**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。

[**现在开始学习 C 编程！**](http://www.runoob.com/cprogramming/c-intro.html)

[C 在线工具](http://www.runoob.com/try/showc.php?filename=helloworld&language=c)

**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;

}

**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) 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。

**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。

# 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上下载 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/](http://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](http://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：

1. 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。
2. 下一行 *int main()* 是主函数，程序从这里开始执行。
3. 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。
4. 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!"。
5. 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：

1. 打开一个文本编辑器，添加上述代码。
2. 保存文件为 *hello.c*。
3. 打开命令提示符，进入到保存文件所在的目录。
4. 键入 *gcc hello.c*，输入回车，编译代码。
5. 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。
6. 现在，键入 *a.out* 来执行程序。
7. 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 |  |  |  |

**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 *void* 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。

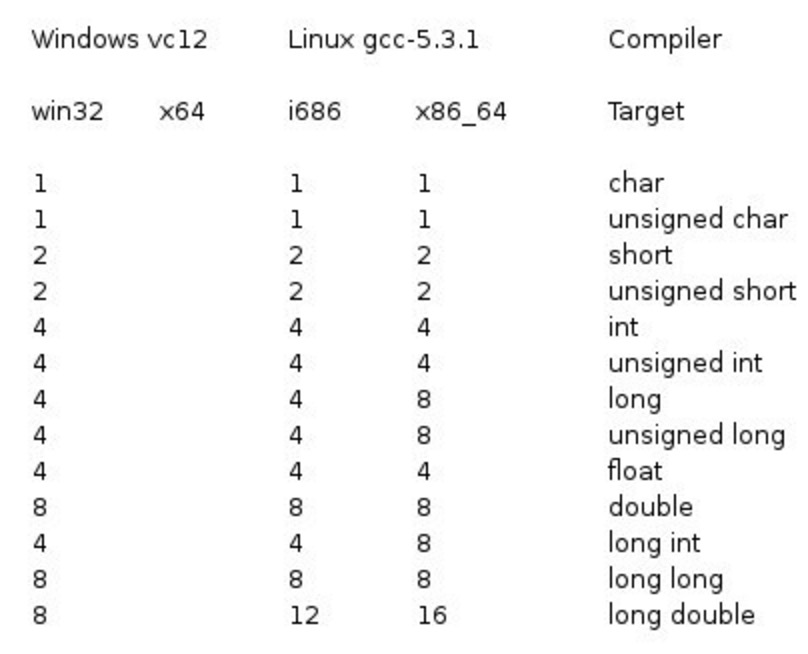
**整数类型**

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）：*

**

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空。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 | 通常是一个八位字节（一个字节）。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。 |
| double | 双精度浮点值。 |
| 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, 这就是单纯的声明

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未定义的。

## 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，或字符串字面值，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'—UNICODE 字符）。

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

字符串字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 //不要加类型声明，不要加分号

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，即在函数内部没有加类型修饰符的变量自动为auto的。

{

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，只创建一次。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。

在 C 编程中，当 **static** 用在类数据成员上时，会导致仅有一个该成员的副本被类的所有对象共享。

**实例**

#include <stdio.h>

/\* 函数声明 \*/

void func(void);

static int count = 5;

/\* 全局变量 \*/

main() {

while(count--) {

func();

}

return 0;

}

/\* 函数定义 \*/

void func(

void

) {

static int i = 5;

/\* 局部静态变量 \*/

i++;

printf("i is %d and count is %d\n", i, count);

}

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，后续的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：

i is 6 and count is 4

i is 7 and count is 3

i is 8 and count is 2

i is 9 and count is 1

i is 10 and count is 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(); 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

Window中：

F:\Html\mannual\_files\_all\C\_study\codes\cainiao\_codes>gcc externDemo.c externDemo01.c -o externDemo01 ，将两个文件编译进一个文件

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 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，划分通常是根据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（类似于java）。
* **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);

当您在一个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](http://www.runoob.com/cprogramming/c-function-call-by-value.html) | 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。 |
| [引用调用](http://www.runoob.com/cprogramming/c-function-call-by-pointer.html) | 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：

* 在函数或块内部的局部变量
* 在所有函数外部的全局变量
* 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。

#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：

#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：

#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：

#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（java中也是如此）。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。数组就是由连续的内存地址来实现的。



**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 ];//这与java中不同，java可以声明不指定大小的数组，初始化的时候在确定数组的大小；而且c的数组的声明语法也与java的有一些不同。

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的图形表示：



**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：

#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](http://www.runoob.com/cprogramming/c-multi-dimensional-arrays.html) | 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](http://www.runoob.com/cprogramming/c-passing-arrays-to-functions.html) | 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](http://www.runoob.com/cprogramming/c-return-arrays-from-function.html) | C 允许从函数返回数组。 |
| [指向数组的指针](http://www.runoob.com/cprogramming/c-pointer-to-an-array.html) | 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：

#include <stdio.h>

int main ()

{

int var1;

char var2[10];

printf("var1 变量的地址： %x\n", &var1 );

printf("var2 变量的地址： %x\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： bff5a400

var2 变量的地址： bff5a3f6

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（符号）。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：

#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: %x\n", &var );

/\* 在指针变量中存储的地址 \*/

printf("Address stored in ip variable: %x\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：

#include <stdio.h>

int main ()

{

int \*ptr = NULL;

printf("ptr 的值是 %x\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](http://www.runoob.com/cprogramming/c-pointer-arithmetic.html) | 可以对指针进行四种算术运算：++、--、+、- |
| [指针数组](http://www.runoob.com/cprogramming/c-array-of-pointers.html) | 可以定义用来存储指针的数组。 |
| [指向指针的指针](http://www.runoob.com/cprogramming/c-pointer-to-pointer.html) | C 允许指向指针的指针。（二级指针） |
| [传递指针给函数](http://www.runoob.com/cprogramming/c-passing-pointers-to-functions.html) | 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](http://www.runoob.com/cprogramming/c-return-pointer-from-functions.html) | 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。 |

\*pi++的效果是pi指向的地址在原先基础上增加了类型所占字节数

pi++与\*pi++的效果一样

（\*pi）++是将存储在指向的内存中的值加1；

就把\*当做一种操作符，他的优先级比++低。

**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；数组的地址就是第一个元素的地址；&arr的值就是&(arr[0])的 值。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：



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]，在这里出现的变量直接就可以使用，相当于是声明了;

**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：

#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 Book \*/

struct Books Book2; /\* 声明 Book2，类型为 Book \*/

/\* 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：

#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 Book \*/

struct Books Book2; /\* 声明 Book2，类型为 Book \*/

/\* 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：

#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 Book \*/

struct Books Book2; /\* 声明 Book2，类型为 Book \*/

/\* 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

# 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，带有多个成员。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：

#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：

#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：

#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。

## 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 位二进位存放一个开关量时，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位二进位。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位分配的。

### 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：

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 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，相当于是起了一个别名。下面的实例为单字节数字定义了一个术语 **BYTE**：

typedef unsigned char BYTE;

typedef int INT; //之后就可以将INT当做int来使用了

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。相当于定义了常量
* **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); //这里的类型指定后，c会自动为我们转换

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，则需要在您的计算机上先创建该目录。

#include <stdio.h>

main()

{

FILE \*fp;

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。

#include <stdio.h>

main()

{

FILE \*fp;

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，就相当于定义常量，而且不用指明变量的类型。

#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 //之后如果不再定义此常量，此常量就不能被访问了

#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(类似PHP中的魔术常量)

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

|  |  |
| --- | --- |
| **宏** | **描述** |
| \_\_DATE\_\_ | 当前日期，一个以 "MMM DD YYYY" 格式表示的字符常量。 |
| \_\_TIME\_\_ | 当前时间，一个以 "HH:MM:SS" 格式表示的字符常量。 |
| \_\_FILE\_\_ | 这会包含当前文件名，一个字符串常量。 |
| \_\_LINE\_\_ | 这会包含当前行号，一个十进制常量。 |
| \_\_STDC\_\_ | 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** 运算符是用在常量表达式中的，用来确定一个标识符是否已经使用 #define 定义过（类比PHP的define和defined方法）。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（就是指定目录）。

#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令牌流：

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（然后使用-D在makefile中定义include规则）：

#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，不管是隐式转换还是显示转换，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 );

}

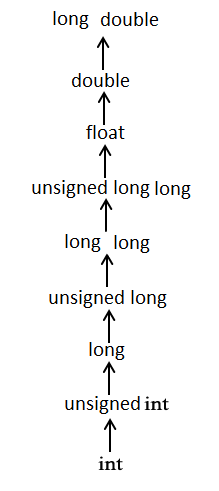
当上面的代码被编译和执行时，它会产生下列结果：

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；这与除法的不同，除法是利用了算术的优先级，先把其中的一个数转换为double在做相除从而得到double的值。

**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：

#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：

#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\n", 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（PHP的参数默认就是可变的）。下面的实例演示了这种函数的定义。

int func(int, ... )

{

.

.

.

}

int main()

{

func(1, 2, 3);

func(1, 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总是 **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，省略号前面的那个参数总是 **int**，表示了参数的个数。
* 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); //该函数根据num初始化valist

/\* 访问所有赋给 valist 的参数 \*/

for (i = 0; i < num; i++)

{

sum += va\_arg(valist, int); //可能是通过指针遍历

}

/\* 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);** 该函数分配一个带有 **num** 个元素的数组，每个元素的大小为 **size** 字节。 |
| 2 | **void free(void \*address);** 该函数释放 address 所指向的h内存块。 |
| 3 | **void \*malloc(int num);** 该函数分配一个 **num** 字节的数组，并把它们进行初始化。 |
| 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需内存大小的字符，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（类似于PHP）。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（类似于PHP），**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""""。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"

Progranm name ./a.out

The argument supplied is testing1 testing2