    {
        fprintf(stderr, "Error - unable to allocate required memory\n");
    }
    else
    {
        strcpy( description, "Zara ali a DPS student in class 10th");
    }
    printf("Name = %s\n", name );
    printf("Description: %s\n", description );
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Name = Zara Ali
Description: Zara ali a DPS student in class 10th
```

上面的程序也可以使用 **calloc()** 来编写，只需要把 malloc 替换为 calloc 即可，如下所示：

```
calloc(200, sizeof(char));
```

当动态分配内存时，您有完全控制权，可以传递任何大小的值。而那些预先定义了大小的数组，一旦定义则无法改变大小。

重新调整内存的大小和释放内存

当程序退出时，操作系统会自动释放所有分配给程序的内存，但是，建议您在不需要内存时，都应该调用函数 **free()** 来释放内存。

或者，您可以通过调用函数 **realloc()** 来增加或减少已分配的内存块的大小。让我们使用 **realloc()** 和 **free()** 函数，再次查看上面的实例：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char name[100];
    char *description;

    strcpy(name, "Zara Ali");

    /* 动态分配内存 */
    description = malloc( 30 * sizeof(char) );
    if( description == NULL )
    {
        fprintf(stderr, "Error - unable to allocate required memory\n");
    }
    else
    {
        strcpy( description, "Zara ali a DPS student.");
    }
    /* 假设您想要存储更大的描述信息 */
    description = realloc( description, 100 * sizeof(char) );
    if( description == NULL )
    {
        fprintf(stderr, "Error - unable to allocate required memory\n");
    }
    else
    {
        strcat( description, "She is in class 10th");
    }

    printf("Name = %s\n", name );
    printf("Description: %s\n", description );

    /* 使用 free() 函数释放内存 */
    free(description);
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Name = Zara Ali
Description: Zara ali a DPS student.She is in class 10th
```

您可以尝试一下不重新分配额外的内存，**strcat()** 函数会生成一个错误，因为存储 **description** 时可用的内存不足。

C 命令行参数

执行程序时，可以从命令行传值给 C 程序。这些值被称为**命令行参数**，它们对程序很重要，特别是当您想从外部控制程序，而不是在代码内对这些值进行硬编码时，就显得尤为重要了。

命令行参数是使用 `main()` 函数参数来处理的，其中，**argc** 是指传入参数的个数，**argv[]** 是一个指针数组，指向传递给程序的每个参数。下面是一个简单的实例，检查命令行是否有提供参数，并根据参数执行相应的动作：

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
    if( argc == 2 )
    {
        printf("The argument supplied is %s\n", argv[1]);
    }
    else if( argc > 2 )
    {
        printf("Too many arguments supplied.\n");
    }
    else
    {
        printf("One argument expected.\n");
    }
}
```

使用一个参数，编译并执行上面的代码，它会产生下列结果：

```
./a.out testing
The argument supplied is testing
```

使用两个参数，编译并执行上面的代码，它会产生下列结果：

```
./a.out testing1 testing2
Too many arguments supplied.
```

不传任何参数，编译并执行上面的代码，它会产生下列结果：

```
./a.out
One argument expected
```

应当指出的是，**argv[0]** 存储程序的名称，**argv[1]** 是一个指向第一个命令行参数的指针，***argv[n]** 是最后一个参数。如果没有提供任何参数，**argc** 将为 1，否则，如果传递了一个参数，**argc** 将被设置为 2。

多个命令行参数之间用空格分隔，但是如果参数本身带有空格，那么传递参数的时候应把参数放置在双引号 "" 或单引号 ' ' 内部。让我们重新编写上面的实例，有一个空间，那么你可以通过这样的观点，把它们放在双引号或单引号 "" 或 ' ' 内部。让我们重新编写上面的实例，向程序传递一个放置在双引号内部的命令行参数：

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
    printf("Program name %s\n", argv[0]);

    if( argc == 2 )
    {
        printf("The argument supplied is %s\n", argv[1]);
    }
    else if( argc > 2 )
    {
```

```
        printf("Too many arguments supplied.\n");
    }
    else
    {
        printf("One argument expected.\n");
    }
}
```

使用一个用空格分隔的简单参数，参数括在双引号中，编译并执行上面的代码，它会产生下列结果：

```
$/a.out "testing1 testing2"
```

```
Program name ./a.out
```

```
The argument supplied is testing1 testing2
```


C 语言实例

[C 语言实例 - 输出 "Hello, World!"](#)

[C 语言实例 - 输出整数](#)

[C 语言实例 - 两个数字相加](#)

[C 语言实例 - 两个浮点数相乘](#)

[C 语言实例 - 字符转 ASCII 码](#)

[C 语言实例 - 两数相除](#)

[C 语言实例 - 计算 int, float, double 和 char 字节大小](#)

[C 语言实例 - 交换两个数的值](#)

[C 语言实例 - 判断奇数/偶数](#)

[C 语言实例 - 判断元音/辅音](#)

[C 语言实例 - 判断三个数中的最大数](#)

[C 语言实例 - 一元二次方程](#)

[C 语言实例 - 判断闰年](#)

[C 语言实例 - 判断正数/负数/零](#)

[C 语言实例 - 判断字母](#)

[C 语言实例 - 计算自然数的和](#)

[C 语言实例 - 输出九九乘法口诀表](#)

[C 语言实例 - 斐波那契数列](#)

[C 语言实例 - 求两数的最大公约数](#)

[C 语言实例 - 求两数最小公倍数](#)

[C 语言实例 - 阶乘](#)

[C 语言实例 - 循环输出26个字母](#)

[C 语言实例 - 判断数字为几位数](#)

[C 语言实例 - 计算一个数的 n 次方](#)

[C 语言实例 - 判断回文数](#)

[C 语言实例 - 判断素数](#)

[C 语言实例 - 判断Armstrong数\(阿姆斯壮数\)](#)

[C 语言实例 - 求一个整数的所有因数](#)

[C 语言实例 - 创建各类三角形图案](#)

[C 语言实例 - 实现简单的计算器](#)

[C 语言实例 - 计算一个数是否可为两个素数之和](#)

[C 语言实例 - 二进制与十进制相互转换](#)

[C 语言实例 - 八进制与十进制相互转换](#)

[C 语言实例 - 八进制与二进制相互转换](#)

[C 语言实例 - 字符串翻转](#)

[C 语言实例 - 计算平均值](#)

[C 语言实例 - 查找数组中最大的元素值](#)

[C 语言实例 - 计算标准偏差](#)

[C 语言实例 - 两个矩阵相加](#)

[C 语言实例 - 矩阵转换](#)

[C 语言实例 - 使用指针访问数组元素](#)

[C 语言实例 - 使用引用循环替换数值](#)

[C 语言实例 - 判断最大值](#)

[C 语言实例 - 查找字符在字符串中的位置](#)

[C 语言实例 - 字符串中各种字符计算](#)

[C 语言实例 - 删除字符串中的特殊字符](#)

[C 语言实例 - 连接字符串](#)

[C 语言实例 - 计算字符串长度](#)

[C 语言实例 - 查找字符在字符串中的位置](#)

[C 语言实例 - 字符串中各种字符计算](#)

[C 语言实例 - 字符串复制](#)

[C 语言实例 - 字符串排序](#)

[C 语言实例 - 使用结构体 \(struct\)](#)

[C 语言实例 - 复数相加](#)

[C 语言实例 - 计算两个时间段的差值](#)

[C 语言实例 - 将字符串写入文件](#)

[C 语言实例 - 从文件中读取一行](#)

[C 语言实例 - 输出当前文件执行代码](#)

C 语言经典100例

- C 语言练习实例1
- C 语言练习实例2
- C 语言练习实例3
- C 语言练习实例4
- C 语言练习实例5
- C 语言练习实例6
- C 语言练习实例7
- C 语言练习实例8
- C 语言练习实例9
- C 语言练习实例10
- C 语言练习实例11
- C 语言练习实例12
- C 语言练习实例13
- C 语言练习实例14
- C 语言练习实例15
- C 语言练习实例16
- C 语言练习实例17
- C 语言练习实例18
- C 语言练习实例19
- C 语言练习实例20
- C 语言练习实例21
- C 语言练习实例22
- C 语言练习实例23
- C 语言练习实例24
- C 语言练习实例25
- C 语言练习实例26
- C 语言练习实例27
- C 语言练习实例28
- C 语言练习实例29
- C 语言练习实例30
- C 语言练习实例31
- C 语言练习实例32
- C 语言练习实例33
- C 语言练习实例34
- C 语言练习实例35
- C 语言练习实例36
- C 语言练习实例37
- C 语言练习实例38
- C 语言练习实例39

C 语言练习实例40

C 语言练习实例41

C 语言练习实例42

C 语言练习实例43

C 语言练习实例44

C 语言练习实例45

C 语言练习实例46

C 语言练习实例47

C 语言练习实例48

C 语言练习实例49

C 语言练习实例50

C 语言练习实例51

C 语言练习实例52

C 语言练习实例53

C 语言练习实例54

C 语言练习实例55

C 语言练习实例56

C 语言练习实例57

C 语言练习实例58

C 语言练习实例59

C 语言练习实例60

C 语言练习实例61

C 语言练习实例62

C 语言练习实例63

C 语言练习实例64

C 语言练习实例65

C 语言练习实例66

C 语言练习实例67

C 语言练习实例68

C 语言练习实例69

C 语言练习实例70

C 语言练习实例71

C 语言练习实例72

C 语言练习实例73

C 语言练习实例74

C 语言练习实例75

C 语言练习实例76

C 语言练习实例77

C 语言练习实例78

C 语言练习实例79

C 语言练习实例80

C 语言练习实例81

C 语言练习实例82

C 语言练习实例83

C 语言练习实例84

C 语言练习实例85

C 语言练习实例86

C 语言练习实例87

C 语言练习实例88

C 语言练习实例89

C 语言练习实例90

C 语言练习实例91

C 语言练习实例92

C 语言练习实例93

C 语言练习实例94

C 语言练习实例95

C 语言练习实例96

C 语言练习实例97

C 语言练习实例98

C 语言练习实例99

C 语言练习实例100

C 标准库 - 参考手册

C 语言是一种通用的、面向过程式的计算机程序设计语言。1972 年，为了移植与开发 UNIX 操作系统，丹尼斯·里奇在贝尔电话实验室设计开发了 C 语言。

C 语言是一种广泛使用的计算机语言，它与 Java 编程语言一样普及，二者在现代软件程序员之间都得到广泛使用。

C 标准库是一组 C 内置函数、常量和头文件，比如 <stdio.h>、<stdlib.h>、<math.h>，等等。这个标准库可以作为 C 程序员的参考手册。

谁适合阅读本教程？

C 标准库可以作为 C 程序员的参考手册，C 程序员在开发系统编程相关的项目时可以参阅这个手册。系统相关的每一个步骤帮助他们参考。我们以易于理解的方式对所有的 C 函数进行讲解，您可以直接在项目中复制使用这些项目。

阅读本教程前，您需要了解的知识：

对 C 语言有基本的了解将有助于您理解本标准库涵盖的 C 内置函数。

编译/执行 C 程序

如果您想要在 Linux 服务器上学习 C 编程，但是又没有相关的配置环境，那么可以访问 compileonline.com。您只需进行简单的点击动作，即可在高端的服务器上体验真实的编程经验。这是完全免费的在线工具。

C 标准库 - <assert.h>

简介

C 标准库的 **assert.h**头文件提供了一个名为 **assert** 的宏，它可用于验证程序做出的假设，并在假设为假时输出诊断消息。

已定义的宏 **assert** 指向另一个宏 **NDEBUG**，宏 **NDEBUG** 不是 <assert.h> 的一部分。如果已在引用 <assert.h> 的源文件中定义 **NDEBUG** 为宏名称，则 **assert** 宏的定义如下：

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

库宏

下面列出了头文件 **assert.h** 中定义的唯一函数：

序号	函数 & 描述
1	<div><div><u>void assert(int expression)</u></div><div>这实际上是一个宏，不是一个函数，可用于在 C 程序中添加诊断。</div></div>

C 标准库 - <ctype.h>

简介

C 标准库的 **ctype.h** 头文件提供了一些函数，可用于测试和映射字符。

这些函数接受 **int** 作为参数，它的值必须是 EOF 或表示为一个无符号字符。

如果参数 **c** 满足描述的条件，则这些函数返回非零（true）。如果参数 **c** 不满足描述的条件，则这些函数返回零。

库函数

下面列出了头文件 **ctype.h** 中定义的函数：

序号	函数 & 描述
1	int isalnum(int c) 该函数检查所传的字符是否是字母和数字。
2	int isalpha(int c) 该函数检查所传的字符是否是字母。
3	int iscntrl(int c) 该函数检查所传的字符是否是控制字符。
4	int isdigit(int c) 该函数检查所传的字符是否是十进制数字。
5	int isgraph(int c) 该函数检查所传的字符是否有图形表示法。
6	int islower(int c) 该函数检查所传的字符是否是小写字母。
7	int isprint(int c) 该函数检查所传的字符是否是可打印的。
8	int ispunct(int c) 该函数检查所传的字符是否是标点符号字符。
9	int isspace(int c) 该函数检查所传的字符是否是空白字符。
10	int isupper(int c) 该函数检查所传的字符是否是大写字母。
11	int isxdigit(int c) 该函数检查所传的字符是否是十六进制数字。

标准库还包含了两个转换函数，它们接受并返回一个 "int"

序号	函数 & 描述
1	int tolower(int c) 该函数把大写字母转换为小写字母。
2	int toupper(int c) 该函数把小写字母转换为大写字母。

字符类

序号	字符类 & 描述
1	数字

	完整的数字集合 {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
2	十六进制数字 集合 {0123456789ABCDEFabcdef}
3	小写字母 集合 {abcdefghijklmnopqrstuvwxyz}
4	大写字母 集合 {ABCDEFGHIJKLMNOPQRSTUVWXYZ}
5	字母 小写字母和大写字母的集合
6	字母数字字符 数字、小写字母和大写字母的集合
7	标点符号字符 集合 !"#\$%&'()*+,-./:;<=>?@[\]^_`{ }~
8	图形字符 字母数字字符和标点符号字符的集合
9	空格字符 制表符、换行符、垂直制表符、换页符、回车符、空格符的集合。
10	可打印字符 字母数字字符、标点符号字符和空格字符的集合。
11	控制字符 在 ASCII 编码中，这些字符的八进制代码是从 000 到 037，以及 177 (DEL)。
12	空白字符 包括空格符和制表符。
13	字母字符 小写字母和大写字母的集合。

C 标准库 - <errno.h>

简介

C 标准库的 **errno.h** 头文件定义了整数变量 **errno**，它是通过系统调用设置的，在错误事件中的某些库函数表明了什么发生了错误。该宏扩展为类型为 `int` 的可更改的左值，因此它可以被一个程序读取和修改。

在程序启动时，**errno** 设置为零，C 标准库中的特定函数修改它的值为一些非零值以表示某些类型的错误。您也可以在适当的时候修改它的值或重置为零。

errno.h 头文件也顶了以一系列表示不同错误代码的宏，这些宏应扩展为类型为 **int** 的整数常量表达式。

库宏

下面列出了头文件 `errno.h` 中定义的宏：

序号	宏 & 描述
1	<code>extern int errno</code> 这是通过系统调用设置的宏，在错误事件中的某些库函数表明了什么发生了错误。
2	<code>EDOM Domain Error</code> 这个宏表示一个域错误，它在输入参数超出数学函数定义的域时发生， <code>errno</code> 被设置为 <code>EDOM</code> 。
3	<code>ERANGE Range Error</code> 这个宏表示一个范围错误，它在输入参数超出数学函数定义的范围时发生， <code>errno</code> 被设置为 <code>ERANGE</code> 。

C 标准库 - <float.h>

简介

C 标准库的 **float.h** 头文件包含了一组与浮点值相关的依赖于平台的常量。这些常量是由 ANSI C 提出的，这让程序更具有可移植性。在讲解这些常量之前，最好先弄清楚浮点数是由下面四个元素组成的：

组件	组件描述
S	符号 (+/-)
b	指数表示的基数，2 表示二进制，10 表示十进制，16 表示十六进制，等等...
e	指数，一个介于最小值 e_{min} 和最大值 e_{max} 之间的整数。
p	精度，基数 b 的有效位数

基于以上 4 个组成部分，一个浮点数的值如下：

floating-point = (S) p x b^e

或

floating-point = (+/-) precision x base^{exponent}

库宏

下面的值是特定实现的，且是通过 **#define** 指令来定义的，这些值都不得低于下边所给出的值。请注意，所有的实例 FLT 是指类型 float，DBL 是指类型 double，LDBL 是指类型 long double。

宏	描述
FLT_ROUNDS	定义浮点加法的舍入模式，它可以是下列任何一个值： -1 - 无法确定 0 - 趋向于零 1 - 去最近的值 2 - 趋向于正无穷 3 - 趋向于负无穷
FLT_RADIX 2	这个宏定义了指数表示的基数。基数 2 表示二进制，基数 10 表示十进制，基数 16 表示十六进制。
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	这些宏定义了 FLT_RADIX 基数中的位数。
FLT_DIG 6 DBL_DIG 10 LDBL_DIG 10	这些宏定义了舍入后不会改变表示的十进制数字的最大值（基数 10）。
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	这些宏定义了基数为 FLT_RADIX 时的指数的最小负整数值。
FLT_MIN_10_EXP -37 DBL_MIN_10_EXP -37 LDBL_MIN_10_EXP -37	这些宏定义了基数为 10 时的指数的最小负整数值。
FLT_MAX_EXP DBL_MAX_EXP	这些宏定义了基数为 FLT_RADIX 时的指数的最大整数值。

LDBL_MAX_EXP	
FLT_MAX_10_EXP +37 DBL_MAX_10_EXP +37 LDBL_MAX_10_EXP +37	这些宏定义了基数为 10 时的指数的最大整数值。
FLT_MAX 1E+37 DBL_MAX 1E+37 LDBL_MAX 1E+37	这些宏定义最大的有限浮点值。
FLT_EPSILON 1E-5 DBL_EPSILON 1E-9 LDBL_EPSILON 1E-9	这些宏定义了可表示的最小有效数字。
FLT_MIN 1E-37 DBL_MIN 1E-37 LDBL_MIN 1E-37	这些宏定义了最小的浮点值。

实例

下面的实例演示了 `float.h` 文件中定义的一些常量的使用。

```
#include <stdio.h>
#include <float.h>

int main()
{
    printf("The maximum value of float = %.10e\n", FLT_MAX);
    printf("The minimum value of float = %.10e\n", FLT_MIN);

    printf("The number of digits in the number = %.10e\n", FLT_MANT_DIG);
}
```

让我们编译和运行上面的程序，这将产生下列结果：

```
The maximum value of float = 3.4028234664e+38
The minimum value of float = 1.1754943508e-38
The number of digits in the number = 7.2996655210e-312
```

C 标准库 - <limits.h>

简介

limits.h 头文件决定了各种变量类型的各种属性。定义在该头文件中的宏限制了各种变量类型（比如 char、int 和 long）的值。这些限制指定了变量不能存储任何超出这些限制的值，例如一个无符号可以存储的最大值是 255。

库宏

下面的值是特定实现的，且是通过 **#define** 指令来定义的，这些值都不得低于下边所给出的值。

宏	值	描述
CHAR_BIT	8	定义一个字节的比特数。
SCHAR_MIN	-128	定义一个有符号字符的最小值。
SCHAR_MAX	127	定义一个有符号字符的最大值。
UCHAR_MAX	255	定义一个无符号字符的最大值。
CHAR_MIN	0	定义类型 char 的最小值，如果 char 表示负值，则它的值等于 SCHAR_MIN，否则等于 0。
CHAR_MAX	127	定义类型 char 的最大值，如果 char 表示负值，则它的值等于 SCHAR_MAX，否则等于 UCHAR_MAX。
MB_LEN_MAX	1	定义多字节字符中的最大字节数。
SHRT_MIN	-32768	定义一个短整型的最小值。
SHRT_MAX	+32767	定义一个短整型的最大值。
USHRT_MAX	65535	定义一个无符号短整型的最大值。
INT_MIN	-32768	定义一个整型的最小值。
INT_MAX	+32767	定义一个整型的最大值。
UINT_MAX	65535	定义一个无符号整型的最大值。
LONG_MIN	-2147483648	定义一个长整型的最小值。
LONG_MAX	+2147483647	定义一个长整型的最大值。
ULONG_MAX	4294967295	定义一个无符号长整型的最大值。

实例

下面的实例演示了 limit.h 文件中定义的一些常量的使用。

```
#include <stdio.h>
#include <limits.h>

int main()
{

    printf("The number of bits in a byte %d\n", CHAR_BIT);

    printf("The minimum value of SIGNED CHAR = %d\n", SCHAR_MIN);
    printf("The maximum value of SIGNED CHAR = %d\n", SCHAR_MAX);
    printf("The maximum value of UNSIGNED CHAR = %d\n", UCHAR_MAX);

    printf("The minimum value of SHORT INT = %d\n", SHRT_MIN);
```

```
printf("The maximum value of SHORT INT = %d\n", SHRT_MAX);

printf("The minimum value of INT = %d\n", INT_MIN);
printf("The maximum value of INT = %d\n", INT_MAX);

printf("The minimum value of CHAR = %d\n", CHAR_MIN);
printf("The maximum value of CHAR = %d\n", CHAR_MAX);

printf("The minimum value of LONG = %ld\n", LONG_MIN);
printf("The maximum value of LONG = %ld\n", LONG_MAX);

return(0);
}
```

让我们编译和运行上面的程序，这将产生下列结果：

```
The number of bits in a byte 8
The minimum value of SIGNED CHAR = -128
The maximum value of SIGNED CHAR = 127
The maximum value of UNSIGNED CHAR = 255
The minimum value of SHORT INT = -32768
The maximum value of SHORT INT = 32767
The minimum value of INT = -32768
The maximum value of INT = 32767
The minimum value of CHAR = -128
The maximum value of CHAR = 127
The minimum value of LONG = -2147483648
The maximum value of LONG = 2147483647
```

C 标准库 - <locale.h>

简介

locale.h 头文件定义了特定地域的设置，比如日期格式和货币符号。接下来我们将介绍一些宏，以及一个重要的结构 **struct lconv** 和两个重要的函数。

库宏

下面列出了头文件 locale.h 中定义的宏，这些宏将在下列的两个函数中使用：

序号	宏 & 描述
1	LC_ALL 设置下面的所有选项。
2	LC_COLLATE 影响 strcoll 和 strxfrm 函数。
3	LC_CTYPE 影响所有字符函数。
4	LC_MONETARY 影响 localeconv 函数提供的货币信息。
5	LC_NUMERIC 影响 localeconv 函数提供的小数点格式化和信息。
6	LC_TIME 影响 strftime 函数。

库函数

下面列出了头文件 locale.h 中定义的函数：

序号	函数 & 描述
1	char *setlocale(int category, const char *locale) 设置或读取地域化信息。
2	struct lconv *localeconv(void) 设置或读取地域化信息。

库结构

```
typedef struct {
    char *decimal_point;
    char *thousands_sep;
    char *grouping;
    char *int_curr_symbol;
    char *currency_symbol;
    char *mon_decimal_point;
    char *mon_thousands_sep;
    char *mon_grouping;
    char *positive_sign;
    char *negative_sign;
    char int_frac_digits;
    char frac_digits;
    char p_cs_precedes;
    char p_sep_by_space;
    char n_cs_precedes;
    char n_sep_by_space;
    char p_sign_posn;
    char n_sign_posn;
```

```
} lconv
```

以下是各字段的描述：

序号	字段 & 描述
1	decimal_point 用于非货币值的小数点字符。
2	thousands_sep 用于非货币值的千位分隔符。
3	grouping 一个表示非货币量中每组数字大小的字符串。每个字符代表一个整数值，每个整数指定当前组的位数。值为 0 意味着前一个值将应用于剩余的分组。
4	int_curr_symbol 国际货币符号使用的字符串。前三个字符是由 ISO 4217:1987 指定的，第四个字符用于分隔货币符号和货币量。
5	currency_symbol 用于货币的本地符号。
6	mon_decimal_point 用于货币值的小数点字符。
7	mon_thousands_sep 用于货币值的千位分隔符。
8	mon_grouping 一个表示货币值中每组数字大小的字符串。每个字符代表一个整数值，每个整数指定当前组的位数。值为 0 意味着前一个值将应用于剩余的分组。
9	positive_sign 用于正货币值的字符。
10	negative_sign 用于负货币值的字符。
11	int_frac_digits 国际货币值中小数点后要显示的位数。
12	frac_digits 货币值中小数点后要显示的位数。
13	p_cs_precedes 如果等于 1，则 currency_symbol 出现在正货币值之前。如果等于 0，则 currency_symbol 出现在正货币值之后。
14	p_sep_by_space 如果等于 1，则 currency_symbol 和正货币值之间使用空格分隔。如果等于 0，则 currency_symbol 和正货币值之间不使用空格分隔。
15	n_cs_precedes 如果等于 1，则 currency_symbol 出现在负货币值之前。如果等于 0，则 currency_symbol 出现在负货币值之后。
16	n_sep_by_space 如果等于 1，则 currency_symbol 和负货币值之间使用空格分隔。如果等于 0，则 currency_symbol 和负货币值之间不使用空格分隔。
17	p_sign_posn

	表示正货币值中正号的位置。
18	n_sign_posn 表示负货币值中负号的位置。

下面的值用于 **p_sign_posn** 和 **n_sign_posn**:

值	描述
0	封装值和 currency_symbol 的括号。
1	放置在值和 currency_symbol 之前的符号。
2	放置在值和 currency_symbol 之后的符号。
3	紧挨着放置在值和 currency_symbol 之前的符号。
4	紧挨着放置在值和 currency_symbol 之后的符号。

C 标准库 - <math.h>

简介

math.h 头文件定义了各种数学函数和一个宏。在这个库中所有可用的功能都带有一个 **double** 类型的参数，且都返回 **double** 类型的结果。

库宏

下面是这个库中定义的唯一的一个宏：

序号	宏 & 描述
1	HUGE_VAL 当函数的结果不可以表示为浮点数时。如果是因为结果的幅度太大以致于无法表示，则函数会设置 errno 为 ERANGE 来表示范围错误，并返回一个由宏 HUGE_VAL 或者它的否定 (- HUGE_VAL) 命名的一个特定的很大的值。 如果结果的幅度太小，则会返回零值。在这种情况下， error 可能会被设置为 ERANGE ，也有可能不会被设置为 ERANGE 。

库函数

下面列出了头文件 **math.h** 中定义的函数：

序号	函数 & 描述
1	<u>double acos(double x)</u> 返回以弧度表示的 x 的反余弦。
2	<u>double asin(double x)</u> 返回以弧度表示的 x 的反正弦。
3	<u>double atan(double x)</u> 返回以弧度表示的 x 的反正切。
4	<u>double atan2(double y, double x)</u> 返回以弧度表示的 y/x 的反正切。y 和 x 的值的符号决定了正确的象限。
5	<u>double cos(double x)</u> 返回弧度角 x 的余弦。
6	<u>double cosh(double x)</u> 返回 x 的双曲余弦。
7	<u>double sin(double x)</u> 返回弧度角 x 的正弦。
8	<u>double sinh(double x)</u> 返回 x 的双曲正弦。
9	<u>double tanh(double x)</u> 返回 x 的双曲正切。
10	<u>double exp(double x)</u> 返回 e 的 x 次幂的值。
11	<u>double frexp(double x, int *exponent)</u> 把浮点数 x 分解成尾数和指数。返回值是尾数，并将指数存入 exponent 中。所得的值是 x = mantissa * 2 ^ exponent 。
12	<u>double ldexp(double x, int exponent)</u> 返回 x 乘以 2 的 exponent 次幂。
13	<u>double log(double x)</u> 返回 x 的自然对数（基数为 e 的对数）。

14	<u><code>double log10(double x)</code></u> 返回 x 的常用对数（基数为 10 的对数）。
15	<u><code>double modf(double x, double *integer)</code></u> 返回值为小数部分（小数点后的部分），并设置 integer 为整数部分。
16	<u><code>double pow(double x, double y)</code></u> 返回 x 的 y 次幂。
17	<u><code>double sqrt(double x)</code></u> 返回 x 的平方根。
18	<u><code>double ceil(double x)</code></u> 返回大于或等于 x 的最小的整数值。
19	<u><code>double fabs(double x)</code></u> 返回 x 的绝对值。
20	<u><code>double floor(double x)</code></u> 返回小于或等于 x 的最大的整数值。
21	<u><code>double fmod(double x, double y)</code></u> 返回 x 除以 y 的余数。

C 标准库 - <setjmp.h>

简介

setjmp.h 头文件定义了宏 **setjmp()**、函数 **longjmp()** 和变量类型 **jmp_buf**，该变量类型会绕过正常的函数调用和返回规则。

库变量

下面列出了头文件 setjmp.h 中定义的变量：

序号	变量 & 描述
1	jmp_buf 这是一个用于存储宏 setjmp() 和函数 longjmp() 相关信息的数组类型。

库宏

下面是这个库中定义的唯一的一个宏：

序号	宏 & 描述
1	<u>int setjmp(jmp_buf environment)</u> 这个宏把当前环境保存在变量 environment 中，以便函数 longjmp() 后续使用。如果这个宏直接从宏调用中返回，则它会返回零，但是如果它从 longjmp() 函数调用中返回，则它会返回一个非零值。

库函数

下面是头文件 setjmp.h 中定义的唯一的一个函数：

序号	函数 & 描述
1	<u>void longjmp(jmp_buf environment, int value)</u> 该函数恢复最近一次调用 setjmp() 宏时保存的环境， jmp_buf 参数的设置是由之前调用 setjmp() 生成的。

C 标准库 - <signal.h>

简介

signal.h 头文件定义了一个变量类型 **sig_atomic_t**、两个函数调用和一些宏来处理程序执行期间报告的不同信号。

库变量

下面是头文件 **signal.h** 中定义的变量类型：

序号	变量 & 描述
1	sig_atomic_t 这是 int 类型，在信号处理程序中作为变量使用。它是一个对象的整数类型，该对象可以作为一个原子实体访问，即使存在异步信号时，该对象可以作为一个原子实体访问。

库宏

下面是头文件 **signal.h** 中定义的宏，这些宏将在下列两个函数中使用。**SIG_** 宏与 **signal** 函数一起使用来定义信号的功能。

序号	宏 & 描述
1	SIG_DFL 默认的信号处理程序。
2	SIG_ERR 表示一个信号错误。
3	SIG_IGN 忽视信号。

SIG 宏用于表示以下各种条件的信号码：

序号	宏 & 描述
1	SIGABRT 程序异常终止。
2	SIGFPE 算术运算出错，如除数为 0 或溢出。
3	SIGILL 非法函数映象，如非法指令。
4	SIGINT 中断信号，如 ctrl-C。
5	SIGSEGV 非法访问存储器，如访问不存在的内存单元。
6	SIGTERM 发送给本程序的终止请求信号。

库函数

下面是头文件 **signal.h** 中定义的函数：

序号	函数 & 描述
1	<u>void (*signal(int sig, void (*func)(int)))(int)</u> 该函数设置一个函数来处理信号，即信号处理程序。
2	<u>int raise(int sig)</u> 该函数会促使生成信号 sig 。sig 参数与 SIG 宏兼容。

C 标准库 - <stdarg.h>

简介

stdarg.h 头文件定义了一个变量类型 **va_list** 和三个宏，这三个宏可用于在参数个数未知（即参数个数可变）时获取函数中的参数。
可变参数的函数通在参数列表的末尾是使用省略号(...)定义的。

库变量

下面是头文件 **stdarg.h** 中定义的变量类型：

序号	变量 & 描述
1	va_list 这是一个适用于 va_start() 、 va_arg() 和 va_end() 这三个宏存储信息的类型。

库宏

下面是头文件 **stdarg.h** 中定义的宏：

序号	宏 & 描述
1	<u>void va_start(va_list ap, last_arg)</u> 这个宏初始化 ap 变量，它与 va_arg 和 va_end 宏是一起使用的。 last_arg 是最后一个传递给函数的已知的固定参数，即省略号之前的参数。
2	<u>type va_arg(va_list ap, type)</u> 这个宏检索函数参数列表中类型为 type 的下一个参数。
3	<u>void va_end(va_list ap)</u> 这个宏允许使用了 va_start 宏的带有可变参数的函数返回。如果在从函数返回之前没有调用 va_end ，则结果为未定义。

C 标准库 - <stddef.h>

简介

`stddef.h` 头文件定义了各种变量类型和宏。这些定义中的大部分也出现在其它头文件中。

库变量

下面是头文件 `stddef.h` 中定义的变量类型：

序号	变量 & 描述
1	<code>ptrdiff_t</code> 这是有符号整数类型，它是两个指针相减的结果。
2	<code>size_t</code> 这是无符号整数类型，它是 <code>sizeof</code> 关键字的结果。
3	<code>wchar_t</code> 这是一个宽字符常量大小的整数类型。

库宏

下面是头文件 `stddef.h` 中定义的宏：

序号	宏 & 描述
1	<u><code>NULL</code></u> 这个宏是一个空指针常量的值。
2	<u><code>offsetof(type, member-designator)</code></u> 这会生成一个类型为 <code>size_t</code> 的整型常量，它是一个结构成员相对于结构开头的字节偏移量。成员是由 <i>member-designator</i> 给定的，结构的名称是在 <i>type</i> 中给定的。

C 标准库 - <stdio.h>

简介

`stdio.h` 头文件定义了三个变量类型、一些宏和各种函数来执行输入和输出。

库变量

下面是头文件 `stdio.h` 中定义的变量类型：

序号	变量 & 描述
1	size_t 这是无符号整数类型，它是 sizeof 关键字的结果。
2	FILE 这是一个适合存储文件流信息的对象类型。
3	fpos_t 这是一个适合存储文件中任何位置的对象类型。

库宏

下面是头文件 `stdio.h` 中定义的宏：

序号	宏 & 描述
1	NULL 这个宏是一个空指针常量的值。
2	_IOFBF 、 _IOLBF 和 _IONBF 这些宏扩展了带有特定值的整型常量表达式，并适用于 setvbuf 函数的第三个参数。
3	BUFSIZ 这个宏是一个整数，该整数代表了 setbuf 函数使用的缓冲区大小。
4	EOF 这个宏是一个表示已经到达文件结束的负整数。
5	FOPEN_MAX 这个宏是一个整数，该整数代表了系统可以同时打开的文件数量。
6	FILENAME_MAX 这个宏是一个整数，该整数代表了字符数组可以存储的文件名的最大长度。如果实现没有限制，则该值应为推荐的最大值。
7	L_tmpnam 这个宏是一个整数，该整数代表了字符数组可以存储的由 tmpnam 函数创建的临时文件名的最大长度。
8	SEEK_CUR 、 SEEK_END 和 SEEK_SET 这些宏是在These macros are used in the fseek 函数中使用，用于在一个文件中定位不同的位置。
9	TMP_MAX 这个宏是 tmpnam 函数可生成的独特文件名的最大数量。
10	stderr 、 stdin 和 stdout 这些宏是指向 FILE 类型的指针，分别对应于标准错误、标准输入和标准输出流。

库函数

下面是头文件 `stdio.h` 中定义的函数：

为了更好地理解函数，请按照下面的序列学习这些函数，因为第一个函数中创建的文件会在后续的函数中使用到。

序号	函数 & 描述
1	<u>int fclose(FILE *stream)</u> 关闭流 stream。刷新所有的缓冲区。
2	<u>void clearerr(FILE *stream)</u> 清除给定流 stream 的文件结束和错误标识符。
3	<u>int feof(FILE *stream)</u> 测试给定流 stream 的文件结束标识符。
4	<u>int ferror(FILE *stream)</u> 测试给定流 stream 的错误标识符。
5	<u>int fflush(FILE *stream)</u> 刷新流 stream 的输出缓冲区。
6	<u>int fgetpos(FILE *stream, fpos_t *pos)</u> 获取流 stream 的当前文件位置，并把它写入到 pos。
7	<u>FILE *fopen(const char *filename, const char *mode)</u> 使用给定的模式 mode 打开 filename 所指向的文件。
8	<u>size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)</u> 从给定流 stream 读取数据到 ptr 所指向的数组中。
9	<u>FILE *freopen(const char *filename, const char *mode, FILE *stream)</u> 把一个新的文件名 filename 与给定的打开的流 stream 关联，同时关闭流中的旧文件。
10	<u>int fseek(FILE *stream, long int offset, int whence)</u> 设置流 stream 的文件位置为给定的偏移 offset，参数 offset 意味着从给定的 whence 位置查找的字节数。
11	<u>int fsetpos(FILE *stream, const fpos_t *pos)</u> 设置给定流 stream 的文件位置为给定的位置。参数 pos 是由函数 fgetpos 给定的位置。
12	<u>long int ftell(FILE *stream)</u> 返回给定流 stream 的当前文件位置。
13	<u>size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)</u> 把 ptr 所指向的数组中的数据写入到给定流 stream 中。
14	<u>int remove(const char *filename)</u> 删除给定的文件名 filename，以便它不再被访问。
15	<u>int rename(const char *old_filename, const char *new_filename)</u> 把 old_filename 所指向的文件名改为 new_filename。
16	<u>void rewind(FILE *stream)</u> 设置文件位置为给定流 stream 的文件的开头。
17	<u>void setbuf(FILE *stream, char *buffer)</u> 定义流 stream 应如何缓冲。
18	<u>int setvbuf(FILE *stream, char *buffer, int mode, size_t size)</u> 另一个定义流 stream 应如何缓冲的函数。
19	<u>FILE *tmpfile(void)</u> 以二进制更新模式(wb+)创建临时文件。

20	<code>char *tmpnam(char *str)</code> 生成并返回一个有效的临时文件名，该文件名之前是不存在的。
21	<code>int fprintf(FILE *stream, const char *format, ...)</code> 发送格式化输出到流 <code>stream</code> 中。
22	<code>int printf(const char *format, ...)</code> 发送格式化输出到标准输出 <code>stdout</code> 。
23	<code>int sprintf(char *str, const char *format, ...)</code> 发送格式化输出到字符串。
24	<code>int vfprintf(FILE *stream, const char *format, va_list arg)</code> 使用参数列表发送格式化输出到流 <code>stream</code> 中。
25	<code>int vprintf(const char *format, va_list arg)</code> 使用参数列表发送格式化输出到标准输出 <code>stdout</code> 。
26	<code>int vsprintf(char *str, const char *format, va_list arg)</code> 使用参数列表发送格式化输出到字符串。
27	<code>int fscanf(FILE *stream, const char *format, ...)</code> 从流 <code>stream</code> 读取格式化输入。
28	<code>int scanf(const char *format, ...)</code> 从标准输入 <code>stdin</code> 读取格式化输入。
29	<code>int sscanf(const char *str, const char *format, ...)</code> 从字符串读取格式化输入。
30	<code>int fgetc(FILE *stream)</code> 从指定的流 <code>stream</code> 获取下一个字符（一个无符号字符），并把位置标识符往前移动。
31	<code>char *fgets(char *str, int n, FILE *stream)</code> 从指定的流 <code>stream</code> 读取一行，并把它存储在 <code>str</code> 所指向的字符串内。当读取 (n-1) 个字符时，或者读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。
32	<code>int fputc(int char, FILE *stream)</code> 把参数 <code>char</code> 指定的字符（一个无符号字符）写入到指定的流 <code>stream</code> 中，并把位置标识符往前移动。
33	<code>int fputs(const char *str, FILE *stream)</code> 把字符串写入到指定的流 <code>stream</code> 中，但不包括空字符。
34	<code>int getc(FILE *stream)</code> 从指定的流 <code>stream</code> 获取下一个字符（一个无符号字符），并把位置标识符往前移动。
35	<code>int getchar(void)</code> 从标准输入 <code>stdin</code> 获取一个字符（一个无符号字符）。
36	<code>char *gets(char *str)</code> 从标准输入 <code>stdin</code> 读取一行，并把它存储在 <code>str</code> 所指向的字符串中。当读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。
37	<code>int putc(int char, FILE *stream)</code> 把参数 <code>char</code> 指定的字符（一个无符号字符）写入到指定的流 <code>stream</code> 中，并把位置标识符往前移动。
38	<code>int putchar(int char)</code> 把参数 <code>char</code> 指定的字符（一个无符号字符）写入到标准输出 <code>stdout</code> 中。

39	<u>int puts(const char *str)</u> 把一个字符串写入到标准输出 <code>stdout</code> ，直到空字符，但不包括空字符。换行符会被追加到输出中。
40	<u>int ungetc(int char, FILE *stream)</u> 把字符 <code>char</code> （一个无符号字符）推入到指定的流 <code>stream</code> 中，以便它是下一个被读取到的字符。
41	<u>void perror(const char *str)</u> 把一个描述性错误消息输出到标准错误 <code>stderr</code> 。首先输出字符串 <code>str</code> ，后跟一个冒号，然后是一个空格。

C 标准库 - <stdlib.h>

简介

`stdlib.h` 头文件定义了四个变量类型、一些宏和各种通用工具函数。

库变量

下面是头文件 `stdlib.h` 中定义的变量类型：

序号	变量 & 描述
1	size_t 这是无符号整数类型，它是 sizeof 关键字的结果。
2	wchar_t 这是一个宽字符常量大小的整数类型。
3	div_t 这是 div 函数返回的结构。
4	ldiv_t 这是 ldiv 函数返回的结构。

库宏

下面是头文件 `stdlib.h` 中定义的宏：

序号	宏 & 描述
1	NULL 这个宏是一个空指针常量的值。
2	EXIT_FAILURE 这是 <code>exit</code> 函数失败时要返回的值。
3	EXIT_SUCCESS 这是 <code>exit</code> 函数成功时要返回的值。
4	RAND_MAX 这个宏是 <code>rand</code> 函数返回的最大值。
5	MB_CUR_MAX 这个宏表示在多字节字符集中的最大字符数，不能大于 <code>MB_LEN_MAX</code> 。

库函数

下面是头文件 `stdlib.h` 中定义的函数：

序号	函数 & 描述
1	<u><code>double atof(const char *str)</code></u> 把参数 <code>str</code> 所指向的字符串转换为一个浮点数（类型为 <code>double</code> 型）。
2	<u><code>int atoi(const char *str)</code></u> 把参数 <code>str</code> 所指向的字符串转换为一个整数（类型为 <code>int</code> 型）。
3	<u><code>long int atol(const char *str)</code></u> 把参数 <code>str</code> 所指向的字符串转换为一个长整数（类型为 <code>long int</code> 型）。
4	<u><code>double strtod(const char *str, char **endptr)</code></u> 把参数 <code>str</code> 所指向的字符串转换为一个浮点数（类型为 <code>double</code> 型）。
5	<u><code>long int strtol(const char *str, char **endptr, int base)</code></u>

把参数 *str* 所指向的字符串转换为一个长整数（类型为 long int 型）。

6 [unsigned long int strtoul\(const char *str, char **endptr, int base\)](#)

把参数 *str* 所指向的字符串转换为一个无符号长整数（类型为 unsigned long int 型）。

7 [void *calloc\(size_t nitems, size_t size\)](#)

分配所需的内存空间，并返回一个指向它的指针。

8 [void free\(void *ptr\)](#)

释放之前调用 *calloc*、*malloc* 或 *realloc* 所分配的内存空间。

9 [void *malloc\(size_t size\)](#)

分配所需的内存空间，并返回一个指向它的指针。

10 [void *realloc\(void *ptr, size_t size\)](#)

尝试重新调整之前调用 *malloc* 或 *calloc* 所分配的 *ptr* 所指向的内存块的大小。

11 [void abort\(void\)](#)

使一个异常程序终止。

12 [int atexit\(void \(*func\)\(void\)\)](#)

当程序正常终止时，调用指定的函数 **func**。

13 [void exit\(int status\)](#)

使程序正常终止。

14 [char *getenv\(const char *name\)](#)

搜索 *name* 所指向的环境字符串，并返回相关的值给字符串。

15 [int system\(const char *string\)](#)

由 *string* 指定的命令传给要被命令处理器执行的主机环境。

16 [void *bsearch\(const void *key, const void *base, size_t nitems, size_t size, int \(*compar\)\(const void *, const void *\)\)](#)

执行二分查找。

17 [void qsort\(void *base, size_t nitems, size_t size, int \(*compar\)\(const void *, const void *\)\)](#)

数组排序。

18 [int abs\(int x\)](#)

返回 *x* 的绝对值。

19 [div_t div\(int numer, int denom\)](#)

分子除以分母。

20 [long int labs\(long int x\)](#)

返回 *x* 的绝对值。

21 [ldiv_t ldiv\(long int numer, long int denom\)](#)

分子除以分母。

22 [int rand\(void\)](#)

返回一个范围在 0 到 *RAND_MAX* 之间的伪随机数。

23 [void srand\(unsigned int seed\)](#)

该函数播种由函数 **rand** 使用的随机数发生器。

24 [int mblen\(const char *str, size_t n\)](#)

返回参数 *str* 所指向的多字节字符的长度。

25	<u><code>size_t mbstowcs(schar_t *pwcs, const char *str, size_t n)</code></u> 把参数 <i>str</i> 所指向的多字节字符的字符串转换为参数 <i>pwcs</i> 所指向的数组。
26	<u><code>int mbtowlc(wchar_t *pwc, const char *str, size_t n)</code></u> 检查参数 <i>str</i> 所指向的多字节字符。
27	<u><code>size_t wcstombs(char *str, const wchar_t *pwcs, size_t n)</code></u> 把数组 <i>pwcs</i> 中存储的编码转换为多字节字符，并把它们存储在字符串 <i>str</i> 中。
28	<u><code>int wctomb(char *str, wchar_t wchar)</code></u> 检查对应于参数 <i>wchar</i> 所给出的多字节字符的编码。

C 标准库 - <string.h>

简介

`string.h` 头文件定义了一个变量类型、一个宏和各种操作字符数组的函数。

库变量

下面是头文件 `string.h` 中定义的变量类型：

序号	变量 & 描述
1	size_t 这是无符号整数类型，它是 sizeof 关键字的结果。

库宏

下面是头文件 `string.h` 中定义的宏：

序号	宏 & 描述
1	NULL 这个宏是一个空指针常量的值。

库函数

下面是头文件 `string.h` 中定义的函数：

序号	函数 & 描述
1	void *memchr(const void *str, int c, size_t n) 在参数 <i>str</i> 所指向的字符串的前 <i>n</i> 个字节中搜索第一次出现字符 <i>c</i> （一个无符号字符）的位置。
2	int memcmp(const void *str1, const void *str2, size_t n) 把 <i>str1</i> 和 <i>str2</i> 的前 <i>n</i> 个字节进行比较。
3	void *memcpy(void *dest, const void *src, size_t n) 从 <i>src</i> 复制 <i>n</i> 个字符到 <i>dest</i> 。
4	void *memmove(void *dest, const void *src, size_t n) 另一个用于从 <i>str2</i> 复制 <i>n</i> 个字符到 <i>str1</i> 的函数。
5	void *memset(void *str, int c, size_t n) 复制字符 <i>c</i> （一个无符号字符）到参数 <i>str</i> 所指向的字符串的前 <i>n</i> 个字符。
6	char *strcat(char *dest, const char *src) 把 <i>src</i> 所指向的字符串追加到 <i>dest</i> 所指向的字符串的结尾。
7	char *strncat(char *dest, const char *src, size_t n) 把 <i>src</i> 所指向的字符串追加到 <i>dest</i> 所指向的字符串的结尾，直到 <i>n</i> 字符长度为止。
8	char *strchr(const char *str, int c) 在参数 <i>str</i> 所指向的字符串中搜索第一次出现字符 <i>c</i> （一个无符号字符）的位置。
9	int strcmp(const char *str1, const char *str2) 把 <i>str1</i> 所指向的字符串和 <i>str2</i> 所指向的字符串进行比较。
10	int strncmp(const char *str1, const char *str2, size_t n) 把 <i>str1</i> 和 <i>str2</i> 进行比较，最多比较前 <i>n</i> 个字节。
11	int strcoll(const char *str1, const char *str2) 把 <i>str1</i> 和 <i>str2</i> 进行比较，结果取决于 <code>LC_COLLATE</code> 的位置设置。
12	char *strcpy(char *dest, const char *src)

	把 <i>src</i> 所指向的字符串复制到 <i>dest</i> 。
13	<u>char*strcpy(char*dest, const char*src, size_t n)</u> 把 <i>src</i> 所指向的字符串复制到 <i>dest</i> ，最多复制 <i>n</i> 个字符。
14	<u>size_t strcspn(const char*str1, const char*str2)</u> 检索字符串 <i>str1</i> 开头连续有几个字符都不含字符串 <i>str2</i> 中的字符。
15	<u>char*strerror(int errnum)</u> 从内部数组中搜索错误号 <i>errnum</i> ，并返回一个指向错误消息字符串的指针。
16	<u>size_t strlen(const char*str)</u> 计算字符串 <i>str</i> 的长度，直到空结束字符，但不包括空结束字符。
17	<u>char*strpbrk(const char*str1, const char*str2)</u> 检索字符串 <i>str1</i> 中第一个匹配字符串 <i>str2</i> 中字符的字符，不包含空结束字符。也就是说，依次检验字符串 <i>str1</i> 中的字符，当被检验字符在字符串 <i>str2</i> 中也包含时，则停止检验，并返回该字符位置。
18	<u>char*strrchr(const char*str, int c)</u> 在参数 <i>str</i> 所指向的字符串中搜索最后一次出现字符 <i>c</i> （一个无符号字符）的位置。
19	<u>size_t strspn(const char*str1, const char*str2)</u> 检索字符串 <i>str1</i> 中第一个不在字符串 <i>str2</i> 中出现的字符下标。
20	<u>char*strstr(const char*haystack, const char*needle)</u> 在字符串 <i>haystack</i> 中查找第一次出现字符串 <i>needle</i> （不包含空结束字符）的位置。
21	<u>char*strtok(char*str, const char*delim)</u> 分解字符串 <i>str</i> 为一组字符串， <i>delim</i> 为分隔符。
22	<u>size_t strxfrm(char*dest, const char*src, size_t n)</u> 根据程序当前的区域选项中的 LC_COLLATE 来转换字符串 src 的前 <i>n</i> 个字符，并把它们放置在字符串 dest 中。

C 标准库 - <time.h>

简介

time.h 头文件定义了四个变量类型、两个宏和各种操作日期和时间的函数。

库变量

下面是头文件 **time.h** 中定义的变量类型：

序号	变量 & 描述
1	size_t 是无符号整数类型，它是 sizeof 关键字的结果。
2	clock_t 这是一个适合存储处理器时间的类型。
3	time_t is 这是一个适合存储日历时间类型。
4	struct tm 这是一个用来保存时间和日期的结构。

tm 结构的定义如下：

```
struct tm {
    int tm_sec;           /* 秒，范围从 0 到 59      */
    int tm_min;           /* 分，范围从 0 到 59      */
    int tm_hour;          /* 小时，范围从 0 到 23    */
    int tm_mday;          /* 一月中的第几天，范围从 1 到 31 */
    int tm_mon;           /* 月，范围从 0 到 11      */
    int tm_year;          /* 自 1900 年起的年数      */
    int tm_wday;          /* 一周中的第几天，范围从 0 到 6 */
    int tm_yday;          /* 一年中的第几天，范围从 0 到 365 */
    int tm_isdst;         /* 夏令时                  */
};
```

库宏

下面是头文件 **time.h** 中定义的宏：

序号	宏 & 描述
1	NULL 这个宏是一个空指针常量的值。
2	CLOCKS_PER_SEC 这个宏表示每秒的处理器时钟个数。

库函数

下面是头文件 **time.h** 中定义的函数：

序号	函数 & 描述
1	<u>char *asctime(const struct tm *timeptr)</u> 返回一个指向字符串的指针，它代表了结构 timeptr 的日期和时间。
2	<u>clock_t clock(void)</u> 返回程序执行起（一般为程序的开头），处理器时钟所使用的时间。
3	<u>char *ctime(const time_t *timer)</u> 返回一个表示当地时间的字符串，当地时间是基于参数 timer 。

4	<u><code>double difftime(time_t time1, time_t time2)</code></u> 返回 time1 和 time2 之间相差的秒数 (time1-time2)。
5	<u><code>struct tm *gmtime(const time_t *timer)</code></u> timer 的值被分解为 tm 结构，并用协调世界时（UTC）也被称为格林尼治标准时间（GMT）表示。
6	<u><code>struct tm *localtime(const time_t *timer)</code></u> timer 的值被分解为 tm 结构，并用本地时区表示。
7	<u><code>time_t mktime(struct tm *timeptr)</code></u> 把 timeptr 所指向的结构转换为一个依据本地时区的 time_t 值。
8	<u><code>size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *timeptr)</code></u> 根据 format 中定义的格式化规则，格式化结构 timeptr 表示的时间，并把它存储在 str 中。
9	<u><code>time_t time(time_t *timer)</code></u> 计算当前日历时间，并把它编码成 time_t 格式。