更多嵌入式 Linux 学习资料，请关注：一口 Linux 回复关键字:1024

个人微信

公众号

目录

第1章 C语言基础

C 语言是计算机编程中最典型最常用的语言，如操作系统、数据库、通信软件和金融、电力等行业软件都是用 C 语言编写。C++语言是 C 语言的扩展，而 JAVA 语言许多语法又与 C 语言类似。所以作者认为，学好计算机请首先学好 C 语言，C 语言是计算机软件行业的地基，学好 C 语言可以对其他计算机语言触类旁通、一通百通，C 语言，编程世界的王者。学好 C 语言，需要理解和掌握指针，理解了指针就一定程度上理解了 C 语言，掌握好指针是晋升 Linux C 高级程序员的必要条件。

1.1 C语言基本概念

计算机语言与人类语言一样，都是一种交流的工具。人类语言是人与人之间交流的工具，计算机语言是人与计算机之间交流的工具。所有语言的都有它的语法、语素和语用；都有它的语法规则，这样才能被交流的双方相互理解；计算机语言也不例外，我们编写的计算机程序也必须遵守一定的语法规则，才能被编译器所识别，最后翻译成能被 CPU 理解执行的机器语言，其中机器语言是 CPU 厂商设计的。

1. 计算机语言相关概念解释

① 编译程序：编译程序又称为编译器，是一个语言翻译程序，它把源语言翻译成目标语言。源语言主要指各种计算机语言，目标语言主要指CPU能识别的机器码。例如英语翻译成汉语的过程中，翻译官相当于编译器。

② 编译：编译源语言，生成目标代码并形成机器码的过程，相当于现实翻译中的笔译。C语言属编译型语言。

③ 解释：边把源语言解释成机器码边执行的过程，相当于现实翻译中的口译。Shell脚本属解释型语言。

④ 编译阶段：编译阶段包括词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成六个阶段，贯穿始终的功能模块有表格管理和出错管理。

⑤ 为什么需要编译程序：就象我们母语是汉语，想与一个埃及人交流一样，但我们不懂埃及语，如果有一个自动语言翻译机（或翻译官），双方的交流才能变得流畅。由于CPU指令是二进制指令，人们难以在此基础上进行有效编程，所以发明了各种各样的计算机高

级语言，然后通过编译器把人们编写的计算机高级语言程序翻译成CPU能理解的二进制指令。

⑥ 程序：通常指的是人们编写的计算机语言源代码和编译后的执行码，其中源代码称为源程序，执行码称为执行程序，程序是静态的。程序相当于一个企业行政部门出台的行政管理文件。

⑦ 进程：程序的一次执行过程，进程是动态的。进程相当于企业各部门拿着行政管理文件的执行过程。

⑧ CPU：CPU主要与内存通信，其主要功能有解释指令、存数据到内存、从内存取数据、数据计算和中断处理，CPU最突出的功能为解释指令和数据计算。CPU的工作是从内存中取出指令并完成指令的自动化执行。

⑨ 内存：内存是存放电脑工作数据的空间，断电后数据丢失。内存的最小单位为字节，内存每一个最小单位空间都有其编号（即内存地址），CPU是通过内存地址访问内存数据。在内存看来，内存里存放的数据都是平等的，都是一串串二进制字符，没有类型，也没有含义，是计算机程序把内存的数据赋予特别的类型和意义。

⑩ 变量：一段内存空间的抽象，变量类型决定了变量存放内存空间的大小。计算机语言编译成机器码后，变量相对于内存的操作就转变为对相应内存地址的存取操作。直接存取是一般变量，间接存取是指针变量。C语言变量原则为先定义，后使用。

⑪ 机器程序：经过编译器编译后形成CPU能理解的机器指令。在机器程序中，只有内存地址，没有变量概念；机器程序中所有变量失去意义，变量的操作都会转换成对内存地址的存取操作。机器程序有如下几种类型的操作：CPU的计算操作、内存空间的申请与释放、CPU把寄存器数据存到内存、取内存数据到CPU寄存器、内存数据从一片空间复制到另一片空间、中断操作等。

⑫ Linux系统编译方法：gcc(或cc) a.c(源代码名称) -o a(执行码)。在这里a.c和a都是程序，其中a.c为源程序，a为可执行程序。在界面上输入 ./a，a就开始了执行之旅，其执行过程称为进程。

2. C语言数据类型种类

图 4-1 画出了 C 语言数据类型种类，在 C 语言中只允许使用下面这些数据类型。

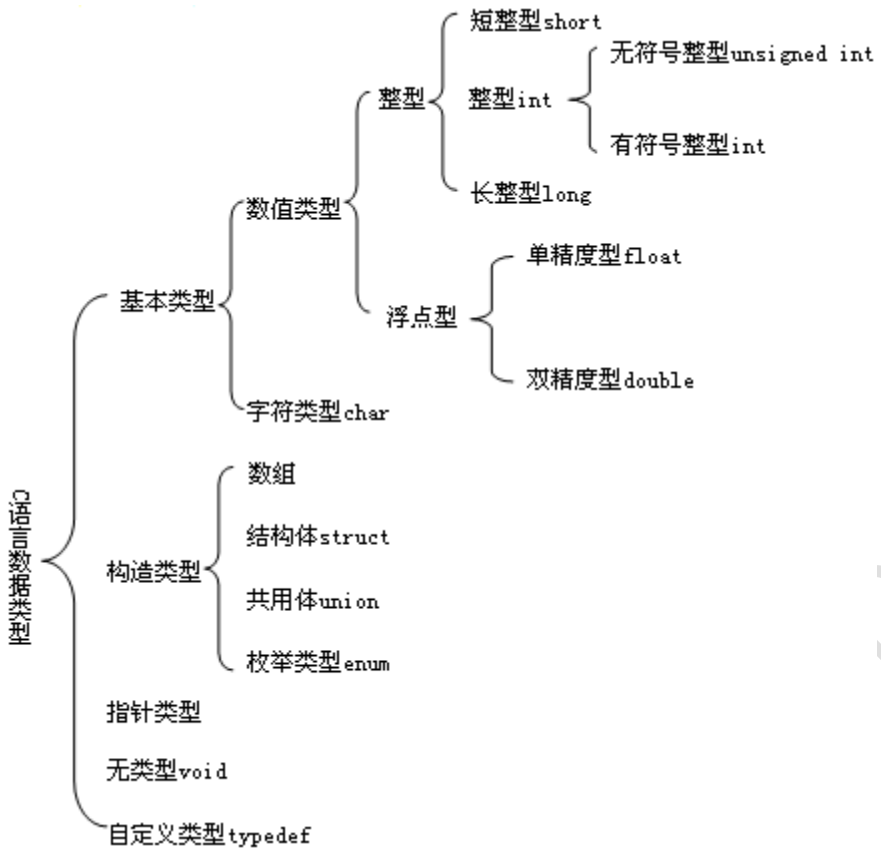


图 4-1　C 语言数据类型种类

3. 32个关键字

C 语言有如下 32 个关键字，这些关键字由系统定义，不能重新作为其他定义。

auto break case char const continue default do double else enum extern float for goto int long register return short signed sizeof static struct switch typedef unsigned union void volatile while

4. 9种控制语句

C 语言控制语句的功能是完成程序流程的控制，C 语言有如下 9 种控制语句。

① if( )-else(条件语句)
② for( )（循环语句）
③ while( )（循环语句）
④ do-while( )（循环语句）
⑤ continue（结束本次循环开始下一次循环语句）
⑥ break（跳出switch或循环语句）
⑦ switch（多分支选择语句）
⑧ goto（跳转语句）
⑨ return（从函数返回语句）

5. C程序格式和结构特点

C 语言习惯用小写字母，大小写敏感；不使用行号，无程序行概念；可使用空行和空格；常用锯齿形书写格式。

下面以一个实例说明 C 语言结构特点。

```
/* example1.1    The   first   C   Program*/    ←——— 注释
#include <stdio.h>    ←——— 编译预处理
main()    ←——— 函数
{
        printf("Hello,World!");          ←语句
}
```

C 语言由函数、语句和注释三个部分组成，这三部分的说明和要求如下。

（1）函数与主函数

一个 C 语言源程序可以由一个或多个源文件组成。每个源文件可由一个或多个函数组成。一个源程序不论由多少个文件组成，都有一个且只能有一个 main 函数，即主函数。

源程序中可以有预处理命令(include 命令仅为其中的一种)，预处理命令通常应放在源文件头。

每一个说明，每一个语句都必须以分号结尾。但预处理命令，函数头和花括号"}"之后不能加分号。

标识符、关键字之间必须至少加一个空格以示间隔，若已有明显的间隔符，也可不再加空格来间隔。

（2）程序语句

C 程序由语句组成，用";"作为语句终止符。

{}表示一个语句的整体。if、for、while 包含多条语句时需要用{}括起来表示一个整体，单条语句则可直接用";"表示语句终止。

（3）注释

/* */为注释，不能嵌套，注释不产生编译代码。

6. C语言程序的开发过程

编辑 ——→ 编译（预编译、编译、链接） ——→ 形成执行码 ——→ 执行 ——→

7. 书写C语言程序时应遵循的规则

一个说明或一个语句应该占一行。

用{}括起来的部分，通常表示了程序的某一层次结构。{}一般与该结构语句的第一个字母对齐，并单独占一行。

使用缩进方式编程，低一层次的语句比高一层次的语句缩进若干空格后书写，以便看起来更加清晰，增加程序的可读性。

有足够的注释。有合适的空行。

8. C语言的字符集

字符是组成语言的最基本的元素。C 语言字符集由字母、数字、空格、标点和特殊字符组成。在字符串和注释中还可以使用汉字或其他可表示的图形符号。

（1）字母

小写字母 a ~ z 共 26 个，大写字母 A ~ Z 共 26 个。

（2）数字

0 ~ 9 共 10 个。

（3）空白符

空格符、制表符、换行符等统称为空白符，空白符只在字符常量和字符串中起作用。在其他地方出现时，只起间隔作用，编译程序对它们忽略不计。在程序中适当的地方使用空白符将

增加程序的清晰性和可读性。

（4）标点和特殊字符

包括“,”、“;”等标点和“[”、“]”、“*”等特殊字符。

9. C语言词汇

在 C 语言中使用的词汇分为六类，分别为标识符、关键字、运算符、分隔符、常量和注释符，具体说明如下。

（1）标识符

在程序中使用的常量名、变量名、函数名等统称为标识符。除库函数的函数名由系统定义外，其余都由用户自定义。C 规定，标识符只能是字母(A ~ Z，a ~ z)、数字(0 ~ 9)、下画线(_)组成的字符串，并且其第一个字符必须是字母或下画线。

a、x、x3、BOOK_1、sum5 这 5 个标识符是合法的标识符。

以下标识符是非法的:

3s　　以数字开头。

s*T　　出现非法字符*。

（2）关键字

关键字是由 C 语言规定的具有特定意义的字符串，通常也称为保留字，用户定义的标识符不应与关键字相同。C 语言的关键字分为以下几类:

① 类型说明符

用于定义、说明变量、函数或其他数据结构的类型，如 int、double 等。

② 语句定义符

用于表示一个语句的功能，如 if~else 就是条件语句的语句定义符。

③ 预处理命令字

用于表示一个预处理命令，如 include。

（3）运算符

C 语言中含有相当丰富的运算符。运算符与变量, 函数一起组成表达式, 表示各种运算功能。运算符由一个或多个字符组成。

（4）分隔符

在 C 语言中采用的分隔符有逗号和空格两种。逗号主要用在类型说明和函数多个形参中分隔各个变量，空格多用于语句各单词之间作间隔符。

（5）常量

C 语言中使用的常量可分为数字常量、字符常量、字符串常量、符号常量、转义字符等多种。

（6）注释符

C 语言的注释符是以“/*”开头并以“*/”结尾的串，在“/*”和“*/”之间的即为注释。程序编译时，不对注释作任何处理，注释可出现在程序中的任何位置，注释是用来向用户提示或解释程序的意义。

10. C语言算法

C 语言是结构化设计语言，结构化程序的灵魂是算法。瑞士 Niklaus Wirth 教授在 20 世纪 60 年代提出了“数据结构+算法=程序”的公式。做任何事情都有一定的步骤，为解决一个问题而采取的方法和步骤，就称为算法。算法具有有穷性、确定性、有零个或多个输入、有一个或多个输出、有效性等特征。C 语言算法是通过结构化设计方法来实现，其特点为自顶向下、逐步细化、模块化设计和结构化编码。

11. C语言编译过程

图 4-2 画出了 C 语言的编译流程。

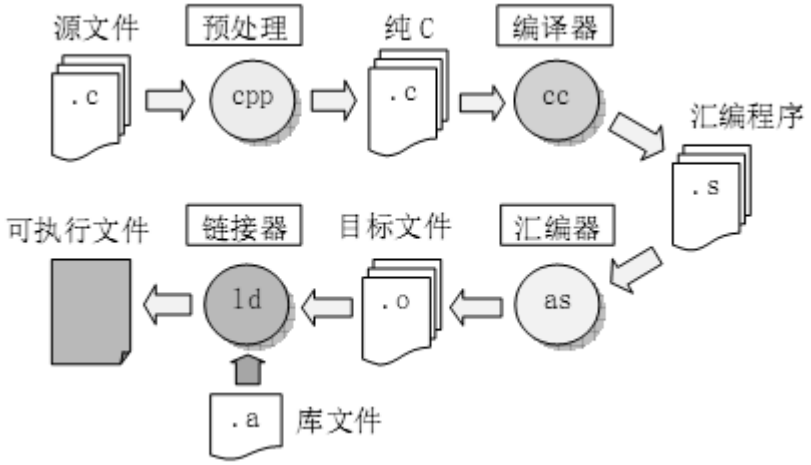


图 4-2 C 语言编译过程

12. 计算机语言说明

数学是描述大自然的语言，会计是人们记录商业活动的语言，人类语言（英语、汉语、法语、西班牙语等）则是人与人之间交流的语言，计算机语言是人们和计算机（CPU）交流的语言。由于 CPU 只能理解机器语言（CPU 指令集），而人们对机器语言（一串二进制数）很难阅读和调式，于是就产生了汇编语言。汇编语言的特点是利用助记符代替相应的机器语言二进制指令，如利用“ADD”助记符代替二进制指令“1000001100000110”，汇编语言较机器语言更有利于人们编程，人们编写的汇编源程序再经汇编编译器翻译成机器语言程序（执行码），执行码就是 CPU 能理解的二进制指令。

由于汇编语言是面向机器的低级语言，与 CPU 的指令集息息相关，是按 CPU 的“思考”方式编写而成的计算机语言。汇编语言与人类思考方法相差较大，于是产生了面向过程的语言（PASCLE、C 等）和面向对象的语言（C++、JAVA 等）。但是计算机只能理解机器语言（CPU 指令集），人们编写的 C 语言、C++语言、JAVA 语言源程序要翻译成机器语言 CPU 才能够执行，执行翻译工作的是编译程序。就这样是一门新的计算机学科（编译原理）产生了，编译原理介绍了编译程序的实现方法。masm 是汇编的编译程序，gcc 是 Linux 下 C 语言和 C++语言的编译程序，JVC 是 JAVA 的编译程序。

万物皆流，万物归宗。计算机源程序，无论是高级语言 C、C++、JAVA 源程序，还是汇编语言源程序，最终都要翻译成机器语言，CPU 才能够执行。

13. CPU的独白

嗨，大家好！我叫 CPU，是中央处理器（Central Processing Unit）的简称，是电子计算机的主要设备之一，其功能主要是解释计算机指令以及处理计算机软件中的数据，所谓的计算机的可编程性主要是指对 CPU 的编程。

在我看来，外部一切都是地址。我只负责从地址上取数据，然后计算数据，计算完毕向地址上存数据。我的工作主要是围绕着存数据、取数据和计算数据在进行。由于外界的地址只有内存和外设两种，CPU 引脚 M/$\overline{IO}$ 高电平意味着与内存通信，低电平意味着与外设端口通信。地址以一个字节（8 位二进制数）为单位进行编排，微机中内存地址和外设端口地址是单独进行编排的，当我看到 MOV 指令就知道是与内存通信，看到 IN 和 OUT 指令就知

道是与外设端口通信。

在我的眼里，地址上（内存和外设端口）的数据都是一堆二进制数，没有类型，也没有任何含义。只有当我开始运行机器语言代码时，这些数据才变得有意义。我是一个优秀且卓越的执行者，完全按照机器语言代码功能进行执行。

人们通过布尔逻辑、数字电路技术把沙粒（二氧化硅）变成了我，我的世界只有 100 多个或几百多个机器指令，这些机器指令早已由 intel 等芯片厂商设计好。设计完成后，机器指令是固定的，不能再修改。因为我是硬件，不同于软件，设计生产完成后，要么是成功的芯片，要么是失败的芯片，没有第三条道路可走。

在我的身体里，利用寄存器暂存数据，利用运算器完成数据运算，利用控制器控制数据与内存或外设的传输。由于我工作的关系，我深刻理解什么叫作分层，什么叫作抽象，什么叫作协议，什么叫作分工，什么叫作转换，什么叫作约定，什么叫作专业，什么叫透明，什么叫作物理，什么叫作逻辑等等。其实计算机技术实现是哲学思想的体现，计算机技术较好的利用分层、抽象、模块化等思想使复杂的问题简单化。

人们常说“谢谢你的存在，世界因你而精彩”。然而我的世界是枯燥的，就是不断的正确执行这一百多个机器指令。也许专业就是简单的事情重复的做，我太过专业，人们赋予我计算专家头衔。

复杂的事情模块化，模块的事情简单化，简单的事情流程化，流程的事情自动化。计算机完成的功能的确非常复杂，要做好这项工作必须懂得分工和协作，每个人只做自己擅长的事情。所以我和我的伙伴们有明确的分工，由于责任明确，大家都完全遵守着各种协议和约定，所以我们配合的很默契，工作得很高效。我和我的同伴们是世界上最好的学生，从来规规矩矩，完全遵守各种协议和约定。内存完成工作数据的存储，硬盘完成长久数据的保存，键盘完成字符的输入，鼠标完成图形按钮的控制，显卡完成显示数据的转换，显示器完成图形显示，声卡完成数字声音向模拟声音的转换，音箱完成声音的播放。我和我的伙伴们总是呆在固定的地点，各自完成自己的工作，大家“鸡犬之声相闻，老死不相往来”，大家通过总线彼此传递着数据。

许多人每天的工作和生活与我们联系在一起，人们与我们电脑从陌生到熟悉；然而许多人觉得我们太有内涵，难以弄懂我们复杂的工作原理和工作个性，又觉得从熟悉走向陌生。其实计算机的世界是按照一系列的协议和规则在运行，就像大自然按照一系列规则在运行一样。大自然是造物主创造的规则，而计算机则是全世界计算机专家们创造的协议与规则。了解计算机首先要了解各个硬件功能以及硬件之间如何分工协作，然后再要知道软件的工作原理。软件是建立在硬件之上，就像精神建立物质之上一样。

许多人把我们电脑当作自我精神愉悦的朋友。许多人说想每天“听听你悦耳的声音，想看看你迷人的笑容，看见你我有一种莫名的快乐。”然而我的声音（声音格式）有各种协议，我的笑容（图像显示）也有各种协议，我的声音和笑容凝聚着全球 IT 工程师辛勤的汗水和智慧的结晶，能为大家带来快乐我也感到很快乐。我们电脑被人们制造出来后，然后我们改变了世界人们工作与生活；有时我们分不清是世界改变了我们，还是我们改变了世界，也许这是相互促进的结果。也许电脑和人脑有相通之处，人脑的思维结构是由天生注定的，人可以通过学习和思考来优化思维软件。电脑的物理结构是硬件决定的了，而软件则是由 IT 工程师编写出来，软件通过硬件来实现电脑价值的提升。

由于我 CPU 是一板一眼，只认识数字世界，人们觉得我 CPU 很难沟通。大家觉得我 CPU 是冰冷的芯下面藏着热情的火焰，在 IT 发展初期，只有能编写计算机机器指令的人才才能点燃 CPU 芯中的火焰。由于机器指令晦涩难懂，这不适应电脑的发展与普及，于是编译器产生了，编译器是我 CPU 与计算机语言之间的翻译官。计算机语言是面向人类思维的语言，而我 CPU 只认识机器指令，所以编译器架起了我们沟通的桥梁。人们许多工作都是

围绕效率、成本、功用三者的平衡，编译器让人们编写的软件更加高效、成本更低、功能更强，编译器推动了软件业的繁荣。不管人们用多少种软件语言，编写软件复杂程度有多高，但最终都要翻译成我能理解为数不多的上百个或几百个机器指令。我用这数量极少的指令展现了色彩缤纷的世界，千变万化的声音。这一切的结果是基于一定的规则和协议，科学就是认识万事万物的规则和规律，世界上只有规则和规律的事情才有意义，美妙的声音是有规律的，而噪音是无规律的。我深深懂得规则和规律的重要，我的一切工作都是按照预定的规则和规律进行的。只有符合规则、约定、协议的数据我和我的同伴们才能相互理解，理解需要建立在协议、约定和规则的基础上。

在电脑世界里，标准的力量常常是无穷的，计算机业标准比任何其他行业都使用得广泛，顺标准则昌，逆标准则亡。由于全世界电脑各种协议需要统一的标准，才能更好把世界软硬件厂商联系起来推动 IT 产业的进步,所以全世界许多 IT 公司都在认识标准、利用标准、制定标准、优化标准中博弈，如高清 DVD 标准之争。

计算机的世界，是数字（数据）的世界。计算机软硬件所做的一切只不过是为了数据的加工、转换、表现（显示、声音）、传输和存储。围绕数据在计算机中的处理产生了许多计算机学科，如数据结构是对数据的算法处理，数据库是对数据的关系处理，计算机网络是对数据的传输处理，而计算机存储是对数据的存储处理，计算机图形学是对数据的转换和显示处理。数据的加工、转换、表现、传输和存储都需要遵照一定的协议和规范。

1.2 常量与变量

C 语言中数据由常量和变量组成，常量顾名思义是其值不可变量的量，变量顾名思义是其值可以改变的量。C 语言变量都有其数据类型，说明其在 C 语言中的类型。数据类型决定了数据占内存的字节数，数据的取值范围和其上可进行的操作。

1. 基本类型的分类及特点

表 4-1 列出了 C 语言基本类型及其说明，使用时要注意这些基本类型的数值范围。

表 4-1 C 语言基本类型分类表

	类型说明符	字节	数值范围
字符型	char	1	C 字符集
基本整型	int	2	-32768 ~ 32767
短整型	short int	2	-32768 ~ 32767
长整型	long int	4	-214783648 ~ 214783647
无符号型	unsigned	2	0 ~ 65535
无符号长整型	unsigned long	4	0 ~ 4294967295
单精度实型	float	4	3/4E-38 ~ 3/4E+38
双精度实型	double	8	1/7E-308 ~ 1/7E+308

2. 标识符

标识符是用来标识变量名、符号常量名、函数名、数组名、类型名、文件名的有效字符序列。标识符组成只能由字母、数字、下画线组成，且第一个字母必须是字母或下画线；大小写敏感；不能使用关键字；长度最长 32 个字符；命名原则要见名知意，做到“顾名思义”。

3. 常量和符号常量

在程序执行过程中，其值不发生改变的量称为常量。常量有直接常量、符号常量两种。

① 直接常量

直接变量有整型常量、实型常量、字符常量、字符串常量、指针常量五种类型。下面是它们的举例说明。

整型常量：12、0、-3。

实型常量：4.6、-1.23。

字符常量：'a'、'b'。

② 符号常量：用标识符代表一个常量。

在 C 语言中，可以用一个标识符来表示一个常量，称之为符号常量。

符号常量在使用之前必须先定义，其一般形式为：

```
#define 标识符 常量
```

其中#define 是一条预处理命令（预处理命令都以“#”开头），称为宏定义命令，其功能是把该标识符定义为其后的常量值。

习惯上符号常量的标识符用大写字母，变量标识符用小写字母，以示区别。

【例 4-1】符号常量的使用

price.c 源代码如下：

```
#include <stdio.h>
#define PRICE 30
void main()
{
    int num,total;
    num=10;
    total=num* PRICE;
    printf("total=%d\n", total);
}
```

由上面实例看出，符号常量有以下特点和好处：

① 符号常量是用一个标识符代表一个常量。

② 符号常量与变量不同，它的值在其作用域内不能改变，也不能再被赋值。

③ 使用符号常量的好处是含义清楚，能做到“一改全改”。

4. 变量

（1）变量说明

其值可以改变的量称为变量，变量在内存中占据一定的存储单元。变量定义必须放在变量使用之前，一般放在函数体的开头部分，不能使用没有定义的变量。每个变量都有一个名字，称为变量名，其值存放在内存中，变量名是变量值的代号。

假设 int a=0，图 4-3 画出了此变量的内存布局。此变量变量名为 a，其值为 0，程序执行时分配内存空间的首地址为 2012，即&a 等于 2012。

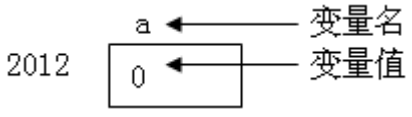


图 4-3 变量的内存布局

变量可以理解为内存相应位置所存数据的抽象，变量类型决定了变量所占空间的大小。字符型变量占用内存空间 1 个字节，整型变量占用内存空间 4 个字节。上述变量 a 可以理解地址为 2012~2009 的 4 个字节内存空间的抽象，其存储的值为 0。一个 C 语言源程序编译成可执行程序后，对变量 a 的操作实际转换为对内存地址 2012~2009 的存取操作。

（2）变量初始化

初始化是变量定义的同时给变量赋以初值的方法。

变量定义中赋初值的一般形式如下：

```
类型说明符 变量 1= 值 1,变量 2= 值 2,……;
例如：
    int a=3;
    int b,c=5;
```

（3）变量转换

变量转换分为显示转换和隐式转换两种。

其中隐式转换分为运算转换、赋值转换、输出转换和函数调用转换。隐式转换时变量的转化顺序为 char,short→int→unsigned→long→float→double。

显式转换的一般形式为：(类型名)(表达式)，例(float)a 是把 a 转换为实型。

5. 整型数据

整型数据有前缀表示法和后缀表示法两种，具体说明如下。

① 前缀表示法。以 0 开头表示八进制，以 0x(或 0X)表示十六进制，默认表示十进制。015(十进制为 13)，0X2A(十进制为 42)。

② 后缀表示法。L(或l)表示长整型，U(或u)表示无符号数。

整型变量赋值要注重其取值范围，超过范围会造成变量的溢出。

6. 转义字符

转义字符是一种特殊的字符常量。转义字符以反斜线“\”开头，后跟一个或几个字符。转义字符具有特定的含义，不同于字符原有的意义，故称“转义”字符。例如，printf 函数格式串中用到的“\n”就是一个转义字符，其意义是“回车换行”。转义字符主要用来表示用一般字符不便于表示的控制代码。表 4-2 列出了常用的转义字符及其含义。

表 4-2 常用的转义字符及其含义

转义字符	转义字符的意义	ASCII 代码
\n	回车换行	10
\t	横向跳到下一制表位置	9
\b	退格	8
\r	回车	13
\f	走纸换页	12
\\	反斜线符"\"	92
\'	单引号符	39
\"	双引号符	34
\a	鸣铃	7
\ddd	1～3 位八进制数所代表的字符	
\xhh	1～2 位十六进制数所代表的字符	

广义地讲，C 语言字符集中的任何一个字符均可用转义字符来表示。表中的\ddd 和\xhh 正是为此而提出的，ddd 和 hh 分别为八进制和十六进制的 ASCII 代码。如\101 表示字母 A，\102 表示字母 B，\x0A 表示换行等。

7. 字符常量

字符常量是用单引号引起来的一个字符。

例如：'a'、'b'、'='、'+'、'?'都是合法字符常量。

在 C 语言中，字符常量有以下特点：

① 字符常量只能用单引号引起来，不能用双引号或其他括号。
② 字符常量只能是单个字符，不能是字符串。
③ 字符可以是字符集中任意字符。但数字被定义为字符型之后就不能参与数值运算。如'5'和 5 是不同的，'5'表示字符 5，对应ASCII码 35，而 5 对应ASCII码 5。

8. 字符变量

字符变量用来存储字符常量，即单个字符，字符变量的类型说明符是 char。

每个字符变量被分配一个字节的内存空间，字符值是以 ASCII 码的形式存放在变量的内存单元之中的。

C 语言允许对整型变量赋以字符值，也允许对字符变量赋以整型值。在输出时，允许把字符变量按整型量输出，也允许把整型量按字符量输出。

短整型量为二字节量，字符量为单字节量，当整型量按字符型量处理时，只有低八位字节参与处理。

如 x 的十进制 ASCII 码是 120，对字符变量 a 赋予'x'可以是 a='x'或 a=120，这两条语句是等价的。

字符变量赋值还可以使用转义字符，如 c='\0'与 c=0 这两条语句也是等价的。

【例 4-2】字符变量使用举例

charint.c 源代码如下：

```
#include <stdio.h>
void main()
{
    char a,b;
    a=120;
    b='y';
    printf("%c,%c\n",a,b);
    printf("%d,%d\n",a,b);
}
```

编译 gcc charint.c -o charint。

执行 ./charint，执行结果如下：

```
x,y
120,121
```

9. 字符串常量

字符串常量是由一对双引号引起来的字符序列。例如"CHINA"、"C program"、"$12.5" 等都是合法的字符串常量。

字符串常量和字符常量是不同的量。它们之间主要有以下区别：

① 字符常量由单引号引起来，字符串常量由双引号引起来。
② 字符常量只能是单个字符，字符串常量则可以含一个或多个字符。
③ 字符常量占一个字节的内存空间。字符串常量占的内存字节数等于字符串中字节数加 1。增加的一个字节中存放字符“\0”(ASCII 码为 0)，这是字符串结束的标志。

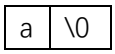
例如：字符串"C program" 在内存中所占的字节为：

C		p	r	O	g	r	a	m	\0

字符常量'a'和字符串常量"a"虽然都只有一个字符，但在内存中的情况是不同的，两者占用内存情况说明如下。

'a'在内存中占一个字节，可表示为:

"a"在内存中占二个字节，可表示为:



1.3 运算符

C 语言中运算符和表达式数量之多，在高级语言中是少见的。正是丰富的运算符和表达式使 C 语言功能十分完善，这也是 C 语言的主要特点之一。
C 言的运算符不仅具有不同的优先级，而且还有一个特点，就是它的结合性,使用时要注意运算符是右结合性还是左结合性。

1. C语言的运算符分类

对 C 语言的运算符分类说明如下:

① 算术运算符：用于各类数值运算。包括加(+)、减(-)、乘(*)、除(/)、求余(或称模运算%)、自增(++)、自减(--)共七种。
② 关系运算符：用于比较运算。包括大于(>)、小于(<)、等于(==)、大于等于(>=)、小于等于(<=)和不等于(!=)六种。
③ 逻辑运算符：用于逻辑运算。包括与(&&)、或(||)、非(!)三种。
④ 位操作运算符：参与运算的量，按二进制位进行运算。包括位与(&)、位或(|)、位非(~)、位异或(^)、左移(<<)、右移(>>)六种。
⑤ 赋值运算符：用于赋值运算，分为简单赋值(=)、复合算术赋值(+=、-=、*=、/=、%=)和复合位运算赋值(&=、|=、^=、>>=、<<=)三类共十一种。
⑥ 条件运算符：这是一个三目运算符，用于条件求值(?:)。
⑦ 逗号运算符：用于把若干表达式组合成一个表达式(，)。
⑧ 指针运算符：用于取内容(*)和取地址(&)二种运算。
⑨ 求字节数运算符：用于计算数据类型所占的字节数(sizeof)。
⑩ 特殊运算符：有优先级()、下标[]、结构成员(→、.)、取负运算符-、强制类型转换(类型)。

2. 基本的算术运算符

对基本的算术运算符说明如下:

① 加法运算符“+”:加法运算符为双目运算符,即应有两个量参与加法运算。如a+b、4+8 等,加法运算符具有左结合性。
② 减法运算符“-”：减法运算符为双目运算符。但“-”也可作负值运算符，此时为单目运算,如-x、-5 等具有左结合性。
③ 乘法运算符“*”：双目运算符，具有左结合性。
④ 除法运算符“/”：双目运算具有左结合性。参与运算量均为整型时，结果也为整型，舍去小数。如果运算量中有一个是实型，则结果为双精度实型。
⑤ 求余运算符(模运算符)“%”：双目运算符，具有左结合性。要求参与运算的量均为整型，求余运算的结果等于两数相除后的余数。

算术单目运算符为右结合性，双目运算符为左结合性，两整数相除，结果为整数，%要求两侧均为整型数据。

3. 表达式、运算符的优先级和结合性

表达式是由常量、变量、函数和运算符组合起来的式子。一个表达式有一个值及其类型,

它们等于计算表达式所得结果的值和类型。表达式求值按运算符的优先级和结合性规定的顺序进行。单个的常量、变量、函数可以看作是表达式的特例。

算术表达式是由算术运算符和括号连接起来的式子。下面是对算术表达式和算术运算符的详细说明。

算术表达式：用算术运算符和括号将运算对象（也称操作数）连接起来的，符合 C 语法规则的式子。

运算符的结合性：C 语言中各运算符的结合性分为两种，即左结合性(自左至右)和右结合性(自右至左)。例如算术运算符的结合性是自左至右，即先左后右。如有表达式 x-y+z 则 y 应先与“-”号结合，执行 x-y 运算，然后再执行+z 的运算。这种自左至右的结合方向就称为“左结合性”。而自右至左的结合方向称为“右结合性”。 最典型的右结合性运算符是赋值运算符。如 x=y=z，由于“=”的右结合性，应先执行 y=z 再执行 x=(y=z)运算。C 语言运算符中有不少为右结合性，应注意区别，以避免理解错误。

运算符的优先级：C 语言中，运算符的运算优先级共分为 15 级。1 级最高，15 级最低。在表达式中，优先级较高的先于优先级较低的进行运算。而在一个运算量两侧的运算符优先级相同时，则按运算符的结合性所规定的结合方向处理。

表 4-3 列出了 C 语言运算符优先级及其说明，此表了解即可，无需记忆，编程时优先级的确定时可使用圆括号“()”来解决。

表 4-3 C 语言运算符优先级表

优先级	运算符	名称或含义	使用形式	结合方向	说明
1	()	圆括号	（表达式）/函数名(形参表)	左到右	
	[]	数组下标	数组名[常量表达式]		
	.	成员选择（对象）	对象.成员名		
	->	成员选择（指针）	对象指针->成员名		
2	-	负号运算符	-表达式	右到左	单目运算符
	(类型)	强制类型转换	(数据类型)表达式		
	++	自增运算符	++变量名/变量名++		单目运算符
	--	自减运算符	--变量名/变量名--		单目运算符
	*	取值运算符	*指针变量		单目运算符
	&	取地址运算符	&变量名		单目运算符
	!	逻辑非运算符	!表达式		单目运算符
	~	按位取反运算符	~表达式		单目运算符
	sizeof	长度运算符	sizeof(表达式)		
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式		双目运算符

	%	余数（取模）	整型表达式/整型表达式		双目运算符
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式		双目运算符
5	<<	左移	变量<<表达式	左到右	双目运算符
	>>	右移	变量>>表达式		双目运算符
6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式		双目运算符
	<	小于	表达式<表达式		双目运算符
	<=	小于等于	表达式<=表达式		双目运算符
7	==	等于	表达式==表达式	左到右	双目运算符
	!=	不等于	表达式!= 表达式		双目运算符
8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10	|	按位或	表达式|表达式	左到右	双目运算符
11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12	||	逻辑或	表达式||表达式	左到右	双目运算符
13	?:	条件运算符	表达式 1? 表达式 2: 表达式 3	右到左	三目运算符
14	=	赋值运算符	变量=表达式	右到左	
	/=	除后赋值	变量/=表达式		
	*=	乘后赋值	变量*=表达式		
	%=	取模后赋值	变量%=表达式		
	+=	加后赋值	变量+=表达式		
	-=	减后赋值	变量-=表达式		
	<<=	左移后赋值	变量<<=表达式		
	>>=	右移后赋值	变量>>=表达式		
	&=	按位与后赋值	变量&=表达式		
	^=	按位异或后赋值	变量^=表达式		
	|=	按位或后赋值	变量|=表达式		
15	,	逗号运算符	表达式,表达式,…	左到右	从左向右顺序运算

强制类型转换运算符

强制类型转换运算符的形式为:

```
(类型说明符) (表达式)
```

其功能是把表达式的运算结果强制转换成类型说明符所表示的类型。

例如:

```
(float)a          把 a 转换为实型
(int)(x+y)        把 x+y 的结果转换为整型
```

自增、自减运算符

自增 1: 自增 1 运算符记为“++”，其功能是使变量的值自增 1。

自减 1: 运算符记为“--”，其功能是使变量值自减 1。

自增 1，自减 1 运算符均为单目运算，都具有右结合性。可有以下几种形式:

```
++i     i 自增 1 后再参与其他运算
--i     i 自减 1 后再参与其他运算
i++     i 参与运算后，i 的值再自增 1
i--     i 参与运算后，i 的值再自减 1
```

【例 4-3】自增自减运算符举例

varadd.c 源代码如下:

```
#include <stdio.h>
void main()
{
    int i=8;
    int a ;
    printf("i=%d\n",++i);
    printf("i=%d\n",i);
    a=++i ;
    printf("a=%d, i=%d\n", a, i);
    a=i++ ;
    printf("a=%d, i=%d\n", a, i);
    a=--i ;
    printf("a=%d, i=%d\n", a, i);
    a=i-- ;
    printf("a=%d, i=%d\n", a, i);
    printf("i=%d\n",--i);
    printf("i=%d\n",i++);
    printf("i=%d\n",i--);
    printf("i=%d\n",-i++);
    printf("i=%d\n",-i--);
}
```

编译 gcc varadd.c -o varadd。

执行 ./varadd，执行结果如下:

```
i=9
i=9
a=10, i=10
a=10, i=11
```

```
a=10, i=10
a=10, i=9
i=8
i=8
i=9
i=-8
i=-9
```

4. 赋值运算符和赋值表达式

（1）赋值运算符

简单赋值运算符号为“=”，由“=”连接的式子称为赋值表达式。其一般形式如下：

```
变量=表达式
```

例如： x=a+b;

a=b=c=5 可理解为 a=(b=(c=5))。

（2）类型转换

如果赋值运算符两边的数据类型不相同，系统将自动进行类型转换，即把赋值号右边的类型换成左边的类型。具体规定如下：

① 实型赋予整型，舍去小数部分。

② 整型赋予实型，数值不变，但将以浮点形式存放，即增加小数部分(小数部分的值为 0)。

③ 字符型赋予整型，由于字符型为一个字节，而整型为二个字节，故将字符的 ASCII 码值放到整型量的低八位中，高八位为 0。整型赋予字符型，只把低八位赋予字符量。

（3）复合的赋值运算符

在赋值符“=”之前加上其他二目运算符可构成复合赋值运算符，如+=、-=、*=、/=、%=、<<=、>>=、&=、^=、|=都是复合赋值运算符。

构成复合赋值表达式的一般形式为：

```
变量  双目运算符=表达式
```

它等效于

```
变量=变量 运算符 表达式
```

例如：

a+=5 等价于 a=a+5;

x*=y+7 等价于 x=x*(y+7);

复合赋值符这种写法，对初学者可能不习惯，但十分有利于编译处理，能提高编译效率并产生质量较高的目标代码。

5. 逗号运算符和逗号表达式

在 C 语言中逗号“，”也是一种运算符，称为逗号运算符。 其功能是把多个表达式连接起来组成一个表达式， 称为逗号表达式。

其一般形式为：

```
表达式 1，表达式 2，…表达式 n
```

其求值过程是依次求表达式的值，并以表达式 n 的值作为整个逗号表达式的值。

6. 关系运算符

关系表达式的语法形式如下：

```
表达式 关系运算符  表达式
```

表 4-4 列出了 C 语言六种关系运算符。在 C 语言中，=表示赋值，==表示数学中的相等关系，使用时请注意=和==的区别。

表 4-4 关系运算符表

运算符	含义
==	等于
!=	不等于
>	大于
<	小于
>=	大于或等于
<=	小于或等于

对于关系表达式，如果表达式所表示的比较关系成立则值为真(True)，否则为假(False)，在 C 语言中分别用 int 型的 1 和 0 分别表示真和假。如果变量 x 的值是-1，那么 x>0 这个表达式的值为 0，x>-2 这个表达式的值为 1。

关系运算符的两个操作数应该是相同类型的，两边都是整型或者都是浮点型可以做比较，但两个字符串不能做比较，需要用字符串函数进行比较。

7. 逻辑运算符

C 语言中提供了三种逻辑运算符：&&是与运算（AND）符，||是或运算(OR)符，!是非运算(NOT)符。

逻辑表达式的语法形式如下：

```
表达式  逻辑运算符  表达式
```

（1）逻辑运算符优先级

逻辑运算符优先级顺序为：!（非）>&&(与)>||(或)。

图 4-4 画出了运算符优先级图，其中赋值运算符优先级最低，!优先级最高。

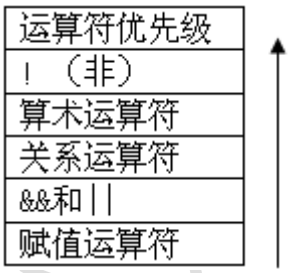


图 4-4：运算符优先级图

按照运算符的优先级顺序可以得出：

```
a>b && c>d      等价于    (a>b)&&(c>d)
!b==c||d<a      等价于    ((!b)==c)||(d<a)
a+b>c&&x+y<b    等价于    ((a+b)>c)&&((x+y)<b)
```

（2）对逻辑运算的值说明

逻辑运算的值为“真”和“假”两种，分别用“1”和“0”来表示。

与运算&&：参与运算的两个量都为真时，结果才为真，否则为假。例：5>0 && 4>2 为真，即为 1。

或运算||: 参与运算的两个量只要有一个为真，结果就为真。两个量都为假时，结果为假。例：5>0||5>8 为真。

非运算!：参与运算量为真时，结果为假；参与运算量为假时，结果为真。例：!(5>0)为

假。

8. 条件运算符和条件表达式

如果在条件语句中，只执行单个的赋值语句时，常可使用条件表达式来实现。这样不但使程序简洁，也提高了运行效率。

条件运算符为?和:，它是一个三目运算符，即有三个参与运算的量。

由条件运算符组成条件表达式的语法形式如下:

```
表达式 1?  表达式 2:  表达式 3
```

其求值规则为：如果表达式 1 的值为真，则以表达式 2 的值作为条件表达式的值，否则以表达式 2 的值作为整个条件表达式的值。条件表达式通常用于赋值语句之中。

例如下面条件语句

```
if(a>b)   max=a;
     else max=b;
```

可用条件表达式写为

```
max=(a>b)?a:b;
```

该语句的语义为如 a>b 为真，则把 a 赋予 max，否则把 b 赋予 max。

使用条件表达式时，还应注意以下几点：

① 条件运算符的运算优先级低于关系运算符和算术运算符，但高于赋值符。

因此

```
max=(a>b)?a:b
```

可以去掉括号而写为

```
max=a>b?a:b
```

② 条件运算符?和:是一对运算符，不能分开单独使用。

③ 条件运算符的结合方向是自右至左。

```
例如：a>b?a:c>d?c:d  应理解为  a>b?a:(c>d?c:d)
```

【例 4-4】条件运算符使用举例

expr.c 源代码如下:

```
#include <stdio.h>
void main()
{
    int a,b,max;
    printf("\n input two numbers:    ");
    scanf("%d%d",&a,&b);
    printf("max=%d",a>b?a:b);
}
```

编译 gcc expr.c -o expr。

执行 ./expr，执行结果如下:

```
input two numbers:    30 90
max=90
```

1.4 C语言控制结构

C 语言有三种基本流程结构，即顺序、选择、循环。选择结构通过 if-else、case-switch、goto 语句实现，循环结构通过 while、for、do-while 语句实现。

1.4.1 if 语句

1. if语句三种形式

if 语句用于分支条件判断，其语法形式有如下三种。

（1）第一种形式为基本形式，只有 if 关键字

if 语句基本形式语法如下：

```
if(表达式) 语句
```

其语义是：如果表达式的值为真，则执行其后的语句，否则不执行该语句。其执行过程可表示为下图 4-5。

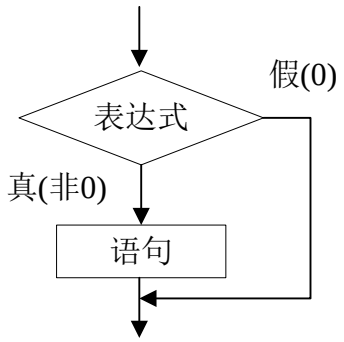


图 4-5 if 语句流程图

if 基本形式应用举例，if1.c 源代码如下：

```
#include <stdio.h>
void main()
{
    int a,b,max;
    printf("\n input two numbers:    ");
    scanf("%d%d",&a,&b);
    max=a;
    if (max<b) max=b;
    printf("max=%d",max);
}
```

编译 gcc if1.c -o if1。

执行 ./if1，执行结果如下：

```
    input two numbers:      3 9
    max=9
```

（2）第二种形式为 if-else 形式

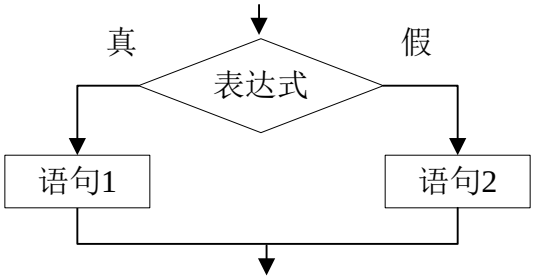
if-else 语句语法形式如下：

```
    if(表达式)
        语句 1;
    else
        语句 2;
```

其语义是：如果表达式的值为真，则执行语句 1，否则执行语句 2。其执行过程可表示为下

图 4-6。

图 4-6 if-else 语句流程图



if-else 形式应用举例，if2.c 源代码如下：

```
#include <stdio.h>
void main()
{
    int a, b;
    printf("input two numbers:      ");
    scanf("%d%d",&a,&b);
    if(a>b)
      printf("max=%d\n",a);
    else
      printf("max=%d\n",b);
}
```

编译 gcc if2.c -o if2。

执行 ./if2，执行结果如下：

```
    input two numbers:      10 0
    max=10
```

（3）第三种形式为 if-else-if 形式

前二种形式的 if 语句一般都用于两个分支的情况。当有多个分支选择时，可采用 if-else-if 语句，其语法形式如下：

```
    if(表达式 1)
            语句 1;
        else  if(表达式 2)
            语句 2;
        else  if(表达式 3)
            语句 3;
            ...
        else  if(表达式 m)
            语句 m;
        else
            语句 n;
```

其语义是：依次判断表达式的值，当出现某个值为真时，则执行其对应的语句，然后跳到整

个 if 语句之外继续执行程序。如果所有的表达式均为假，则执行语句 n，然后继续执行后续程序。其执行过程可表示为下图 4-7。

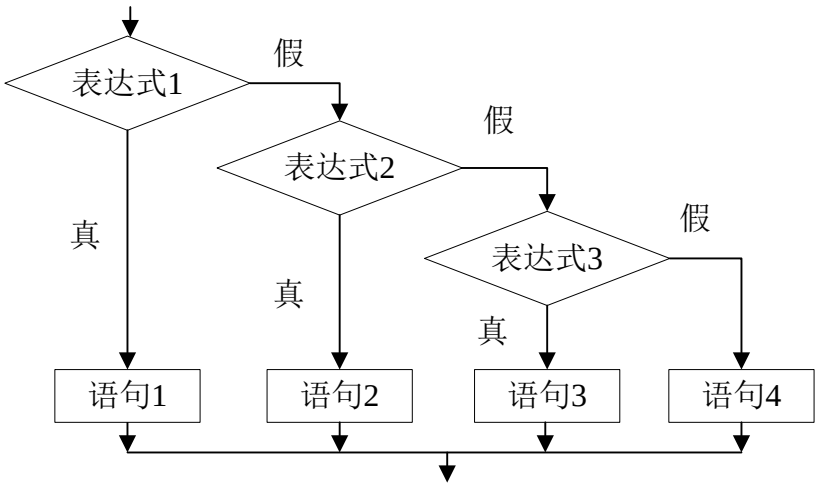


图 4-7 if-else-if 语句流程图

if-else-if 形式应用举例，if3.c 源代码如下：

```
#include   <stdio.h>
void main()
{
      char c;
      printf("input a character:       ");
      c=getchar();
      if(c<32)
         printf("This is a control character\n");
      else if(c>='0'&&c<='9')
         printf("This is a digit\n");
      else if(c>='A'&&c<='Z')
         printf("This is a capital letter\n");
      else if(c>='a'&&c<='z')
         printf("This is a small letter\n");
      else
         printf("This is an other character\n");
}
```

编译 gcc if3.c -o if3。

执行 ./if3，执行结果如下：

```
input a character:      X
This is a capital letter
```

2. if语句的嵌套

当 if 语句中的执行语句又是 if 语句时，则构成了 if 语句嵌套的情形，为了避免这种二义性，C 语言规定，else 总是与它前面最近的 if 配对。

下面的 if 语句就构成了 if 语句的嵌套，使用 if 语句应尽量避免此种形式，应尽量使用“{}”来说明 else 匹配哪一个 if。

```
if(表达式 1)
    if(表达式 2)
        语句 1;
    else
        语句 2;
```

1.4.2 switch 语句

1. switch 语句说明

switch 语句用于多分支选择，其语法形式如下：

```
switch(表达式){
    case 常量表达式 1:  语句 1;
    case 常量表达式 2:  语句 2;
    ...
    case 常量表达式 n:  语句 n;
    default          :  语句 n+1;
    }
```

其语义是：计算表达式的值。并逐个与其后的常量表达式值相比较，当表达式的值与某个常量表达式的值相等时，即执行其后的语句，然后不再进行判断，继续执行后面所有 case 后的语句。如表达式的值与所有 case 后的常量表达式均不相同时，则执行 default 后的语句。所以 case 语句需要用 break 进行跳出，否则将对后面的语句顺序执行。

2. switch 语句注意事项

switch 有如下注意事项：

① 在case后的各常量表达式的值不能相同，否则会出现错误。
② 在case后，允许有多个语句，可以不用{}括起来。
③ 各case和default子句的先后顺序可以变动，且不会影响程序执行结果。
④ default子句可以省略不用，但不提倡省略。
⑤ switch表达式结果只能为整型变量和字符型变量。

3. switch 关键字应用举例

switch1.c 源代码如下：

```
#include <stdio.h>
void main()
{
    int a;
    printf("input integer number:      ");
    scanf("%d",&a);
    switch (a){
      case 1:printf("Monday\n");break;
      case 2:printf("Tuesday\n"); break;
      case 3:printf("Wednesday\n");break;
      case 4:printf("Thursday\n");break;
      case 5:printf("Friday\n");break;
      case 6:printf("Saturday\n");break;
      case 7:printf("Sunday\n");break;
```

```
        default:printf("error\n");
    }
}
```

编译 gcc switch1.c -o switch1。

执行 ./switch1，执行结果如下：

```
    input integer number:      1
    Monday
```

switch2.c 源代码如下：

```
#include <stdio.h>
void main()
{
    float a,b;
    char c;
    printf("input expression: a+(-,*,/)b \n");
    scanf("%f%c%f",&a,&c,&b);
    switch(c){
        case '+': printf("%.2f\n",a+b);break;
        case '-': printf("%.2f\n",a-b);break;
        case '*': printf("%.2f\n",a*b);break;
        case '/': printf("%.2f\n",a/b);break;
        default: printf("input error\n");
    }
}
```

编译 gcc switch2.c -o switch2。

执行 ./switch2，执行结果如下：

```
    input expression: a+(-,*,/)b
    3*7
    21.00
```

1.4.3 goto 语句

1. goto 语句说明

goto 语句是一种无条件转移语句。其语法形式如下：

```
    goto   语句标号;
```

标号是一个有效的标识符，这个标识符加上一个“:”一起出现在函数内某处,执行 goto 语句后，程序将跳转到该标号处并执行其后的语句。

标号必须与 goto 语句同处于一个函数中，但可以不在一个循环层中。

goto 语句经典使用场合为程序处理多个错误时跳转到结尾进行错误处理，达到错误处理代码复用的目的，并减少 return 语句。其他场合不建议使用。

2. goto 语句应用举例

goto.c 源代码如下：

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

```
{
    int i=0;
    scanf("%d", &i) ;
    if ( i<0 )
    {
        goto err_end ;
    }
    if ( i >100 )
    {
        goto err_end ;
    }
    printf("input number validity:%d\n", i) ;
    return 0 ;
err_end:
    printf("input number overflow, please guess again\n");
    return -1 ;
}
```

编译 gcc goto.c -o goto。

执行 ./goto，执行结果如下:

```
    900
    input number overflow, please guess again
```

1.4.4 while 语句

1. while语句说明

while 语句用来实现循环结构，其语法形式如下:

```
    while(表达式) 语句
```

其中表达式是循环条件，语句为循环体。

while 语句的语义是：计算表达式的值，当值为真(非 0)时， 执行循环体语句。其执行过程可用下图 4-8 表示。

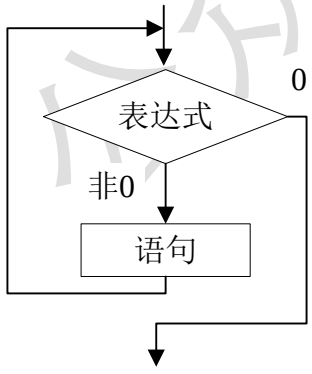


图 4-8 while 语句流程图

2. while 语句举例

用 while 语句求 $\sum_{n=1}^{100} n$ 。

while.c 源代码如下：

```
#include <stdio.h>
void main()
{
    int i,sum=0;
    i=1;
    while(i<=100)
          {
             sum=sum+i;
             i++;
           }
    printf("%d\n",sum);
}
```

编译 gcc while.c -o while。

执行 ./while，执行结果如下：

```
sum=5050
```

1.4.5 do-while 语句

1. do-while 语句原型

do-while 语句同样是用来实现循环结构，其语法形式如下：

```
do
  语句
while(表达式);
```

这个循环与 while 循环的不同在于：它先执行循环中的语句，然后再判断表达式是否为真，如果为真则继续循环；如果为假，则终止循环。因此，do-while 循环至少要执行一次循环语句。其执行过程可用下图 4-9 表示。

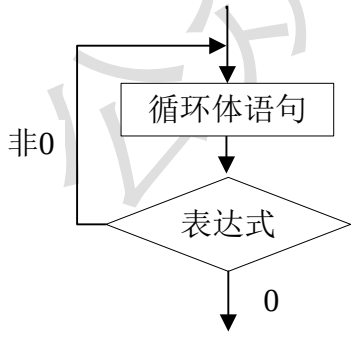


图 4-9 do-while 语句流程图

2. do-while 语句举例

用 do-while 语句求 $\sum_{n=1}^{100} n$ 。

dowhile.c 源代码如下：

```
#include <stdio.h>
void main()
{
    int i,sum=0;
    i=1;
    do
    {
        sum=sum+i;
        i++;
    }while(i<=100)
    printf("sum=%d\n",sum);
}
```

编译 gcc dowhile.c -o dowhile。

执行 ./dowhile，执行结果如下：

```
sum=5050
```

1.4.6 for 语句

1. for 语句原型

for 语句也是用来实现循环结构，其语法形式如下：

```
for(表达式 1；表达式 2；表达式 3) 语句
```

它的执行过程如下：

1) 先求解表达式 1。
2) 求解表达式 2，若其值为真（非 0），则执行 for 语句中指定的内嵌语句，然后执行下面第 3）步；若其值为假（0），则结束循环，转到第 5）步。
3) 求解表达式 3。
4) 转回上面第 2）步继续执行。
5) 循环结束，执行 for 语句下面的一个语句。

其执行过程可用下图 4-10 表示。

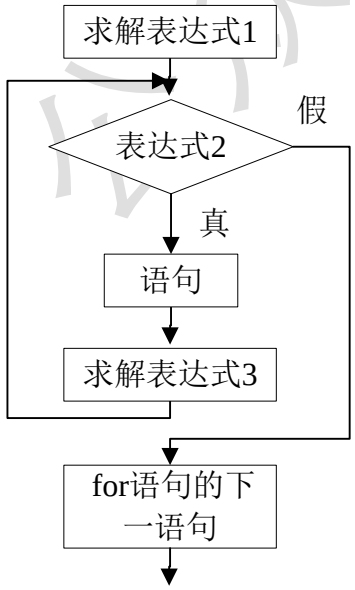


图 4-10 for 语句流程图

for 语句最简单的应用形式也是最容易理解的形式，其使用形式如下：

```
for(循环变量赋初值；循环条件；循环变量增量) 语句
```

注意 for 循环三个部分之间要用“；”分开，举例说明如下。

```
for(i=1; i<=100; i++)sum=sum+i;  /*利用 for 语句实现上述 1 到 100 的相加功能*/
```

2. for 语句的多种使用方法

for 语句有如下多种使用方法。

```
for(i=1; i<=100; i++)    /*常见使用方法*/
for(i=1;;i++)             /*死循环，语句中碰到合适条件时用 break 跳出*/
for(i=1;i<=100;)          /*循环增加放到语句中*/
for(;i<=100;)             /*初始化放到语句外，循环增量放到语句中*/
for(;;)                  /*相当于 while(1)语句*/
```

3. for 语句应用举例

for.c 源代码如下：

```
#include <stdio.h>
void main()
{
    int i, j, k;
    printf("i j k\n");
    for (i=0; i<2; i++)
       for(j=0; j<2; j++)
          for(k=0; k<2; k++)
              printf("%d %d %d\n", i, j, k);
 }
```

编译 gcc for.c -o for。

执行 ./for，执行结果如下：

```
i j k
i=0 j=0 k=0
i=0 j=0 k=1
i=0 j=1 k=0
i=0 j=1 k=1
i=1 j=0 k=0
i=1 j=0 k=1
i=1 j=1 k=0
i=1 j=1 k=1
```

4. 几种循环的比较

while、do-while、for 三种循环都可以用来处理同一个问题，一般可以互相代替，for 语句功能最强。

while 和 do-while 循环，循环体中应包括使循环趋于结束的语句。

用 while 和 do-while 循环时，循环变量初始化的操作应在 while 和 do-while 语句之前完成，而 for 语句可以在表达式 1 中实现循环变量的初始化。

1.4.7 break 和 continue 语句

1. 两语句说明

continue 语句终止当前循环后又回到循环体的开头准备执行下一次循环。

break 语句跳出当前循环。

2. break 和 continue 语句举例

breakcon.c 源代码如下：

```
#include <stdio.h>
int is_prime(int n)
{
	int i;
	for (i = 2; i < n; i++)
		if (n % i == 0)
			break;
	if (i == n)
		return 1;
	else
		return 0;
}
int main(void)
{
	int i;
	for (i = 1; i <= 100; i++) {
		if (!is_prime(i))
			continue;
		printf("%d  ", i);
	}
	printf("\n") ;
	return 0;
}
```

编译 gcc breakcon.c -o breakcon。

执行 ./breakcon，执行结果如下：

```
2  3  5  7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67  71  73  79  83
89  97
```

程序说明如下：

is_prime 函数从 2 到 n-1 依次检查有没有能被 n 整除的数，如果有就说明 n 不是素数，立刻跳出循环而不执行 i++，所以 is_prime 函数中使用的是 break。

在主程序中，从 1 到 100 依次检查每个数是不是素数，如果不是素数，并不直接跳出循环，而是 i++后继续执行下一次循环，因此用 continue 语句。

第2章 C语言函数

2.1 函数简述

在学生时代的数学课上，老师用 y=f(x,a,…)来说明数学中函数。C 语言是函数式语言，C 语言函数的名称其实也是借鉴数学中的函数。

函数是按照模块化设计思想，实现特殊控制流程的程序块。

函数在内存表现为内存中的一段二进制代码，可以被 CPU 执行的一段机器码。而程序的执行只不过程序代码段的顺序执行和跳转执行而已。函数调用属于跳转执行代码，将程序代码段指针跳到函数起始地址处开始执行代码，函数执行完后代码段指针指向调用函数程序的下一行。

1. 函数概述

函数的基本思想为将一个大的程序按功能分割成一些小模块。

函数特点为各模块相对独立、功能单一、结构清晰、接口简单，控制了程序设计的复杂性，提高元件的可靠性，缩短开发周期，避免程序开发的重复劳动，易于维护和功能扩充。函数开发方法为自上向下，逐步分解，分而治之。

C 是函数式语言，必须有且只能有一个名为 main 的主函数，C 程序的执行总是从 main 函数开始，在 main 中结束，函数不能嵌套定义，可以嵌套调用。

函数分类从用户角度分为标准函数（库函数）和用户自定义函数。从函数调用或被调用形式上分为有参函数和无参函数。从返回值上划分为有返回值函数和无返回值函数，无返回值函数需要在函数名前加 void 关键字。

有参函数在函数定义及函数说明时都有参数，这些参数称为形式参数(简称为形参)。在函数调用时也必须给出参数，这些参数称为实际参数(简称为实参)。进行函数调用时，主调函数将把实参的值传送给形参，供被调函数使用。

使用库函数时应注意函数功能，函数参数的数目和顺序，各参数意义和类型，函数返回值意义和类型，需要包含的头文件。

函数的返回值形式为：return (表达式)、return 表达式、 return 三种。返回的作用是使程序控制从被调用函数返回到调用函数中，同时把返回值带给调用函数。函数中可有多个 return 语句（但良好的设计应保持只有一个 return 语句），若无 return 语句，遇}时，自动返回调用函数，若函数类型与 return 语句中表达式值的类型不一致，按照函数类型为准进行自动转换。函数返回值类型默认为 int 型。

函数调用形式为“函数名(实参表)”，调用时要求函数的实参与形参个数相等、类型一致、按顺序一一对应。

在同一文件，使用后面的函数（除返回值是 int 类型和 void 类型而且不带形参的函数）需要在文件头进行声明。如果有多个源文件，也可以把函数声明放在一个单独的头文件中，其他源文件包含此头文件。

2. 从用户角度划分函数

从用户角度分可划分为库函数和用户自定义函数，具体说明如下：

① 库函数：由C系统提供，用户无须定义，也不必在程序中作类型说明，只需在程序前包含有该函数原型的头文件即可在程序中直接调用。如printf、scanf、strcpy、strcat等函数均属此类。

② 用户定义函数：由用户按需要自己写的函数。对于用户自定义函数，不仅要在程序中定义函数本身，而且在主调函数程序中还必须对该被调函数进行类型说明，然后才能使用。

3. 从功能角度划分函数

从功能角度可以把 C 语言函数划分为如下 13 类：

① 字符类型测试函数：用于对字符类型进行测试。
② 转换函数：用于字符或字符串的转换，在字符量和各类数字量(整型、实型等)之间进行转换。
③ 目录路径函数：用于文件目录和路径操作。
④ 诊断函数：用于内部错误检测。
⑤ 图形函数：用于屏幕管理和各种图形功能。
⑥ 输入输出函数：用于完成输入输出功能。
⑦ 字符串函数：用于字符串操作和处理。
⑧ 内存管理函数：用于内存管理。
⑨ 数学函数：用于数学函数计算。
⑩ 日期和时间函数：用于日期，时间转换操作。
⑪ 进程控制函数：用于进程管理和控制。
⑫ 文件操作函数：用于对文件的操作。
⑬ 其他函数：用于其他各种功能的操作。

5. 程序执行时的内存布局

当一个源代码通过 gcc 编译成 a.out，执行 a.out 时，程序便开始了执行之旅（即进程）。操作系统为进程分配堆栈空间，随后把程序执行码放入文本段，把程序中经过初始化的全局变量和静态变量放入数据段，把程序中未初始化的全局变量和静态变量放入 bss 段并对 bss 段数据初始化为 0。之后 CPU 代码段指针指向 main 的入口，CPU 堆栈段指针指向栈顶，代码段指针从 main 的入口地址顺序读取指令代码并进行执行，碰到局部变量和函数调用时需要在栈顶分配空间并把堆栈段指针下移，碰到 malloc 等动态分配内存函数就在堆上分配内存。图 5-1 画出了程序执行时堆栈空间图，此图对于理解程序的运行机理非常重要。

栈段（.stack） （局部于函数的数据）
堆段（.heap） （malloc 申请的内存区域）
bss 段（.bss） （未初始化的数据）
数据段（.data） （经过初始化的全局变量和静态变量）
文本段（.text） （又称代码段，存放程序执行码）

图 5-1 程序执行堆栈图

下面是对图 5-1 各段的具体说明。

.text：文本段，又称代码段，存放程序执行码。

.data：数据段，存放已初始化全局/静态变量，在整个程序执行过程中有效。

.bss：bss 段，存放未初始化全局/静态变量，在整个程序执行过程中有效。

.stack：栈段，存放函数调用栈和函数局部变量，其中的内容在函数执行期间有效，并由编译器负责分配和收回。

.heap：堆段，由程序显式分配和收回，如果不回收就会产生内存泄漏。

数据段和 bss 段有时也统称为数据区。

2.2 函数变量

1. 函数变量简述

在 C 语言中，每个变量和函数有两个属性：数据类型和数据的存储类别。

变量存储类型的属性按生存期（时间）分为静态变量与动态变量，按照作用域（空间）分为局部变量与全局变量。

局部变量即内部变量,在函数内定义,只在本函数内有效。main 中定义的变量只在 main 中有效，所以也属于局部变量。不同函数中的同名变量，占用不同内存单元。函数中的形参属于局部变量。局部变量可用存储类型有 auto、register、static 三种类型，局部变量默认为 auto 类型。

静态存储是程序运行前分配固定存储空间，如 bss 段和数据段。

动态存储是程序运行期间根据需要动态分配的存储空间。如程序运行时使用的栈段和使用 malloc 函数申请空间使用的堆段。

静态变量从程序开始执行前分配空间到程序执行结束才释放空间。静态变量属于静态存储，静态变量包括全局变量、静态全局变量、静态局部变量。

动态变量作用域是从包含该变量定义的函数开始执行至此函数执行结束。动态变量属于动态存储，函数中的变量属于动态变量。

2. 变量存储类型关键字说明

变量的存储类型有 auto（自动型）、register（寄存器型）、static（静态型）、extern（外部型）四种，下面是这四种存储类型的具体说明。

① auto：局部变量。编译器在默认的情况下，所有变量都是auto。

② register：寄存器变量。变量存在于CPU寄存器中，CPU寄存器不足时此变量当做auto变量来处理。寄存器变量只能是单个值，长度小于或等于整型变量，由于此变量存在于CPU的寄存器中，不能用“&”来获得地址。把经常使用的变量放入寄存器中是为了获得高速度。

③ static：静态变量。在文件头定义的静态变量是全局静态变量，在函数中定义的静态变量是局部静态变量。静态变量在执行前将分配内存空间,在程序执行成后才释放内存空间。全局静态变量作用域为整个文件，局部静态变量作用域为单个函数。

④ extern：外部变量。引用的变量是其在其他文件头进行定义的。

2.3 函数定义与调用

2.3.1 函数定义

函数使用前必须先定义,函数定义从函数调用或被调用形式上可分为有参函数和无参函数两种，下面是对这两种函数定义形式的具体说明。

1. 无参函数的定义形式

无参函数定义一般形式如下:

```
类型标识符 函数名()
{   变量定义部分
     语句
}
```

函数名是由用户定义的标识符，下面是一个无参无返回值的函数。

```
void Hello()
{
        printf ("Hello,world \n");
```

```
}
```

2. 有参函数定义的一般形式

有参函数比无参函数多了一个内容，即形式参数表列。在形参表中给出的参数称为形式参数，它们可以是各种类型的变量，各参数之间用逗号间隔。在进行函数调用时，主调函数将赋予这些形式参数实际的值。形参既然是变量，必须在形参表中给出形参的类型说明。

有参函数定义一般形式如下:

```
类型标识符 函数名(形参类型  [形参名],…)
{
   变量定义部分
      语句
}
```

例如，定义一个函数，用于求两个数中的大数，可写为:

```
int max(int a, int b)
{
      if (a>b) return a;
      else return b;
}
```

第一行说明 max 函数是一个整型函数，其返回的函数值是一个整数，形参 a 和 b 均为整型量。

3. 静态函数

用 static 修饰的函数为静态函数。静态函数的作用域范围局限于本文件，又称为内部函数。使用内部函数的好处是不同的人在编写函数时，不用担心自己定义的函数，是否与其他文件中的函数同名。

2.3.2 函数的参数与返回值

1. 函数的形参与实参

发生函数调用时，主调函数把实参的值传送给被调函数的形参从而实现主调函数向被调函数的数据传送。

函数的形参和实参具有以下特点:

C 语言函数调用方式为传值调用方式。

C 语言函数调用时，为形参分配单元，并将实参的值复制到形参中；函数调用结束，形参单元被释放，实参单元仍保留并维持原值。

C 语言值传递的特点为形参与实参占用不同的内存单元，值为单向传递。

实参和形参在数量上，类型上，顺序上应严格一致，否则会发生类型不匹配的错误。

2. 函数的返回值

函数的值是指函数被调用之后，执行函数体中的程序段所取得的并返回给主调函数的值。

函数的值只能通过 return 语句返回主调函数。

return 语句的一般形式如下:

```
return 表达式;
或者为 return (表达式) ;
或者为 return ;
```

该语句的功能是计算表达式的值，并返回给主调函数。在函数中允许有多个 return 语

句，但每次调用只能有一个 return 语句被执行，因此只能返回一个值。

函数返回值类型应与函数类型保持一致，如果两者不一致，则以函数类型为准，自动进行类型转换。

如函数返回值为整型，在函数定义时可以省去类型说明。

没有返回值的函数，可以明确定义为“空类型”，类型说明符为“void”。

3．形参实参、变量与返回值综合举例

（1）有参函数举例

formreal.c 源代码如下：

```
#include <stdio.h>
int lable1 ;
int lable2 = 0 ;
static int lable3 ;
int formreal(int x, int y)
{
    static int lable4 ;
    int z ;
    printf("&lable4=%d\n", &lable4) ;
    printf("&x=%d, &y=%d\n",&x, &y) ;
    x= x*5 ;
    y= y*5 ;
    printf("x=%d, y=%d,lable1=%d,lable2=%d,lable3=%d,\
        lable4=%d\n",x,y,lable1,lable2,lable3,lable4) ;
    return 0 ;
}
int main()
{
    int a,b,c,d;
    printf("&lable1=%d,&lable2=%d,&lable3=%d\n", \
            &lable1, &lable2, &lable3 ) ;
    printf("&a=%d,&b=%d,&c=%d,&d=%d\n",&a,&b,&c,&d) ;
    printf("input two numbers:\n");
    scanf("%d%d",&a,&b);
    c=formreal(a,b);
    printf("a=%d, b=%d,c=%d, d=%d\n",a,b,c,d);
    return 0 ;
}
```

编译 gcc formreal.c -o formreal。

执行 ./formreal，执行结果如下：

```
&lable1=134518780,&lable2=134518768,&lable3=134518776
&a=-1075307972,&b=-1075307976,&c=-1075307980,&d=-1075307984
input two numbers:
2 3
&lable4=134518772
```

```
&x=-1075308016, &y=-1075308012
x=10, y=15,lable1=0,lable2=0,lable3=0,          lable4=0
a=2, b=3,c=0, d=-1208725516
```

在上例中，x、y 属于形参，调用 c=formreal(a,b)语句时，CPU 执行将堆栈指针下移，为函数、形参 x、y 分配内存单元，同时将实参 a 的值复制给形参 x，将实参 b 的值复制给形参 y。图 5-2 画出了函数调用时实参传值给形参的方式。

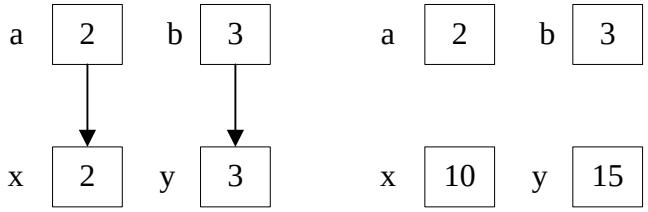


图 5-2 函数调用时实参传值给形参方式图

（2）formreal 程序执行堆栈表

表 5-1 列出了 formreal 程序执行时堆栈空间的内存模型，与实际执行堆栈空间并不完全一致。但可以让读者更好的理解全局变量、静态变量、实参和形参，特别是实参和形参是怎样进行传值调用的。

对表 5-1 设计的具体说明如下：

由于上面实例中使用的都是整型变量，整型变量占用内存空间为 4 个字节，所以这里以 4 个字节为单位对堆栈空间进行说明。

函数的返回值一般是通过 CPU 数据寄存器 EAX 返回，这里为了好理解，依然为函数返回值分配内存空间。

当执行./formreal 时，操作系统会为执行码分配好代码段、静态数据段、bss 段和栈段，然后代码指针指向 main 的入口。所以上述 lable1~lable4 变量是在程序执行前分配到数据段和 bss 段并完成赋值。在这里补充说明的是，静态局部变量作用范围是在编译时进行检查的，在编译后，静态局部变量和静态全局变量在执行程序看来，没有本质的差别。

从下面的执行过程可以看出，调用函数时系统为形参分配空间，将实参值复制给形参，所以说 C 语言传值调用为单向传值调用。

在代码编译成二进制码时，所有的变量失去意义，变量的操作其实都转化为对应内存地址的操作，变量其实只是一段内存空间值的抽象。编译完成后，机器代码中没有变量的存在，只有对内存线性逻辑地址的操作。

表 5-1 formreal 程序堆栈空间表

数据段说明	内存地址	实际内存
栈段	30000	main 函数返回值的存放空间
	29996	a
	29992	b
	29988	c
	29984	d
	29980	formreal 函数返回值的存放空间
	29976	x
	29972	y
	29968	z
	……	

堆段		
	……	
bss 段	10012	lable1
静态数据段	10008	lable3
	10004	lable4
	10000	lable2
代码段		1.formreal 函数地址入口 2.在栈顶分配 z 变量空间 3.打印 lable4 变量的地址到屏幕 4.打印 x、y 变量的地址到屏幕 5.x 的值送到 CPU 乘以 5，数据结果由 cpu 返传到 x 的内存地址处 6.y 的值送到 CPU 乘以 5，数据结果由 cpu 返传到 y 的内存地址处 7.打印 x、y、lable1 到 lable4 的值到屏幕上 8.返回 0 给函数返回值空间，即 29980~29977 处 4 个字节的空间 9.函数结束返回 10.main 函数地址入口 11.在栈顶分配 main 函数返回值、a、b、c、d 变量空间 12.打印 lable1 到 lable3 的地址到屏幕上 13.打印输入两个数字的提示信息 14.从屏幕上读取两个数字分别存放到 a、b 的内存空间里。此时 &a 代表内存地址 29996，而 a 代表地址从 29993~29996 的 4 个字节内存抽象，即向这 4 个字节里存放第一个读取的值。b 变量同理 15.调用 formreal(a,b)函数，在栈顶分配 formreal 返回值空间，x、y 变量空间。并将实参 a 的值复制给 x，实参 b 的值复制给 y。然后即跳转到 1(formreal 函数入口)的地方去执行 16.将 formreal 函数的返回值赋给 c，即将 29980~29977 地址空间的值复制到 29988~29985。栈顶运行指针移到 29980 处，代表 formreal 函数运行栈空间已经释放 17.打印 a、b、c、d 的值到屏幕上 18.把返回值 0 赋值到 main 入口空间，即地址为 30000~29997 的内存空间处 19.程序执行完成，释放所有内存空间

2.3.3 函数调用

1. 函数调用简述

函数声明是说明函数的调用形式，函数声明必须是已存在的函数。使用库函数需要包含对应的头文件，包含方法为#include <*.h>。使用用户自定义函数需在文件内进行函数声明或将函数声明放在自定义头文件中然后进行包含，包含用户自定义头文件方法为#include "*.h"。

函数声明的一般形式为：函数类型　函数名(形参类型　[形参名],…)或函数类型 函数名

()。函数声明的作用告诉编译系统函数类型、参数个数及类型，以便检验。函数定义与函数声明不同，函数定义是函数的实现。函数声明是说明函数的调用形式，方便其他程序按接口调用。函数声明位置可在函数内或外。

函数调用时实参必须有确定的值,形参必须指定类型,形参与实参类型一致,个数相同。若形参与实参类型不一致，自动按形参类型转换。形参在函数被调用前不占内存，函数调用时为形参分配内存，调用结束，内存释放。

函数定义不可嵌套，但可以嵌套调用函数。

2. 函数调用的方式

函数调用有如下三种方式:

① 函数表达式：函数作为表达式中的一项出现在表达式中，以函数返回值参与表达式的运算，这种方式要求函数是有返回值。例如：z=max(x,y)是一个赋值表达式，把max的返回值赋予变量z。

② 函数语句：函数调用的一般形式加上分号即构成函数语句。例如：printf ("%d",a)和scanf ("%d",&b)都是以函数语句的方式调用函数。

③ 函数实参：函数作为另一个函数调用的实际参数出现。这种情况是把该函数的返回值作为实参进行传送，因此要求该函数必须是有返回值的。例如：printf("%d",max(x,y))，即把max调用的返回值又作为printf函数实参来使用的。在函数调用中还应该注意的一个问题是求值顺序的问题，所谓求值顺序是指对实参表中各量是自左至右使用还是自右至左使用，对此，各系统的规定不一定相同。

3. 被调用函数的声明

在主调函数中调用某函数之前应对该被调函数进行声明,这与使用变量之前要先进行变量说明是一样的。在主调函数中对被调函数作说明的目的是使编译系统知道被调函数返回值的类型，以便在主调函数中按此种类型对返回值作相应的处理。

其一般形式为:

```
类型说明符 被调函数名(类型 形参，类型 形参…);
```

或为:

```
类型说明符 被调函数名(类型，类型…);
```

函数声明时圆括号内需给出形参类型和形参名，或只给出形参类型。这便于编译系统进行检错，以防止可能出现的错误。

例 main 函数中对 max 函数的说明为:

```
int max(int a,int b);
或写为:int max(int,int);
```

C 语言中规定在以下几种情况时可以省去函数声明。

如果被调函数的返回值是 int 或 void 型且没有形参时，可以不对被调函数作说明，而直接调用，这时系统将自动对被调函数返回值按整型处理。但不提倡不进行声明。

当被调函数的函数定义出现在主调函数之前时，在主调函数中也可以不对被调函数再作说明而直接调用。

如在所有函数定义之前或头文件中对函数原型进行了声明,则在以后的各主调函数中,可不再对被调函数作说明。

4.函数的递归调用

一个函数在它的函数体内调用它自身称为递归调用，这种函数称为递归函数。C 语言允许函数的递归调用，在递归调用中，主调函数又是被调函数。执行递归函数将反复调用其自

身，每调用一次就进入新的一层。直到碰到结束条件然后递归返回。递归的次数是有限的，常用的办法是加条件判断，满足某种条件后就不再作递归调用，然后逐层返回。

5.用递归法计算 n!

（1）n!的递归程序实现

用递归法计算 n!，可用下述公式表示：

n!=1　　(n=0，1)

n×(n-1)!　　(n>1)

按公式编程实现如下，recursion.c 源代码如下：

```
#include <stdio.h>
int ff(int n)
{
    int f;
    if(n<0) printf("n<0,input error");
    else if(n==0||n==1) f=1;
    else f=ff(n-1)*n;
    return(f);
}
void main()
{
    int n;
    int y;
    printf("\n input a inteager number:\n");
    scanf("%d",&n);
    y=ff(n);
    printf("%d!=%ld",n,y);
}
```

编译 gcc recursion.c -o　recursion。

执行 ./recursion，执行结果如下：

```
input a inteager number:
3
3!=6
```

程序中给出的函数 ff 是一个递归函数。主函数调用 ff 后即进入函数 ff 执行，如果 n<0、n==0 或 n==1 时都将结束函数的执行，否则就递归调用 ff 函数自身。由于每次递归调用的实参为 n-1，即把 n-1 的值赋予形参 n，最后当 n-1 的值为 1 时形参 n 的值也为 1，将使递归终止，然后可逐层回退。

（2）recursion 程序执行堆栈表

表 5-2 列出程序 recursion 递归调用时堆栈变化情况。

表 5-2 递归调用堆栈表

数据段说明	内存地址	实际内存
栈段	30000	main 函数返回值存放空间
	29996	n =3 (main 中局部变量)
	29992	y

	29988	ff(3)
	29984	n=3 (ff 函数中局部变量，下同，不再说明)
	29980	f=ff(2)*3
	29976	ff(2)
	29972	n=2
	29968	f=ff(1)*2
	29964	ff(1)
	29960	n=1
	29956	f=1
	……	

根据表 5-2，对递归调用流程说明如下：

① 递归调用时栈不停向下增长，直到碰到结束条件才依次向上返回。
② 如ff(1)碰到结束条件f=1 返回，此时ff(1)=1。
③ 上一级ff(2)中函数f=ff(1)*2，f=1*2=2，f返回给ff(2)，ff(2)=2。
④ 再往上一级ff(3)中f=ff(2)*3，f=2*3=6，f返回给ff(3)，ff(3)=6。
⑤ 将ff(3)赋值给main函数中变量y，所以打印出来的值为 6。

第3章 C语言数组、结构体及指针

本章节介绍 C 语言数组、结构体及指针的知识，由于 C 语言中数组、结构体经常以指针的方式进行使用，所以将这三部分放入一个章节中。

3.1 C语言数组

3.1.1 数组概述

在程序设计中，为了处理方便，把具有相同类型的若干变量按有序的形式组织起来，这些按序排列的同类数据元素的集合称为数组。在 C 语言中，数组属于构造数据类型。一个数组可以分解为多个数组元素，这些数组元素可以是基本数据类型或是构造类型。C 语言数组按维数分有一维、二维和多维，按数组元素的类型又可分为数值数组、字符数组、指针数组、结构体数组等各种类别。

对数组特点概括如下：

① 数组是有序数据的集合，用数组名标识。
② 数组元素需属同一数据类型，用数组名和下标确定。
③ 一维数组的定义：数据类型　数组名[常量表达式]。
④ 数组名表示的是内存首地址，是地址常量，所以只能右值（=号的右边），不能是左值（=号的左边，当变量使用）。
⑤ 程序运行时对数组分配连续的内存空间，数组空间大小=数组维数*sizeof(元素数据类型)。
⑥ 数组元素个数的下标从 0 开始。
⑦ C语言对数组不作越界检查，使用时要注意。int a[5]只能用a[0]~a[4]，用a[5]就发生了越界。
⑧ 只能逐个引用数组元素，不能一次引用整个数组。
⑨ 数组不初始化时，其元素值为随机数。
⑩ 对static数组元素不赋初值时，系统会自动赋以 0 值。
⑪ 当全部数组元素赋初值时，可不指定数组长度。

⑫ C语言中无字符串变量，是用字符数组处理字符串，字符串结束标志为字符'\0'。

3.1.2 一维数组

1. 一维数组定义

在 C 语言中使用数组必须先进行定义。

一维数组的定义方式如下：

```
类型说明符 数组名[常量表达式];
```

其中：

① 类型说明符是任一种基本数据类型或构造数据类型。
② 数组名是用户定义的数组标识符。
③ 方括号中的常量表达式表示数据元素的个数，也称为数组的长度。

例如：

```
int a[10];              说明整型数组 a，有 10 个元素。
float b[10],c[20];      说明实型数组 b，有 10 个元素；实型数组 c，有 20 个元素。
char ch[20];            说明字符数组 ch，有 20 个元素。
```

对于数组类型说明应注意以下几点：

① 数组的类型实际上是指数组元素的取值类型，对于同一个数组，其所有元素的数据类型都是相同的。
② 数组名的书写规则应符合标识符的书写规定。
③ 数组名不能与其他变量名相同。
④ 方括号中常量表达式表示数组元素的个数，但是其下标从 0 开始计算。如a[3]表示数组a有 3 个元素，这 3 个元素分别为a[0]、a[1]、a[2]。
⑤ 注意数组的越界，达到数组的最大序号就是越界。
⑥ 不能在方括号中用变量来表示元素的个数，但是可用符号常数或常量表达式。

例如：

```
#define FD 5
main()
{
    int a[3+2],b[7+FD];
    ……
}
```

是合法的。

但是下述说明方式是错误的。

```
 main()
{
        int n=5;
        int a[n];
        ……
}
```

2. 一维数组元素的引用

数组元素是组成数组的基本单元，数组元素也是一种变量，其标识方法为数组名后跟一个下标，下标表示了元素在数组中的顺序号。

数组元素引用形式如下：

```
数组名[下标]
```

其中下标只能为整型常量或整型表达式。如为小数时，C 编译将自动取整。

例如：

```
a[5]、a[i+j]、a[i++]都是合法的数组元素。
```

数组元素通常也称为下标变量，必须先定义数组，才能使用下标变量。在 C 语言中只能逐个地使用下标变量，而不能一次引用整个数组。

例如，输出有 10 个元素的数组必须使用循环语句逐个输出各下标变量，输出方法如下：

```
for(i=0; i<10; i++)
    printf("%d",a[i]);
```

不能用一个语句输出整个数组，下面的写法是错误的：

```
printf("%d",a);
```

3. 一维数组内存格局

【例 6-1】以 int a[3]和 char aa[5]为例说明数组内存的格局。

表 6-1 列出例 6-1 中数组变量内存格局图，其中“==”表示左右两边变量相等，后文同。int 说明数组 a 中每个元素类型为 int，每个元素占用 4 个字节，下标 3 表示元素个数为 3，此数组元素为 a[0]~a[3]。a 同时又是该数组的首地址，是一地址常量。

char 说明数组 aa 中每个元素类型为 char，每个元素占用 1 个字节，下标 5 表示元素个数为 5，此数组元素为 aa[0]~a[4]。aa 同时又是该数组的首地址。

由于 a 和 aa 代表的是内存地址，编译后逻辑地址固定，所以是地址常量。且 a 和 aa 本身代表地址时本身不占内存空间，不像指针变量有专门 4 个字节来存放变量的内存地址，数组名为地址常量，而指针变量是地址变量。

表 6-1 一维数组内存布局表

<table>
<tr><td colspan="4">一维数组的内存格局，假设变量的堆栈起始地址为 3000</td></tr>
<tr><td>内存地址</td><td>内存</td><td>假设的内存值</td><td>说明</td></tr>
<tr><td>3000</td><td rowspan="4">a[2]</td><td rowspan="4">9</td><td rowspan="4">编译完成后机器代码只认识地址，不认识变量。如 a[2]=9 时编译后的机器代码为向内存地址 3000~2997 内存里写入数据 9</td></tr>
<tr><td>2999</td></tr>
<tr><td>2998</td></tr>
<tr><td>2997</td></tr>
<tr><td>2996</td><td rowspan="4">a[1]</td><td rowspan="4">6</td><td rowspan="4">a[1]代表内存地址 2996~2993 空间的抽象，&a1 为内存地址 2996。由于此数组类型为 int，每个元素占用 4 个字节，如 a[1]=6 编译完成后机器代码转换为对相应内存地址的操作，就是向地址为&a[1](即 2996)相邻 4 个字节的内存里写入 6</td></tr>
<tr><td>2995</td></tr>
<tr><td>2994</td></tr>
<tr><td>2993</td></tr>
<tr><td>2992</td><td rowspan="4">a[0]</td><td rowspan="4">3</td><td rowspan="4">a[0]代表内存地址 2992~2989 内存空间的抽象，其存放的值为 3，此时 a[0]可以与 3 划上等号。&a[0]等于内存地址 2992，同时数组名 a 也代表内存地址 2992，所以 a==&a[0]</td></tr>
<tr><td>2991</td></tr>
<tr><td>2990</td></tr>
<tr><td>2989</td></tr>
<tr><td>2988</td><td>aa[4]</td><td></td><td rowspan="5">编译器通过元素类型确定每个元素的大小，通过元素类型和元素下标确定每个数组元素的内存地址，公式为“数组首地址+下标*单个元素长度”</td></tr>
<tr><td>2987</td><td>aa[3]</td><td></td></tr>
<tr><td>2986</td><td>aa[2]</td><td></td></tr>
<tr><td>2985</td><td>aa[1]</td><td></td></tr>
<tr><td>2984</td><td>aa[0]</td><td></td></tr>
</table>

4. 一维数组的初始化

给数组赋值的方法除了用赋值语句对数组元素逐个赋值外，还可采用初始化赋值和动态赋值的方法。

数组初始化赋值是指在数组定义时给数组元素赋予初值，数组初始化是在编译阶段进行的。这样将减少运行时间，提高效率。

初始化赋值的一般形式为：

```
类型说明符 数组名[常量表达式]={值，值……值};
```

其中在{ }中的各数据值即为各元素的初值，各值之间用逗号间隔。

例如：

```
int a[10]={ 0,1,2,3,4,5,6,7,8,9 };
```

相当于 a[0]=0;a[1]=1...a[9]=9;

C 语言对数组的初始化赋值还有以下几点规定：

(1) 可以只给部分元素赋初值。

当{ }中值的个数少于元素个数时，只给前面部分元素赋值。

例如：

```
int a[10]={0,1,2,3,4};
```

表示只给 a[0] ~ a[4]5 个元素赋值，而后 5 个元素自动赋 0 值。

(2) 只能给元素逐个赋值，不能给数组整体赋值。

例如给十个元素全部赋 1 值，只能写为：

```
int a[10]={1,1,1,1,1,1,1,1,1,1};
```

而不能写为：

```
int a[10]=1;
```

(3) 如给全部元素赋值，则在数组说明中，可以不给出数组元素的个数。

例如：

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

可写为：

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

5. 一维数组应用举例

【例 6-2】输入 10 个数，并输出最大的数。

arrmax.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int i,max,a[10];
    printf("input 10 numbers:\n");
    for(i=0;i<10;i++)
        scanf("%d",&a[i]);
    max=a[0];
    for(i=1;i<10;i++)
        if(a[i]>max) max=a[i];
    printf("maxmum=%d\n",max);
```

```
    return 0 ;
}
```

编译 gcc arrmax.c -o arrmax。

执行 ./arrmax，执行结果如下：

```
    input 10 numbers:
    3 9 1 900 100 800 700 600 90 0
    maxmum=900
```

3.1.3 二维数组

1. 二维数组的定义

二维数组定义的一般形式如下：

```
类型说明符 数组名[常量表达式 1][常量表达式 2] ;
```

其中常量表达式 1 表示第一维下标的长度，常量表达式 2 表示第二维下标的长度。例如：

```
    char a[3][4];
```

说明了一个三行四列的数组，数组名为 a，其下标变量的类型为整型。该数组的下标变量共有 3×4 个，即：

```
    a[0][0],a[0][1],a[0][2],a[0][3],
    a[1][0],a[1][1],a[1][2],a[1][3],
    a[2][0],a[2][1],a[2][2],a[2][3]
```

二维数组在概念上是二维的，即是说其下标在两个方向上变化，下标变量在数组中的位置也处于一个平面之中，而不是象一维数组只是一个向量。但是，实际的硬件存储器却是连续编址的，也就是说存储器单元是按一维线性排列的，二维数组也是线性存储在内存中。

二维数组长度计算公式如下：

```
二维数组长度=存储类型长度*第一维下标*第二维下标
```

表 6-2 列出了 char a[3][4]的内存布局表，从表中可以看出，二维数组内存空间是连续分配的，数组名和数组的一维下标表示的都是内存地址。

表 6-2 char a[3][4]的内存布局表

一维数组的内存格局，假设变量的堆栈起始地址为-2889		
内存地址	内存	补充说明
-2889	a[2][3]	
-2990	a[2][2]	
-2991	a[2][1]	
-2992	a[2][0]	此时 a[2]==&a[2][0]==-2992
-2993	a[1][3]	
-2994	a[1][2]	
-2995	a[1][1]	
-2996	a[1][0]	此时 a[1]==&a[1][0]==-2996
-2997	a[0][3]	
-2998	a[0][2]	
-2999	a[0][1]	
-3000	a[0][0]	此时 a==a[0]==&a[0][0]==-3000

2. 二维数组的初始化

二维数组初始化也是在类型说明时给各下标变量赋以初值。二维数组可按行分段赋值，也可按行连续赋值。

例如对数组 a[5][3]，可用如下两种方法初始化。

① 按行分段赋值可写为:

```
int a[5][3]={ {80,75,92},{61,65,71},{59,63,70},{85,87,90},{76,77,85} };
```

② 按行连续赋值可写为:

```
int a[5][3]={ 80,75,92,61,65,71,59,63,70,85,87,90,76,77,85};
```

这两种赋初值的结果是完全相同的。

对于二维数组初始化赋值还有以下说明:

(1) 可以只对部分元素赋初值，未赋初值的元素自动取 0 值。

例如：int a[3][3]={{1},{2},{3}};

是对每一行的第一列元素赋值，未赋值的元素取 0 值。 赋值后各元素的值为:

```
1 0 0
2 0 0
3 0 0
```

int a [3][3]={{0,1},{0,0,2},{3}};

赋值后的元素值为:

```
0 1 0
0 0 2
3 0 0
```

(2) 如对全部元素赋初值，则第一维的长度可以不给出。

```
例如：int a[3][3]={1,2,3,4,5,6,7,8,9};
可以写为：int a[][3]={1,2,3,4,5,6,7,8,9};
```

(3) 数组是一种构造类型的数据，二维数组可以看作是由一维数组的嵌套而构成的，假设一维数组的每个元素都又是一个数组，就组成了二维数组。当然，前提是各元素类型必须相同。根据这样的分析，一个二维数组也可以分解为多个一维数组，C 语言允许这种分解。

如二维数组 a[3][4]，可分解为三个一维数组，其数组名分别为 a[0]、a[1]、a[2]，这三个一维数组都有 4 个元素。例如：一维数组 a[0]的元素为 a[0][0]、a[0][1]、a[0][2]、a[0][3]。

必须强调的是，a[0]、a[1]、a[2]不能当做下标变量使用，它们是数组名，不是一个单纯的下标变量。但 a[0]、a[1]、a[2]代表数组第二维的首地址。

3. 多维数组定义

C 语言允许构造多维数组，多维数组元素有多个下标，以标识它在数组中的位置，所以也称为多下标变量。多维数组可由二维数组类推而得到。

多维数组定义语法形式如下:

```
类型    数组名[长度 1][长度 2]……[长度 N];
```

例如：int m[9][9][8][9]占用空间为 4*9*9*8*9。

4. 二维数组程序举例

【例 6-3】一个学习小组有 5 个人，每个人有三门课的考试成绩，求全组分科的平均成绩和各科总平均成绩。表 6-3 列出此 5 个人的三门课的考试成绩。

表 6-3 二维数组应用案例表

	张	王	李	赵	周
Math	80	61	59	85	76
C	75	65	63	87	77
Java	92	71	70	90	85

此时可设一个二维数组 a[5][3]存放五个人三门课的成绩，再设一个一维数组 v[3]存放所求得各分科平均成绩，设变量 average 为全组各科总平均成绩。

twoarray.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int i,j,s=0,average,v[3],a[5][3];
    printf("input score\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<5;j++)
        { scanf("%d",&a[j][i]);
            s=s+a[j][i];}
            v[i]=s/5;
            s=0;
    }
    average =(v[0]+v[1]+v[2])/3;
    printf("math:%d\nc languag:%d\nJava:%d\n",v[0],v[1],v[2]);
    printf("total:%d\n", average );
    return 0 ;
}
```

编译 gcc twoarray.c -o twoarray。

执行 ./twoarray，执行结果如下：

```
input score
80      61      59      85      76
75      65      63      87      77
92      71      70      90      85
math:72
c languag:73
Java:81
total:75
```

3.1.4 字符数组

1. 字符数组说明

在 C 语言中没有专门的字符串变量，通常用一个字符数组来存放一个字符串。前面介绍字符串常量时，已说明字符串总是以'\0'作为串的结束符。因此当把一个字符串存入一个数组时，也把结束符'\0'存入数组，并以此作为该字符串是否结束的标志。有了'\0'标志后，就不必再用字符数组的长度来判断字符串的长度了。

C 语言允许用字符串的方式对数组作初始化赋值。

例如：

```
    char c[]={'c', ' ','p','r','o','g','r','a','m'};
```

可写为:

```
char c[]={"C program"};
```

或去掉{}写为:

```
char c[]="C program";
```

用字符串方式赋值比用字符逐个赋值要多占一个字节，用于存放字符串结束标志'\0'。上面的后两种数组 c 在内存中的实际存放情况为:

C		p	r	o	g	r	a	m	\0

'\0'是由 C 编译系统自动加上的。由于采用了'\0'标志，所以在用字符串赋初值时一般无须指定数组的长度，而由系统自行处理。

截断字符数组或者增加结束标志方法为：c[6]= '\0'或 c[6]=0。

2. 字符串处理说明

如果字符数组存放的是字符串，输入可使用 scanf("%s",字符数组名)，输出可使用 printf("%s",字符数组名)。

字符串处理需要包含<string.h>头文件，需要使用 strcpy、strlen 等字符串函数，具体使用见第 9 章。

3. 字符数组举例

strarray.c 源代码如下:

```
#include <stdio.h>
#include <string.h>
int      main()
{
    char st[15];
    char st1[15] ;
    printf("input string:\n");
    scanf("%s",st);
    printf("%s\n",st);
    strcpy(st1, st) ;
    printf("cmp=%d\n", strcmp(st, st1)) ;
    return 0 ;
}
```

编译 gcc strarray.c -o strarray。

执行 ./strarray，执行结果如下:

```
input string:
hello
hello
cmp=0
```

3.1.5 冒泡法排序

冒泡法排序实例是 C 语言中的一个经典算法，实现多个数值的排序。方法是多次循环进行比较，每次比较时将最大数移动到最上面。每次循环时，找出剩余变量里的最大值，然后减小查询范围。这样经过多次循环以后，就完成了对这个数组的排序。冒泡法排序使用了

反复循环和比较的算法，执行了下面这些步骤。

① 在第一次循环时，拿第一个数与第二个数进行比较，如果第一个数小于第二个数，就用一个中间变量使这两个数交换。这样就使第一和第二个数从大到小排列。

② 再用第一个数与第三个数比较，使这两个数从大到小排列。用同样的方法，用第一个数与后面所有的数进行比较。

③ 经过了第一轮循环比较以后，第一个数一定是所有数里面最大的数。

④ 进行第二轮比较，这时让第二个数与后面所有的数比较，使这个数是这些数里面的最大数。

⑤ 用循环的方法，依次用后面的一个数与这个数后面的所有数进行比较，这样就完成了这些数的从大到小排序。

图 6-1 是 5 个变量的冒泡法排序原理图，其排序流程如上所述。

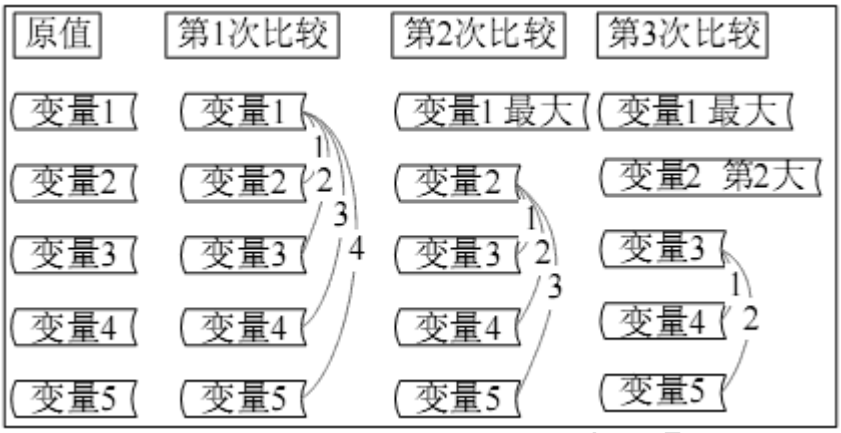


图 6-1 冒泡法排序图

下面的程序是冒泡法排序的例子，冒泡法是通过两次循环比较实现的。

airbubb.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int a[10];
    int i,j,temp;        /*定义循环变量和中间变量*/
    printf("please enter a number:\n");   /*输出提示*/
    for(i=0;i<10;i++)
    {
        scanf("%d",&a[i]);   /*进行循环输入变量*/
    }
    for(i=0;i<10;i++)
    {
        for(j=i+1;j<10;j++)
        {
            if(a[i]<a[j])
            {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
```

```
            }
        }
        for(i=0;i<10;i++)
        {
            printf("%d   ",a[i]);
        }
        return 0 ;
}
```

编译 gcc airbubb.c -o airbubb。

执行 ./airbubb，执行结果如下：

```
please enter a number:
3 9 1 900 100 800 700 600 90 0
900   800   700   600   100   90   9   3   1   0
```

3.2 C语言结构体

3.2.1 结构概念

1．结构存在的意义

存在是合理的，许多事物的存在是在不断解决问题引入的，当然有更好的方法出现时改变也是合理的。在实际问题中，一组数据往往具有不同的数据类型。例如，在学生登记表中，姓名应为字符型，学号可为整型或字符型，年龄应为整型，性别应为字符型，成绩可为整型或实型。显然不能用一个数组来存放这一组数据，因为数组中各元素的类型和长度都必须一致，以便于编译系统处理。为了解决这个问题，C 语言中给出了另一种构造数据类型—“结构（structure）”或叫“结构体”，它相当于其他高级语言中的记录。“结构”是一种构造类型，它是由若干“成员”组成的，每一个成员可以是一个基本数据类型或者又是一个构造类型。结构既是一种“构造”而成的数据类型，那么在说明和使用之前必须先定义它，也就是先构造它，如同在声明和调用函数之前要先定义函数一样。

定义一个结构的一般语法形式如下：

```
struct  结构名
{
    成员表列
};
```

成员表列由若干个成员组成，每个成员都是该结构的一个组成部分。对每个成员也必须作类型说明，其形式如下：

```
    类型说明符  成员名;
```

成员名的命名应符合标识符的书写规定。结构定义举例说明如下：

```
struct stu
{
    int num;
    char name[20];
    char sex;
    float score;
};
```

在这个结构定义中，结构名为 stu，该结构由 4 个成员组成。第一个成员是 num，为整型变

量; 第二个成员是 name, 为字符数组; 第三个成员是 sex, 为字符变量; 第四个成员是 score, 为实型变量。应注意在括号后的分号是不可缺少的。结构定义之后, 即可进行结构变量定义, 凡定义为结构 stu 的变量都由上述 4 个成员组成。由此可见，结构是一种复杂的数据类型, 是数目固定，类型不同的若干有序变量的集合。

2．结构概念小结

结构是一种构造数据类型。

结构的用途是把不同类型的数据组合成一个整体，是自定义数据类型。

结构类型定义语法形式如下:

```
struct       [结构名]
{
        类型标识符      成员名;
        类型标识符      成员名;
        ……
}[变量名 1, 变量名 2……];
```

定义结构变量方法为：struct　结构名　变量名。

结构变量引用规则为：结构变量不能整体引用，只能引用变量成员，引用方式为结构变量名.成员名，可以将一个结构变量赋值给另一个结构变量，结构嵌套时需要逐级引用。

3.2.2 结构变量

1. 结构变量有以下三种定义方法

(1) 先定义结构，再定义结构变量

```
struct stu
 {
        int num;
        char name[8];
        char sex;
        float score;
 };
 struct stu boy1,boy2;
```

(2) 在定义结构类型的同时定义结构变量

```
struct stu
{
        int num;
        char name[8];
        char sex;
        float score;
}boy1,boy2;
```

(3) 直接定义结构变量

```
struct
{
        int num;
        char name[8];
        char sex;
```

```
    float score;
}boy1,boy2;
```

第三种方法与第二种方法的区别在于第三种方法中省去了结构名，而直接给出结构变量。三种方法中说明的 boy1、boy2 变量都具有下图 6-2 所示的结构。

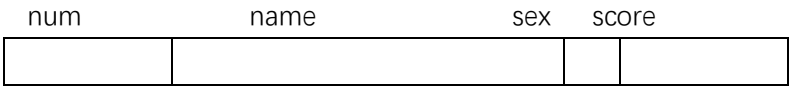


图 6-2 结构变量组成图

2. 结构变量内存布局

所有 C 语言中的变量经过编译后都会转换为对内存线性地址的操作。各种变量的引入是面向人类思维进行的，方便人们高效的编程，而 C 语言源代码经过 gcc 编译后，形成执行码时，所有变量与符号失去意义，变量与符号转变为对相应内存地址的操作。int 类型代表着 4 个字节相应地址的内存空间的抽象，而结构变量也是代表着一片内存空间的抽象（结构名不代表内存地址），所以相同类型结构变量与 int 类型变量一样可以用=号直接赋值(如 boy2=boy1)，两个结构变量用=号赋值代表两片相同大小空间的复制拷贝。

内存的物理地址是线性的，一维的，而结构变量往往代表二维或多维。这种二维或多维结构是方便人们编程时对变量进行管理，是一种面向人们思维的逻辑结构，但结构变量最终物理表现在内存中还是一维线性结构。

下面以上面定义的 boy1 来说明结构变量在内存中存储方式。表 6-4 列出了结构变量 boy1 的内存布局，此时 boy1 代表着-19 到 0 这片内存空间的抽象。&boy1 代表结构变量的地址，即为 0。结构成员（如 boy1.num）其实也代表着对应地址内存空间的抽象，对结构成员的操作编译器最后会转化为对相应内存地址的操作。

表 6-4 boy1 结构变量内存布局表

假定栈空间起始地址为 0		
内存地址	内存	说明
0	boy1.num	
-1		
-2		
-3		
-4	boy1.name[7]	
-5	boy1.name[6]	
-7	boy1.name[5]	
-8	boy1.name[4]	
-9	boy1.name[3]	
-10	boy1.name[2]	
-11	boy1.name[1]	
-12	boy1.name[0]	body.name 表示内存地址-12
-13	boy1.sex	
-14		float 为 4 个字节，需要-16 处才能对齐，所以空两个字节
-15		
-16	boy1.score	
-17		
-18		

-19		

3. 结构变量成员的引用

表示结构变量成员引用的一般形式如下:

```
结构变量名.成员名
```

例如:

```
boy1.num          即第一个人的学号
boy2.sex          即第二个人的性别
```

如果成员本身又是一个结构则必须逐级找到最低级的成员才能引用。

4. 结构变量的赋值

结构变量的赋值就是给各成员赋值，可用输入语句或赋值语句来完成。

【例 6-4】给结构变量赋值并输出其值。

stru1.c 源代码如下:

```
#include <stdio.h>
int main()
{
    struct stu
    {
        int num;
        char name[20];
        char sex;
        float score;
    } boy1,boy2;
    boy1.num=102;
    boy1.name="Zhang ping";
    printf("input sex and score\n");
    scanf("%c %f",&boy1.sex,&boy1.score);
    boy2=boy1;
    printf("Number=%d\nName=%s\n",boy2.num,boy2.name);
    printf("Sex=%c\nScore=%f\n",boy2.sex,boy2.score);
    return 0 ;
}
```

编译 gcc stru1.c -o stru1。

执行 ./stru1，执行结果如下:

```
input sex and score
M 90
Number=102
Name=Zhang ping
Sex=M
Score=90.000000
```

5. 结构变量的初始化

【例 6-5】对结构变量初始化。

stru2.c 源代码如下：

```
#include <stdio.h>
int main()
{
    struct stu
    {
        int num;
        char *name;
        char sex;
        float score;
    }boy2,boy1={102,"Zhang ping",'M',78.5};
    boy2=boy1;
    printf("Number=%d\nName=%s\n",boy2.num,boy2.name);
    printf("Sex=%c\nScore=%f\n",boy2.sex,boy2.score);
    return 0 ;
}
```

编译 gcc stru2.c -o stru2。

执行 ./stru2，执行结果如下：

```
Number=102
Name=Zhang ping
Sex=M
Score=78.500000
```

6. 结构数组的定义

结构数组的定义与初始化方法如下：

```
struct stu
{
    int num;
    char name[20];
    char sex;
    float score;
}boy[5]={
    {101,"Li ping","M",45},
    {102,"Zhang ping","M",62.5},
    {103,"He fang","F",92.5},
    {104,"Cheng ling","F",87},
    {105,"Wang ming","M",58},
};
```

当对全部元素作初始化赋值时，也可不给出数组长度。

引用结构数组成员的方法如下：

```
结构[下标].成员变量
如 boy[0].num, boy[1].name
```

3.3 指针

指针，C 语言的精华，它在 C 语言中，表现得最优秀也最危险。

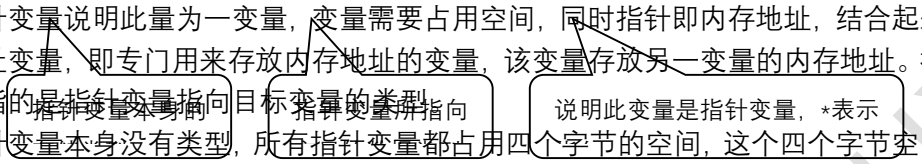
3.3.1 指针概念

1. 指针概述

内存中每个字节有一个编号，即地址。

变量是对数据存储内存空间的抽象，一般变量（如 int 等）是对变量的直接访问，而指针变量是对变量的间接访问。

指针变量说明此量为一变量，变量需要占用空间，同时指针即内存地址，结合起来就是内存地址变量，即专门用来存放内存地址的变量，该变量存放另一变量的内存地址。指针变量类型指的是指针变量指向目标变量的类型。



指针变量本身没有类型，所有指针变量都占用四个字节的空间，这个四个字节空间里存放的是内存地址。如 int *p1、char *p2、double *p3 中的指针变量 p1、p2、p3 都占用四个字节的内存空间。

理解指针变量指向的变量，int *p1 中 p1 为指针变量，*p1 为指针变量指向的变量。指针变量只能指向定义时所规定类型的变量，此时 p1 只能指向 int 型变量，即给 p1 赋值时只能赋 int 型变量的地址。指针变量定义后，变量值不确定，应用前必须先赋值。

&与*运算符互为逆运算，*表示取指针变量所指向变量的内容，&表示取变量的地址。假设 i_pointer 为指针变量，它的内容是地址量，*i_pointe 为指针变量指向的变量，它的内容是数据，&i_pointer 为指针变量本身内存的地址。

存放变量地址的变量是指针变量。int *p1 说明 p1 是指针变量，该指针变量名为 p1，p1 的值为内存地址。*p1 是 p1 所指向的变量，int 说明指针变量指向变量的类型，即*p1 的类型，而不是 p1 的类型。

数组名是表示数组首地址，是地址常量。

数组名作函数参数，是地址传递，数组名和指针变量作为函数传递参数时可以通用，数组名作为形参时被调用函数将数组名当做指针变量处理。

2. 指针变量与其所指向的变量之间的关系

图 6-3 画出了指针变量与其所指向变量之间的关系图，⇔符号表示等价，其中变量 i 的地址为 2000，i_pointer 为指针变量，*i_pointer 为指针变量所指向的变量。

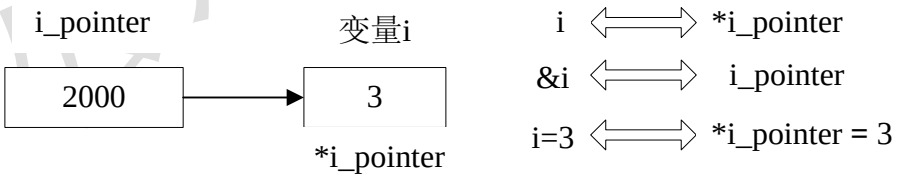


图 6-3 指针变量与其所指向变量之间的关系图

3. 指针变量定义

图 6-4 画出了指针变量定义图，说明了定义时各字段的含义。

```
[存储类型]          数据类型          *指针变量名
```

图 6-4 指针变量定义图

指针变量定义说明:

```
int *p1;
float *p2;
```

对上述指针变量，具体解释如下:

指针变量名是 p1、p2，不是*p1、*p2。

指针变量只能指向定义时所规定类型的变量。p1 只能指向 int 变量的地址，p2 只能指向 float 变量的地址。

指针变量定义后，变量值不确定，使用前必须先赋值。

4. 指针的引用

指针变量的赋值只能赋予地址，决不能赋予任何其他数据，否则将引起错误。在 C 语言中，变量的地址是由编译系统分配的，对用户完全透明，用户不知道变量的具体地址。与指针有如下两个有关的运算符:

```
① &：取地址运算符。
② *：指针运算符（或称“间接访问”运算符）。
```

C 语言中提供了地址运算符&来表示变量的地址，其一般形式为：&变量名；如&a 表示变量 a 的地址，&b 表示变量 b 的地址，变量本身必须预先进行定义。

设有指向整型变量的指针变量 p，如要把整型变量 a 的地址赋予 p 可以有以下两种方式:

(1) 指针变量初始化的方法如下

```
int a;
int *p=&a;
```

(2) 对指针变量赋值的方法如下

```
int a;
int *p;
p=&a;
```

5. 指针引用图解

指针变量和一般变量一样，存放在它们之中的值是可以改变的，也就是说可以改变它们的指向，假设

```
char i,j,*p1,*p2;
i='a';
j='b';
p1=&i;
p2=&j;
```

上述语句建立了如下图 6-5 所示的联系:

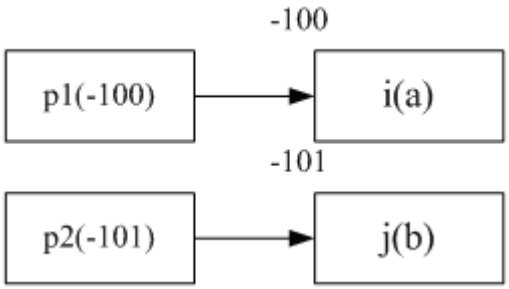


图 6-5 指针引用图 1

在这里-100 为 i 的内存地址（即&i），-101 为 j 的内存地址(即&j)。

赋值 p2=p1，就使 p2 与 p1 指向同一对象 i，此时*p2 就等于 i，而不是 j。如下图 6-6 所示:

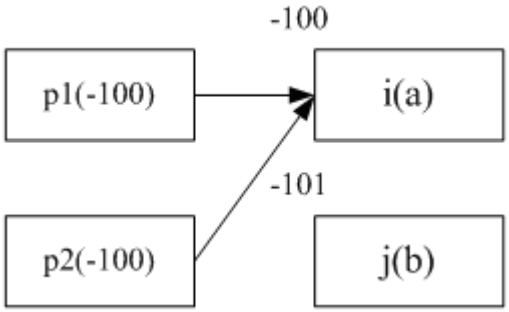


图 6-6 指针引用图 2

如果执行如下表达式 *p2=*p1，则表示把 p1 指向的内容赋给 p2 所指的内存区域，此时就变成下图 6-7 所示:

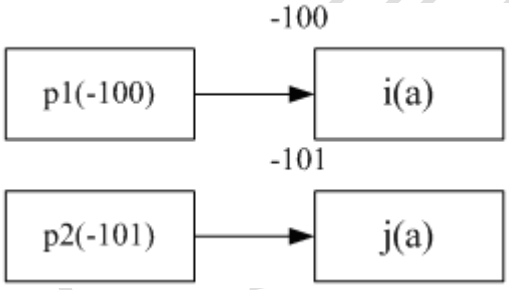


图 6-7 指针引用图 3

表 6-5 列出了上述变量在栈空间上的内存布局，当 p2=p1，就是把 p2 的值改为-100，而*(-100)代表变量 i。当*p2=*p1，即*(-101)=*(-100)，是对两个内存地址指向的空间进行赋值，相当于 j=i，而 p2 本身的值并没有改变。*p1 是地址-100 内存空间的抽象，*p2 是地址-101 内存空间的抽象，而 p1 是地址-104~-107 内存空间的抽象，p2 是地址-108~-111 内存空间的抽象。

表 6-5 上述变量在栈空间上的内存布局表

假设变量的堆栈起始地址为-100,()中代表变量的值		
内存地址	内存	说明
-100	i(a)	
-101	j(b)	

-102		由于整型变量需要对齐，地址必须被 4 整除，指针本身编译器把它当做整型数据处理，所以空两个字节
-103		
-104	p1 (-100)	此时 p1 等于-100，而-100 是变量 i 的内存地址，*(-100)代表指向内存地址-100 的变量，即变量 i
-105		
-106		
-107		
-108	p2 (-101)	
-109		
-110		
-111		

6. 理解指针变量

假设

```
int a=3, *p1;
char *p2, aa='X';
p1=&a ;
p2=&aa ;
```

表 6-6 列出了上述变量在内存中的布局并给出其详细说明。

表 6-6 指针变量内存布局表

假设变量的堆栈起始地址为 3000，()中为变量的值		
内存地址	内存	说明
3000	a (3)	变量是相应地址内存空间的抽象，变量 a 是内存地址 3000~2997 这片内存空间的抽象，其值初始化等于 3，int 型说明此变量占用空间大小为 4 个字节。&a 等于变量 a 的内存地址，即 3000
2999		
2998		
2997		
2996	p1 (3000)	p1 是一个指针变量(即地址变量)，保存的值为 3000。*p1 是指针变量 p1 所指向的变量，此时*p1 等于*3000，即地址 3000 所指向的变量（即内存地址 3000 所存的变量值，也就是变量 a），*p1 此时可以与变量 a 划等号，值为 3
2995		
2994		
2993		
2992	p2 (2888)	char *p2 说明指针变量指向的变量类型为 char，p2 等于 2888，是变量 aa 的内存地址，*p2 等于*2888，即变量 aa，其值为 X
2991		
2990		
2889		
2888	aa(X)	aa 是一个 char 类型的变量，占用一个字节的内存空间

对上述指针变量的解释如下:

指针变量是一个特殊的变量，它里面存储的数值是内存里的一个地址。不管指针变量前面的类型如何，32 位机器中指针变量都一律占用内存空间 4 个字节，这 4 个字节存放的就是其他变量的内存地址。所以指针变量可以理解成内存地址变量，*可理解为指向。

理解指针变量和指针变量指向的变量，上述 p1、p2 是指针变量，*p1、*p2 是指针变量指向的变量。

理解指针变量的值和指针变量指向变量的值，指针变量的值为 p1 等于 3000，p2 等于 2888。指针变量指向变量的值即内存地址指向变量的值，*p1 即*3000，*3000 等于变量 a，即 3；*p2 即*2888，*2888 等于变量 aa，即字符 X。

理解指针变量类型，通常所说指针变量类型是指针变量所指向变量的类型。*p1 代表 p1 指向的类型为 int 型，占用 4 个字节，*p2 代表 p2 指向的类型为 char 型，占用 1 个字节的内存空间。

理解指针变量本身类型，所有指针变量本身占用 4 个字节，存放内存地址。本身类型在编程中毫无意义，可以理解成无类型，也可以理解成是 4 个字节的整型数据。

int a 说明 a 是整型变量，其 a 有两重限定含义，其一为变量，其二类型为整型，两者合在一起即为整型变量；char aa 说明 aa 是字符变量。int *p 中的*说明 p 是指针变量，指针是变量的限定词，说明此变量的类型是指针（内存地址），int 说明此指针变量所指向变量的类型为 int。

通过指针变量是对目标变量的间接操作。例如我们找小强，可以通过直接操作找小强；也可以通过间接操作，先找到小强的爸，通过小强爸的指引找到小强。上述 a=3 是的直接操作，而 p1=&a、*p1=3 则是间接操作，虽然两者结果相同，但过程不同。直接操作是直接在相应内存地址上读写数据，而间接操作则需两次或多次读写内存，间接操作是 CPU 首先从内存中读到变量地址，再根据变量地址从内存中读写数据。

3.3.2 sizeof、void、const 说明

1. sizeof 运算符

sizeof 运算符是 C 语言的关键字，用于求一个对象所占用的字节数。使用 sizeof 运算符计算对象大小是一个良好的编程习惯。

【例 6-6】利用 sizeof 更深刻的理解指针变量。

sizeof.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int *p_int ;
    char *p_char ;
    float *p_float ;
    double *p_double ;
    int var_int ;
    char var_char ;
    float var_float ;
    double var_double ;
    struct stu
    {
        int num;
        char name[8];
        char sex;
        float score;
    } ;
    struct stu *p_str, var_str ;
    p_int=&var_int ;
    p_char=&var_char ;
    p_float=&var_float ;
    printf("p_int=%d, *p_int=%d\n", sizeof(p_int), sizeof(*p_int) ) ;
```

```
    printf("p_char=%d, *p_char=%d\n", sizeof(p_char), sizeof(*p_char) ) ;
    printf("p_float=%d, *p_float=%d\n", sizeof(p_float), sizeof(*p_float) ) ;
    printf("p_double=%d, *p_double=%d\n", sizeof(p_double), sizeof(*p_double) ) ;
    printf("p_str=%d, *p_str=%d\n", sizeof(p_str), sizeof(*p_str) ) ;
    printf("int length=%d\n", sizeof(int) ) ;
    printf("char length=%d\n", sizeof(char) ) ;
    printf("float length=%d\n", sizeof(float) ) ;
    printf("double length=%d\n", sizeof(double) ) ;
    printf("struct stu length=%d\n", sizeof(struct stu) ) ;
    return 0 ;
}
```

编译 gcc sizeof.c -o sizeof。

执行 ./sizeof，执行结果如下：

```
p_int=4, *p_int=4
p_char=4, *p_char=1
p_float=4, *p_float=4
p_double=4, *p_double=8
p_str=4, *p_str=20
int length=4
char length=1
float length=4
double length=8
struct stu length=20
```

从上面运行结果可以看出，指针变量无论类型是什么，都只占用四个字节的内存空间，而指针变量指向的变量就是变量类型定义的大小。

2. void 类型指针

void 表示无类型，通常用来修饰指针变量，使用时需要强制转换。

```
void *p ;
p=(int *)malloc(sizeof(int)*100) ;
```

3. const 关键字说明

类型声明中 const 用来修饰一个常量。修饰时有以下情况，请读者理解掌握。

```
const int nValue;   //nValue 是 const
const char *pContent; //*pContent 是 const, pContent 可变
const (char *) pContent;//pContent 是 const,*pContent 可变
char* const pContent; //pContent 是 const,*pContent 可变
const char* const pContent; //pContent 和*pContent 都是 const
```

3.3.3 指针变量作为函数参数

【例 6-7】指针变量作函数参数的运行机理

swap_p.c 源代码如下：

```
#include <stdio.h>
int swap(int *p1,int *p2)
{
    int temp;
```

```
    temp=*p1;
    *p1=*p2;
    *p2=temp;
    return 0 ;
}
int main()
{
    int a,b;
    int *pointer_1,*pointer_2;
    scanf("%d,%d",&a,&b);
    pointer_1=&a;pointer_2=&b;
    if(a<b) swap(pointer_1,pointer_2);
    printf("%d,%d\n",a,b);
    return 0 ;
}
```

编译 gcc swap_p.c -o swap_p。

执行 ./swap_p，执行结果如下：

```
80,90
90,80
```

对上述程序的说明：

变量的赋值只不过是把内存一个地址上的值复制到另一个地址而已。

形参 pointer_1 传给形参 p1 后，由于 pointer_1 的值为&a，所以在 swap 函数中对*p1 的操作，其实是对*(&a)的操作，即对 a 的操作。

一个程序中指针变量可以指向该执行空间堆栈段任何内存地址，所以即使在 swap 函数中，也能达到对该程序中任何变量进行操作，只要形参传入该变量地址即可完成对该地址上变量的操作。如*p1 操作的是 main 函数中的 a 变量，因为 p1 的值等于 a 变量的地址。

指针变量可以指向执行程序（进程）整个堆栈空间，指针变量的值（内存地址）不正确或越界会造成程序执行错误或 coredump，指针变量的值不正确或越界是常见的编程错误。

表 6-7 列出了 swap_p 程序运行内存堆栈空间表，此表可以更好的理解指针作函数形参时的调用原理，假设栈段起始值为 30000，下表圆括号()中的值是变量的值。

表 6-7 swap_p 程序运行时内存堆栈表

数据段说明	内存地址	说明
栈段	30000	main 函数返回值存放空间
	29996	a(80)　此时 a 等于 80，&a 等于 29996，*(&a)等于 a
	29992	b(90)
	29988	pointer1(29996)
	29984	pointer2(29992)
	29980	swap 函数返回值存放空间（0）
	29976	p1 (29996)
	29972	p2 (29992)
	29968	temp
堆段	……	……
bss 段	……	……

静态数据段	.....	.....
代码段		1. swap 函数地址入口 2. 在栈顶分配整型变量 temp 空间 3. 执行 temp=*p1，即 temp=*29996==a==80 4. 执行*p1=*p2，即*29996=*29992，即*(&a)=*(&b)，即 a=b==90，即 a=90 5. 执行*p2=temp，即*29992=80，即*(&b)=80，即 b=80 6. 返回 0 给函数返回值空间，即内存地址 29980~29977 处 7. 函数结束返回。 8. main 函数地址入口 9. 在栈顶分配 main 函数返回值、a、b、pointer1、pointer2 变量空间 10. 从屏幕上读取两个数字分别存放到 a、b 变量的内存空间里，此时&a 等于内存地址 29996，而 a 代表内存地址 29996~29993 内存空间抽象，读入变量 a 的值存放到内存地址 29996~29993 的内存空间里。变量 b 同理 11. 执行 pointer_1=&a，即 pointer_1=29996 12. 执行 pointer_2=&b，即 pointer_2=29992 13. 如果 a<b，调用 swap(pointer_1,pointer_2)函数，在栈顶分配 swap 返回值空间、p1、p2 变量空间，并将实参 pointer_1 的值复制给 p1，实参 pointer_2 的值复制给 p2。然后即跳转到 1(swap 函数入口)的地方去执行 14. 打印 a、b 的值到屏幕上 15. 返回值 0 写入到 main 返回值存放空间，即内存地址 30000~29997 内存空间处 16. 程序执行完成，释放所有内存空间

3.3.4 指针的运算

1. 加减算术运算

对于指向数组的指针变量，可以加上或减去一个整数 n。设 pa 是指向数组 a 的指针变量，则 pa+n、pa-n、pa++、++pa、pa--、--pa 运算都是合法的。

指针变量的加减运算只能对数组指针变量进行，对指向其他类型变量的指针变量作加减运算是毫无意义的。

指针变量加减偏移量与其前面定义类型密切相关，int 类型指针变量加 1 地址偏移 4 位，char 类型指针变量加 1 偏移 1 位。

【例 6-8】指针变量加减运算地址的偏移量

p_airth.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int x[10], *px ;
    char y[10], *py ;
```

```
    px=x ;
    py=&y[0] ;
    printf("px=%d, py=%d\n", px, py) ;
    px++ ;
    py++ ;
    printf("px1=%d, py1=%d\n", px, py) ;
    return 0;
}
```

编译 gcc p_airth.c -o p_airth。

执行 ./p_airth，执行结果如下：

```
px=-1074644316, py=-1074644266
px1=-1074644312, py1=-1074644265
```

2. 两个指针变量之间的运算

只有指向同一数组的两个指针变量之间才能进行运算，否则运算毫无意义。两个指针变量之间的运算有下面两种情况。

① 两指针变量相减：两指针变量相减所得之差除以指针变量指向变量的类型长 度是两个指针所指数组元素之间相差的元素个数。

```
int a[10],*p1, *p2 ;
p1=&a[0];
p2=&a[7];
```

此时(p2-p1)/4 等于 7，因为 int 占用 4 个字节，数组在内存中是顺序存储的。

② 两指针变量进行关系运算：指向同一数组的两指针变量进行关系运算可表示它们所指数组元素之间的关系。

例如：

pf1==pf2 表示 pf1 和 pf2 指向同一数组元素。

pf1>pf2 表示 pf1 处于高地址位置。

pf1<pf2 表示 pf2 处于低地址位置。

指针变量还可以与 0 比较，设 p 为指针变量，则 p==0 表明 p 是空指针，它不指向任何变量，p!=0 表示 p 不是空指针，空指针是对指针变量赋予 0 值而得到的。

例如：

```
#define NULL 0
int *p=NULL;
```

对指针变量赋 0 值和不赋值是不同的。指针变量未赋值时，可以是任意值，是不能当右值使用的，否则将造成意外错误。而指针变量赋 0 值后，则是安全的，它不指向任何具体的变量，指针变量不使用时应赋 0 值。

3. 指针的赋值运算

指针的赋值运算有下面 6 种情形。

① 指针变量初始化赋值，前面已作介绍。

② 把一个变量的地址赋予指向相同数据类型的指针变量。

```
例如：int a,*pa;
pa=&a;      /*把整型变量 a 的地址赋予整型指针变量 pa*/
```

③ 把一个指针变量的值赋予指向相同类型变量的另一个指针变量。

```
如：  int a,*pa=&a,*pb;
```

```
pb=pa;      /*把 a 的地址赋予指针变量 pb*/
```

由于 pa、pb 均为指向整型变量的指针变量，因此可以相互赋值。

④ 把数组的首地址赋予指向数组的指针变量。

```
例如：int a[5],*pa;
      pa=a; /*数组名表示数组的首地址，故可赋予指向数组的指针变量 pa*/
也可写为：pa=&a[0];  /*数组第一个元素的地址也是整个数组的首地址，也可赋予 pa*/
```

当然也可采取初始化赋值的方法：

```
int a[5],*pa=a;
```

⑤ 把字符串的首地址赋予指向字符类型的指针变量。

```
例如：  char *pc;
   pc="C Language"; /*编译程序先开辟静态数据区存放字符串，然后指针变量 pc 指向它*/
```

或用初始化赋值的方法写为：

```
      char *pc="C Language";
此时相当于  static char st[]={"C Language"}, *pc=&st ;
```

⑥ 把函数的入口地址赋予指向函数的指针变量。

```
例如：
      int (*pf)();
      pf=f;       /*f 为函数名*/
```

其实函数也有其入口地址，指针变量指向函数地址入口，就代表对此函数的执行。

3.3.5 指向数组的指针变量

1.数组与指针的关系

在许多程序员眼里，数组与指针有说不清、道不明的差别，剪不断、理还乱的关系。它们俩什么时候是等价，什么时候又不等价，为什么许多时候可以等价使用，又有一些时候判若两人呢？为了揭开它们俩差别的神秘面纱，还得从 CPU 这个关键人物说起，CPU 看待变量，只不过是一段内存空间的抽象而已。数组变量也好，指针变量也罢，最终都会转化对内存地址的操作，对于数组和指针，是编译器让这两位兄弟在右值时等价是编译器让数组与指针让这两位兄弟在右值时等价。又因为数组名是地址常量（数组名代表数组首地址，不能改变，编译时确定），而指针变量是地址变量（有 4 个字节的内存空间，是变量理所当然执行时可以不断改变），所以在左值时只能有指针变量可以使用，而数组名是地址常量，不占内存空间，当然就不能被其他变量或常量对其赋值。

一个变量有一个地址，一个数组包含若干元素，每个数组元素都在内存中占用一定的存储单元，它们都有相应的内存地址，而且数组元素是按顺序占用连续的内存空间。所谓数组的指针是指数组的起始地址，数组元素的指针是指某个数组元素的地址。

2.一级指针变量与一维数组的关系

假设定义: int *p 与 int q[10], 下面详细说明两者的联系与区别, “⇔”表示两者等价。

数组名是指针（地址）常量。

当 p=q 时，p+i 是 q[i]的地址。

数组元素的表示方法有下标法和指针法两种, 若 p=q, 则 p[i]⇔q[i] ⇔ *(p+i) ⇔*(q+i) 。

形参是数组时实质上是形参变量是指针变量，即 int q[]⇔int *q，调用时该形参变量占用 4 个字节，存放数组的首地址。

在定义指针变量（不是形参）时，不能把 int *p 写成 int p[]。

系统只给 p 分配 4 字节的内存区(32 位机器)，而给 q 分配 4*10 字节的内存区。

3.指针变量与数组名的等价使用

如果指针变量 p 已指向数组中的一个元素，则 p+1 指向同一数组中的下一个元素。引入指针变量后，就可以用下标法和指针法两种方法来访问数组元素了。

```
int a[10], p=&a[0];
```

此时 p 的初值为&a[0]，则:

1) p+i 和 a+i 就是 a[i]的地址，或者说它们指向 a 数组的第 i 个元素。

图 6-8 画出了指针变量与数组名等价使用图。

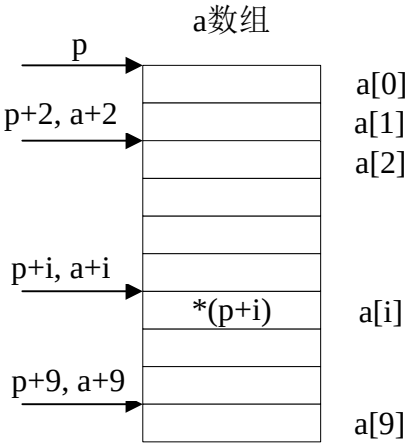


图 6-8 指针变量与数组名等价使用图

2) *(p+i)或*(a+i)就是 p+i 或 a+i 所指向的数组元素，即 a[i]。

例如，*(p+5)或*(a+5)就是 a[5]。

3) 指向数组的指针变量也可以带下标，如 p[i]与*(p+i)等价。

根据以上叙述，引用一个数组元素可以有如下两种方法:

① 下标法：用a[i]形式访问数组元素。

② 指针法：若p=a，可采用*(a+i)或*(p+i)形式访问数组元素，其中a是数组名，p是指向数组的指针变量。

注意指针变量可以实现本身值的改变，如 p++是合法的，而 a++是错误的。因为 a 是数组名，是地址常量，不能进行自加减运算。

【例 6-9】指针变量与数组名的等价使用

p_array.c 源代码如下:

```
#include <stdio.h>
int main()
{
    int a[10]={0,1,2,3,4,5,6,7,8,9} ;
    int *pa ;
    pa=a ;
    printf("a[9]=%d\n", a[9]) ;
    printf("pa[9]=%d\n", pa[9]) ;
    printf("*(a+9)=%d\n", *(a+9)) ;
    printf("*(pa+9)=%d\n", *(pa+9)) ;
    return 0 ;
```

```
}
```

编译 gcc p_array.c -o p_array。

执行 ./p_array，执行结果如下：

```
a[9]=9
pa[9]=9
*(a+9)=9
*(pa+9)=9
```

从上面的执行结果可以看出，这四种表示法在右值时是等价的。假设 a 代表地址 3000，这四种表示法经编译过后都是*(3000+9)，都代表 a[9]。

3.3.6 数组名作函数参数

1.形参是数组的本质特征

形参是数组时实质上形参变量是指针变量，即 int q[]⇔int *q，调用时该形参变量 q 占用 4 个字节，存放数组的首地址。

```
f(int x[],int n)等同于 f(int *x,int n)
```

2.形参是数组名时与指针可等价使用

fun_array.c 源代码如下：

```
#include <stdio.h>
// void inv(int *x, int n)
void inv(int x[],int n)     /*形参 x 是数组名*/
{
      int temp,i,j,m=(n-1)/2;
      for(i=0;i<=m;i++)
      {j=n-1-i;
            temp=x[i];x[i]=x[j];x[j]=temp;}
     //           temp=*(x+i); *(x+i)=*(x+j); *(x+j)=temp; }
            return;
}
int main()
{int i,a[10]={3,7,9,11,0,6,7,5,4,2};
      printf("The original array:\n");
      for(i=0;i<10;i++)
            printf("%d,",a[i]);
      printf("\n");
      inv(a,10);
      printf("The array has benn inverted:\n");
      for(i=0;i<10;i++)
            printf("%d,",a[i]);
      printf("\n");
      return 0 ;
}
```

编译 gcc fun_array.c -o fun_array。

执行 ./fun_array，执行结果如下：

```
The original array:
3,7,9,11,0,6,7,5,4,2,
The array has benn inverted:
2,4,5,7,6,0,11,9,7,3,
```

可以看出，数组名作形参时可以与指针变量互换。

```
此时 void inv(int *x, int n)与 void inv(int x[],int n)等价
    temp=x[i];x[i]=x[j];x[j]=temp 与 temp=*(x+i); *(x+i)=*(x+j); *(x+j)=temp 等价
```

上面两两组合可构成四种情形。

3.多维数组数组名含义

若 int a[3][4]，图 6-9 画出了二维数组 a 的地址，每个方格中是多维数组元素的地址，此时 a、a[0]、a[1]、a[2]都是地址常量，编译时确定，其值说明如下。

```
a==a[0]==&a[0][0]==1000
a+1==a[1]==&a[1][0]==1008
a+2==a[2]==&a[2][0]==1016
```

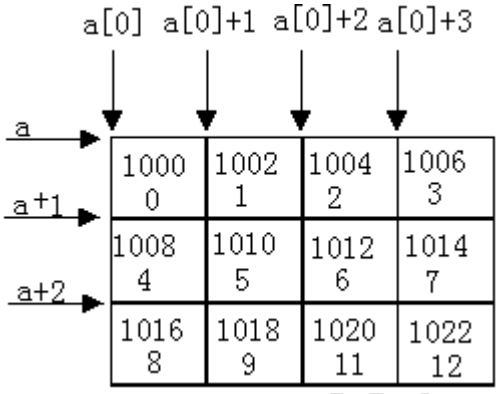


图 6-9 多维数组地址图

3.3.7 函数指针变量

在 C 语言中，一个函数总是占用一段连续的内存区，而函数名就是该函数所占内存区的首地址。可以把函数的这个首地址(或称入口地址)赋予一个指针变量，使该指针变量指向该函数，然后通过指针变量就可以找到并调用这个函数。这种指向函数的指针变量称为“函数指针变量”。

函数指针变量定义的一般形式如下:

```
类型说明符  (*指针变量名)();
```

其中“类型说明符”表示被指函数的返回值类型。“(* 指针变量名)”表示定义的是指针变量，最后的空圆括号表示指针变量所指的是一个函数。

例如:

```
int (*pf)();
```

pf 是一个指向函数入口的指针变量，该函数的返回值(函数值)是整型。

【例 6-10】函数指针变量举例

p_fun.c 源代码如下:

```
#include <stdio.h>
int max(int a,int b)
{
```

```
    if(a>b)return a;
    else return b;
}
int main()
{
    int max(int a,int b);
    int(*pmax)();
    int x,y,z;
    pmax=max;
    printf("input two numbers:\n");
    scanf("%d%d",&x,&y);
    z=(*pmax)(x,y);
    printf("maxmum=%d\n",z);
    return 0 ;
}
```

编译 gcc p_fun.c -o p_fun。

执行 ./p_fun，执行结果如下：

```
input two numbers:
2000 3000
maxmum=3000
```

3.3.8 返回指针类型函数

所谓函数类型是指函数返回值的类型。在 C 语言中允许一个函数的返回值是一个指针(即地址)，这种返回指针值的函数称为指针型函数。

定义指针型函数的一般形式如下：

```
类型说明符 *函数名(形参表)
{
    ……      /*函数体*/
}
```

其中函数名之前加了“*”号表明这是一个指针型函数，即返回值是一个指针，类型说明符表示返回的指针变量所指向的数据类型。

【例 6-11】本程序是通过指针函数，输入一个 1～7 之间的整数，输出对应的星期名。

p_week.c 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i;
    char *day_name(int n);
    printf("input Day No:\n");
    scanf("%d",&i);
    if(i<0) exit(1);
    printf("Day No:%2d-->%s\n",i,day_name(i));
    return 0 ;
```

```
}
char *day_name(int n){
    static char *name[]={ "Illegal day",
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"};
    return((n<1||n>7) ? name[0] : name[n]);
}
```

编译 gcc p_week.c -o p_week。

执行 ./p_week，执行结果如下：

```
input Day No:
3
Day No: 3-->Wednesday
```

3.3.9 指向指针的指针

如果一个指针变量存放的又是另一个指针变量的地址，则称这个指针变量为指向指针的指针变量。

通过指针访问变量称为间接访问。由于指针变量直接指向变量，所以称为“单级间址”，而如果通过指向指针的指针变量来访问变量则构成“二级间址”，图 6-10 画出二维指针变量实现图。

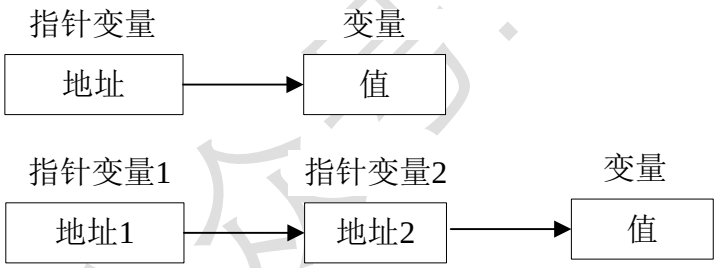


图 6-10 二维指针变量实现图

【例 6-12】一个指针数组元素指向数据的简单例子。

p2_array.c 源代码如下：

```
#include <stdio.h>
int   main()
{
     static int a[5]={1,3,5,7,9};
    int *num[5]={&a[0],&a[1],&a[2],&a[3],&a[4]};
    int **p,i;
    p=num;
    for(i=0;i<5;i++)
    {
        printf("%d\t",**p);
```

```
            p++;
        }
        printf("\n") ;
        return 0 ;
}
```

编译 gcc p2_array.c -o p2_array。

执行 ./p2_array，执行结果如下：

```
    1       3       5       7       9
```

3.3.10 结构指针

1.结构指针变量定义

一个指针变量用来指向一个结构变量时，称之为结构指针变量。结构指针变量中的值是所指向的结构变量首地址，通过结构指针即可访问该结构变量，这与数组指针和函数指针的情况是相同的。

结构指针变量定义语法形式如下：

```
struct 结构名 *结构指针变量名
```

2.结构作函数形参特点

用结构变量的成员作形参或结构变量作形参都为值传递，效率低，用结构指针变量作参数为地址传递。

结构指针变量也是地址变量，占用 4 个字节的内存空间，存放的是结构变量的内存地址。结构变量代表一片内存空间的抽象，与 int 等直接变量类似，所以结构变量作形参是值传递，结构指针变量作形参是地址传递。

3.指针变量指向结构变量

```
struct stu
{
        int num;
        char name[30];
 };
struct stu girl;
struct stu *pstu;
pstu=&girl;
```

有了结构指针变量，就能更方便地访问结构变量的各个成员。

结构指针变量访问结构成员语法形式如下：

```
(*结构指针变量).成员名
或为:   结构指针变量->成员名
```

此时(*pstu).num 等价于 pstu->num。

4. 结构指针变量作形参

p_struct.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
struct student
{
    char name[20];
    int age;
```

```
    int sex;
    int height;
};
void showstu(struct student *p)
{
    printf("A student:\n");
    printf("    Name    : %s \n",p->name);
    printf("    Age     : %d \n",p->age);
    printf("    Sex     : %d \n",p->sex);
    printf("    Height: %d \n\n",p->height);

}
void main()
{
    struct student stu1;
    struct student *p1;
    p1=&stu1;
    stu1.age=17;
    stu1.sex=1;
    stu1.height=176;
    strcpy(stu1.name,"Jim");
    showstu(p1);
}
```

编译 gcc p_struct.c -o p_struct。

执行 ./p_struct，执行结果如下：

```
A student:
   Name    : Jim
   Age     : 17
   Sex     : 1
   Height: 176
```

3.3.11 动态存储分配

变量空间是在栈上分配，动态存储分配是在堆上分配。动态内存分配需要用 free 函数释放空间，没有用 free 释放会造成内存泄露。

1.动态存储函数原型

申请内存空间函数有 malloc、calloc、realloc 三个函数，这里主要介绍 malloc 函数，另外两个函数将在 Linux 进程编程章节加以介绍，这三个函数申请的内存空间需要用 free 函数来释放。

malloc 函数和 free 函数的函数原型如下：

malloc（动态申请内存空间）	
所需头文件	#include <stdlib.h>
函数说明	动态申请内存空间
函数原型	void *malloc(size_t size)
函数传入值	size：申请内存空间的大小

函数返回值	成功：返回申请内存空间的首地址
	失败：NULL

free（释放原先申请内存空间）	
所需头文件	#include <stdlib.h>
函数说明	参数 ptr 为指向先前由 malloc()、calloc()或 realloc()所返回的内存指针，调用 free()后 ptr 所指的内存空间便会被收回
函数原型	void free(void *ptr)
函数传入值	ptr：申请内存空间的首地址

2.动态存储函数举例

malloc.c 源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    struct stu
    {
        int num;
        char *name;
        char sex;
        float score;
    }  *ps;
    ps=(struct stu*)malloc(sizeof(struct stu));
    ps->num=102;
    ps->name="Zhang ping";
    ps->sex='M';
    ps->score=62.5;
    printf("Number=%d\nName=%s\n",ps->num,ps->name);
    printf("Sex=%c\nScore=%.2f\n",ps->sex,ps->score);
    free(ps);
    return 0 ;
}
```

编译 gcc malloc.c -o malloc。

执行 ./malloc，执行结果如下：

```
Number=102
Name=Zhang ping
Sex=M
Score=62.50
```

3.3.12 指针链表

链表是通过系统不断申请内存，然后通过结构体中的指针变量将所申请的空间一级一级链接起来。

以学生链表为例，下面是链表的结构体定义。

```
struct stu
{
        int num;
        int score;
        struct stu *next;
}
```

图 6-11 为一简单链表的示意图，通过指针变量把各节点链在一起。

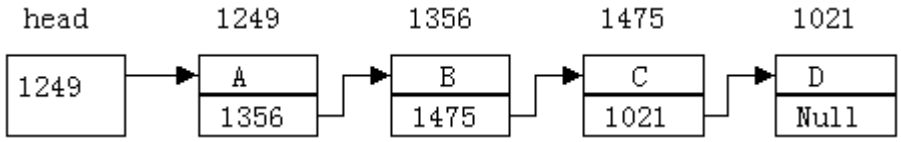


图 6-11 指针链表图

图 6-11 中，第 0 个结点称为头结点，它存放第一个结点的首地址，它没有数据，只是一个指针变量。以下的每个结点都分为两个域，一个是数据域，存放各种实际的数据，如成绩 score 等；另一个域为指针域，存放下一结点的首地址。链表中的每一个结点都是同一种结构类型。

对链表的主要操作有以下几种:
① 建立链表;
② 结构体的查找与输出;
③ 插入一个结点;
④ 删除一个结点。

【例 6-13】建立一个三个结点的链表，存放学生数据，假定学生数据结构中只有学号和年龄两项，可编写一个建立链表的函数 creat。

creat.c 源代码如下:

```
#define NULL 0
#define TYPE struct stu
#define LEN sizeof (struct stu)
struct stu
{
    int num;
    int age;
    struct stu *next;
};
TYPE *creat(int n)
{
    struct stu *head,*pf,*pb;
    int i;
    for(i=0;i<n;i++)
    {
        pb=(TYPE*) malloc(LEN);
        printf("input Number and   Age\n");
        scanf("%d%d",&pb->num,&pb->age);
        if(i==0)
```

```
            pf=head=pb;
        else pf->next=pb;
        pb->next=NULL;
        pf=pb;
    }
    return(head);
}
```

3.3.13 指针数据类型小结

表 6-8 列出了指针数据类型，并对其含义进行了说明。

表 6-8 指针数据类型小结表

定义	含　　义
int i	定义整型变量 i
int *p	p 为指向整型数据的指针变量
int a[n]	定义整型数组 a，它有 n 个元素
int *p[n]	定义指针数组 p，它由 n 个指向整型数据的指针元素组成
int (*p)[n]	p 为指向含 n 个元素的一维数组的指针变量
int f()	f 为带回整型函数值的函数
int *p()	p 为带回一个指针的函数，该指针指向整型数据
int (*p)()	p 为指向函数的指针，该函数返回一个整型值
int **p	p 是一个指针变量，它指向一个指向整型数据的指针变量

第4章 C语言预处理

所谓预处理是指在进行编译的第一遍扫描(词法扫描和语法分析)之前所作的工作。预处理是 C 语言的一个重要功能，它由预处理程序负责完成。当对一个源文件进行编译时，系统将自动引用预处理程序对源程序中的预处理部分作处理，处理完毕自动进入对源程序的编译。

4.1 define宏定义

C 语言提供了多种预处理功能，如宏定义、文件包含、条件编译等。

在 C 语言源程序中允许用一个标识符来表示一个字符串，称为“宏”。被定义为“宏”的标识符称为“宏名”。在编译预处理时，对程序中所有出现的“宏名”，都用宏定义中的字符串去代换，这称为“宏代换”或“宏展开”。

宏定义是由源程序中的宏定义命令完成的。宏代换是由预处理程序自动完成的。在 C 语言中，“宏”分为有参数和无参数两种。

C 语言对宏定义用#define 命令来实现，注意宏定义后字符串需要用圆括号括起来，防止宏代换时产生歧义和隐含错误。

1. 无参宏定义

无参宏定义的一般形式如下:

```
#define  标识符  字符串
```

如：#define M (y*y+3*y)，可用#undef M 解除定义。

2. 有参宏定义

有参宏定义的一般形式如下:

```
#define   宏名(形参表)   字符串
```

上述宏定义在字符串中含有各个形参。

有参宏调用的一般形式如下:

```
宏名(实参表);
```

例如:

```
#define M(y) y*y+3*y          /*宏定义*/
   ……
k=M(5);                          /*宏调用*/
……
```

在宏调用时，用实参 5 去代替形参 y，经预处理宏展开后的语句为:

```
k=5*5+3*5
```

4.2 typedef重定义

typedef 为 C 语言的关键字，作用是为一种数据类型定义一个新名字。这里的数据类型包括内部数据类型（int、char 等）和自定义的数据类型（struct 等）。

在编程中使用 typedef 目的一般有两个，一个是给变量一个易记且意义明确的新名字，另一个是简化一些比较复杂的类型声明。

1. 给已知数据类型 long 起个新名字，叫 byte_4，定义方式如下:

```
typedef long byte_4;
```

2. 结构体重定义

结构体重定义有下面两种方式:

```
struct tagMyStruct
{
   int iNum;
   long lLength;
};
typedef struct tagMyStruct MyStruct;
```

这里 MyStruct 实际上相当于 struct tagMyStruct，可以使用 MyStruct varName 来定义变量。

```
typedef struct{
      char plat_no[3+1];
      char plat_name[60+1];
   } T_PLAT_PARA;
T_PLAT_PARA   s_para ;
```

可以用 T_PLAT_PARA s_para 来定义变量 s_para。

4.3 inline关键字

1. inline关键字说明

inline 为把函数替换为函数展开语句，展开在汇编阶段开始。函数的展开是由编译器决定的，这一点对程序员而言是透明的。只有代码很短的情况下，函数才会被展开，递归调用函数不会被展开。如果过度地使用 inline 关键字，编译器将不会展开函数，以防止代码体积的恶性膨胀。

2. inline关键字举例

假设有如下内联函数:

```
inline int add(int a, int b)
{
    return a+b ;
}
int c ;
```

当调用 c=add(1,9)时函数语句展开变为 c=1+9。

4.4 条件编译

1. 条件编译的三种方式

预处理程序提供了条件编译的功能。可以按不同条件去编译不同的程序部分，因而产生不同的目标代码文件，这对于程序的移植和调试是很有用的。

条件编译有三种形式，具体说明如下。

(1) 第一种形式

```
#ifdef  标识符
  程序段 1
#else
  程序段 2
#endif
```

它的功能是，如果标识符已被#define 命令定义过则对程序段 1 进行编译；否则对程序段 2 进行编译。如果没有程序段 2(它为空)，格式中的#else 可以没有，即可以写为:

```
#ifdef  标识符
程序段
 #endif
```

用“#define 标识符”来表明对标识符进行了定义，如“#define XXX”表明对标识符 XXX 进行了定义。

(2) 第二种形式

```
 #ifndef 标识符
    程序段 1
 #else
    程序段 2
 #endif
```

与第一种形式的区别是将“ifdef”改为“ifndef”。它的功能是，如果标识符未被#define 命

令定义过则对程序段 1 进行编译，否则对程序段 2 进行编译。这与第一种形式的功能正相反。

如果防止某标识符被重复定义，实现方法如下:

```
#ifndef INADDR_NONE
#define INADDR_NONE        0xffffffff
#endif
```

(3) 第三种形式

```
#if 常量表达式
    程序段 1
#else
    程序段 2
#endif
```

它的功能是，如常量表达式的值为真(非 0)，则对程序段 1 进行编译，否则对程序段 2 进行编译。

2. #if与if的区别

#if 是在编译时起作用，进行条件编译；而 if 是在程序运行时起作用，进行条件判断。

3. #if可经常使用的场合

我在项目中，经常使用#if 0 注释掉程序中一段语句，用#if 1 打开程序中一段语句，这比/* */注释更清晰更好使用，因为/* */不能嵌套。使用方法如下:

```
#if 0
    程序段
#endif
```

4.5 头文件的使用

1. 防止头文件重复包含

假设头文件名为 somefile.h，可在头文件名前后加__，并将头文件名大写，然后按如下方式定义防止头文件重复包含问题。

```
#ifndef __SOMEFILE_H__
#define __SOMEFILE_H__
    ... ... // 声明、定义语句
#endif
```

2. 头文件的使用建议及说明

系统头文件用<>符号进行包含，自定义头文件用" "符号进行包含。

建议将全局函数和类型定义集中于一个头文件中，方便程序引用；将常用的系统头文件集中在一个自定义的头文件中，方便程序引用。

没有正确包含库函数头文件有时编译不通过；有时编译通过，但会产生警告信息；有时编译通过连警告信息都不产生;没有正确包含头文件时少量时候代码执行时会产生莫名其妙的错误。

第5章 格式化I/O函数

格式化 I/O 函数分为输出函数和输入函数两大类，输入和输出格式是编程应该掌握的细节，同时也是编程时经常需要使用到的知识。

5.1 格式化输出函数

5.1.1 输出函数原型

格式化 I/O 输出函数原型如下:

```
#include <stdio.h>
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);

#include <stdarg.h>
int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

具体说明如下:

返回值：上述函数成功返回格式化输出的字节数（不包括字符串的结尾'\0'），出错返回一个负值，错误原因存在于 error 中。

printf 函数会把格式化串打印到标准输出。

fprintf 函数会把格式化串输出到指定的文件 stream 中。

sprintf 函数会把格式化串输出到缓冲区 str 中，并在末尾加'\0'，当 str 空间不够时，会造成缓冲区溢出。

snprintf 函数会把格式化串输出到缓冲区 str 中，并在末尾加'\0'，但格式化串长度超过 size-1 字节时，对格式化串按长度 size-1 进行截断，因此 snprintf 函数比 sprintf 函数使用起来更加安全。

上面列出的后四个函数在前四个函数名的前面多了个 v，表示可变参数，不是以...的形式传进来，而是以 va_list 类型传进来。

5.1.2 输出函数格式说明

1. format格式说明

```
%(flags)(width)(.prec)type
```

以上圆括号括起来的参数为选择性参数，而%与 type 则是必要的。下面是 format 各参数的详细说明。

type 选项说明	
选项	说明
整数	
d	整数的参数会被转成一有符号的十进制数字
u	整数的参数会被转成一无符号的十进制数字

o	整数的参数会被转成一无符号的八进制数字
x	整数的参数会被转成一无符号的十六进制数字，并以小写 abcdef 表示
X	整数的参数会被转成一无符号的十六进制数字，并以大写 ABCDEF 表示浮点型数
f	double 型的参数会被转成十进制数字，并取到小数点以下六位，四舍五入
e	double 型的参数以指数形式打印，有一个数字会在小数点前，六位数字在小数点后，而在指数部分会以小写的 e 来表示
E	与%e 作用相同，唯一区别是指数部分将以大写的 E 来表示
g	double 型的参数会自动选择以%f 或%e 的格式来打印，其标准是根据欲打印的数值及所设置的有效位数来决定
G	与%g 作用相同，唯一区别在以指数形态打印时会选择%E 格式
字符串	
c	整型数的参数会被转成 unsigned char 型打印出
s	指向字符串的参数会被逐字输出，直到出现 NULL 字符为止
p	如果是参数是“void *”型指针则使用十六进制格式显示

pre 选项说明
① 正整数的最小位数
② 在浮点型数中代表小数位数
③ 在%g 格式代表有效位数的最大值
④ 在%s 格式代表字符串的最大长度
⑤ 若为×符号则代表下个参数值为最大长度

width 选项说明
width 为参数的最小长度，若此栏并非数值，而是*符号，则表示以下一个参数当做参数长度

flag 选项说明	
选项	说明
#	此旗标会根据其后转换字符的不同而有不同含义。当在类型为 o 之前（如%#o），则会在打印八进制数值前多印一个 o。而在类型为 x 之前（%#x）则会在打印十六进制数前多印'0x'，在型态为 e、E、f、g 或 G 之前则会强迫数值打印小数点。在类型为 g 或 G 之前时则同时保留小数点及小数位数末尾的零
u	一般在打印负数时，printf（）会加印一个负号，整数则不加任何负号。此旗标会使得在打印正数前多一个正号（+）
o	整数的参数会被转成一无符号的八进制数字
0	当有指定参数时，无数字的参数将补上 0。默认是关闭此标记，所以一般会打印出空白字符
-	格式化后的内容居左，右边可以留空格

2. 输出字符串中字符类型说明

参数 format 字符串可包含下列 3 种字符类型:

① 一般文本，伴随直接输出。
② ASCII控制字符，如\t、\n等。
③ 格式转换字符。

格式转换为一个百分比符号(%)及其后的格式字符所组成。一般而言，每个%符号在其后都必需有一参数与之相呼应（只有当%%转换字符出现时会直接输出%字符）。

3. 常用格式化输出说明

格式化输出常用格式为：%(+|-|0)m.n。
对格式化输出常用格式解释如下：
① m：输出数据域宽，数据长度<m，左补空格，否则按实际输出。
② .n：对实数，指定小数点后位数(四舍五入)；对字符串，指定字符串实际输出位数，超过指定长度则进行截断。
③ -：输出数据在域内左对齐（默认右对齐）。
④ +：指定在有符号数的正数前显示正号(+)。
⑤ 0：输出数值时指定左边不使用的空位置自动填 0。
⑥ 输出百分号需要用两个%%。

4. 常用格式化输出使用举例

在实际应用编程中，常用的格式化输出有下面 6 种。
① 数字前补 0：sprintf(acStr,"%06dSECCTL ",30)。
② 左对齐：sprintf(acStr,"%-6.6s","05023")。
③ 右对齐：sprintf(acStr,"%6.6s","05023")。
④ 两位小数点输入：sprintf(acStr,"%.2f",19.79)。
⑤ 增加输出百分号：sprintf(acStr,"%%.2f",19.79)。
⑥ 指定最长输出位数：sprintf(acStr,"%6.6s","testTEST")。

5. 输出函数应用举例

printf.c 代码如下：

```
#include <stdio.h>
int main()
{
	int i = 150;
	int j = -100;
	double k = 3.14159;
	printf("%d %f %x\n",j,k,i);
	printf("%010d|%-6d|%4d|%*d\n",i,i,i,2,i); /*参数 2 会代入格式*中，而与%2d 同意义*/
	return 0 ;
}
```

编译 gcc printf.c –o printf。
执行 ./printf，执行结果如下：

```
-100 3.141590 96
0000000150|150    | 150|150
```

变参使用范例，了解即可。
vprintf.c 代码如下：

```
#include <stdio.h>
#include <stdarg.h>
int my_printf( const char *format,...)
{
```

```
    va_list ap;
    int retval;
    va_start(ap,format);
    printf("my_printf( ):");
    retval = vprintf(format,ap);
    va_end(ap);
    return retval;
}
int main()
{
    int i = 150,j = -100;
    double k = 3.14159;
    my_printf("%d %f %x\n",j,k,i);
    my_printf("%2d %*d\n",i,2,i);
    return 0 ;
}
```

编译 gcc vprintf.c –o vprintf。

执行 ./vprintf，执行结果如下:

```
my_printf( ):-100 3.141590 96
my_printf( ):150 150
```

5.2 格式化输入函数

5.2.1 输入函数原型

格式化 I/O 输入函数原型如下:

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

具体说明如下:

返回值: 上述函数返回成功匹配和赋值的参数个数，成功匹配的参数可能少于所提供的赋值参数，返回 0 表示一个都不匹配，出错返回-1 并设置 errno。

scanf 函数会将输入的数据根据参数 format 字符串来转换并格式化数据。

fscanf 函数从指定的文件 stream 中读字符。

sscanf 函数从指定的字符串 str 中读字符。

5.2.2 输入函数格式说明

1. format类型

```
%[*][size][l][h]type
```

以上圆括号括起来的参数为选择性参数，而%与 type 则是必要的。具体说明如下:

- *代表该对应的参数数据忽略不保存。
- size 为允许参数输入的数据长度。
- l 输入的数据数值以long int 或double型保存。
- h 输入的数据数值以short int 型保存。

type 选项说明	
选项	说明
d	输入的数据会被转换成一有符号的十进制数字（int）
i	输入的数据会被转换成一有符号的十进制数字，若输入数据以“0x”或“0X”开头代表转换十六进制数字，若以“0”开头则转换八进制数字，其他情况代表十进制
o	输入的数据会被转换成一无符号的八进制数字
u	输入的数据会被转换成一无符号的正整数
x	输入的数据为无符号的十六进制数字，转换后存于 unsigned int 型变量
X	同%x
f	输入的数据为有符号的浮点型数，转换后存于 float 型变量
e	同%f
E	同%f
g	同%f
s	输入数据为以空格字符为终止的字符串
c	输入数据为单一字符
[]	读取数据但只允许括号内的字符。如[a-z]
[^]	读取数据但不允许中括号的^符号后的字符出现，如[^0-9]

2. 输入字符串中字符类型说明

格式化输入字符串中字符类型包括如下 3 种:

① 空格或Tab，在处理过程中被忽略。
② 普通字符（不包括%），和输入字符中的非空白字符相匹配，输入字符中的空白字符是指空格、Tab、\r、\n、\v、\f。
③ 格式转换是以%开头，以转换字符结尾，中间有若干个可选项。

3. scanf函数特别说明

scanf 函数使用格式为 scanf("格式控制串"，地址表)，输入变量默认间隔是空格，如果用“,”号做间隔，输入时也要输入“,”号。

4. 输入函数应用举例

scanf.c 源代码如下:

```
#include <stdio.h>
int main()
{
	int i;
	unsigned int j;
	char s[5];
	scanf("%d %x %5[a-z] %*s %f",&i,&j,s,s);
	printf("%d %d %s\n",i,j,s);
	return 0 ;
}
```

编译 gcc scanf.c -o scanf。

执行 ./scanf，执行结果如下:

```
输入：  10 0x1b aaaaaaaaaa bbbbbbbbbb
```

```
10 27 aaaaa
```

5. 变参实用项目实例

下面是一个打印源代码文件名称、报错位置和信息的实用日志函数，具体说明如下:

程序中 syslog 函数打印信息长度不限，报错参数个数和信息不限，并打印该进程进程号。

日志位置存放在自定义的目录下，日志文件名称可自定义。

日志文件按天生成，打印函数自动检测日志文件是否存在，如不存在建立相应的日志文件。

变参一般都需使用 va_start 和 va_end 函数，编程模式类似，读者可模仿掌握。

syslog.c 源代码如下:

```
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdarg.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
/*-------------------------------------------------------------------------
 * Function Name : syslog
 * Description    : 写日志
 * Input             : sysname      --   系统代号
 *                     : modname      --   功能模块名
 *                     : file_name   --   文件名
 *                     : line_num     --   行号
 *                     : format       --   消息
 * Output           :
 * Return           : success:0
 *                     : fail    :-1
 *-------------------------------------------------------------------------*/
int syslog( char *sysname, char *modname,char *file_name, int line_num, char *format, ...   )
{
    va_list ap ;
    struct tm *ts;
    time_t now;
    char acTimeStr[16] ;
    char filename[128] ;
    char syscmd[256] ;
    FILE *LogFileID ;
    memset( filename, 0x00, sizeof( filename ) );
    now = time(NULL);
    ts = localtime( &now );
```

```
    sprintf( filename, "%s/log/%s/%s.%02d%02d.log",
                getenv("HOME"), sysname, modname, ts->tm_mon+1, ts->tm_mday );
    LogFileID=fopen( filename, "a" );
    if ( LogFileID == NULL )
    {
        sprintf( syscmd, "/bin/mkdir -p %s/log/%s/", getenv("HOME"), sysname );
        if ( system( syscmd ) != 0 )
        {
            fprintf(stderr,"cread log folder fail [%s]", syscmd );
            return -1;
        }
        LogFileID=fopen( filename, "a" );
        if ( LogFileID == NULL )
        {
            fprintf(stderr,"open file fail [%s]", filename );
            return -1 ;
        }
    }
    sprintf( acTimeStr, "%02d-%02d %02d:%02d:%02d",
                ts->tm_mon+1, ts->tm_mday, ts->tm_hour, ts->tm_min, ts->tm_sec);
    fprintf( LogFileID, "[%s %d %s:%d] ", acTimeStr, getpid(), file_name, line_num );
    va_start(ap, format );
    vfprintf( LogFileID, format, ap );
    va_end( ap );
    fputc( 0x0a, LogFileID );
    fflush( LogFileID );
    fclose( LogFileID );
    return 0 ;
}
int main()
{
    syslog( "newSys", "SOCK", __FILE__, __LINE__, "hello world! %s %d", "what who why where when how",
999 );
    syslog( "newSys", "SOCK", __FILE__, __LINE__, "every day is new day!");
    return 0;
}
```

编译 gcc syslog.c –o syslog。

执行 ./syslog，在$HOME/log/newSys 下产生了 SOCK.0116.log 文件，文件内容如下：

```
[01-16 19:51:04 11921 syslog.c:66] hello world! what who why where when how 999
[01-16 19:51:04 11921 syslog.c:67] every day is new day!
```

第6章 字符串和内存操作函数

6.1 字符串操作函数说明

对一串字符的处理在应用编程中无处不在,其操作函数主要有两类:一类以 str 开头的函数,主要针对字符串进行处理；一类为 mem 开头的函数，针对一片内存进行处理，此类函数可以处理字符串和结构体。

6.1.1 字符串操作函数总结说明

1. str与mem的含义

str 开头的函数为字符串函数，字符串函数遇到'\0'结束符即终止操作，字符串最后一个字符即为'\0'字符。

mem 开头的函数为一片内存操作函数，函数原型中包括处理长度，表示对一片内存空间的处理，这片内存中可以包含'\0'字符。

2. 会对目的串自动补’\0’字符的函数

常见有 strcpy、sprintf、gets、strcat 等不指定长度的字符串函数。

3. strcpy与memcpy的比较

对 strcpy、memcpy 两函数差别说明如下:

① strcpy用于字符串的复制，strcpy碰到源串'\0'字符即终止复制。
② memcpy为一片内存空间的复制，复制时可以包括'\0'字符，一片内存空间或结构体变量之间的复制常用此函数。
③ strcpy(dest,src)表示把源字符串src复制到目的串dest中，当dest空间小于src长度时会造成内存溢出，从而让程序coredump。
④ memcpy (dest，src, n)表示复制src所指的内存内容前n个字节到dest所指的内存地址上。如果是字符串复制可以用strncpy代替，不过strncpy遇到'\0'字符会终止复制，可能复制不到n个字节。
⑤ 两结构体变量之间的复制只能用memcpy函数。

4. strcpy与strncpy的比较

对 strcpy、strncpy 两函数差别说明如下:

① 用strcpy函数复制时目的串后会自动补 0，用strncpy函数复制时目的串后不会自动补 0。
② strncpy(dest, src, n)表示从src源串复制n个字节到dest目的串，但碰到'\0'字符时会提前终止复制。
③ 用strncpy函数时,有时需要给目的串设置结束符('\0'),给一串字符设置'\0'的方式有两种,分别为string[n]=0 或string[n]= '\0'。因为字符可以当做整型单独赋值，0 表示ASCII编号的 0;也可以当做字符赋值,字符赋值时特殊字符需用\来转义。字符'0'的ASCII编号为 30,而字符'\0'的ASCII编号为 0。

6.2 字符串函数操作

字符串函数是编程中常用到的函数，字符串按照使用用途可分为字符串初始化、取字符串长度、复制字符串、连接字符串、比较字符串、搜索字符串、分割字符串七大类。

1. 字符串初始化

（1）函数原型

memset（将一段内存空间填入某值）	
所需头文件	#include <string.h>
函数说明	memset()会将参数 s 所指的内存区域前 n 个字节以参数 c 填入，然后返回指向 s 的指针。在编写程序时，若需要将某一数组作初始化，memset()会相当方便
函数原型	void * memset (void *s ,int c, size_t n)
函数传入值	s：字符串地址
	c：初始化字符
	n：初始化字符串长度
函数返回值	返回指向 s 的指针
附加说明	参数 c 虽声明为 int，但必须是 unsigned char，所以范围在 0 到 255 之间
使用场合	① 初始化字符串 ② 初始化结构体 ③ 初始化 malloc 申请的内存空间

（2）memset 函数说明

memset 函数完成内存一片空间的初始化，对此函数使用方法说明如下：

① 字符串初始化：memset(string,0x00,sizeof(string))，string表示字符数组。

② 结构变量初始化：memset(&struct,0x00,sizeof(&struct))，struct表示结构变量。因为结构变量表示一片内存空间的抽象，需要用&得到结构变量地址。

③ 字符串和结构变量使用前最好用memset函数进行初始化。

（3）memset 应用代码范例

memset.c 代码如下：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    char s[30];
    char *p ;
    struct stu
    {
        int num;
        char name[20];
        char sex;
        float score;
    };
    struct stu boy1;
    p=(char *)malloc(1024);
    memset (s, 0x00, sizeof(s));
    memset(p, 0x00, 1024) ;
```

```
    memset(&boy1, 0x00, sizeof(struct stu)) ;
    return 0 ;
}
```

2. 取字符串长度

（1）函数原型

strlen（返回字符串长度）	
所需头文件	#include <string.h>
函数说明	strlen()用来计算指定的字符串 s 的长度，不包括结束字符'\0'
函数原型	size_t strlen (const char *s)
函数传入值	s：字符串首地址
函数返回值	返回字符串 s 的字符数

（2）应用代码范例

strlen.c 代码如下：

```
#include <stdio.h>
#include <string.h>
int main()
{
    char *str = "12345678";
    printf("str length = %d\n", strlen(str));
    return 0 ;
}
```

编译 gcc strlen.c -o strlen。

执行 ./strlen，执行结果如下：

```
str length = 8
```

3. 复制字符串

（1）函数原型

strcpy（复制字符串）	
所需头文件	#include <string.h>
函数说明	strcpy()会将参数 src 字符串复制至参数 dest 所指的地址
函数原型	char *strcpy(char *dest,const char *src)
函数传入值	dest：目的串地址
	src：源串地址
函数返回值	返回参数 dest 的字符串起始地址
附加说明	如果参数 dest 所指的内存空间不够大，可能会造成缓冲溢出(buffer Overflow)的错误情况，在编写程序时请特别留意，或者用 strncpy()来取代

strncpy（根据长度复制字符串）	
所需头文件	#include <string.h>
函数说明	strncpy()会将参数 src 字符串复制前 n 个字符至参数 dest 所指的地址
函数原型	char * strncpy(char *dest,const char *src,size_t n)
函数传入值	dest：目的串地址

	src：源串地址
	n：复制的字符数
函数返回值	返回参数 dest 的字符串起始地址

memcpy（复制内存内容）	
所需头文件	#include <string.h>
函数说明	memcpy()用来复制 src 所指的内存内容前 n 个字节到 dest 所指的内存地址上。与 strcpy()不同的是，memcpy()会完整的复制 n 个字节，不会因为遇到字符串结束'\0'而结束
函数原型	void * memcpy (void * dest ,const void *src, size_t n)
函数传入值	dest：目的串地址
	src：源串地址
	n：复制字节数
函数返回值	返回指向 dest 的指针
附加说明	① 指针 src 和 dest 所指的内存区域不可重叠 ② 结构体之间的复制需要用 memcpy，而不能用 strcpy

memmove（复制内存内容）	
所需头文件	#include <string.h>
函数说明	memmove()与 memcpy()一样都是用来复制 src 所指的内存内容前 n 个字节到 dest 所指的地址上。不同的是，当 src 和 dest 所指的内存区域重叠时，memmove()仍然可以正确的处理，不过执行效率上会比使用 memcpy()略慢些
函数原型	void * memmove(void *dest,const void *src,size_t n)
函数传入值	dest：目的串地址
	src：源串地址
	n：复制字节数
函数返回值	返回指向 dest 的指针

（2）应用代码

strcpy.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[30]="string(1)";
    char b[]="string(2)";
    printf("before strcpy() :%s\n",a);
    printf("after strcpy() :%s\n",strcpy(a,b));

    strncpy( &a[9], b, 9) ;
    a[18]='\0' ;
    printf("after strncpy():%s\n", a) ;
    return 0 ;
```

```
}
```

编译 gcc strcpy.c -o strcpy。

执行 ./strcpy，执行结果如下：

```
before strcpy() :string(1)
after strcpy() :string(2)
after strncpy():string(2)string(2)
```

memcpy.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
int main()
{
	char a[30];
	char b[30]="string\0string";
	int i;

	memset( a, 0x00, sizeof(a) ) ;
	strcpy(a,b);
	printf("strcpy():", a);
	for(i=0;i<30;i++)
		printf("%c",a[i]);
	printf("\n") ;
	memcpy(a,b,30);
	printf("memcpy():");
	for(i=0;i<30;i++)
		printf("%c",a[i]);
	printf("\n") ;
	return 0 ;
}
```

编译 gcc memcpy.c -o memcpy。

执行 ./memcpy，执行结果如下：

```
strcpy():string
memcpy():stringstring
```

4. 连接字符串

（1）函数原型

strcat（连接两字符串）	
所需头文件	#include <string.h>
函数说明	strcat()会将参数 src 字符串复制到参数 dest 所指的字符串尾。第一个参数 dest 要有足够的空间来容纳要复制的字符串
函数原型	char *strcat (char *dest,const char *src)
函数传入值	dest：目的串地址
	src：源串地址
函数返回值	返回参数 dest 的字符串起始地址

附加说明	如果参数 dest 所指的内存空间不够大，可能会造成缓冲溢出(buffer Overflow)的错误情况，在编写程序时请特别留意。

strncat（根据长度连接两字符串）	
所需头文件	#include <string.h>
函数说明	strncat()会将参数 src 字符串复制 n 个字符到参数 dest 所指的字符串尾。第一个参数 dest 要有足够的空间来容纳要复制的字符串
函数原型	char * strncat(char *dest,const char *src,size_t n)
函数传入值	dest：目的串地址
	src：源串地址
	n：复制的字符数
函数返回值	返回参数 dest 的字符串起始地址
附加说明	strncat 与 strncpy 不同，系统会对目的串结尾处自动补 0

（2）应用代码范例

strcat.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[30]="string(1)";
    char b[]="string(2)";
    printf("before strcat(): %s\n",a);
    printf("after strcat(): %s\n",strcat(a,b));
    printf("after strncat(): %s\n",strncat(a,b,6));
    return 0 ;
}
```

编译 gcc strcat.c -o strcat。

执行 ./strcat，执行结果如下：

```
before strcat(): string(1)
after strcat(): string(1)string(2)
after strncat(): string(1)string(2)string
```

5. 比较字符串

（1）函数原型

strcmp（比较字符串）	
所需头文件	#include <string.h>
函数说明	strcmp()用来比较参数 s1 和 s2 字符串。字符串大小的比较是以 ASCII 码表上的顺序来决定，此顺序亦为字符的值。strcmp()首先将 s1 第一个字符值减去 s2 第一个字符值，若差值为 0 则再继续比较下个字符，若差值不为 0 则将差值返回。例如字符串"Ac"和"ba"比较则会返回字符'A'(65)和'b'(98)的差值(-33)
函数原型	int strcmp(const char *s1,const char *s2)
函数传入值	s1：串 1

	s2：串 2
函数返回值	若参数 s1 和 s2 字符串相同则返回 0。s1 若大于 s2 则返回大于 0 的值。s1 若小于 s2 则返回小于 0 的值

strncmp（比较规定长度的字符串）	
所需头文件	#include <string.h>
函数说明	用来将参数 s1 中前 n 个字符和参数 s2 字符串作比较
函数原型	int strcmp(const char *s1,const char *s2, size_t n)
函数传入值	s1：串 1
	s2：串 2
	n：比较字符串的长度
函数返回值	若参数 s1 和 s2 字符串相同则返回 0。s1 若大于 s2 则返回大于 0 的值。s1 若小于 s2 则返回小于 0 的值

memcmp（比较内存内容）	
所需头文件	#include <string.h>
函数说明	memcmp()用来比较 s1 和 s2 所指的内存区间前 n 个字符。字符串大小的比较是以 ASCII 码表上的顺序来决定，次顺序亦为字符的值。memcmp()首先将 s1 第一个字符值减去 s2 第一个字符的值，若差为 0 则再继续比较下个字符，若差值不为 0 则将差值返回。例如，字符串"Ac"和"ba"比较则会返回字符'A'(65)和'b'(98)的差值(-33)
函数原型	int memcmp (const void *s1,const void *s2,size_t n)
函数传入值	s1：串 1
	s2：串 2
	n：比较字符串的长度
函数返回值	若参数 s1 和 s2 字符串相同则返回 0。s1 若大于 s2 则返回大于 0 的值。s1 若小于 s2 则返回小于 0 的值
附加说明	strcmp、strncmp 碰到 0 字符即终止，而 memcmp 比较内容中可以包含 0 字符

忽略大小写比较字符串	
所需头文件	#include <string.h>
函数说明	用来将参数 s1 中前 n 个字符和参数 s2 字符串作比较
函数原型	int strcasecmp(const char *s1, const char *s2) int strncasecmp(const char *s1, const char *s2, size_t n)
函数传入值	s1：串 1
	s2：串 2
	n：比较字符串的长度
函数返回值	若参数 s1 和 s2 字符串相同则返回 0。s1 若大于 s2 则返回大于 0 的值。s1 若小于 s2 则返回小于 0 的值

（2）应用代码范例

strcmp.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
int main()
{
	char *a="aBcDeF";
	char *b="AbCdEf";
	char *c="aacdef";
	char *d="aBcDeF";

	printf("strcmp(a,b) : %d\n",strcmp(a,b));
	printf("strcmp(a,c) : %d\n",strcmp(a,c));
	printf("strcmp(a,d) : %d\n",strcmp(a,d));
	printf("strncmp(a,d,5 ) : %d\n",strncmp(a,d, 5));
	printf("memcmp(a,d,3) : %d\n",memcmp(a,d,3));
	printf("memcmp(a,b) : %d\n",strcasecmp(a,b));
	printf("memcmp(a,b,4) : %d\n",strncasecmp(a,b,4));

	return 0 ;
}
```

编译 gcc strcmp.c -o strcmp。

执行 ./strcmp，执行结果如下：

```
strcmp(a,b) : 1
strcmp(a,c) : -1
strcmp(a,d) : 0
strncmp(a,d,5 ) : 0
memcmp(a,d,3) : 0
memcmp(a,b) : 0
memcmp(a,b,4) : 0
```

6. 搜索字符串

（1）函数原型

strstr（在一字符串中查找指定的字符串）	
所需头文件	#include <string.h>
函数说明	strstr()会从字符串 haystack 中搜寻字符串 needle，并将第一次出现的地址返回。
函数原型	char *strstr(const char *haystack, const char *needle)
函数传入值	haystack：源串
	needle：搜索串
函数返回值	返回指定字符串第一次出现的地址，否则返回 0

strchr（查找字符串中第一个出现的指定字符）	
所需头文件	#include <string.h>
函数说明	strchr()用来找出参数 s 字符串中第一个出现的参数 c 字符的地址，然后将该字符出现的地址返回

函数原型	char * strchr (const char *s,int c)
函数传入值	s：源串
	c：匹配字符
函数返回值	如果找到指定的字符则返回该字符所在地址，否则返回 0

strrchr（查找字符串中最后出现的指定字符）	
所需头文件	#include <string.h>
函数说明	strrchr()用来找出参数 s 字符串中最后一个出现的参数 c 字符的地址，然后将该字符出现的地址返回
函数原型	char * strrchr(const char *s, int c)
函数传入值	s：源串
	c：匹配字符
函数返回值	如果找到指定的字符则返回该字符所在地址，否则返回 0

（2）应用代码范例

strstr.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
int main()
{
    char * s="01234567890123456789012345678890";
    char *p;

    p= strstr(s,"901");
    printf("%s\n",p);
    p=strchr(s,'5');
    printf("%s\n",p);
    p=strrchr(s, '8');
    printf("%s\n",p);

    return 0 ;
}
```

编译 gcc strstr.c -o strstr。

执行 ./strstr，执行结果如下：

```
9012345678901234567890
5678901234567890123456789O
890
```

7. 分割字符串

（1）函数原型

strtok（分割字符串）	
所需头文件	#include <string.h>

函数说明	strtok()用来将字符串分割成一个个片段。参数 s 指向欲分割的字符串，参数 delim 则为分割字符串，当 strtok()在参数 s 的字符串中发现到参数 delim 字符时则会将该字符改为'\0'字符。在第一次调用时，strtok()必需给予参数 s 字符串，往后的调用则将参数 s 设置成 NULL。每次调用成功则返回下一个分割后的字符串指针
函数原型	char * strtok(char *s,const char *delim)
函数传入值	s：源串
	delim：分割符
函数返回值	返回下一个分割后的字符串指针，如果已无从分割则返回 NULL

（2）应用代码范例

strtok.c 源代码如下：

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[] = "root:x::0:root:/root:/bin/bash:";
    char *token;
    token = strtok(str, ":");
    printf("%s\n", token);
    while ( (token = strtok(NULL, ":")) != NULL)
        printf("%s\n", token);
    return 0;
}
```

编译 gcc strtok.c -o strtok。

执行 ./strtok，执行结果如下：

```
root
x
0
root
/root
/bin/bash
```

6.3 字符类型测试函数

字符类型测试用来测试单个字符的类型。

（1）字符类型测试函数原型

头文件	#include <ctype.h>
函数原型	函数说明
int isalnum(int c)	测试字符是否为英文或数字
int isalpha(int c)	测试字符是否为英文字母
int isascii(int c)	测试字符是否为 ASCII 码字符
int iscntrl(int c)	测试字符是否为 ASCII 码的控制字符
int isdigit(int c)	测试字符是否为阿拉伯数字

int isgraph(int c)	测试字符是否为可打印字符(ASCII 码 33-126 之间的字符)
int islower(int c)	测试字符是否为小写字母
int isprint(int c)	测试字符是否为包含空格在内可打印字符（ASCII 码 32-126 之间的字符)
int isspace(int c)	测试字符是否为空格字符
int ispunct(int c)	测试字符是否为标点符号或特殊符号
int isupper(int c)	测试字符是否为大写英文字母
int isxdigit(int c)	测试字符是否为 16 进制数字
返回值	测试正确：返回 TRUE 或大于零的值
	测试错误：NULL(0)

（2）应用代码范例

ischar.c 源代码如下：

```
#include <stdio.h>
#include <ctype.h>
int main()
{
	char str[]="123@#FDsP[e?";
	int i;
	for(i=0;str[i]!=0;i++)
		if(islower(str[i])) printf("%c is a lower-case character\n",str[i]);
	printf("1=%d\n", isxdigit('1')) ;
	printf("2=%d\n", isxdigit('2')) ;
	printf("v=%d\n", isxdigit('v')) ;
	printf("U=%d\n", isupper('U')) ;
	printf("u=%d\n", isupper('u')) ;

	return 0 ;
}
```

编译 gcc ischar.c -o ischar。

执行./ischar，执行结果如下：

```
s is a lower-case character
e is a lower-case character
1=4096
2=4096
v=0
U=256
u=0
```

6.4 字符串转换函数

字符串转换函数完成数字到字符串、字符串到数字的转换功能，字符串转换函数是在应用编程中经常使用到的函数。

（1）字符串转换函数原型

atof（将字符串转换成浮点型数）	
所需头文件	#include <stdlib.h>
函数说明	atof()会扫描参数 nptr 字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时('\0')才结束转换，并将结果返回。参数 nptr 字符串可包含正负号、小数点或 E(e)来表示指数部分，如 123.456 或 123e-2
函数原型	double atof(const char *nptr)
函数传入值	nptr：浮点型指针
函数返回值	返回转换后的浮点型数

atoi（将字符串转换成整型数）	
所需头文件	#include <stdlib.h>
函数说明	atoi()会扫描参数 nptr 字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时('\0')才结束转换，并将结果返回
函数原型	int atoi(const char *nptr)
函数传入值	nptr：短整型指针
函数返回值	返回转换后的整型数

atol（将字符串转换成长整型数）	
所需头文件	#include <stdlib.h>
函数说明	atol()会扫描参数 nptr 字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时('\0')才结束转换，并将结果返回
函数原型	long atol(const char *nptr)
函数传入值	nptr：长整型指针
函数返回值	返回转换后的长整型数

整型、长整型、浮点型转换为字符串	
所需头文件	#include <stdio.h>
函数说明	通常可以用 sprintf 作变量类型转换
函数原型	int sprintf(char *str,const char * format, ···)
转换例子	整型转换为字符串：sprintf(str, "%d", 30)
	长整型转换为字符串：sprintf(str, "%ld", 300000)
	浮点型转换为字符串：sprintf(str, "%.2f", 90.90)

（2）应用代码范例

ato.c 源代码如下：

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

```
    char a[]="-100";
    char aa[]="1000000000";
    char *aaa="-100.23";
    int c;
    long cc ;
    float ccc ;
    char str[30];

    c=atoi(a);
    cc = atol(aa) ;
    ccc = atof(aaa) ;
    printf("c=%d\n",c);
    printf("cc=%ld\n",cc);
    printf("ccc=%.2f\n",ccc);
    sprintf(str,"%d%ld%.2f",c, cc,ccc) ;
    printf("str=%s\n", str) ;

    return 0 ;
}
```

编译 gcc ato.c -o ato。
执行 ./ato，执行结果如下:

```
c=-100
cc=1000000000
ccc=-100.23
str=-10010000000000-100.23
```

第7章 标准I/O文件编程

标准 I/O 又称为带缓存的 I/O，标准 I/O 库是由 ANSI C 标准进行规范和说明的，基本所有的操作系统上都支持此库。标准 I/O 库处理了很多细节，例如，缓存分配、优化长度执行 I/O 等。这样，用户不必担心如何选择使用正确的块长度。标准 I/O 库是在系统调用函数基础上构造的，它便于用户使用，但是如果不较深入地了解库的操作，也会带来一些问题。

1.流和 FILE 对象

缓冲型文件系统把每个设备都转换成一个逻辑设备，叫做流。所有的流具有相同的行为。流基本上与设备无关。有两种类型的流：文本流和二进制流。

所有的初级 I/O 函数都是针对文件描述符的。当打开一个文件时，即返回一个文件描述符，然后该文件描述符就用于后续的 I/O 操作。而对于标准 I/O 库，它们的操作则是围绕流（stream）进行的。

当用标准 I/O 库打开或创建一个文件时，会使一个流与一个文件相结合。当打开一个流时，标准 I/O 函数 fopen 返回一个指向 FILE 对象的指针。该对象通常是一个结构，它包含了 I/O 库为管理该流所需要的所有信息：用于实际 I/O 的文件描述符、指向流缓存的指针、缓存的长度、当前在缓存中的字符数、出错标志等。

应用程序没有必要检验 FILE 对象。为了引用一个流，需将 FILE 指针作为参数传递给每

个标准 I/O 函数。在本书中，我们称指向 FILE 对象的指针(类型为 FILE *)为文件指针。

2.文件概述

所谓“文件”，是指一组相关数据的有序集合，这个数据集叫做文件，数据集的名字叫做文件名。实际上在前面的各章中我们已经多次使用了文件，例如源程序文件、目标文件、可执行文件、库文件 (头文件)等。

文件通常是驻留在外部介质(如磁盘等)上的，在使用时才调入内存中。从不同的角度可对文件进行不同的分类。从用户的角度看，文件可分为普通文件和设备文件两种。

普通文件是指驻留在磁盘或其他外部介质上的一个有序数据集，可以是源文件、目标文件、可执行程序，也可以是一组待输入处理的原始数据，或者是一组输出的结果。对于源文件、目标文件、可执行程序可以称为程序文件，对输入输出数据可称为数据文件。

设备文件是指与主机相联的各种外部设备，如显示器、打印机、键盘等。在操作系统中，把外部设备也看做是一个文件来进行管理，把它们的输入、输出等同于对磁盘文件的读和写。

通常把显示器定义为标准输出文件，一般情况下，在屏幕上显示有关信息就是向标准输出文件输出。如前面经常使用的 printf 函数就是这类输出。

通常把键盘定义为标准输入文件，从键盘上输入就意味着从标准输入文件上输入数据。scanf 函数就属于这类输入。

从文件编码的方式来看，文件可分为 ASCII 码文件和二进制码文件两种。ASCII 文件也称为文本文件，这种文件在磁盘中存放时每个字符对应一个字节，用于存放对应的 ASCII 码。

例如，数 5678 的存储形式为：

ASCII 码: 00110101 00110110 00110111 00111000

↓ ↓ ↓ ↓

十进制码: 5 6 7 8

共占用 4 个字节。

ASCII 码文件可在屏幕上按字符显示。例如，源程序文件就是 ASCII 文件，由于是按字符显示，因此，人们能读懂其文件内容。

二进制文件是按二进制的编码方式来存放文件的。

例如， 数 5678 的存储形式为：

00010110 00101110

只占 2 个字节。二进制文件虽然也可在屏幕上显示，但其内容无法读懂。C 系统在处理这些文件时，并不区分类型，都看成是字符流，按字节进行处理。

输入/输出字符流的开始和结束只由程序控制而不受物理符号(如回车符)的控制，因此也把这种文件称为“流式文件”。

流式文件的操作主要有打开、关闭、读、写、 定位等各种操作。

7.1 文件打开方式

1.文件打开的方式

标准 I/O 库打开文件的方式共有 12 种，表 10-1 列出了文件的打开方式及其意义说明。

表 10-1 文件打开方式表

文件打开方式	意义说明
rt	只读打开一个文本文件，只允许读数据
wt	只写打开或建立一个文本文件，只允许写数据

at	追加打开一个文本文件，并在文件末尾写数据
rb	只读打开一个二进制文件，只允许读数据
wb	只写打开或建立一个二进制文件，只允许写数据
ab	追加打开一个二进制文件，并在文件末尾写数据
rt+	读写打开一个文本文件，允许读和写
wt+	读写打开或建立一个文本文件，允许读写
at+	读写打开一个文本文件，允许读，或在文件末追加数据
rb+	读写打开一个二进制文件，允许读和写
wb+	读写打开或建立一个二进制文件，允许读和写
ab+	读写打开一个二进制文件，允许读，或在文件末追加数据

2.文件打开方式说明

对于文件的打开方式，有以下几点说明：

文件的打开方式由 r、w、a、t、b、+共 6 个字符组成，各字符的含义如下：

① r(read) ：读。

② w(write)：写。

③ a(append)：追加。

④ t(text) ：文本文件，可省略不写。

⑤ b(banary)：二进制文件。

⑥ +: 读和写。

凡用"r"打开一个文件时，该文件必须已经存在，且只能从该文件读出。

用"w"打开的文件只能向该文件写入。若打开的文件不存在，则以指定的文件名建立该文件，若打开的文件已经存在，则将该文件删去，重建一个新文件。

若要向一个已存在的文件追加新的信息，只能用"a"方式打开文件。但此时该文件必须是存在的，否则将会出错。

在打开一个文件时，如果出错，fopen 将返回一个空指针值 NULL。在程序中可以用这一信息来判别是否完成打开文件的工作，并做相应的处理。

把一个文本文件读入内存时，要将 ASCII 码转换成二进制码，而把文件以文本方式写入磁盘时，也要把二进制码转换成 ASCII 码。因此，文本文件的读写要花费较多的转换时间，而对二进制文件的读写则不存在这种转换。

标准输入文件(键盘)、标准输出文件(显示器)、标准出错输出(显示器)是由系统打开的，可直接使用。

3.标准 I/O 文件函数分类说明

表 10-2 列出标准 I/O 文件函数的分类及其简要说明。

表 10-2 标准 I/O 文件函数分类表

文件函数分类	函数名称
打开关闭文件函数	fopen()和 fclose()
字符读写函数	fgetc()和 fputc()
字符串读写函数	fgets()和 fputs()
数据块读写函数	fread()和 fwrite()
格式化读写函数	fprintf()和 fscanf()
取得文件流的读取位置	ftell()
移动文件流的读写位置	fseek()

文件结束检测函数	feof()

7.2 标准I/O函数说明及程序范例

1. 打开关闭文件

（1） 函数原型

fopen（打开文件）	
所需头文件	#include <stdio.h>
函数说明	根据文件路径打开文件
函数原型	FILE * fopen(const char * path, const char * mode)
函数传入值	path：打开的文件路径及文件名
	mode：代表流的形态。参见文件处理方式
函数返回值	成功：指向该流的文件指针就会被返回
	出错：返回 NULL，并把错误代码存在 errno 中

fdopen（将文件描述符转为文件指针）	
所需头文件	#include <stdio.h>
函数说明	fdopen()会将参数 fildes 的文件描述符转换为对应的文件指针后返回
函数原型	FILE * fdopen(int fildes,const char * mode)
函数传入值	fildes：文件描述符
	mode：代表文件指针的流形态，此形态必须和原先文件描述符读写模式相同。mode 方式参见文件处理方式
函数返回值	成功：转换成功时返回指向该流的文件指针
	错误：返回 NULL，并把错误代码存在 errno 中

freopen（打开文件）	
所需头文件	#include <stdio.h>
函数说明	参数 path 字符串包含欲打开的文件路径及文件名，参数 mode 请参考 fopen()说明。参数 stream 为已打开的文件指针。freopen()会将原 stream 所打开的文件流关闭，然后打开参数 path 的文件
函数原型	FILE * freopen(const char * path,const char * mode,FILE * stream)
函数传入值	path：文件路径全称
	mode：参见文件处理方式
	stream：原先打开的文件流
函数返回值	成功：文件顺利打开后，指向该流的文件指针就会被返回
	错误：返回 NULL，并把错误代码存在 errno 中

fclose（关闭文件）	
所需头文件	#include <stdio.h>
函数说明	fclose()用来关闭先前 fopen()打开的文件。此动作会让缓冲区内的数据写入文件中，并释放系统所提供的文件资源
函数原型	int fclose(FILE * stream)

函数传入值	stream：文件指针
函数返回值	成功：0
	出错：返回 EOF，并把错误代码存到 errno

fileno（返回文件流所使用的文件描述符）	
所需头文件	#include <stdio.h>
函数说明	fileno()用来取得参数 stream 指定的文件流所使用的文件描述符
函数原型	int fileno(FILE * stream)
函数传入值	stream：已打开的文件指针
函数返回值	返回文件描述符

（2） 打开关闭函数举例

fopen.c 源代码如下:

```
#include <stdio.h>
int main()
{
	FILE * fp;
	int fd ;
	fp=fopen("noexist","a+");
	if(fp==NULL)
	{
		perror("fopen error") ;
		return;
	}

	fd = fileno( fp ) ;
	printf("fildes no=%d\n", fd ) ;
	fclose(fp);

	return 0 ;
}
```

编译 gcc fopen.c -o fopen。

执行 ./fopen，执行结果如下:

```
fildes no=3
```

fdopen.c 源代码如下:

```
#include <stdio.h>
int main()
{
	FILE *fp =fdopen(0,"w+");
	fprintf(fp,"%s\n","hello!");
	fclose(fp);
	return 0 ;
}
```

编译 gcc fdopen.c -o fdopen。
执行 ./fdopen，执行结果如下：

```
hello!
```

freopen.c 源代码如下：

```
#include <stdio.h>
int main()
{
    FILE * fp;
    fp=fopen("/etc/passwd","r");
    if ( NULL == fp )
    {
        perror("fopen error") ;
        return -1;
    }
    printf("first fildes no=%d\n", fileno(fp)) ;
    fp=freopen("/etc/group","r",fp);
    if ( NULL == fp )
    {
        perror("fopen error") ;
        return -1;
    }
    printf("second fildes no=%d\n", fileno(fp)) ;
    fclose(fp);
    return 0 ;
}
```

编译 gcc freopen.c -o freopen。
执行 ./freopen，执行结果如下：

```
first fildes no=3
second fildes no=3
```

可见 freopen 是先关闭文件描述符，然后重新打开文件描述符。

2. 以字符串为单位的I/O函数

(1) 函数原型

fgets（从文件中读取一字符串）	
所需头文件	#include <stdio.h>
函数说明	用来从参数 stream 所指的文件内读入字符并存到参数 s 所指的内存空间，直到出现换行字符、读到文件尾或是已读了 size-1 个字符为止，最后会加上 NULL 作为字符串结束
函数原型	char * fgets(char * s,int size,FILE * stream)
函数传入值	s：读取内容存放字符串地址
	size：所读的最大长度
	stream：文件指针
函数返回值	成功：返回 s 的指针

	出错：返回 NULL

fputs（将一指定的字符串写入文件内）	
所需头文件	#include <stdio.h>
函数说明	用来将参数 s 所指的字符串写入到参数 stream 所指的文件内
函数原型	int fputs(const char * s,FILE * stream)
函数传入值	s：读取内容存放字符串地址
	stream：文件指针
函数返回值	成功：返回写出的字符个数
	出错：返回 EOF 则表示有错误发生

(2) 函数举例

fgets.c 源代码如下：

```
#include <stdio.h>
int main()
{
    char s[80];
    fputs(fgets(s,80,stdin),stdout);
    return 0 ;
}
```

编译 gcc fgets.c -o fgets。

执行 ./fgets，执行结果如下：

```
this is a test /*输入*/
this is a test /*输出*/
```

3. 以块为单位的I/O函数

(1) 函数原型

fread（从文件流读取数据）	
所需头文件	#include <stdio.h>
函数说明	用来从文件流中读取数据，读取的字符数由参数 size*nmemb 来决定。fread()会返回实际读取到的 nmemb 数目，如果此值比参数 nmemb 小，则代表可能读到了文件尾或有错误发生，这时必须用 feof()或 ferror()来决定发生什么情况
函数原型	size_t fread(void * ptr,size_t size,size_t nmemb,FILE * stream)
函数传入值	ptr：指向欲存放读取进来的数据空间
	size：读取单个块长度
	nmemb：读取的块数
	stream: 已打开的文件指针
函数返回值	实际读取到的 nmemb 数目

fwrite（将数据写至文件流）	
所需头文件	#include <stdio.h>
函数说明	用来将数据写入文件流中。总共写入的字符数由参数 size*nmemb 来决定。fwrite()会返回实际写入的 nmemb 数目

函数原型	size_t fwrite(const void * ptr,size_t size,size_t nmemb,FILE * stream)
函数传入值	ptr：指向欲写入的数据地址
	size：写入单个块长度
	nmemb：写入的块数
	stream：已打开的文件指针
函数返回值	实际写入的 nmemb 数目

(2) 函数举例

fwrite.c 源代码如下：

```
#include <stdio.h>
struct record {
    char name[10];
    int age;
};
int main(void)
{
    struct record array[2] = {{"Ken", 24}, {"Knuth", 28}};
    FILE *fp = fopen("recfile", "w");
    if (fp == NULL) {
        perror("Open file recfile");
        return -1;
    }
   /*变量在内存中为对齐存放，整型占 4 个字节，所以整型变量要存放在能除断 4 的内存地址上*/
    printf("record length=%d\n", sizeof( struct record )) ;
    printf("record name =%d, address=%d\n", sizeof(array[0].name), &array[0].name) ;
    printf("record age=%d, address=%d\n ", sizeof(array[0].age), &array[0].age) ;
    fwrite(array, sizeof(struct record), 2, fp);
    fclose(fp);
    return 0;
}
```

编译 gcc fwrite.c -o fwrite。

执行 ./fwrite，执行结果如下：

```
record length=16
record name =10, address=-1081549780
record age=4, address=-1081549768
```

fread.c 源代码如下：

```
#include <stdio.h>
struct record {
    char name[10];
    int age;
};
int main(void)
{
```

```
    struct record array[2];
    FILE *fp = fopen("recfile", "r");
    if (fp == NULL) {
        perror("Open file recfile");
        return -1;
    }
    fread(array, sizeof(struct record), 2, fp);
    printf("Name1: %s\tAge1: %d\n", array[0].name, array[0].age);
    printf("Name2: %s\tAge2: %d\n", array[1].name, array[1].age);
    fclose(fp);
    return 0;
}
```

编译 gcc fread.c -o fread。

执行 ./fread，执行结果如下：

```
Name1: Ken          Age1: 24
Name2: Knuth        Age2: 28
```

4. 以字节为单位的I/O函数

(1) 函数原型

fputc（将一指定字符写入文件流中）	
所需头文件	#include <stdio.h>
函数说明	将参数 c 转为 unsigned char 后写入参数 stream 指定的文件中
函数原型	int fputc(int c, FILE * stream)
函数传入值	c：写入成功的字符
	stream：已打开的文件指针
函数返回值	成功：写入的字符，即参数 c
	错误：返回 EOF（-1）则代表写入失败
附加说明	putc 原型同 fputc，其作用也相同，但 putc()为宏定义，非真正的函数调用

fgetc（将从文件中读取一个字符）	
所需头文件	#include <stdio.h>
函数说明	从参数 stream 所指的文件中读取一个字符。若读到文件尾无数据时便返回 EOF
函数原型	int fputc(FILE * stream)
函数传入值	stream：已打开的文件指针
函数返回值	成功：返回读到的字符
	错误：返回 EOF（-1）则代表读取失败
附加说明	getc 原型同 fgetc，其作用也相同，但 getc()为宏定义，非真正的函数调用

getchar（从标准输入设备内读取一个字符）	
所需头文件	#include <stdio.h>
函数说明	getchar()用来从标准输入设备中读取一个字符。然后将该字符从 unsigned char 转换成 int 后返回

函数原型	int getchar(void)
函数返回值	getchar()会返回读取到的字符，若返回 EOF，则表示有错误发生

	putchar（将指定的字符写到标准输出设备）
所需头文件	#include <stdio.h>
函数说明	putchar()用来将参数 c 字符写到标准输出设备
函数原型	int putchar (int c)
函数返回值	putchar()会返回输出成功的字符，即参数 c。若返回 EOF，则代表输出失败
附加说明	putchar()函数是通过 putc(c，stdout)的宏定义实现的

(2) 函数举例

fgetc.c 源代码如下：

```
#include <stdio.h>
int main()
{
    FILE *fp;
    FILE *fp1;
    int c;
    fp=fopen("exist.txt","r");
    if ( NULL == fp )
    {
        perror("fopen error") ;
        return -1;
    }
    fp1= fopen("noexist.txt","w");
    if ( NULL == fp1 )
    {
        perror("fopen error") ;
        return -1;
    }
    while((c=fgetc(fp))!=EOF)
    {
        fputc(c, fp1) ;
    }
    fclose(fp);
    fclose(fp1);
    return 0 ;
}
```

编译 gcc fgetc.c -o fgetc。

创建 exist.txt 文件并输入内容，执行./fgetc。查看结果发现产生了 noexist.txt，且两文件内容一样。

5. 操作读写位置的函数

(1) 函数原型

fseek（移动文件流的读写位置）			
所需头文件	#include <stdio.h>		
函数说明	用来移动文件流的读写位置。参数 offset 为根据参数 whence 来移动读写位置的位移数		
函数原型	int fseek(FILE * stream,long offset,int whence)		
函数传入值	stream：已打开的文件指针		
	offset：根据参数 whence 来移动读写位置的位移数，可为正、负、0		
	whence(既可以用宏，也可以用数字)		
	起始点	宏表示符号	数字表示
	文件首	SEEK_SET	0
	当前位置	SEEK_CUR	1
	文件末尾	SEEK_END	2
函数返回值	成功：0		
	错误：-1，errno 会存放错误代码		
常见使用方法	① 欲将读写位置移动到文件开头时用 fseek(FILE *stream,0,SEEK_SET) ② 欲将读写位置移动到文件尾时用 fseek(FILE *stream,0,0SEEK_END)		

rewind（重设文件流的读写位置为文件开头）	
所需头文件	#include <stdio.h>
函数说明	rewind()用来把文件流的读写位置移至文件开头。参数 stream 为已打开的文件指针，此函数相当于调用 fseek(stream,0,SEEK_SET)
函数原型	void rewind(FILE * stream)
函数传入值	stream：已打开的文件指针

ftell（取得文件流的读取位置）	
所需头文件	#include <stdio.h>
函数说明	ftell()用来取得文件流目前的读写位置
函数原型	long ftell(FILE * stream)
函数传入值	stream：已打开的文件指针
函数返回值	成功：目前的读写位置
	错误：-1，errno 会存放错误代码
错误代码	EBADF:参数 stream 无效或文件流的读写位置无效

feof（检查文件流是否读到了文件尾）	
所需头文件	#include <stdio.h>
函数说明	用来侦测是否读取到了文件尾。如果已到文件尾，则返回非零值，其他情况返回 0
函数原型	int feof(FILE * stream)
函数传入值	stream：已打开的文件指针
函数返回值	返回非零值代表已到达文件尾

(2) 函数举例

fseek.c 源代码如下：

```
#include <stdio.h>
int main()
{
	FILE * stream;
	long offset;
	stream=fopen("/etc/passwd","r");
	fseek(stream,5,SEEK_SET);
	printf("offset=%d\n",ftell(stream));
	rewind(stream);
	printf("offset = %d\n",ftell(stream));
	printf("file eof status=%d\n", feof(stream)) ;
	fseek(stream,0,SEEK_END);
	printf("offset = %d\n",ftell(stream));
	printf("eof=%d\n", fgetc(stream)) ;
	printf("file eof status=%d\n", feof(stream)) ;
	fclose(stream);
	return 0 ;
}
```

编译 gcc fseek.c -o fseek。

执行 ./fseek，结果如下：

```
offset=5
offset = 0
file eof status=0
offset = 2676
eof=-1
file eof status=1
```

ftell.c 源代码如下：

```
#include <stdio.h>
int main()
{
	FILE *fp;
	long flen;
	char data[10000] ;
	if( ( fp = fopen( "/etc/passwd" , "r" ) ) == NULL ){
		perror("fopen error") ;
		return -1 ;
	}
	fseek(fp,0L,SEEK_END); /* 定位到文件末尾 */
	flen=ftell(fp); /* 得到文件大小 */
	fseek(fp,0L,SEEK_SET); /* 定位到文件末尾 */
	fread(data,flen,1,fp); /* 一次性读取全部文件的内容 */
	fclose(fp);
```

```
    printf("passwd file content:\n%s\n", data ) ;
    return 0 ;
}
```

编译 gcc ftell.c -o ftell。

执行 ./ftell，执行结果如下：

```
passwd file content:
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
.....
```

6. 格式化读写函数

(1) 函数原型

fprintf（格式化输出数据至文件）	
所需头文件	#include <stdio.h>
函数说明	fprintf()会根据参数 format 字符串来转换并格式化数据，然后将结果输出到参数 stream 指定的文件中，直到出现字符串结束('\0')为止
函数原型	int fprintf(FILE * stream, const char * format, ...)
函数传入值	stream：文件描述符
	format：格式项
函数返回值	成功：返回实际输出的字节数
	失败：-1，并把错误代码存在于 errno 中

fscanf（格式化字符串输入）	
所需头文件	#include <stdio.h>
函数说明	fscanf()会从参数 stream 的文件流中读取字符串，再根据参数 format 字符串来转换并格式化数据。格式化转换形式请参考 scanf()。转换后的结构存于对应的参数内
函数原型	int fscanf(FILE * stream, const char *format, ...)
函数传入值	stream：文件描述符
	format：格式项
函数返回值	成功：返回参数数目
	失败：-1，并把错误代码存在于 errno 中

(2) 函数举例

fprintf.c 源代码如下：

```
#include <stdio.h>
int main()
{
    int i = 150;
    int j = -100;
    double k = 3.14159;
    int fp ;
    fp = fprintf(stdout,"%d %f %x \n",j,k,i);
```

```
    fp = fprintf(stdout,"%2d %*d\n",i,2,i);
    printf(" fp=%d\n", fp) ;
    return 0 ;
}
```

编译 gcc fprintf.c -o fprintf。

执行 ./fprintf，执行结果如下：

```
-100 3.141590 96
150 150
fp=8
```

fscanf.c 源代码如下：

```
#include<stdio.h>
int main()
{
    int i;
    unsigned int j;
    char s[5];
    int fd ;
    fd =fscanf(stdin,"%d %x %5[a-z] %*s %f",&i,&j,s,s);
    printf("%d %d %s \n",i,j,s);
    printf("fd=%d\n", fd) ;
    return 0 ;
}
```

编译 gcc fscanf.c -o fscanf。

执行 ./fscanf，执行结果如下：

```
10 0x1b aaaaaaaaa bbbbbbbbbb /*从键盘输入*/
10 27 aaaaa
fd=3
```

7. 产生临时文件

mktemp 函数只产生唯一的临时文件名，并不产生临时文件，而且存在安全隐患，不建议使用。mkstemp 函数会产生唯一的临时文件。

(1) 函数原型

mktemp（产生唯一的临时文件名）	
所需头文件	#include <stdlib.h>
函数说明	mktemp()用来产生唯一的临时文件名，并不创建文件。参数 template 所指的文件名称字符串中最后六个字符必须是 XXXXXX，产生后的文件名会借字符串指针返回
函数原型	char * mktemp(char * template)
函数传入值	template：所指的文件名称字符串，最后六个字符必须是 XXXXXX
函数返回值	成功：以指针方式返回产生的文件名
	失败：NULL，并把错误代码存于 errno 中

附加说明	参数 template 所指的文件名称字符串必须声明为数组，如: char template[ ]= "template-XXXXXX"; 不可用 char * template="template-XXXXXX";

mkstemp（建立唯一的临时文件）	
所需头文件	#include <stdlib.h>
函数说明	mkstemp()用来建立唯一的临时文件。参数 template 所指的文件名称字符串中最后六个字符必须是 XXXXXX。mkstemp()会以可读写模式来打开该文件，如果该文件不存在则会建立该文件。
函数原型	int mkstemp(char * template)
函数传入值	template：所指的文件名称字符串，最后六个字符必须是 XXXXXX
函数返回值	成功：文件顺利打开后，返回该文件的文件描述符
	失败：如果文件打开失败，则返回 0，并把错误代码存于 errno 中
附加说明	参数 template 所指的文件名称字符串必须声明为数组，如: char template[ ]= "template-XXXXXX"; 不可用 char * template="template-XXXXXX";

(2) 函数举例

mktemp.c 源代码如下:

```
#include <stdio.h>
int main()
{
    char template[ ]="template-XXXXXX";
    char *file ;
    file = mktemp(template);
    printf("template=%s\n",template);
    printf("file=%s\n", file);
    return 0 ;
}
```

编译 gcc mktemp.c -o mktemp。

执行 ./mktemp，执行结果如下:

```
template=template-oh2jT8
file=template-oh2jT8
```

mkstemp.c 源代码如下:

```
#include <stdio.h>
int main()
{
    int fd;
    char template[ ]="template-XXXXXX";
    fd=mkstemp(template);
    printf("template = %s\n",template);
    printf("fd = %d\n", fd);
    close(fd);
```

```
    return 0 ;
}
```

编译 gcc mkstemp.c -o mkstemp。

执行 ./mkstemp，执行结果如下:

```
template = template-rPGs2W
fd = 3
```

8. 错误检测与清除

(1) 函数原型

ferror（检查文件流是否有错误发生）	
所需头文件	#include <stdio.h>
函数说明	ferror()用来检查参数 stream 所指定的文件流是否发生了错误情况，如有错误发生则返回非 0 值
函数原型	int ferror(FILE *stream)
函数传入值	stream:已打开的文件指针
函数返回值	如果文件流有错误发生则返回非 0 值

clearerr（清除文件流的错误标记）	
所需头文件	#include <stdio.h>
函数说明	clearerr（）清除参数 stream 指定的文件流所使用的错误标记
函数原型	void clearerr(FILE * stream)
函数传入值	stream:已打开的文件指针
函数返回值	无

9. 文件编程综合实例

源代码 file.c 如下:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
 /*去掉字符串左边指定字符*/
char *LTrimChar( char *pBuf, char cDel )
{
    char *pTmp = pBuf ;
    if ( ( cDel != '\0' ) && ( pTmp != NULL ) )
    {
        while ( *pTmp++ ==  cDel ) ;
        strcpy( pBuf, pTmp - 1 );
    }
    return pBuf ;
}
/*去掉字符串右边指定字符*/
char *RTrimChar( char *pBuf, char cDel )
{
    char *pTmp = pBuf;
```

```
    if ( ( cDel != '\0' ) && ( pTmp != NULL ) )
    {
        for ( pTmp = pBuf + strlen( pBuf ) -1 ;
                ( *pTmp ==  cDel ) && ( pTmp >= pBuf ) ; *pTmp--  = '\0' );
    }
    return pBuf ;

}
/*去掉字符串两边指定字符*/
char *AllTrimChar( char *pBuf, char cDel )
{
    return  LTrimChar( RTrimChar( pBuf, cDel  ) , cDel );
}
int main(int argc, char **argv)
{
    char buff_read[128];
    long line_length ;
    char seri_no[8+1];
    char cust_acct_no[22+1] ;
    char amt[17+1] ;  /*原文件金额除以 100 进行转换*/
    char *ptr;
    FILE *fpr; /*读文件指针*/
    FILE *fpw; /*写文件指针*/
    char file_read[64];
    char file_write[64];
    sprintf( file_read,"%s", "read.txt" );
    if( ( fpr = fopen( file_read,"r" ) ) == NULL  ) {
        perror("fopen error\n");
        return -1;
    }
    strcpy( file_write, "write.txt") ;
    fpw = fopen( file_write , "w" );
    if( fpw == NULL )  {
        perror("fopen error\n");
        return -1;
    }
    memset( buff_read , 0x00 , sizeof( buff_read ));
    line_length= sizeof(buff_read) ;
    while( fgets( buff_read, line_length, fpr ) != NULL ) {
        memset(seri_no, 0x00, sizeof(seri_no)) ;
        memset(cust_acct_no, 0x00, sizeof(cust_acct_no)) ;
        memset(amt, 0x00, sizeof(amt)) ;
        memcpy( seri_no , buff_read+0 , 8 );
        AllTrimChar( seri_no, ' ' ) ; /*去掉两边的空格*/
```

```
            memcpy( cust_acct_no , buff_read+8 , 22 );
            AllTrimChar( cust_acct_no, ' ' ) ;
            memcpy( amt , buff_read+30 , 17 );
            AllTrimChar( amt, ' ' ) ;
            fprintf(fpw,"%-8.8s%-22.22s%-17.2lf\n", seri_no,cust_acct_no, atof(amt)/100 ) ;
        }
        fclose( fpr );
        fclose( fpw );
        return 0 ;
}
```

编译 gcc file.c –o file。

read.txt 文件内容为：

```
123456   12345678909876543210   1234567
123457   12345678909876543230   1234565
```

执行 ./file。发现建立了 write.txt 文件，其文件内容为：

```
123456   12345678909876543210   12345.67
123457   12345678909876543230   12345.65
```