第 1 章　编译与调试

1.1　编译的概念和理解

在进行 C 程序开发时，编译就是将编写的 C 语言代码变成可执行程序的过程，这一过程是由编译器来完成的。编译器就是完成程序编译工作的软件，在进行程序编译时完成了一系列复杂的过程。

1.1.1　程序编译的过程

在执行这一操作时，程序完成了复杂的过程。一个程序的编译，需要完成词法分析、语法分析、中间代码生成、代码优化、目标代码生成。本章将讲解这些步骤的作用与原理。

（1）词法分析。指的是对由字符组成的单词进行处理，从左至右逐个字符地对源程序进行扫描，产生一个个的单词符号。然后把字符串的源程序改造成为单词符号串的中间程序。在编译程序时，这一过程是自动完成的。编译程序会对代码的每一个单词进行检查。如果单词发生错误，编译过程就会停止并显示错误。这时需要对程序中的错误进行修改。

（2）语法分析。语法分析器以单词符号作为输入，分析单词符号串是否形成符合语法规则的语句。例如，需要检查表达式、赋值、循环等结构是否完整和符合使用规则。在语法分析时，会分析出程序中错误的语句，并显示出结果。如果语法发生错误，编译任务是不能完成的。

（3）中间代码生成。中间代码是源程序的一种内部表示，或称中间语言。程序进行词法分析和语法分析以后，将程序转换成中间代码。这一转换的作用是使程序的结构更加简单和规范。中间代码生成操作是一个中间过程，与用户是无关的。

（4）代码优化。代码优化是指对程序进行多种等价变换，使得从变换后的程序能生成更有效的目标代码。用户可以在编译程序时设置代码优化的参数，可以针对不同的环境和设置进行优化。

（5）目标代码生成。目标代码生成指的是产生可以执行的应用程序，这是编译的最后一个步骤。生成的程序是二进制的机器语言，用户只能运行这个程序，而不能打开这个文件查看程序的代码。

1.1.2　编译器

所谓编译器，是将编写出的程序代码转换成计算机可以运行的程序的软件。在进行 C 程序开发时，编写出的代码是源程序的代码，是不能直接运行的。需要用编译器编译成可以运行的二进制程序。

在不同的操作系统下面有不同的编译器。C 程序是可以跨平台运行的。但并不是说 Windows 系统下 C 语言编写的程序可以直接在 Linux 下面运行。Windows 下面 C 语言编写的程序，被编译成 exe 文件。这样的程序只能在 Windows 系统下运行。如果需要在 Linux 系统下运行，需要将这个程序的源代码在 Linux 系统重新编译。不同的操作系统下面有不同的编译器。Linux 系统下面编译生成的程序是不能在 Windows 系统上运行的。

1.2　gcc 编译器

gcc 是 Linux 下的 C 程序编译器，具有非常强大的程序编译功能。在 Linux 系统下，C 语言编写的程序代码一般需要通过 gcc 来编译成可执行程序。

1.2.1　gcc 编译器简介

Linux 系统下的 gcc 编译器（GNU C Compiler）是一个功能强大、性能优越的编译器。gcc 支持多种平台的编译，是 Linux 系统自由软件的代表作品。gcc 本来只是 C 编译器的，但是后来发展为可在多种硬体平台上编译出可执行程序的超级编译器。各种硬件平台对 gcc 的支持使得其执行效率与一般的编译器相比平均效率要高 20%~30%。gcc 编译器能将 C、C++源程序、汇程语言和目标程序进行编译链接成可执行文件。通过支持 make 工具，gcc 可以实施项目管理和批量编译。

经过多年的发展，gcc 已经发生了很大的变化。gcc 已经不仅仅能支持 C 语言，还支持 Ada 语言、C++语言、Java 语言、Objective C 语言、Pascal 语言、COBOL 语言等更多的语言集的编译。gcc 几乎支持所有的硬件平台，使得 gcc 对于特定的平台可以编译出更高效的机器码。

gcc 在编译一个程序时，一般需要完成预处理（preprocessing）、编译（compilation）、汇编（assembly）和链接（linking）过程。使用 gcc 编译 C 程序时，这些过程是使用默认的设置自动完成的，但是用户可以对这些过程进行设置，控制这些操作的详细过程。

1.2.2　gcc 对源程序扩展名的支持

扩展名指的是文件名中最后一个点的这个点以后的部分。例如下面是一个 C 程序源文件的扩展名。

```
5.1.c
```

那么这个文件的文件名是“5.1.c”，扩展名是“.c”。通常来说，源文件的扩展名标识源文件所使用的编程语言。例如 C 程序源文件的扩展名一般是“.c”。对编译器来说，扩展名控制着缺省语言的设定。在默认情况下，gcc 通过文件扩展名来区分源文件的语言类型。然后根据这种语言类型进行不同的编译。gcc 对源文件的扩展名约定如下所示。

- .c 为扩展名的文件，为 C 语言源代码文件。
- .a 为扩展名的文件，是由目标文件构成的库文件。
- .C,.cc 或.cpp 为扩展名的文件，标识为 C++源代码文件。
- .h 为扩展名的文件，说明文件是程序所包含的头文件。
- .i 为扩展名的文件，标识文件是已经预处理过的 C 源代码文件，一般为中间代码文件。

- .ii 为扩展名的文件，是已经预处理过的 C++源代码文件，同上也是中间代码文件。
- .o 为扩展名的文件，是编译后的目标文件，源文件生成的中间目标文件。
- .s 为扩展名的文件，是汇编语言源代码文件。
- .S 为扩展名的文件，是经过预编译的汇编语言源代码文件。
- .o 为扩展名的文件，是编译以后的程序目标文件（Object file），目标文件经过连接成可执行文件

此外，对于 gcc 编译器提供两种显示的编译命令，分别对应于编译 C 和 C＋＋源程序的编译命令。

1.3 C 程序的编译

本章以一个实例讲述如何用 gcc 编译 C 程序。在编译程序之前，需要用 VIM 编写一个简单的 C 程序。在编译程序时，可以对 gcc 命令进行不同的设置。

1.3.1 编写第一个 C 程序

本节将编写第一个 C 程序。程序实现一句文本的输出和判断两个整数的大小关系。本书中编写程序使用的编辑器是 VIM。程序编写步骤如下所示。

① 打开系统的终端。单击“主菜单”|“系统工具”|“终端”命令，打开一个系统终端。

② 在终端中输入下面的命令，在用户根目录“root”中建立一个目录。

```
mkdir c
```

③ 在终端界面中输入“vim”命令，然后按“Enter”键，系统会启动 VIM。

④ 在 VIM 中按“i”键，进入到插入模式。然后在 VIM 中输入下面的程序代码。

```
#include <stdio.h>

int max(int i,int j )
{
    if(i>j)
    {
        return(i);
    }
    else
    {
        return(j);
    }
}

void main()
{
   int i ,j,k;
   i=3;
   j=5;
   printf("hello ,Linux.\n");
```

```
    k=max(i,j);
    printf("%d\n",k);
}
```

⑤ 代码输入完成以后，按“Esc”键，返回到普通模式。然后输入下面的命令，保存文件。

```
:w /root/c/a.c
```

这时，VIM 会把输入的程序保存到 c 目录下的文件 a.c 中。

⑥ 再输入“:q”命令，退出 VIM。这时，已经完成了这个 C 程序的编写。

1.3.2　用 gcc 编译程序

上面编写的 C 程序，只是一个源代码文件，还不能作为程序来执行。需要用 gcc 将这个源代码文件编译成可执行文件。编译文件的步骤如下所示。

① 打开系统的终端。单击“主菜单”|“系统工具”|“终端”命令，打开一个系统终端。这时进入的目录是用户根目录“/root”。然后输入下面的命令，进入到 c 目录。

```
cd c
```

② 上一节编写的程序就存放在这个目录中。输入“ls”命令可以查看这个目录下的文件。显示的结果如下所示。

③ 输入下面的命令，将这个代码文件编译成可执行程序。

```
gcc a.c
```

④ 查看已经编译的文件。在终端中输入“ls”命令，显示的结果如下所示。

```
a.c a.out
```

⑤ 输入下面的命令对这个程序添加可执行权限。

```
chmod +x a.out
```

⑥ 输入下面的命令，运行这个程序。

```
./a.out
```

⑦ 程序的运行结果如下所示。

```
hello ,Linux.
5
```

从上面的操作可知，用 gcc 可以将一个 C 程序源文件编译成一个可执行程序。编译以后的程序需要添加可执行的权限才可以运行。在实际操作中，还需要对程序的编译进行各种设置。

1.3.3 查看 gcc 的参数

gcc 在编译程序时可以有很多可选参数。在终端中输入下面的命令，可以查看 gcc 的这些可选参数。

```
gcc --help
```

在终端中显示的 gcc 的可选参数如下所示。进行程序编译时，可以设置下面的这些参数。

```
用法：gcc [选项] 文件...
选项：
-pass-exit-codes：在某一阶段退出时返回最高的错误码
--help：显示此帮助说明
--target-help：显示目标机器特定的命令行选项
-dumpspecs：显示所有内建 spec 字符串
-dumpversion：显示编译器的版本号
-dumpmachine：显示编译器的目标处理器
-print-search-dirs：显示编译器的搜索路径
-print-libgcc-file-name：显示编译器伴随库的名称
-print-file-name=<库>：显示 <库> 的完整路径
-print-prog-name=<程序>：显示编译器组件 <程序> 的完整路径
-print-multi-directory：显示不同版本 libgcc 的根目录
-print-multi-lib：显示命令行选项和多个版本库搜索路径间的映射
-print-multi-os-directory：显示操作系统库的相对路径
-Wa,<选项>：  将逗号分隔的 <选项> 传递给汇编器
-Wp,<选项>：将逗号分隔的 <选项> 传递给预处理器
-Wl,<选项>：将逗号分隔的 <选项> 传递给链接器
-Xassembler <参数>：将 <参数> 传递给汇编器
-Xpreprocessor <参数>：将 <参数> 传递给预处理器
-Xlinker <参数>：将 <参数> 传递给链接器
-combine：将多个源文件一次性传递给汇编器
-save-temps：不删除中间文件
-pipe：使用管道代替临时文件
-time：为每个子进程计时
-specs=<文件>：  用 <文件> 的内容覆盖内建的 specs 文件
-std=<标准>：指定输入源文件遵循的标准
--sysroot=<目录>：将 <目录> 作为头文件和库文件的根目录
-B <目录>：将 <目录> 添加到编译器的搜索路径中
-b <机器>：为 gcc 指定目标机器(如果有安装)
-V <版本>：运行指定版本的 gcc(如果有安装)
-v：显示编译器调用的程序
-###：与 -v 类似，但选项被引号括住，并且不执行命令
-E：仅作预处理，不进行编译、汇编和链接
-S：编译到汇编语言，不进行汇编和链接
-c：编译、汇编到目标代码，不进行链接
-o <文件>：输出到 <文件>
-x <语言>：指定其后输入文件的语言。允许的语言包括 c、c++、assembler 等。
以 -g、-f、-m、-O、-W 或 --param 开头的选项将由 gcc 自动传递给其调用的不同子进程。若要向这些进程传递其他选项，必须使用 -W<字母> 选项。
```

1.3.4　设置输出的文件

在默认情况下，gcc 编译出的程序为当前目录下的文件 a.out。-o 参数可以设置输出的目标文件。例如下面的命令，可以设置将代码编译成可执行程序 do。

```
gcc a.c -o do
```

也可以设置输出目录文件为不同的目录。例如下面的命令，是将目录文件设置成/tmp 目录下的文件 do。

```
gcc a.c -o /tmp/do
```

输入下面的命令，查看生成的目录文件。结果如下所示，在编译程序时生成的目录为/tmp 目录下的文件 do。

```
-rwxrwxr-x 1 root root 5109 12-28 13:33 /tmp/do
```

1.3.5　查看编译过程

参数-v 可以查看程序的编译过程和显示已经调用的库。输入下面的命令，在编译程序时输出编译过程。

```
gcc -v a.c
```

显示的结果如下所示。

```
使用内建 specs。
目标：i386-redhat-linux
配置为：../configure --prefix=/usr --mandir=/usr/share/man
 --infodir=/usr/share/info --enable-shared --enable-threads=posix
 --enable-checking=release --with-system-zlib --enable-__cxa_atexit
--disable-libunwind-exceptions
--enable-languages=c,c++,objc,obj-c++,java,fortran,ada
--enable-java-awt=gtk --disable-dssi --enable-plugin
--host=i386-redhat-linux
线程模型：posix
gcc 版本 4.1.2 20070925 (Red Hat 4.1.2-33)
/usr/libexec/gcc/i386-redhat-linux/4.1.2/cc1
-quiet -v a.c -quiet -dumpbase a.c -mtune=generic -auxbase a -version
-o /tmp/cc8P7rzb.s
忽略不存在的目录“/usr/lib/gcc/i386-redhat-linux/4.1.2/../../../../i386-
redhat-linux/include”
#include "..." 搜索从这里开始：
#include <...> 搜索从这里开始：
 /usr/local/include
 /usr/lib/gcc/i386-redhat-linux/4.1.2/include
 /usr/include
搜索列表结束。
GNU C 版本 4.1.2 20070925 (Red Hat 4.1.2-33) (i386-redhat-linux)
由 GNU C 版本 4.1.2 20070925 (Red Hat 4.1.2-33) 编译。
GGC 准则：--param ggc-min-expand=64 --param ggc-min-heapsize=64394
```

```
    Compiler executable checksum: ab322ce5b87a7c6c23d60970ec7b7b31
    a.c: In function 'main':
    a.c:16: 警告: 'main' 的返回类型不是 'int'
     as -V -Qy -o /tmp/ccEFPrYh.o /tmp/cc8P7rzb.s
    GNU assembler version 2.17.50.0.18 (i386-redhat-linux) using BFD version
version 2.17.50.0.18-1 20070731
    /usr/libexec/gcc/i386-redhat-linux/4.1.2/collect2
    --eh-frame-hdr --build-id -m elf_i386 --hash-style=gnu -dynamic-linker
    /lib/ld-linux.so.2 /usr/lib/gcc/i386-redhat-linux/4.1.2/../../../crt1.o
```

从显示的编译过程可知，gcc 自动加载了系统的默认配置，调用系统的库函数完成了程序的编译过程。

1.3.6 设置编译的语言

gcc 可以对多种语言编写的源代码。如果源代码的文件扩展名不是默认的扩展名，gcc 就无法编译这个程序。可以用-x 选择来设置程序的语言。可以用下面的步骤来练习这一操作。

① 输入下面的命令，将 C 程序文件复制一份。

```
cp a.c a.u
```

② 复制出的文件 a.u 是一个 C 程序文件，但扩展名不是默认的扩展名。这时输入下面的命令编译这个程序。

```
gcc a.u
```

③ 显示的结果如下所示，表明文件的格式不能识别。

```
a.u: file not recognized: File format not recognized
collect2: ld 返回 1
```

④ 这时，用-x 参数设置编译的语言，命令如下所示。这样就可以正常地编译文件 a.u。

```
gcc -x 'c' a.u
```

需要注意的是，这里的 c 需要用单引号扩起来。当编译扩展名不是.c 的 C 程序时，需要使用-x 参数。

1.3.7 -asci 设置 ANSIC 标准

ANSIC 是 American National Standards Institute（ANSI：美国标准协会） 出版的 C 语言标准。使用这种标准的 C 程序可以在各种编译器和系统下运行通过。gcc 可以编译 ANSIC 的程序，但是 gcc 中的很多标准并不被 ANSIC 所支持。在 gcc 编译程序时，可以用-ansic 来设置程序使用 ANSIC 标准。例如下面的命令，在设置程序编译时，用 ANSIC 标准进行编译。

```
gcc -asci a.out
5.3.8
```

1.3.8 g++编译 C++程序

gcc 可以编译 C++程序。编译 C 程序和 C++程序时，使用的是不同的命令。编译 C++程

序时，使用的命令是 g++。该命令的使用方法与 gcc 是相似的。下面是使用 g++命令编译 C++程序的实例。

下面是一个 C++程序的代码，实现与 1.3.1 节程序同样的功能。C++程序的代码与 C 程序的代码非常相似。

```
#include <iostream>
int max(int i,int j )
{
    if(i>j)
    {
        return(i);
    }
    else
    {
        return(j);
    }
}

int main()
{

   int i ,j,k;
   i=3;
   j=5;
   printf("hello ,Linux.\n");
   k=max(i,j);
   printf("%d\n",k);
   return(0);
}
```

输入下面的命令，编译这个 C++程序。

```
g++ 5.2.cpp -o 5.2.out
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x 5.2.ou
```

输入下面的命令，运行这个程序，程序的代码如下所示。

```
hello ,Linux.
5
```

从结果可知，这个程序与 1.3.1 节中的程序运行结果是相同的。

1.4　编译过程的控制

编译过程指的是 gcc 对一个程序进行编译时完成的内部处理和步骤。编译程序时会自动完成预处理（Preprocessing）、编译（Compilation）、汇编（Assembly）和链接（Linking）4 个步骤。本节将讲解如何对这 4 个步骤进行控制。

1.4.1 编译过程简介

gcc 把一个程序的源文件，编译成一个可执行文件，中间包括很多复杂的过程。可以用图 1-1 来表示编译中 4 个步骤的作用和关系。

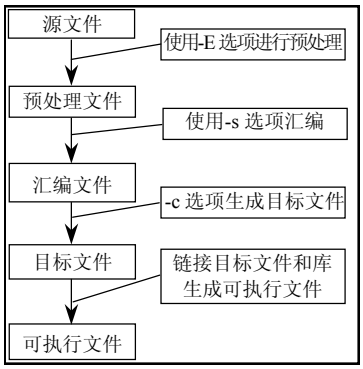

图 1-1 gcc 编译源文件到可执行文件的过程

在 4 个过程中，每一个操作都完成了不同的功能。编译过程的功能如下所示。

- 预处理：在预处理阶段，主要完成对源代码中的预编译语句（如宏定义 define 等）和文件包含进行处理。需要完成的工作是对预编译指令进行替换，把包含文件放置到需要编译的文件中。完成这些工作后，会生成一个非常完整的 C 程序源文件。
- 编译：gcc 对预处理以后的文件进行编译，生成以.s 为后缀的汇编语言文件。该汇编语言文件是编译源代码得到的汇编语言代码，接下来交给汇编过程进行处理。汇编语言是一种比 C 语言更低级的语言，直接面对硬盘进行操作。程序需要编译成汇编指令以后再编译成机器代码。
- 汇编：汇编过程是处理汇编语言的阶段，主要调用汇编处理程序完成将汇编语言汇编成二进制机器代码的过程。通常来说，汇编过程是将.s 的汇编语言代码文件汇编为.o 的目标文件的过程。所生成的目标文件作为下一步链接过程的输入文件。
- 链接：链接过程就是将多个汇编生成的目标文件以及引用的库文件进行模块链接生成一个完整的可执行文件。在链接阶段，所有的目标文件被安排在可执行程序中的适当的位置。同时，该程序所调用到的库函数也从各自所在的函数库中链接到程序中。经过了这个过程以后，生成的文件就是可执行的程序。

1.4.2 控制预处理过程

参数-E 可以完成程序的预处理工作而不进行其他的编译工作。下面的命令，可以将本章编写的程序进行预处理，然后保存到文件 a.cxx 中。

```
gcc -E -o a.cxx a.c
```

输入下面的命令，查看经过预处理以后的 a.cxx 文件。

```
vim a.cxx
```

可以发现，文件 a.cxx 约有 800 行代码。程序中默认包含的头文件已经被展开写到到这个

文件中。显示的文件 a.cxx 前几行代码如下所示。可见，在程序编译时，需要调用非常多的头文件和系统库函数。

```
# 1 "a.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "a.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 335 "/usr/include/features.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 360 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/bits/wordsize.h" 1 3 4
# 361 "/usr/include/sys/cdefs.h" 2 3 4
# 336 "/usr/include/features.h" 2 3 4
# 359 "/usr/include/features.h" 3 4
# 1 "/usr/include/gnu/stubs.h" 1 3 4
# 1 "/usr/include/bits/wordsize.h" 1 3 4
# 5 "/usr/include/gnu/stubs.h" 2 3 4
# 1 "/usr/include/gnu/stubs-32.h" 1 3 4
# 8 "/usr/include/gnu/stubs.h" 2 3 4
# 360 "/usr/include/features.h" 2 3 4
# 29 "/usr/include/stdio.h" 2 3 4
```

1.4.3　生成汇编代码

参数-S 可以控制 gcc 在编译 C 程序时只生成相应的汇编程序文件，而不继续执行后面的编译。下面的命令，可以将本章中的 C 程序编译成一个汇编程序。

```
gcc -S -o a.s a.c
```

输入下面的命令，查看汇编文件 a.s。可以发现这个文件一共有 60 行代码。这些代码是这个程序的汇编指令。部分汇编程序代码如下所示。

```
    .file   "a.c"
    .text
.globl max
    .type   max, @function
max:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    8(%ebp), %eax
    cmpl    12(%ebp), %eax
    jle .L2
    movl    8(%ebp), %eax
    movl    %eax, -4(%ebp)
    jmp .L4
.L2:
    movl    12(%ebp), %eax
```

```
        movl    %eax, -4(%ebp)
    .L4:
        movl    -4(%ebp), %eax
        leave
        ret
        .size   max, .-max
        .section    .rodata
    .LC0:
        .string "hello ,Linux."
    .LC1:
        .string "%d\n"
        .text
    .globl main
        .type   main, @function
    main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
    .......
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.1.2 20070925 (Red Hat 4.1.2-33)"
        .section    .note.GNU-stack,"",@progbits
```

1.4.4 生成目标代码

参数-c 可以使得 gcc 在编译程序时只生成目录代码而不生成可执行程序。输入下面的命令，将本章中的程序编译成目录代码。

```
gcc -c -o a.o  a.c
```

输入下面的命令，查看这个目录代码的信息。

```
file a.o
```

显示文件 a.o 的结果如下所示，显示文件 a.o 是一个可重定位的目标代码文件。

```
a.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

1.4.5 链接生成可执行文件

gcc 可以把上一步骤生成的目录代码文件生成一个可执行文件。在终端中输入下面的命令。

```
gcc a.o -o aa.out
```

这时生成一个可执行文件 aa.out。输入下面的命令查看这个文件的信息。

```
file aa.out
```

显示的结果如下所示，表明这个文件是可在 Linux 系统下运行的程序文件。

```
aa.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.9, not stripped
```

1.5　gdb 调试程序

所谓调试，指的是对编好的程序用各种手段进行查错和排错的过程。进行这种查错处理时，并不仅仅是运行一次程序检查结果，而是对程序的运行过程、程序中的变量进行各种分析和处理。本节将讲解使用 gdb 进行程序的调试。

1.5.1　gdb 简介

gdb 是一个功能强大的调试工具，可以用来调试 C 程序或 C++程序。在使用这个工具进行程序调试时，主要使用 gdb 进行下面 5 个方面的操作。

- 启动程序：在启动程序时，可以设置程序运行环境。
- 设置断点：断点就是可以在程序设计时暂停程序运行的标记。程序会在断点处停止，用户便于查看程序的运行情况。这里的断点可以是行数、程序名称或条件表达式。
- 查看信息：在断点停止后，可以查看程序的运行信息和显示程序变量的值。
- 分步运行：可以使程序一个语句一个语句的执行，这时可以及时地查看程序的信息。
- 改变环境：可以在程序运行时改变程序的运行环境和程序变量。

1.5.2　在程序中加入调试信息

为了使用 gdb 进行程序调试，需要在编译程序中加入供 gdb 使用的调试信息。方法是在编译程序时使用一个-g 参数。在终端中输入下面的命令，在编译程序时加入调试信息。

```
gcc -g -o a.debug a.c
```

这时，编译程序 a.c，生成一个 a.bedug 的可执行程序。这个可执行程序中加入了供调试所用的信息。

1.5.3　启动 gdb

在调试文件以前，需要启动 gdb。在终端中输入下面的命令。

```
gdb
```

这时，gdb 的启动信息如下所示。这些提示显示了 gdb 的版本和版权信息。

```
GNU gdb Red Hat Linux (6.6-35.fc8rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb)
```

1.5.4 在 gdb 中加载需要调试的程序

使用 gdb 调试一个程序之前，需要加载这个程序。加载程序的命令是 file。在(gdb)提示符后面输入下面的命令加载程序 a.debug。

```
file a.debug
```

命令的运行结果如下所示，显示已经加载了这个文件，并且使用了系统库文件。

```
Reading symbols from /root/c/a.debug...done.
Using host libthread_db library "/lib/libthread_db.so.1".
```

1.5.5 在 gdb 中查看代码

用 gcc 命令编译程序加入了-g 命令以后，编译后的 a.debug 程序中加入了断点。可以用 list 命令显示程序的源代码和断点。下面的步骤是查看加入断点以后的代码。

① 在(gdb)提示符后面输入下面的命令。

```
list 1
```

② 这时，gdb 会显示第一个断点以前的代码。显示的代码如下所示。

```
1       #include <stdio.h>
2
3       int max(int i,int j )
4       {
5               if(i>j)
6               {
7                       return(i);
8               }
9               else
10              {
(gdb)
```

③ 这时，按“Enter”键，显示下一个断点以前的代码。结果如下所示。

```
11                      return(j);
12              }
13      }
14
15      void main()
16      {
17         int i ,j,k;
18         i=3;
19         j=5;
20         printf("hello ,Linux.\n");
(gdb)
```

④ 按“Enter”键，显示下一个断点以前的代码。结果如下所示。

```
21         k=max(i,j);
22         printf("%d\n",k);
```

```
23      }
(gdb)
```

1.5.6　在程序中加入断点

程序会运行到断点的位置停止下来，等待用户处理信息或者查看中间变量。如果自动设置的断点不能满足调试要求，可以用 break 命令增加程序的断点。例如需要在程序的第 6 行增加一个断点，可以输入下面的命令。

```
break 6
```

这时 gdb 显示的结果如下所示。

```
Breakpoint 1 at 0x8048402: file a.c, line 6.
```

输入下面的命令，在程序的第 18 行、19 行、21 行增加断点。

```
break 18
break 19
break 21
```

1.5.7　查看断点

命令 info breakpoint 可以查看程序中设置的断点。输入“info breakpoint”命令，结果如下所示。显示程序中所有的断点。

```
1   breakpoint     keep y   0x08048402 in max at a.c:6
2   breakpoint     keep y   0x08048426 in main at a.c:18
3   breakpoint     keep y   0x0804842d in main at a.c:19
4   breakpoint     keep y   0x08048440 in main at a.c:21
```

加上相应的断点编号，可以查看这一个断点的信息。例如下面的命令就是查看第二个断点。

```
info breakpoint 2
```

显示的结果如下所示。

```
2   breakpoint     keep y   0x08048426 in main at a.c:18
```

1.5.8　运行程序

gdb 中的 run 命令可以使这个程序以调试的模式运行。下面的步骤是分步运行程序，对程序进行调试。

① 在(ddb)提示符后输入“run”命令，显示的结果如下所示。

```
Starting program: /root/c/a.debug
warning:      Missing      the      separate      debug      info      file:
/usr/lib/debug/.build-id/ac/2eeb206486bb7315d6ac4cd64de0cb50838ff6.debug
warning:      Missing      the      separate      debug      info      file:
/usr/lib/debug/.build-id/ba/4ea1118691c826426e9410cafb798f25cefad5.debug
Breakpoint 2, main () at a.c:18
18        i=3;
```

② 结果显示了程序中的异常，并将异常记录到了系统文件中。然后在程序的第二个断点的位置第 18 行停下。

③ 这时输入“next”命令，程序会在下一行停下，结果如下所示。

```
19          j=5;
```

④ 输入“continue”命令，程序会在下一个断点的位置停下。结果如下所示。

```
Continuing.
Breakpoint 3, main () at a.c:19
21          k=max(i,j);
```

⑤ 输入“continue”命令，程序运行到结束。结果如下所示，表明程序已经运行完毕正常退出。

```
5
Program exited with code 02.
```

⑥ step 命令与 next 命令的作用相似，对程序实现单步运行。不同之处是，在遇上函数调用时，step 函数可以进行到函数内部。而 next 函数只是一步完成函数的调用 。

1.5.9 变量的查看

print 命令可以在程序的运行中查看一个变量的值。本节将用下面的步骤来讲解变量的查看方法。

① 输入下面的命令，运行程序。

```
run
```

② 程序在第一个断点位置停下。显示的结果如下所示。

```
Breakpoint 2, main () at a.c:18
18          i=3;
```

③ 程序进入第 18 行之前停下，并没有对 i 进行赋值。可以用下面的命令来查看 i 的值。

```
print i
```

④ 显示的结果如下所示，表示 i 现在只是一个任意值。

```
$5 = -1076190040
```

⑤ 输入下面的命令，使程序运行一步。

```
step
```

⑥ 显示的结果如下所示。

```
19          j=5;
```

⑦ 这时程序在 19 行以前停下，这时输入下面的命令，查看 i 的值。

```
print i
```

⑧ 这时显示的 i 的结果如下所示。表明 i 已经赋值为 3。

```
$6 = 3
```

⑨ 这时输入“step”命令，再次输入“step”命令，显示的结果如下所示。

```
21        k=max(i,j);
```

⑩ 这时输入“step”命令，会进入到子函数中，结果如下所示。这时，显示了传递给函数的变量和值。

```
max (i=3, j=5) at a.c:5
5             if(i>j)
```

⑪ 这时，输入“step”命令，显示的结果如下所示，表明函数会返回变量 j。

```
11                    return(j);
```

⑫ 输入下面的命令，查看 j 的值。

```
print j
```

⑬ 显示的结果如下所示，表明 j 的值为 5。

```
$7 = 5
```

⑭ 这时再运行两次“step”命令，显示的结果如下所示。

```
22        printf("%d\n",k);
```

⑮ 这时，输入下面的命令，查看 k 的值。

```
print k
```

⑯ 显示的结果如下所示，表明 k 的值为 5。

```
$8 = 5
```

⑰ 完成了程序的调试运行以后，输入“q”命令，退出 gdb。

1.6　程序调试实例

本节讲解一个程序调试实例。先编写一个程序，在程序运行时，发现结果与预想结果有些不同。然后用 gdb 工具进行调试，通过对单步运行和变量的查看，查找出程序的错误。

1.6.1　编写一个程序

本节将编写一个程序，要求程序运行时可以显示下面的结果。

```
1+1=2
2+1=3 2+2=4
3+1=4 3+2=5 3+3=6
4+1=5 4+2=6 4+3=7 4+4=8
```

很明显，这个程序是通过两次循环与一次判断得到的。程序中需要定义三个变量。下面用这个思路来编写这个程序。

① 打开一个终端。在终端中输入“vim”命令，打开 VIM。

② 在 VIM 中按“i”键，进入到插入模式。然后在 VIM 中输入下面的代码。

```
#include <stdio.h>
main()
{
    int i,j,k;
    for(i=1;i<=4;i++)
    {
        for(j=1;j<=4;j++);
        {
            if(i>=j)
            {
                k=i+j;
                printf("%d+%d=%d ",i,j,k);
            }
        }
        printf("\n");
    }
}
```

③ 在 VIM 中按“Esc”键，返回到普通模式。然后输入下面的命令，保存这个文件。

```
:w /root/c/test.c
```

④ 输入“:q”命令退出 VIM。很容易发现，在第二个循环后

1.6.2 编译文件

本节将对上一节编写的程序进行编译和运行。在运行程序时，会发现程序有错误。

① 在终端中输入下面的命令，编译这个程序。

```
gcc /root/c/test.c
```

② 程序可以正常编译通过，输入下面的命令，运行这个程序。

```
/root/c/a.out
```

③ 程序的显示结果是 4 个空行，并没有按照预想的要求输出结果。

④ 输入下面的命令，对这个程序进行编译。在编译加入-g 参数，为 gdb 调试做准备。

```
gcc -g -o test.debug 6.2.c
```

⑤ 这时，程序可以正常编译通过。输出的文件是 test.debug。这个文件中加入了文件调试需要的信息。

1.6.3 程序的调试

本节将讲述使用 gdb 对上一节编写的程序进行调试，查找出程序中的错误。

① 在终端中输入“gdb”命令，进入到 gdb，显示的结果如下所示。

```
GNU gdb Red Hat Linux (6.6-35.fc8rh)
Copyright (C) 2006 Free Software Foundation, Inc.
```

```
    GDB is free software, covered by the GNU General Public License, and you are
    welcome to change it and/or distribute copies of it under certain conditions.
    Type "show copying" to see the conditions.
    There is absolutely no warranty for GDB.  Type "show warranty" for details.
    This GDB was configured as "i386-redhat-linux-gnu".
```

② 导入文件。在 gdb 中输入下面的命令。

```
    file /root/c/test.debug
```

③ 这时显示的结果如下所示。表明已经成功加载了这个文件。

```
    Reading  symbols  from  /root/c/test.debug...(no  debugging  symbols
found)...done.
    Using host libthread_db library "/lib/libthread_db.so.1".
```

④ 查看文件。在终端中输入下面的命令。

```
    list
```

⑤ 显示的文件查看结果如下所示。

```
    1       #include <stdio.h>
    2
    3       main()
    4       {
    5               int i,j,k;
    6               for(i=1;i<=4;i++)
    7               {
    8                       for(j=1;j<=4;j++);
    9                       {
    10                              if(i>=j)
    (gdb)
    11                              {
    12                                      k=i+j;
    13                                      printf("%d+%d=%d ",i,j,k);
    14                              }
    15                      }
    16                      printf("\n");
    17              }
    18      }
    (gdb)
    Line number 19 out of range; 6.2.c has 18 lines.
```

⑥ 在程序中加入断点。从显示的代码可知，需要在第 6 行、第 11 行、第 12 行和第 13 行加入断点。在 gdb 中输入下面的命令。

```
    break 6
    break 11
    break 12
    break 13
```

⑦ gdb 显示的添加断点的结果如下所示。

```
Breakpoint 1 at 0x8048405: file 6.2.c, line 6.
Breakpoint 2 at 0x8048429: file 6.2.c, line 11.
Breakpoint 3 at 0x8048429: file 6.2.c, line 12.
Breakpoint 4 at 0x8048432: file 6.2.c, line 13.
```

⑧ 输入下面的命令，运行这个程序。

```
run
```

⑨ 运行到第一个断点显示的结果如下所示。

```
Breakpoint 1, main () at 6.2.c:6
6             for(i=1;i<=4;i++)
```

⑩ 输入“step”命令，程序运行一步，结果如下所示。

```
8                     for(j=1;j<=4;j++);
```

⑪ 这说明程序已经进入了 for 循环。这时输入下面命令，查看 i 的值。

```
print i
```

⑫ 显示的结果如下所示。

```
$2 = 1
```

⑬ 这时再输入“step”命令，显示的结果如下所示。

```
10                          if(i>=j)
```

⑭ 这时再输入“step”命令，显示的结果如下所示。

```
16                    printf("\n");
```

⑮ 这表明，在进行 j 的 for 循环时，没有反复执行循环体。这时再输入“step”命令，显示的结果如下所示。

```
 for(i=1;i<=4;i++)
```

⑯ 这表明，程序正常的进行了 i 的 for 循环。这是第二次执行 for 循环。

⑰ 输入“step”命令，显示的结果如下所示。

```
8                   for(j=1;j<=4;j++);
```

⑱ 这表明，程序执行到 for 循环。这时再次输入“step”命令，显示的结果如下所示。

```
10                          if(i>=j)
```

⑲ 输入“step”命令，显示的结果如下所示。

```
16                    printf("\n");
```

⑳ 输入“step”命令，显示的结果如下所示。

```
 6             for(i=1;i<=4;i++)
```

㉑ 这说明，程序正常的进行了 i 的 for 循环，但是没有执行 j 的 for 循环。这一定是 j 的

for 循环语句有问题。这时就不难发现 j 的 for 循环后面多了一个分号。

㉒ 输入“q”命令，退出 gdb。

1.6.4　gdb 帮助的使用

gdb 有非常多的命令。输入“help”命令可以显示这些命令的帮助信息。本节将讲解帮助信息的使用。

① 在 gdb 输入“help”命令，显示的帮助信息如下所示。

```
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

② 上面的帮助信息显示，输入“help all”会输出所有帮助信息。

③ 在“help”命令后面加上一个命令名称，可以显示这个命令的帮助信息。例如输入“help file”，显示的 file 命令帮助信息如下所示。

```
Use FILE as program to be debugged.
It is read for its symbols, for getting the contents of pure memory,
and it is the program executed when you use the `run' command.
If FILE cannot be found as specified, your execution directory path
($PATH) is searched for a command of that name.
No arg means to have no executable file and no symbols.
```

1.7　gdb 常用命令

除了前面讲述的 gdb 命令以外，gdb 还有很多种命令。这些命令可以完成程序调试的各种功能。其他的常用命令含义如下所示。

- backtrace：显示程序中的当前位置和表示如何到达当前位置的栈跟踪（同义词：where）。
- breakpoint：在程序中设置一个断点。
- cd：改变当前工作目录。
- clear：删除刚才停止处的断点。

- commands：命中断点时，列出将要执行的命令。
- continue：从断点开始继续执行。
- delete：删除一个断点或监测点，也可与其他命令一起使用。
- display：程序停止时显示变量和表达式。
- down：下移栈帧，使得另一个函数成为当前函数。
- frame：选择下一条 continue 命令的帧。
- info：显示与该程序有关的各种信息。
- info break：显示当前断点清单，包括到达断点处的次数等。
- info files：显示被调试文件的详细信息。
- info func：显示所有的函数名称。
- info local：显示当函数中的局部变量信息。
- info prog：显示被调试程序的执行状态。
- info var：显示所有的全局和静态变量名称。
- jump：在源程序中的另一点开始运行。
- kill：异常终止在 gdb 控制下运行的程序。
- list：列出相应于正在执行的程序的源文件内容。
- next：执行下一个源程序行，从而执行其整体中的一个函数。
- print：显示变量或表达式的值。
- pwd：显示当前工作目录。
- pype：显示一个数据结构（如一个结构或 C++类）的内容。
- quit：退出 gdb。
- reverse-search：在源文件中反向搜索正规表达式。
- run：执行该程序。
- search：在源文件中搜索正规表达式。
- set variable：给变量赋值。
- signal：将一个信号发送到正在运行的进程。
- step：执行下一个源程序行，必要时进入下一个函数。
- undisplay display：命令的反命令，不要显示表达式。
- until：结束当前循环。
- up：上移栈帧，使另一函数成为当前函数。
- watch：在程序中设置一个监测点（即数据断点）。
- whatis：显示变量或函数类型。

1.8 编译程序常见的错误与问题

在编写程序时，无论是逻辑上还是语法上，不可能一次做到完全正确。于是在编译程序时，就会发生编译错误。本节将讲述程序编译时常见的错误类型与处理方法。

1.8.1　逻辑错误与语法错误

在编程时，出现的错误可能有逻辑错误和语法错误两种。这两种错误的发生原因和处理方法是不同的。本节将讲述这两种错误的处理方法。

- 逻辑错误指的是程序的设计思路发生了错误。这种错误在程序中是致命的，程序可能正常编译通过，但是结果是错误的。当程序正常运行而结果错误时，一般都是编程的思路错误。这时，需要重新考虑程序的运算方法与数据处理流程是否正确。
- 语法错误：语法错误指的是程序的思路正确，但是在书写语句时，发生了语句错误。这种错误一般是编程时不小心或是对语句的错误理解造成的。在发生语句错误时，程序一般不能正常编译通过。这时会提示错误的类型和错误的位置，按照这些提示改正程序的语法错误即可完成错误的修改。

1.8.2　C 程序中的错误与异常

C 程序中的错误，根据严重程序的不同，可以分为异常与警误两类。在编译程序时，这两种情况对编译的影响是不同的，对错误与异常的处理方式是不同的。

1．什么是异常

异常指的是代码中轻微的错误，这些错误一般不会影响程序的正常运行，但是不完全符合编程的规范。在编译程序时，会产生一个“警告”，但是程序会继续编译。下面的程序会使程序发生异常，在编译时产生一个警告错误。

- 在除法中，0 作除数。
- 在开方运算时，对负数开平方。
- 程序的主函数没有声明类型。
- 程序的主函数没有返回值。
- 程序中定义了一个变量，但是没有使用这个变量。
- 变量的存储发生了溢出。

2．什么是错误

错误指的是程序的语法出现问题，程序编译不能正常完成，产生一个错误信息。这时会显示错误的类型与位置。根据这些信息可以对程序进行修改。

1.8.3　编译中的警告提示

在编译程序时，如果发生了不严重的异常，会输出一个错告错误，然后完成程序的编译。例如下面的内容是一个程序在编译时产生的警告。

```
5.1.c: In function 'main':
5.1.c:16: 警告：‘main’ 的返回类型不是 ‘int’
5.1.c:18: 警告：被零除
```

这些的含义如下所示。

（1）“In function 'main':”表示发生的异常在 main 函数内。

（2）“5.1.c:16:”表示发生异常的文件是 5.1.c，位置是第 16 行。

（3）下面的信息是第 16 行的异常，表明程序的返回类型不正确。

```
‘main’ 的返回类型不是 ‘int’
```

（4）下面的警告信息表明程序的第 18 行有除数为 0 的错误。

```
5.1.c:18: 警告：被零除
```

1.8.4 找不到包含文件的错误

程序中的包含文件在系统或工程中一定要存在，否则程序编译时会发生致命错误。例如下面的语句包含了一个不正确的头文件。

```
#include <stdio1.h>
```

编译程序时，会发生错误，错误信息如下所示。

```
5.1.c:2:20: 错误：stdio2.h：没有那个文件或目录
```

1.8.5 错误地使用逗号

程序中逗号的含义是并列几个内容，形成某种算法或结构。程序中如果错误地使用逗号，会使程序在编译时发生致命错误。例如下面的代码，是程序中的 if 语句后面有一个错误的逗号。

```
int max(int i,int j )
{
    if(i>j),
    {
        return(i);
    }
    else
    {
        return(j);
    }
}
```

程序编译时输出的错误信息如下所示。表明 max 函数中逗号前面的表达式有错误，实际上的错误是多一个逗号。

```
5.1.c: In function ‘max’:
5.1.c:4: 错误：expected expression before ‘,’ token
5.1.c: In function ‘max’:
```

1.8.6 括号不匹配错误

程序中的引号、单引号、小括号、中括号、大括号等符号必须成对出现。这方面的错误会使程序发生符号不匹配的错误。发生这种错误后，编译程序往往不能理解代码的含义，也不能准确显示错误的位置，而是显示表达式错误。例如下面的代码，在最后一行上了一个花括号。

```
int max(int i,int j )
```

```
{
    if(i>j)
    {
        return(i);
    }
    else
    {
        return(j);
    }
```

编译程序时，会显示下面的错误信息。

```
5.1.c:22: 错误: expected declaration or statement at end of input
```

1.8.7　小括号不匹配错误

程序中的小括号一般在一行内成对出现并且相匹配。小括号不匹配时，程序发生致命错误。例如下面的代码，第一行多了一个右半边括号。

```
if(i>j))
    {
        return(i);
    }
    else
    {
        return(j);
    }
```

编程程序时，会发生下面的错误。显示括号前面有错误，并且导致下面的 else 语句也有错误。

```
5.1.c:4: 错误: expected statement before ‘)’ token
5.1.c:8: 错误: expected expression before ‘else’
```

1.8.8　变量类型或结构体声明错误

程序中的变量或结构体的名称必须正确，否则程序会发生未声明的错误。例如下面的代码，用一个不存在的类型来声明一个变量。

```
ch a;
```

程序在运行时，会显示出这个变量错误，并且会显示有其他的错误。

```
5.1.c:17: 错误: ‘ch’ 未声明 (在此函数内第一次使用)
5.1.c:17: 错误: (即使在一个函数内多次出现，每个未声明的标识符在其
5.1.c:17: 错误: 所在的函数内只报告一次。)
5.1.c:17: 错误: expected ‘;’ before ‘a’
```

1.8.9　使用不存在的函数的错误

如果程序引用了一个不存在的函数，会使用程序发生严重的错误。例如下面的代码，引用了一个不存在的函数 add。

```
k=add(i,j);
```

程序显示的错误信息如下所示，表明在 main 函数中的 add 函数没有定义。

```
/tmp/ccYQfDJy.o: In function `main':
5.1.c:(.text+0x61): undefined reference to `add'
collect2: ld 返回 1
```

5.8.10 大小写错误

C 程序对代码的大小写是敏感的，不同的大小写代表不同的内容。例如下面的代码，将小写的“int”错误的写成了“Int”。

```
Int t;
```

程序显示的错误信息如下所示，表明“Int”类型不存在或未声明。发生这个错误时，会输出多行错误提示。

```
5.1.c:16: 错误：‘Int’ 未声明 (在此函数内第一次使用)
5.1.c:16: 错误：(即使在一个函数内多次出现，每个未声明的标识符在其
5.1.c:16: 错误：所在的函数内只报告一次。)
5.1.c:16: 错误：expected ‘;’ before ‘t’
```

1.8.11 数据类型的错误

程序中的某些运算，必须针对相应的数据类型，否则这个运算会发生数据类型错误。例如下面的代码，错误地将两个整型数进行求余运算。

```
float a,b;
a= a %b ;
```

程序编译时，输出下面的错误，表明“%”运算符的操作数无效。

```
5.1.c:19: 错误：双目运算符 % 操作数无效
```

1.8.12 赋值类型错误

任何一个变量，在赋值时必须使用相同的数据类型。例如下面的代码，错误地将一个字符串赋值给一个字符。

```
char c;
c="a";
```

程序编译时的结果如下所示，表明赋值时数据类型错误。

```
5.1.c:19: 警告：赋值时将指针赋给整数，未作类型转换
```

1.8.13 循环或判断语句中多加分号

分号在程序中的作用是表示一个语句结束。在程序的语句中用一个单独的分号表示一个空语句。但是在循环或判断结构的后面，一个分号会导致程序的逻辑发生错误。关于这些结构的使用方法，后面的章节将会详细讲到。下面的程序，在 for 语句的后面，错误的添加了一

个分号，导致程序不能正常地进行循环。

```
#include <stdio.h>
main()
{
  int sum, j;
  sum=0;
  for(j=0;j<11;j++);
  {
    sum=sum+j;
  }
printf("%d",sum);
}
```

这个程序的本意是要求出 10 以内的整数和。但是在 for 语句的后面，错误地使用了一个分号。这时，程序不能正确地进行循环，而是把分号作为一个语句进行循环，所以程序输出的结果为“11”。

1.9　小结

程序的编译和调试是编程的一个重要环节。本章讲解了 Linux 系统中 C 编程的编译器 gcc 和编译器 gdb 的使用。使用 gcc 时，需要对编译进行各种设置，需要理解 gcc 各项参数的作用。gdb 的学习重点是 gdb 单步运行程序的理解，通过程序的单步运行发现程序中的问题。

第 2 章　C 语言基础

2.1　C 程序的基本概念

C 语言有着严格的格式和语法规则，用户需要按照这些要求来编写程序。本章将讲解 C 程序的组成、语句、注释等基本概念。

2.1.1　C 程序的基本结构

C 程序由语句、函数、包含文件等部分组成。本节将以一个简单的 C 程序实例来讲解 C 程序的基本结构问题。这个程序实现两个整数的大小判断功能。

（1）单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。在终端中输入“vim”命令，打开 VIM。

（2）输入程序。在 VIM 中按“i”键，进入到插入模式，然后在 VIM 中输入下面的代码。需要注意的是，/* */符号中的内容是注释，程序中不必输入这些注释。

```
#include <stdio.h>          /*包含文件。*/
int max(int i , int j)
{
                            /*注释：这是一个自定义函数，实现两个整数的大小比较功能。 */
if(i>=j)
  return(i);                /*返回较大的数。*/
else
  return(j);
}
int main()                  /*主函数。*/
{
int i =3;                   /*定义三个变量，并且赋值。*/
int j =5;
int t;
t=max(3,5);                 /*将较大的数赋值给 t。*/
printf("the big number is %d\n",t);     /*输出文本和结果。*/
}
```

（3）保存文件。输入这些代码以后，按“Esc”键返回到普通模式，输入命令“:w 2.1.c”，保存这个文件。然后输入命令“:q”退出 VIM。

（4）在终端中输入下面的命令，编译这个程序。

```
gcc -o 2.1.out 2.1.c
```

（5）在终端中输入下面的命令，对这个程序添加可执行权限。

```
chmod +x 2.1.out
```

（6）输入下面的命令，运行这个程序。

```
./2.1.out
```

（7）程序的运行结果如下所示，显示较大值是 5。

```
the big number is 5
```

由上面的例子可知，C 程序的源代码有下面的特点。

（1）程序一般用小写字母书写。

（2）大多数语句结尾必须要分号作为终止符，表示一个语句结束。同一个语句需要写在一行上。

（3）每个程序必须有一个主函数，主函数用 main()声明，并且只能有一个主函数。在 Linux 系统中，main 主函数应该是 int 类型。

（4）每个程序中的自定义函数和主函数，需要用一对大括号括起来。

（5）程序需要使用#include <>语句来包含系统文件，这些系统文件完成系统函数的定义。

（6）一个较完整的程序大致包括下面这些内容。

- 包含文件：一组#include <*.h>语句。
- 用户自定义函数：用户已编写的完成特定功能的模块。
- 主函数：程序自动执行的程序体。
- 变量定义：定义变量以存储程序中的数据。
- 数据运算：程序通过运算完成各种逻辑功能。这些运算由各种语句和函数实现的。
- 注释：注释写在/* */符号之间，不是程序的必需部分，但是可以增强程序的可读性。

需要注意的是，C 程序是严格区分大小写的，程序中同一个字母的和大写小写字母代表不同的内容。可以将多个语句写在一行上，但是每个语句必须有一个分号结束。例如本节代码中的 max 函数，也可以写成下面的形式。

```
int max(int i , int j){  if(i>=j)  return(i);  else  return(j);}
```

这段程序也是可以正常执行的，但是不便于程序的阅读与修改。在编程中需要使用正确的格式，每个语句写在一行上，并且使用正确的注释与缩进，使代码的层次清晰明了。

2.1.2　C 程序的一般格式

通过上一节的例子可知程序的主要部分是主函数和自定义函数。C 程序的一般结构如下所示。

```
程序的包含文件
子函数类型声明
全局变量定义
main 主函数()
{
   局部变量定义
   <程序体>
}
```

```
自定义函数 1()
{
   局部变量定义
   <程序体>
}
自定义函数 2()
{
   局部变量定义
   <程序体>
}
……
自定义函数 N()
{
   局部变量定义
   <程序体>
}
```

主函数与自定义函数的位置是随意的。自定义函数可以写在主函数的前面或后面，如果放在后面，需要在程序的前面声明这个函数。

2.1.3 C 程序中的注释

所谓注释，就是在编写程序时对代码的补充说明，不影响程序。任何一个程序员，都不可能完全记忆所写过的代码，注释的作用是对代码的阅读和理解进行提示。程序在编译时，会自动去除注释的内容。

在编写程序时，需要正确地书写注释，养成良好的注释习惯。例如下面代码中使用的注释方法就是常用的注释方法。其中，程序右侧的注释可以按“Tab”键使注释对齐。

```
/*  作者：小明。
    内容：变量的输入与输出。
    时间：2007.11.18
    程序的最前面注释出代码的信息。*/
#include <stdio.h>                          /*包含文件。*/
int main()                                  /*主函数。*/
{
int i ;                                     /*定义一个变量。*/
scanf("%d",&i);                             /*输入一个变量。*/
printf("%d",i)                              /*输出变量。*/
}
```

2.2 数据类型

数据类型指的是一类数据的集合，是对数据的抽象描述。数据类型的不同决定了所占存储空间的大小不同。每个变量在使用之前必须定义其数据类型。C 程序有整型（int）、浮点型（float）、字符型（char）、指针型（*）、无值型（void）这些常用数据类型。还有结构体（struct）和联合体（union）两种自定义数据类型。本章将讲解前三种基本的数据类型。

2.2.1　整型（int）

整型可以简单理解为整数。与整数不同的是，一个整型变量有一定的字节长度和存储数据范围。整型变量是有正负的，在定义整型变量时，需要注意变量的正负问题。如表 2.1 所示是不同长度与正负的整型变量。

表 2.1　整型变量

类　型	说　明
signed short int	有符号短整型数。简写为 short 或 int，字长为 2 字节，数的范围是-32768~32767
signed long int	有符号长整型数。简写为 long，字长为 4 字节，数的范围是-2147483648~2147483647
unsigned short int	无符号短整型数。简写为 unsigned int，字长为 2 字节，数的范围是 0~65535
unsigned long int	无符号长整型数。简写为 unsigned long，字长为 4 字节，数的范围是 0~4294967295

例如下面的代码中定义和使用了整型变量。

```
#include <stdio.h>
int main()
{
int i ;                          /*定义一个 signed short int 变量 i。*/
long j;                          /*定义一个 signed long int 变量 i。*/
unsigned short k;                /*定义一个 unsigned short int 变量 i。*/
i=123;                           /*i 赋值为 123。*/
j=1234L;                         /*j 赋值为 1234。*/
k=0x5e;                          /*k 赋值为十六进制的 5e。*/
printf("%d\n",i);
printf("%ld\n",j);
printf("%d\n",k);
}
```

用下面的命令编译这段代码。

```
gcc 2.2.c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
123
1234
94
```

在程序中整型变量有以下三种表示方法。

- 普通十进制数，在程序中除了 0 以外不以 0 开始。如 0，123，-123 等。
- 八进制数，程序中以 0 开始，如 013，-013 等。
- 十六进制数，程序中以 0x 或 0X 开始，如 0x1E、0XEF 等。

另外，可在整型常数后添加一个“L”或“l”字母表示该数为长整型数，如代码中的 j=1234L，表示 j 为长整型数。

2.2.2 浮点型（float）

浮点型数指的是数值的小数点在存储空间中可以移动，可以用来表示不同精度与大小的数值。小数、科学计数法的值都是用浮点型变量来存储的。根据所占的存储空间与表示范围不同，浮点型数可以分为下面的两种。

- float 单精度浮点数，字长为 4 个字节共 32 位二进制数。可表示数的范围是 3.4x10-38E~3.4x10+38E。
- double 双精度浮点数，字长为 8 个字节共 64 位二进制数，可表示数的范围是 1.7x10-308E~1.7x10+308E。

```
#include <stdio.h>
int main()
{
float m;                              //定义一个 float 变量。
double n ;                            //定义一个 double 变量。
m=1.234;                              //对 m 赋值。
n=.1234;
printf("%f\n",m);                     //输出变量。
printf("%f\n",n);
}
```

输入下面的命令，编译这个程序。

```
gcc 2.3.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
1.234000
0.123400
```

与整数变量相比，浮点型变量在下面这些方面是不同的。

- 浮点常数只有十进制一种进制。
- 绝对值小于 1 的浮点数，其小数点前面的零可以省略。如 0.123 可写为.123。
- 用默认格式输出浮点数时，都是保留 6 位小数，没有小数的后面加零。

2.2.3 字符型(char)

字符型变量在计算机中以 ASCII 码方式表示，长度为一个字节。各种字母、符号都可以是一个字符型的变量。数字 0~9 也可以存储为一个字符型变量。当数字存储为字符型变量时，

保存的是这个数字的 ASCII 码的值，与 int 型保存数值是不相同的。

> **注意**：ASCII 码是计算机内部的字符编码集。所有的信息都是以 ASCII 码的形式保存在计算机中的。一个字符变量对应 ASCII 码表中的一个字符。

在程序中，字符可直接用单引号引起来。如‘A’、‘b’、‘9’、‘@’都是一个字符型变量，也可以用这一字符的 ASCII 码值来表示一个字符，例如 87 表示大写字母‘W’。

除了屏幕上可以直接显示的字符以外，还有一些字符是用来进行格式控制，并不能直接显示在屏幕上。例如\n 表示一个换行符，ASCII 码为八进制 010，而不是一个字符\加一个字符 n。C 程序中常用的转义字符如表 2.2 所示。

表 2.2　转义字符

字　　符	名　　称	ASCII 码
\a	响铃（BEL）	007
\b	退格（BS）	008
\f	换页（FF）	012
\n	换行（LF）	010
\r	回车（CR）	013
\t	水平制表（HT）	009
\v	垂直制表（VT）	011
\\	反斜杠	092
\?	问号字符	063
\'	单引号字符	039
\"	双引号字符	034
\0	空字符（NULL）	000

在程序中字符变量需要用 char 来定义。例如下面的代码，是字符变量的定义与使用实例。

```
#include <stdio.h>
int main()
{
char m;
char n;
char t;
m='a';
n='!';
t='\n';
printf("%c\n",m);                    /*输出 a 再输出一个换行。*/
printf("%c\n",n);                    /*输出!再输出一个换行。*/
printf("%c\n",t);                    /*输出的 t 为一个换行.*/
}
```

输入下面的命令，编译这个程序。

```
gcc 2.4.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。结果如下所示。

```
a
!
```

2.2.4 变量名

变量名只能由字母、数字、下画线组成，且头一个字符只能是字母或下画线，变量名之间不能有空格或其他字符。这样的变量才是合法的。例如下面这些变量名是合法变量名。

```
ab   _a   a2  AAA   A_B   __AB
```

下面这些变量名，不符合变量名的要求，是不合法的。

```
3a                  /*不能以数字开头。*/
a.b                 /*变量名之间不能有特殊符号。*/
$ab                 /*不能以特殊符号开头。*/
a b                 /*变量名中不能有空格。*/
```

不合法的变量名会引起程序的错误。除了这些变量名以外，系统关键字也不能作为变量名。例如下面这些词是系统关键字，作为变量名时会引发错误。

```
int   printf   float   exit
```

2.2.5 字符 NULL

NULL 是一个特殊的字符，在 C 程序中有着重要的功能。NULL 在 ASCII 码表中是 0，在程序中表示空字符，或者没有这个数值。在定义变量时，如果要对变量赋值为空值，可将这个变量赋值为 NULL，如下面代码所示。

```
int a=NULL;
char b=NULL;
float c=NULL;
```

> **注意**：ASCII 码是计算机字符的内部编码，每一个字符对应 ASCII 码表中的一个编号。详细介绍见本书 8.5.3 节中的内容。

2.3 变量的赋值与输出

变量的赋值指的是对一个变量指定一个值，这个值可以用于程序的运算。变量的输出指的是将变量的值显示在屏幕上，一般情况下，程序执行完毕以后需要将结果输出。

2.3.1 变量的赋值

赋值符号是等号，含义是将等号右边的值或表达式结果传递给等号左边的变量。例如下面的代码就是定义变量和赋值的例子。

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

```
{
int i =0,j=1,k=123;                    /*定义三个整型变量，定义变量同时赋值。*/
int s;
char a,b,c;                            /*定义三个字符型变量。*/
s=j;                                   /*将变量 j 的值赋值给 s。*/
a='a';                                 /*将字符 a 赋值给变量 a。*/
b=a;                                   /*把变量 a 的值赋值给 b。*/
c=b;                                   /*把变量 b 的值赋值给 c。*/
}
```

2.3.2　printf 函数输出变量

在前面的代码中，都是使用 printf()函数进行变量的输出。printf()函数称为格式输出函数，功能是按用户指定的格式，把指定的数据显示到屏幕上。

Printf()函数是一个标准库函数，它的函数原型在头文件“stdio.h”中，该函数的使用方法如下所示。

```
printf(“格式控制字符串”,输出变量列表)
```

输出变量列表中给出了各个输出变量。格式控制字符串用于指定输出格式，由格式字符串和非格式字符串两种组成。

- 格式字符串是以%开头的字符串，在%后面跟有各种格式字符。不同格式字符串的作用是说明输出数据的类型、形式、长度、小数位数等。如“%d”表示按十进制整型输出，“%ld”表示按十进制长整型输出。这些格式字符串与输出变量列表中的变量一一对应。
- 格式字符串以外的内容是非格式字符串，在输出时按照原有的字符直接输出。

格式字符串的内容和意义如下所示。

- %c：输出单个字符，参数为该字符的 ASCII 码。
- %d：以十进制形式输出带符号整数（正数不输出符号）。
- %e 或%E：以指数形式输出单、双精度实数，默认保留 6 位小数。
- %f：以小数形式输出单或双精度实数，默认保留 6 位小数。
- %g 或%G：以%f 或%e 中较短的输出宽度输出单、双精度实数。如果指数小于-4 或大于等于默认精度，则使用%e 或%E 格式输出。否则用%f 格式输出，省略末尾多余的 0。
- %i：以十进制形式输出带符号整数，同%d。
- %o：以八进制形式输出无符号整数（不输出前缀 0）。
- %s：输出字符串，参数为 char 指针，显示字符串中所有的字符。
- %u：以十进制形式输出无符号整数。
- %x 或%X：以十六进制形式输出无符号整数，%x 表示输出小写，%X 表示输出大写。

例如下面的代码是 printf()函数格式化输出的实例。

```
#include <stdio.h>
int main()
{
  int i=117;                                   /*定义整型变量 i=117。*/
  char a='m';                                  /*定义字符定量 a。*/
```

```
    float m=123.1234567 ;                    /*定义一个浮点型变量。*/
    printf("%d\n",i);                        /*以整型输出 i。*/
    printf("%c\n",i);                        /*以字符型输出 i。*/
    printf("%o\n",i);                        /*以八进制输出 i。*/
    printf("%x\n",i);                        /*以十六进制小写输出 i。*/
    printf("%X\n",i);                        /*以十六进制大写输出 i。*/
    printf("%c\n",a);                        /*以字符型输出 a。*/
    printf("%d\n",a);                        /*以整型输出 a。*/
    printf("%o\n",a);                        /*以八进制输出 a。*/
    printf("%x\n",a);                        /*以十六进制小写输出 a。*/
    printf("%X\n",a);                        /*以十六进制大写输出 a。*/
    printf("%f\n",m);                        /*以浮点型输出 f。*/
    printf("%e\n",m);                        /*以科学计数法输出 m。*/
}
```

输入下面的命令，编译这一个文件。

```
gcc 2.5.c
```

输入下面的命令，对编译的文件添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这一个程序。

```
./a.out
```

程序的运行结果如下所示。

```
117
u
165
75
75
m
109
155
6d
6D
123.123459
1.23123e+02
```

从程序的运行结果可知，字符是以 ASCII 码的形式保存的，相对应一相整型的。当整型数 117 以整型输出时是 117，而以字符型输出时是 u。

2.3.3 scanf 函数从键盘读入变量

scanf()函数是键盘格式化输入函数，可以从键盘输入读取信息，这个函数的使用方法如下所示。

```
scanf("<格式化字符串>", <地址表>);
```

这里的格式化字符串与上一节中的格式化字符串含义是相同的。当有多个要读取的变量

时，可以用空格或其他符号隔开，在键盘输入时也相应地输入这些符号。例如下面的代码就是使用 scanf()函数从键盘读取输入，再输出变量的例子。

```
#include <stdio.h>
int main()
{
   int i ,j ;                                    /*定义程序中的变量。*/
   int m;
   printf("please enter a char:\n");              /*提示信息。*/
   scanf("%c",&m);                               /*输入一个字母。*/
   printf("the char is %c.\n",m);                 /*输出一个字母。*/
   printf("please enter a number:\n");
   scanf("%d",&i);                               /*输入一个整数。*/
   printf("please enter another number:\n");
   scanf("%d",&j);
   printf("%d + %d = %d \n",i ,j ,i+j);          /*输出两个整数和这两个整数的和。*/
}
```

输入下面的命令，编译这一个文件。

```
gcc 2.9.c
```

输入下面的命令，对编译的文件添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这一个程序。

```
./a.out
```

运行程序时，光标等待输入内容。输入内容以后，按“Enter”键继续向后执行。程序的运行结果如下所示。

```
please enter a char:
```

这时从键盘输入一个字母 a，然后按“Enter”键。这时程序的结果如下所示。

```
the char is a.
please enter a number:
```

这时输入一个数字 5，然后按“Enter”键，程序显示结果如下所示。

```
please enter another number:
```

这时输入一个数字 3，然后按“Enter”键，程序显示结果如下所示。

```
5 + 3 = 8
```

2.4　运算符

运算符指的是用来完成各种运算的符号。C 程序中有着丰富的运算符，很多运算都是通过运算符来实现的。本节将讲解常用运算符的操作。

2.4.1 算术运算符

算术运算符是指进行简单数学计算的运算符，是程序中最简单最常用的运算符。算术运算符的种类和使用方法如表 2.3 所示。

表 2.3 算术运算符

运 算 符	含 义
+ -	加号和减号
* /	乘号和除号
++	自加，在原来的值上面加 1
--	自减，在原来的值上面减 1
+= -=	相加赋值和相减赋值，对原有值加或者减去一个值
*= /=	相乘赋值或相除赋值，对原有值乘或除一个值
%	求余运算，第一个数除以第二个数求余数

例如下面的代码是使用这些运算符进行简单的数学运算的例子。

```
#include <stdio.h>
int main()
{
   int i =3 ,j = 7 ,k;                    /*定义变量和赋值。*/
   float m=2.5 , n = 3.8,t;
   k= 3+7;                                /*做加法运算。*/
   printf("%d\n",k);
   k*=5;                                  /* *=运算，表示把 k 乘以 5 再赋值给 k。*/
   printf("%d\n",k);
   k=k%j;                                 /*求余运算。*/
   printf("%d\n",k);
   printf("%d\n",i+j);                    /*可以直接在参数列表中进行运算。*/
   t= 3.8/2.5;
   printf("%f\n",t);                      /*以浮点数的形式输出。*/
   printf("%f\n",m*n+5);                  /*直接在参数列表中进行运算。*/
}
```

输入下面的命令编译这个程序。

```
gcc 2.2.c
```

输入下面的命令，对编译后的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
10
50
1
10
```

```
1.520000
14.500000
```

对于++、--运算的使用，将符号写在前面和写在后面的含义是不同的。++写在后面时，表示取得值然后再自加，写在前面时，表示先自加再取值。例如下面的代码，就是自加和自减运算符使用的例子。

```
#include <stdio.h>
int main()
{
   int i =3 ,j = 7;
   i++;                                    /*i 的值为 4。*/
   printf("%d\n",i);
   printf("%d\n",i++);                     /*输出 i 的值为 4，然后自加得 5。*/
   printf("%d\n",i++);                     /*输出 i 的值为 5，然后自加得 6。*/
   printf("%d\n",++i);                     /*自加得 7，输出结果得 7。*/
   printf("%d\n",--i);                     /*自减得 6，输出结果为 6。*/
   printf("%d\n",i++ + j++);               /*先取值相加得 13，然后分别自加。*/
}
```

输入下面的命令编译这个程序。

```
gcc 2.7.c
```

输入下面的命令，对编译后的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
4
4
5
7
6
13
```

2.4.2　关系运算符

所谓关系运算符，指的是比较两个数的大小关系或相等关系的符号。关系运算符的种类与使用方法如表 2.4 所示。

表 2.4　关系运算符

运 算 符	含　义
>	大于。第一个数大于第二个数时返回真
>=	大于等于。第一个数大于等于第二个数时返回真
<	小于。第一个数小于第二个数时返回真
<=	小于等于。第一个数小于等于第二个数时返回真

==	等于。两个数相等时返回真
!=	不等于。两个数不相等时返回真

关系运算符的返回值是真（true）和假（false）的概念。在 C 语言中，真值 true 可以是不为 0 的任何值。而假值 false 则为 0。使用关系运算符，若表达式为真则返回 1，否则表达式为假则返回值为 0。例如下面的关系运算符的使用。

```
6>5                    返回 1
10=9                   返回 0
```

2.4.3 逻辑运算符

所谓逻辑运算符，指的是用形式逻辑原则来建立数值间关系运算的符号。逻辑运算有与、或、非三种，使用方法如表 2.5 所示。

表 2.5 逻辑运算符

运 算 符	含 义	返 回 值
&&	逻辑与	所有的条件为真则返回真
||	逻辑或	只要有一个条件为真则返回真
!	逻辑非	真值取非返回假，假值取反返回真

例如下面的代码，是关系运算符与逻辑运算符的使用实例。

```
#include <stdio.h>
int main()
{
   int i=5,j=3;
   printf("%d\n",i>j);                    /*判断 i 是否大于等于 j，返回真，输出 1。*/
   printf("%d\n",i==j);                   /*判断 i 是否等于 j，返回假，输出 0。*/
   printf("%d\n",i<j);
   printf("%d\n",i!=j);
   printf("%d\n",i!=j);
   printf("%d\n",(i>j)&&(i>=j));          /*判断两个条件，然后做逻辑与运算。*/
   printf("%d\n",(i>j)&&(i<j));
   printf("%d\n",1&&1&&1);                /*所有条件为真则返回真。*/
   printf("%d\n",1&&0&&1);                /*有一个条件为假则返回假。*/
   printf("%d\n",0||0||1);                /*有一个条件为真则返回真。*/
   printf("%d\n",!0);                     /*非假返回真。*/
}
```

输入下面的命令编译这个程序。

```
gcc 2.8.c
```

输入下面的命令，对编译后的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
1
0
0
1
1
1
0
1
0
1
1
```

2.5　小结

本章讲解了数据类型、变量赋值与输出、常用运算符等知识。这些知识是编写 C 程序的基础，通过这些知识的学习，可以理解 C 程序的一些概念，编写简单的 C 语言程序。在本章的学习中，数据类型与运算符的使用是重点，需要详细地理解数据类型的概念和输出方式。

第 3 章　C 程序的常用语句

3.1　流程控制语句

在程序中，需要对语句的执行进行分支选择，或者重复执行某些语句，这些实现程序逻辑功能或多次循环执行运算的语句就是流程控制语句。流程控制语句有条件语句与循环语句两种。条件语句实现程序的逻辑功能，循环语句实现程序的重复执行功能。

3.1.1　if 条件语句

if 条件语句的作用，是对一个条件进行判断。如果判断的结果为真，则执行条件后面的语句。如果执行判断的结果为假，则跳过后面的语句。最基本的 if 条件语句结构如下所示。

```
if(条件)
{
    条件成立时需要执行的内容;
}
```

这种条件语句的执行流程如图 3-1 所示。

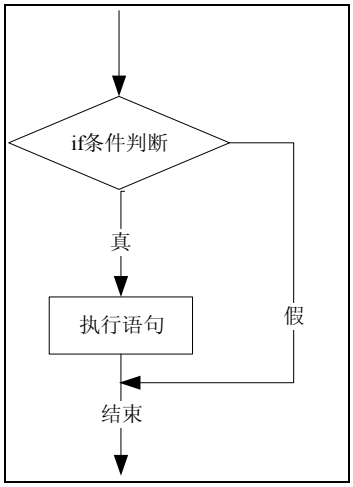

图 3-1　if 条件语句的流程图

如果 if 后面的判断结果只有两种情况，即第一种条件不成立时一定是第二种情况。则条件语句可以使用 if else 结构，这种条件语句的使用方法如下所示。

```
if(条件)
{
```

```
    条件成立时需要执行的内容;
}
else
{
    条件不成立时执行的内容;
}
```

这种条件语句中，先判断 if 的条件内容。如果返回的结果为真，则执行 if 后面的语句，否则就执行 else 后面的语句。程序的执行流程如图 3-2 所示。

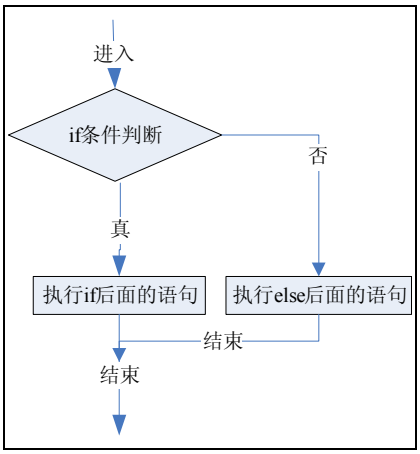

图 3-2　if else 条件语句的流程图

例如下面的实例是实现从键盘输入一个整数，用 if 语句判断这个整数是奇数还是偶数，然后输出判断结果。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。在终端中输入“vim”命令，然后按“Enter”键打开 VIM。

② 在 VIM 中按“i”键进入到插入模式，然后输入下面的代码。

```
#include <stdio.h>
int main()
{
   int i ,j ;                                    /*定义两个整型变量。*/
   printf("please input a number:\n");           /*提示输入。*/
   scanf("%d",&i);                               /*从键盘读取一个整数。*/
   j =i %2;                                      /*输入的数对 2 求余数。*/
   if(j==0)                                      /*如果余数为 0 则输入是偶数。*/
   {
      printf("%d oushu.\n",i);
   }
   else                                          /*否则输出为奇数。*/
   {
      printf("%d jishu.\n",i);
   }
}
```

③ 在 VIM 按“Esc”键返回到普通模式。然后输入“:w 7.1.c”命令，将这个文件保存

到主目录下的文件 7.1.c。然后输入“:q”命令退出 VIM。

④ 编译程序，在终端中输入下面的命令，然后按“Enter”键。

```
gcc 7.1.c
```

⑤ 对编译生成的程序添加可执行权限，在终端中输入下面的命令，然后按“Enter”键。

```
chmod +x a.out
```

⑥ 在终端中输入下面的命令，运行这个程序。

```
./a.out
```

⑦ 程序在运行时，显示下面的信息，提示输入一个整数。

```
please input a number:
```

⑧ 在键盘上输入一个整数 12，程序的显示结果如下所示。

```
12 oushu
```

3.1.2 if 语句的嵌套

所谓 if 语句嵌套，指的是在一个 if 语句的执行内容中再进行 if 判断语句。当判断情况很多时，需要使用 if 语句嵌套。下面就是一种简单的 if 语句嵌套。

```
if(条件)
{
   if(条件)
   {
      执行的内容;
}
   else
   {
      执行的内容;
   }
}
else
{
   if(条件)
   {
      执行的内容;
}
   else
   {
      执行的内容;
   }
}
}
```

在这个嵌套结构中，先进行一次条件判断，然后判断的结果进行另外一次条件判断。这种结构的执行流程可用图 3-3 来表示。

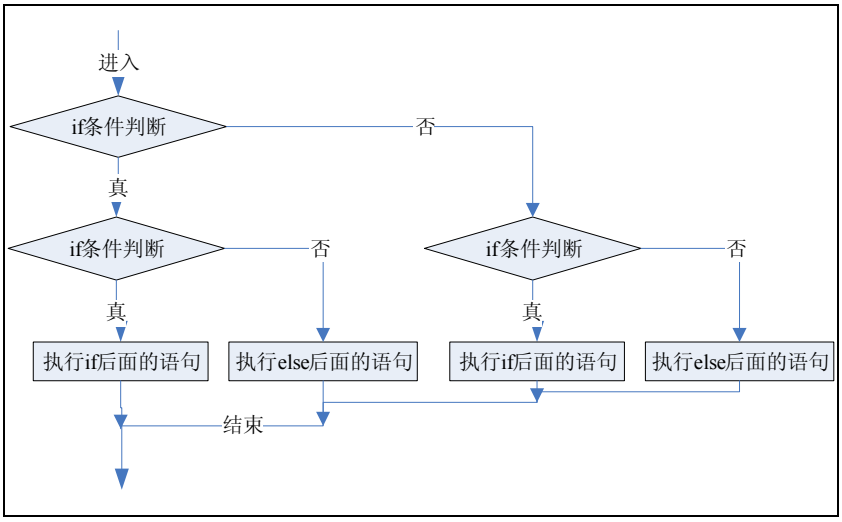

图 3-3　if 嵌套语句的流程

假设对考试分数进行 A、B、C、D 评级，85 分以上为 A，70~85 分为 B，60~70 分为 C，60 分以下为 D。要实现这个功能，需要对分数进行多次 if 判断，可以使用如图 3-3 所示的 if 嵌套结构。程序的代码如下所示。

```
#include <stdio.h>
int main()
{
   int i;                                  /*定义一个变量。*/
   printf("please input a number:\n");     /*提示输入信息。*/
   scanf("%d",&i);                         /*从读取一个变量。*/
   if(i>=70)                               /*第一次 if 判断。*/
   {
      if(i>=85)                            /*85 分以上的输出 A。*/
      {
         printf("A");
      }
      else                                 /*其他的分数就是 70~85 分的，输出 B。*/
      {
         printf("B");
      }
   }
   else                                    /*这里其他的就是 70 分以下的。*/
   {
       if(i>=60)                           /*60~70 分的输出 C。*/
      {
         printf("C");
      }
      else
      {
         printf("D");                      /*其他的一定是 60 分以下的，输出 D。*/
      }
   }
```

```
}
```

编译并运行这个程序，在提示后面输入一个数字 77，然后按“Enter”键。程序的 if 嵌套结构会判断分数的级别，显示的结果是 B。

3.1.3 switch 选择执行语句

当 if 语句的判断情况很多时，可以使用 switch 选择执行语句。在这种结构中，switch 对一个条件进行判断，然后分别列出可能的结果，每个结果执行不同的语句。switch 语句的使用方法如下所示。

```
switch(条件)
{
   case 结果 1 :
        执行内容 1; break ;
   case 结果 2 :
        执行内容 2; break ;
……
   case 结果 n :
        执行内容 n; break ;

}
```

这种结构的执行流程如图 3-4 所示。

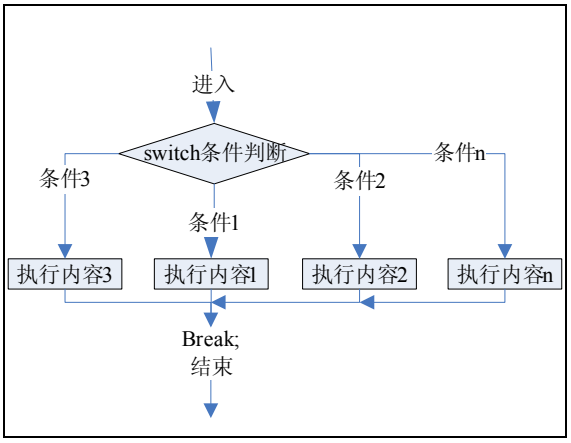

图 3-4　switch 语句的流程图

下面是一个 switch 语句的使用实例。用 scanf()函数从键盘读取一个数字，如果数字在 0~6 之间，输出对应的星期一到星期六，否则提示错误。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。在终端中输入“vim”命令，然后按“Enter”键打开 VIM。

② 在 VIM 中按“i”键进入到插入模式，然后输入下面的代码。

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

```
    int i;                                          /*定义一个变量。*/
    printf("please input a number:\n");             /*提示输入信息。*/
    scanf("%d",&i);                                 /*从键盘读取一个数字。*/
    if(i<0 || i >6)                                 /*如果这个数字小于0或大于6则输出错误
提示信息。*/
    {
    printf("input error.\n");
    }
    else                                            /*否则执行判断。*/
    {
       switch(i)                                    /*执行判断。*/
       {
          case 0:
            printf("Sunday\n") ; break;             /*7 个判断条件和结果。*/
          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;

       }
    }
}
```

③ 在 VIM 按“Esc”键返回到普通模式。然后输入“:w 7.2.c”命令，将这个文件保存到主目录下的文件 7.1.c。然后输入“:q”命令退出 VIM。

④ 输入下面的命令，编译这个程序。

```
gcc 7.2.c
```

⑤ 输入下面的命令，对编译后的程序添加可执行权限。

```
chmod +x a.out
```

⑥ 输入下面的命令，运行这个程序。

```
./a.out
```

⑦ 程序运行时，会输出下面的提示信息要求输入一个数字。

```
please input a number:
```

⑧ 这时输入一个数字 5，然后按“Enter”键。程序显示的结果如下所示。

```
Friday
```

3.1.4 for 循环语句

循环语句指的是使用循环结构使一个程序过程多次执行。for 循环语句是最常用的循环语句，这种结构的用法如下所示。

```
for(起始条件;循环条件;循环变量变化)
{
   循环体执行的内容;
}
```

例如下面的程序，实现从 1～6 之间 6 个数字的输出。

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.     int i;                                  /*定义循环变量。*/
5.     for(i=1;i<=6;i++)                       /*开始 for 循环*/
6.     {
7.       printf("%d\n",i);                     /*循环体，输出 i 的值和换行。*/
8.     }
9.  }
```

输入下面的命令，编译这个程序。

```
gcc 7.4.c
```

输入下面的命令，对编译后的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
1
2
3
4
5
6
```

这个程序的执行流程可用图 3-5 来表示。

根据图 3-5 可知，for 循环是用反复判断循环条件、执行循环体、改变循环变量的方法来实现程序的循环流程的。上述代码的执行过程如下所示。

① 第 4 行定义一个循环变量 i。

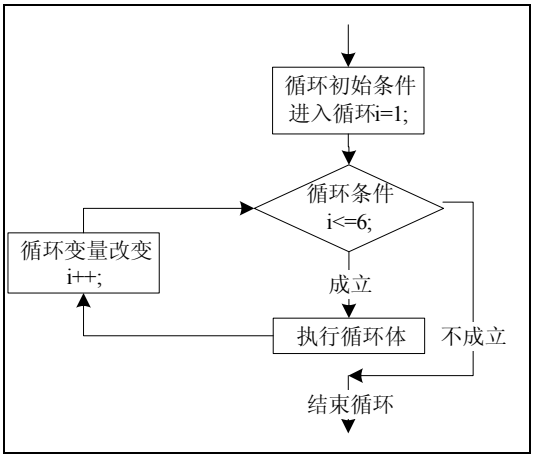

图 3-5　for 循环的程序流程

② 第 5 行，先执行循环初始条件，对 i 赋值为 1。

③ 进行第一次条件判断 i 是否小于等于 6。判断的结果为真则执行下面的循环体，为假时则跳出循环。

④ 循环体执行完毕以后，进入循环条件改变运算 i++。然后再次进行条件判断，从而完成程序的循环。

⑤ 当 i 等于 7 时，条件判断不成立，这时跳出循环，结束程序。

需要注意的是，循环变量的改变和条件判断要确保 for 循环可以中断，否则就会构成死循环。死循环指的是循环语句无法达到停止条件，永远执行下去，可能引起计算机的死机。下面是一个 for 循环实例，求出 100 以内的整数和。

```
#include <stdio.h>
int main()
{
  int i ;                                    /*定义循环变量。*/
  int sum = 0;                               /*定义求和的结果。*/
  for(i=0;i<=100;i++)                        /*循环语句。*/
  {
     sum =sum +i ;                           /*循环体。*/
  }
  printf("sum = %d",sum);                    /*输出结果。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 7.5.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序会从 1 到 100 进行循环，每次循环时，与 sum 相加的结果赋值给 sum，这样就得到了 100 以内整数的和。结果如下所示。

```
sum = 5050
```

需要注意的是，定义求和结果 sum 时，需要对变量 sum 赋初值 0，这样可以与后面的数相加而结果不变。进行求积运算时，需要对变量赋初值 1。

3.1.5 for 循环的嵌套

在 for 循环结构的循环体中，再次执行一个循环语句，这种结构就是 for 循环嵌套。在循环嵌套结构中，每执行一次外层循环，就会多次执行内层循环。for 循环嵌套的结构如下所示。

```
for(起始条件;循环条件;循环变量变化)
{
   for(起始条件;循环条件;循环变量变化)
   {
      循环体;
   }
}
```

这种结构的执行流程如图 3-6 所示。

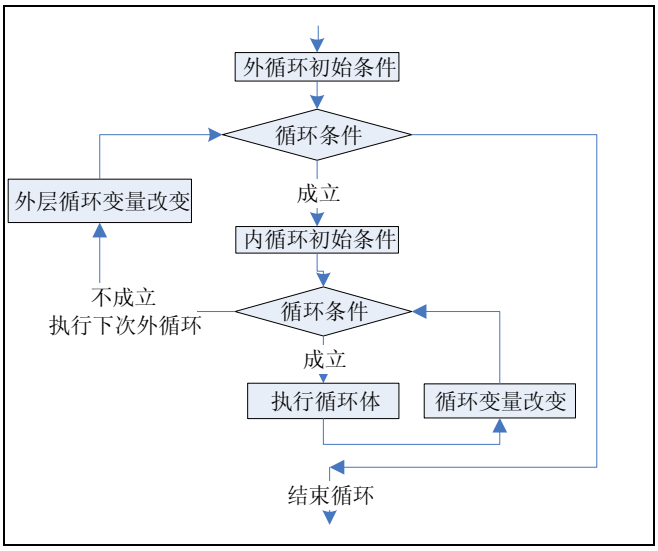

图 3-6 for 循环的程序流程

例如下面是一个循环嵌套结构实例，输出一个星号图案。在外层循环中，实现个数和行数的计数，在内层循环中实现星号的输出。

```
#include <stdio.h>
int main()
{
   int i ,j;                                    /*定义程序中的变量。*/
   for(i=1;i<7;i++)                             /*外层循环。*/
   {
```

```
        for(j=1;j<=i;j++)                          /*内层循环。*/
        {
         printf("* ");                             /*输出星号和一个空格。*/
        }
         printf("\n");                             /*输出一个换行。*/
    }
}
```

输出下面的命令，编译这个程序。

```
gcc 7.6.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
```

3.1.6　for 循环应用实例：输出九九乘法口诀表

乘法口诀表需要用 for 循环嵌套来实现，一个循环变量作被乘数，另一个循环变量作乘数。在输出结果时，需要用一个 if 判断来实现排列。每一行的循环需要输出一个换行。程序的代码如下所示。

```
#include <stdio.h>
int main()
{
    int i,j;                                       /*定义两个循环变量。*/
    for(i=1;i<=9;i++)                              /*外层循环。*/
    {
        for(j=1;j<=i;j++)                          /*内层循环。*/
        {
          if (j<=i)                                /*判断 i 与 j 的大小，实现排列。*/
            {
               printf("%d*%d=%d  ",j ,i , i*j );  /*输出乘法式。*/
            }
        }
        printf("\n");                                  /*输出一个换行。*/
     }
}
```

输出下面的命令，编译这个程序。

```
gcc 7.7.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
1*1=1
1*2=2  2*2=4
1*3=3  2*3=6  3*3=9
1*4=4  2*4=8  3*4=12  4*4=16
1*5=5  2*5=10  3*5=15  4*5=20  5*5=25
1*6=6  2*6=12  3*6=18  4*6=24  5*6=30  6*6=36
1*7=7  2*7=14  3*7=21  4*7=28  5*7=35  6*7=42  7*7=49
1*8=8  2*8=16  3*8=24  4*8=32  5*8=40  6*8=48  7*8=56  8*8=64
1*9=9  2*9=18  3*9=27  4*9=36  5*9=45  6*9=54  7*9=63  8*9=72  9*9=81
```

在这个程序中，需要注意乘法式的输出，下面是代码中的输出语句。

```
printf("%d*%d=%d  ",j ,i , i*j );
```

在输出格式字符串中，三个%d 分别对应输出列表中的 j、i、i*j。而字符*、=与空格按照原来的格式输出，这就实现了乘法式的输出。

3.1.7 while 循环语句

while 语句是另一种循环语句，可以实现与 for 循环一样的功能。while 循环的使用方法如下所示。

```
while(循环条件)
{
   循环体语句;
   循环条件改变;
}
```

在 while 循环结构中，先判断循环的条件是否成立。如果条件成立则执行循环体中的语句，然后继续判断条件是否成立。如果条件不成立则跳出循环，执行后面的语句。while 循环的执行流程可用图 3-7 来表示。

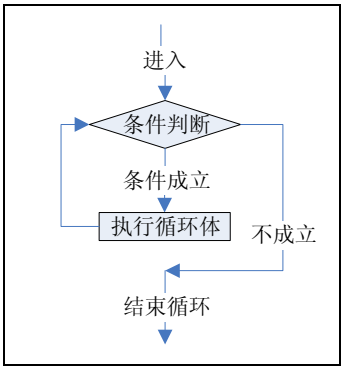

图 3-7　while 循环的执行流程图

因为 while 循环直接进入循环，而没有对循环变量赋值，所以需要在执行循环以前对循环变量赋值。在循环体中也需要更改循环变量的值。下面的实例是用 while 循环求出 1 到 100 之间的整数和。

```
#include <stdio.h>
void main()
{
  int i,sum;                          /*定义程序的变量。*/
  i=0;                                /*变量赋初值。*/
  sum=0;                              /*求和变量赋初值。*/
  while(i<=100)                       /*进入 while 循环。*/
  {
     sum=sum+i;                       /*sum 与 i 求和赋值给 sum。*/
     i++;                             /*i 自加。*/
  }
  printf("sum = %d \n",sum);          /*输出结果。*/
}
```

输出下面的命令，编译这个程序。

```
gcc 7.8.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
sum = 5050
```

在这个程序中需要注意下面两个问题。

（1）需要对循环变量与求和变量赋初值。while 循环在进入循环时没有赋值语句，如果在循环以外没有对循环变量赋初值，则程序会发生异常。

（2）在循环体中一定要改变循环变量的值，否则 while 循环的条件就始终成立，无法跳出

循环。这种情况可能造成计算机死机。

3.1.8 do while 循环语句

do while 循环与 while 循环相似，实现程序对循环体的多次执行。与 while 循环不同的是，do while 循环先执行一次循环体，然后判断条件是否成立。如果条件成立则执行下一次循环，否则结束循环。这种循环结构的使用方法如下所示。

```
do
{
   循环体;
}
while(循环条件);
```

do while 循环的程序流程如图 3-8 所示。

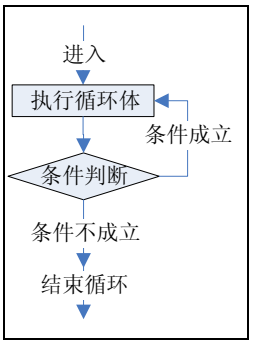

图 3-8　do while 循环的执行流程图

与 while 循环一样，do while 循环在进入循环以前，需要对循环变量赋初值。在循环体中需要更改循环变量的值。下面是一个使用 do while 循环的实例，可以完成 0 到 100 之间偶数的求和。

```
#include <stdio.h>
void main()
{
  int i,sum;                          /*定义两个程序变量。*/
  i=0;                                /*循环变量赋初值。*/
  sum=0;                              /*循环结果赋初值。*/
  do                                  /*开始 do 循环。*/
  {
     sum=sum+i;                       /*求和。*/
     i=i+2;                           /*变量加 2，使 i 是 100 以内的偶数。*/
  }while(i<=100);                     /*判断条件。*/
  printf("sum = %d \n",sum);
}
```

输出下面的命令，编译这个程序。

```
gcc 7.9.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
sum = 2550
```

3.1.9　转移控制语句：continue

有时需要在判断或循环的程序执行过程中实现程序流程的强制转移，这就需要使用转移控制语句。C 程序中有 continue, break 和 return。

在循环语句中，continue 语句可以中断循环体后面语句的执行，直接进入下一次循环。下面是一个在 for 循环中使用 continue 语句的实例，实现 100 以内偶数的求和。在程序中，用对变量 i 除 2 求余判断结果的方法，来判断是不是一个偶数。如果不是偶数，则用 continue 语句进行下一次循环。

```
#include <stdio.h>
void main()
{
  int i,sum;                        /*定义程序的变量。*/
  sum=0;                            /*对求和变量赋初值为 0。*/
  for(i=0;i<=100;i++)               /*for 语句实现 1 到 100 的循环。*/
  {
     if(i%2==1)                     /*判断是否为奇数。*/
     {
        continue;                   /*如是奇数则跳过下面的语句，进入下一次循环。*/
     }
     sum = sum +i;                  /*sum 与 i 求和的结果赋值给 sum。*/
  }
  printf("sum = %d \n",sum);        /*输出结果。*/
}
```

输出下面的命令，编译这个程序。

```
gcc 7.10.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序进入 for 循环以后，从 1 到 100 执行 100 次循环。在每次循环中，对变量 i 除 2 求余。如果结果为 1，则表明 i 为奇数，则用 continue 语句跳过后面循环体中语句的执行，进入下一次循环，否则进行求和运算。程序的运行结果如下所示。

```
sum = 2550
```

3.1.10 转移控制语句：break

break 语句可以中断循环语句的执行。例如在 for 循环、while 循环的语句中，可以不设置循环条件，让循环自动执行。在循环体中对循环变量进行判断，当符合循环中止条件时，使用 break 语句使循环中止。例如下面用 for 循环与 break 语句实现 100 以内偶数的求和。

```
#include <stdio.h>
void main()
{
  int i,sum;                          /*定义程序的变量。*/
  sum=0;                              /*求和变量赋初值为 0。*/
  for(i=0;  ;i++)                     /*进入 for 循环，不设置结束条件。*/
  {
     if(i>100)                        /*如果 i 的值大于 100，则用 break 语句中断循环。*/
     {
        break;                        /*break 语句跳出循环。*/
     }
     sum = sum +i;                    /*求和。*/
  }
  printf("sum = %d \n",sum);          /*输出结果。*/
}
```

输出下面的命令，编译这个程序。

```
gcc 7.11.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
sum = 2550
```

3.1.11 转移控制语句：return

程序中的 return 语句可以中止 main 主函数或自定义函数的执行。return 语句在 main 主函数与自定义函数中的使用是不同的。

- 在 main 主函数中使用 return 语句时，会中断当前程序的执行，程序中后面的所有部分都不再执行。
- 在自定义函数中使用 return 语句时，会中断函数的执行，返回到主函数的执行。如果 return 后面有参数，则参数就是函数的返回值。

例如下面的程序使用了 return 语句来中断程序的执行。

```
#include <stdio.h>
void main()
{
   printf("good morning.\n");
```

```
    return ;
    printf("good evening.\n");
}
```

编译并运行这个程序，结果如下所示。

```
good morning.
```

return 语句中断了程序的执行，return 以后的语句都是不执行的。如果程序中出现异常或错误，可以用 return 语句来中止程序。

3.2　流程控制语句实例

流程控制语句可以在程序中实现复杂的功能。判断语句可以实现程序的逻辑功能，循环语句可以完成程序的多次运算。本节将讲解几个流程控制语句的使用实例。

3.2.1　三个数字的排序

用 if 语句可以比较两个变量的大小，可以用一个中间变量来实现两个数字的排列。用同样的方法，对三个数字每两个进行一次从小到大的排列，就可以实现三个数字的从小到大排列。这个程序的算法如下所示。

（1）定义 4 个变量 x、y、z、t。

（2）输出提示，然后用 scanf()函数分别从键盘读取三个数字，分别赋值给 x、y、z。

（3）取出 x 与 y 进行比较，如果 x 大于 y，则 x 赋值给 t，y 赋值给 x，t 再赋值给 y。三次赋值操作实现了 x 与 y 的交换。这一步骤的操作实现了 x 与 y 的从小大到排列。

（4）用上一步骤同样的办法，实现 x 与 z 的从小到大排列，再实现 y 与 z 的从小到大排列。从而实现了三个数字的从小到大排列。

（5）输出排列的结果。

下面是完成这个程序的步骤。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。在终端中输入“vim”命令，然后按“Enter”键打开 VIM。

② 在 VIM 中按“i”键进入到插入模式，然后输入下面的代码。

```
#include <stdio.h>
void main()
{
   int x, y, z, t;                          /*定义 4 个变量。*/
   printf("please input x:\n");             /*分别提示和输入三个变量。*/
   scanf("%d", &x);
   printf("please input y:\n");
   scanf("%d", &y);
   printf("please input x:\n");
   scanf("%d", &z);
   if(x > y)                                /*判断是否 x 大于 y。*/
   {
      t = x;                                /*三次赋值操作实现 x 与 y 的交换。*/
      x = y;
```

```
        y = t;
    }
    if(x > z)                                   /*用同样的方法实现 x 与 z 的排列。*/
    {
        t = x;
        x = z;
        z = t;
    }
    if(y > z)                                   /*实现 y 与 z 的排列。*/
    {
        t = y;
        y = z;
        z = t;
    }
    printf("%d  %d  %d",x,y,z);                 /*输出结果。*/
}
```

③ 输出下面的命令，编译这个程序。

```
gcc 7.13.c
```

④ 输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

⑤ 输入下面的命令，运行这个程序。

```
./a.out
```

⑥ 程序提示输入一个数字，显示结果如下所示。

```
please input x:
```

⑦ 输入一个数字 5，然后按“Enter”键。在后面的提示中再输入两个数字 8 和 2。按“Enter”键以后，显示的结果如下所示。

```
2  5  8
```

3.2.2　解一元二次方程

解一元二次方程的关键算法是根据根的判别式的情况判断方程是否有根，然后根据方程的求根公式求出方程的根。在解方程时要注意下面这些问题。

（1）一元二次方程的标准形式如下所示。

```
aX²+bX+c=0
```

（2）在方程中，需要输入 a、b、c 三个参数。参数是浮点型数字。

（3）在方程的三个参数中，a 是不能为 0 的，b 和 c 是可以为 0 的。

（4）方程根的判别式如下所示。

$$s=b*b-4*a*c$$

（5）当根的判别式 s 是否为 0 时有下面三种情况。

- $s<0$ 时，方程无解。
- $s=0$ 时，方程有一个解。
- $s>0$ 时，方程有两个不同的解。

（6）当判别式 s 大于等于 0 时，方程的求根公式如下所示。

$$X=\frac{-b\pm\sqrt{b*b-4*a*c}}{2*a}$$

解方程的程序流程图如图 3-9 所示。

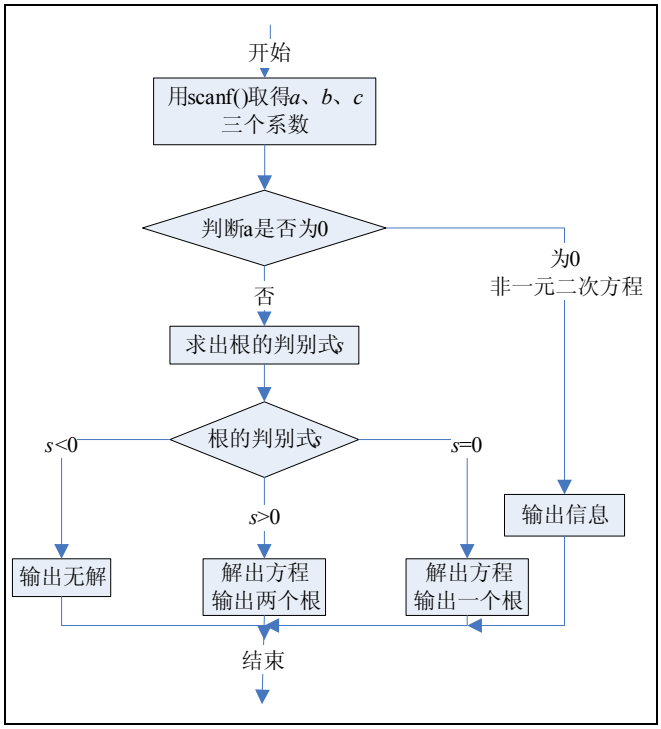

图 3-9　解一元二次方程的程序流程图

在 C 程序中，开方运算需要用 sqrt 函数，这个函数在 math.h 头文件中，需要在程序开始处包含这个文件。下面的步骤是编写这个解一元二次方程的程序。

① 打开 VIM，在 VIM 插入模式中编写下面的代码。

```
#include <stdio.h>
#include <math.h>
void main()
{
   float a ,b ,c ,s;                          /*定义三个变量。*/
   float x1,x2;                               /*结果变量。*/
   printf("please input a:\n");               /*输入变量 a。*/
   scanf("%f", &a);                           /*注意，需要用%f 表示浮点型输入。*/
   if(a==0)                                   /*如果 a 为 0 则中止执行。*/
   {
      printf("error.\n");
      return;                                 /*return 中止程序。*/
   }
   printf("please input b:\n");               /*输入变量 b 和 c。*/
   scanf("%f", &b);
   printf("please input c:\n");
   scanf("%f", &c);
   s=b*b - 4*a*c;                             /*求出根的判别式。*/
   if(s<0)                                    /*判别式小于 0 时无解。*/
   {
       printf("no result.\n");
   }
   else
   {
      if(s==0)                                /*判别式为 0 时有一个解。*/
      {
         printf("one result.\n");
         x1=(-b+sqrt(s))/(2*a);               /*输出这个解。*/
         printf("reslut is x = %d\n",x1);
      }
      else
      {
         printf("two result.\n");             /*判别式大于 0 时有两个解。*/
         x1=(-b+sqrt(s))/(2*a);
         x2=(-b-sqrt(s))/(2*a);
         printf("x1 = %f\n",x1);
         printf("x2 = %f\n",x2);
      }
   }
}
```

② 输入下面的命令，编译这个程序。

```
gcc 7.14.c
```

③ 输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

④ 输入下面的命令，运行这个程序。

```
./a.out
```

⑤ 程序运行结果如下所示，提示输入参数 a。

```
please input a:
```

⑥ 这时从键盘输入 3，然后按“Enter”键。用同样的方法，输入 b 和 c 的参数分别为 8 和 4。程序显示方程有两个解，然后列出两个解，结果如下所示。

```
two result.
x1 = -0.666667
x2 = -2.000000
```

3.3　两种特殊语句结构

在 C 程序中，有一些特殊语句结构的用法。本节将讲解三元操作符?和块语句的用法。?结构相当于一个 if else 条件语句，块语句是作为一个单元进行处理的语句组。

> **注意**：三元运算符指的是一个运算符号有三个参数。

3.3.1　?三元操作符

问号（?）操作符是 C 程序中的一个强有力的操作符，实现逻辑判断的功能，可以代替程序中的 if else 语句。这种结构的用法如下所示。

```
条件表达式 1?结果表达式 1:结果表达式 2
```

?操作符在执行时实现了一个逻辑功能。先求条件表达式的值。如果条件表达式的结果为真，则求出结果表达式 1 的值并返回。如果条件表达式的值为假，则求出结果表达式 2 的值并返回。两个结果表达式之间用冒号（:）隔开。

例如下面是一个简单的?操作符使用。

```
int x ,y;
x=15;
y=x>10?20:30;
```

?操作符的含义是，如果 x 大于 10，则 y 被赋值为 20，否则 y 被赋值为 30。在这个语句中，条件表达式的值为真，所以 y 被赋值为 20。

这个语句结构也可以改写成如下所示的 if 语句结构。

```
int x ,y;
x=15;
if(x>10)
   y=20;
else
   y=30;
```

下面是一个?操作符使用实例，求出一个数的绝对值。在程序中，需要对输入的值进行判断，如果大于等于 0，则输出原来的变量。如果小于 0，则输入这个数的相反数。程序的代码

如下所示。

```
#include <stdio.h>
void main()
{
   float x,y;                                          /*定义变量。*/
   printf("please input a number:\n");                 /*输出提示信息。*/
   scanf("%f", &x);                                    /*从键盘读取一个数。*/
   y=x>=0?x:(-x);                                      /*用?操作符判断和赋值。*/
   printf("number: %f \nvalue: %f\n",x,y);             /*输出原来变量和绝对值。*/
}
```

3.3.2 块语句

块语句是使用两个花括号（{}）括起来的一个语句集，形成一个语句体。循环语句、判断语句要执行的内容，都是一个块语句。在程序中，块语句是作为一个整体来执行的，要么全部执行，要么都不执行。可以将块语句看成一个语句。块语句的格式如下所示。

```
{
   语句列表;
}
```

例如下面的程序就使用块语句作为循环语句的循环体。

```
#include <stdio.h>
void main()
{
  int i,sum;                                  /*定义程序的变量。*/
  i=0;                                        /*变量赋初值。*/
  sum=0;                                      /*求和变量赋初值。*/
  while(i<=100)                               /*进入 while 循环。*/
  {
     sum=sum+i;                               /*sum 与 i 求和赋值给 sum。*/
     i++;                                     /*i 自加。*/
  }
  printf("sum = %d \n",sum);                  /*输出结果。*/
}
```

在这个程序中，while 语句的循环体是两个语句，这两个语句用花括号构成块语句。

3.4 小结

本章讲述了 C 程序中的流程控制语句。流程控制语句实现了程序的逻辑功能，可以完成各种复杂的程序运算，是编程开发的基础和重点。其中，循环语句和条件语句是本章的重点，需要理解这些语句的结构和执行流程。

第 4 章　数组与指针

4.1　数组的理解与操作

数组实际上就是一组相同数据类型变量。如果一个数组中的一个变量也是一个数组，就构成了二维数组。用同样的方法可以构成多维数组。在使用数组以前需要定义一个数组。

4.1.1　什么是数组

可以用一个实例来理解数组。假设在一个程序中要存储 100 个人的年龄，可以写成下面代码来定义 100 个整型变量。

```
int age1=12;
int age2=18;
int age3=15;
……
int age100=21;
```

如果这样编写程序，程序中产生大量类似的语句，在调用变量时，变量名没有统一的规则。可以将所有同类的数据存放在一个数组变量中。所有人的年龄可以建立一个数组，然后用编号来表示数组的一个变量。可以用下面的代码来表示这些数据。

```
int a[100]
age[0]=12;
age[1]=18;
age[2]=15;
……
age[99]=21;
```

这样就可以用一个变量名和一个下标来表示数组中的一个变量。例如 age[0]表示数组中的第一个变量，age[1]表示数组中的第二个变量。age 是这个数组的变量名，后面的数字是这个数组的下标，下标需要使用中括号括起来。例如下面的程序就是定义一个整型变量的数组，然后用循环语句对数组赋值 10~19，再用循环语句输出这些数据变量。

```
#include <stdio.h>
void main()
{
   int a[10];
   int i;
   for(i=0;i<=9;i++)
   {
      a[i]=i+10;
```

```
    }
    for(i=0;i<=9;i++)
    {
      printf("%d  ",a[i]);
    }
}
```

编译并运行这个程序，结果如下所示。

```
10  11  12  13  14  15  16  17  18  19
```

在使用数组时，需要注意下面这些下标和数据类型的问题。

（1）在定义数组时，已经定义了数组的数据类型，在访问时需要按照数组的数据类型进行访问和赋值。

（2）数组的下标是从 0 开始的向后排列整数，不能是其他的下标。第一个下标是 0 而不是 1。而下标 n 表示第 n+1 个变量。

（3）定义数组 a[n]以后，有 n 个元素，但是没有 a[n]这个元素。访问不存在的数组变量时，程序就会发生溢出错误。

> **注意**：溢出指的是程序中的变量不够存储空间，或者访问了不存在的变量，从而使程序发生错误。

在本质上，数组是内存上一组同类信息列出的一个表。假设上面例子中的数组 a[10]是从内存编号为 1000 的存储单元开始存储的，数组的内容和值如表 4.1 所示。

表 4.1　数组的存储与值

变量	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
值	10	11	12	13	14	15	16	17	18	19
存储单元	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009

4.1.2　数组的定义与访问

数组的定义指的是在内存中开辟一块存储空间，生成一个空数组。数组的定义与变量的定义相似，需要指定数据类型及变量的多少，变量数目用中括号括起来。例如下面的代码就是定义不同的数组。

```
int a[10];              /*定义 10 个变量的整型数组。*/
float f[20];            /*定义有 20 个变量的浮点型数组。*/
char s[5];              /*定义有 5 个变量的字符型数组。*/
```

数组在定义时，可以不指定变量的个数，在访问变量时可以动态改变数组中变量的个数。例如下面的代码是定义一个有不定变量的字符型数组。

```
char a[];               /*定义一个含有不定个数变量的字符数组。*/
```

数组的访问很简单。用数组变量的名称与下标就可以访问这个数组。例如下面的代码是输出数组元素的值和对数组元素进行赋值。

```
printf("%d",a[2]);        /*输出数组 a 中第 3 个元素的值。*/
a[3]=5;                   /*对数组中的第 4 个变量进行赋值。*/
```

4.1.3 数组使用实例

数组可以实现变量的循环访问功能，和循环语句一起可以方便地实现数组的访问与输出。下面的实例，定义一个字符型数组，然后从键盘上输入字符存储到这个数组中。当输入一个“#”停止输入，然后用循环语句输出这些字符。程序代码如下所示。

```
#include <stdio.h>
void main()
{
   char a[100];                          /*定义一个字符型数组。*/
   char c='a';                           /*定义一个字符变量，赋一个初值。*/
   int i=0 ;                             /*定义一个计数变量，实现下标的计数。*/
   while(1)                              /*进入一个循环。*/
   {
     scanf("%c",&c);                     /*输入一个字符。*/
     if(c=='#')                          /*如果字符为#则中断循环。*/
     {
       break;                            /*结束循环。*/
     }
     a[i]=c;                             /*把输入的变量赋值给数组中的一个元素。*/
     i++ ;                               /*计数变量自加。*/
   }
   i=0;                                  /*输出时，计数变量赋值为 0。*/
   while(a[i]!=NULL)
   {
     printf("%c",a[i]);                  /*输出数组的中字符。*/
     i++ ;                               /*下标自加。*/
   }
}
```

输入下面的命令，编译这个程序。

```
gcc 8.2.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

在程序中输入一个字符，然后按“Enter”键。输入多个字符以后，输入“#”结束输入。程序会输出这些字符。

这个程序是用“#”号来结束循环输入的。每次输入一个字符时，判断这个字符是不是“#”，如果是“#”则用 break 语句来中断循环。在输出时，判断输出的字符是不是存在。在定义数组时，数组中的字符都是空字符 NULL。

4.2　指针

指针是一种特殊的数据类型，用来存储一个变量的地址。通过一个指针，可以访问这个指针所指向的变量。在使用指针时需要考虑到变量的存储关系。

4.2.1　指针的理解

程序中的变量都是以字节的形式存储在内存单元中的，这些内存单元都有一个编号。这个编号就是程序中的指针。如表 4.2 所示是一个程序中的内存地址和变量值的关系。

表 4.2　指针和变量

内存地址	内存中的变量
1005	1
1006	2
1007	3
1008	b
1009	c
1010	d

在这个内存区域中，每一个变量都有一个内存存储单元编号，这个编号就是变量的指针。通过这个指针可以访问这个变量的值。

4.2.2　指针操作符

指针操作符有*与&两个，分别实现取变量和取地址的操作。程序中就是通过这两个操作符实现指针的定义与访问的。

“&”可以实现取一个变量的地址的功能。取出的变量地址可能是一个很复杂的数据类型，但是操作时并不关心数值的多少，只需要保存到一个指针变量上面。例如下面的代码是取地址操作。

```
int *p;                          /*定义一个指针变量。*/
int i=5;                         /*定义一个整变量。*/
p=&i;                            /*取变量 i 的地址赋值给 p。*/
```

“*”实现取一个指针所指向的变量的功能。例如下面的代码就是通过一个变量的指针来访问变量。

```
int *p;                          /*定义一个指针变量。*/
int i=5 , j;                     /*定义一个整变量。*/
p=&i;                            /*取变量 i 的地址赋值给 p。*/
j=*p;                            /*取指针 p 的值赋值给 j。*/
```

4.2.3 指针的定义与访问

指针虽然是一种特殊类型，但定义指针变量时，需要考虑到这个指针所指向变量的数据类型。定义指针的方法是用定义变量的方法，再指针类型名称前面加一个“*”。例如下面的代码是指向不同数据类型的指针。

```
int *p1;                          /*定义一个指向整型变量的指针。*/
char *p2;                         /*定义指向一个字符型变量的指针。*/
float *p3;                        /*定义一个指向浮点型变量的指针。*/
```

一个变量是用“&”取地址的。一个指针变量可以用“*”来取这个变量指向的数值。例如下面的代码是使用指针访问变量的实例。

```
#include <stdio.h>
void main()
{
   int i ,j;                                  /*定义两个整型变量。*/
   int *p;                                    /*定义一个指向整型的指针变量。*/
   char a ,b;                                 /*定义两个字符变量。*/
   char *q;                                   /*定义一个指向字符变量的指针。*/
   i=5;                                       /*i 赋值为 5。*/
   p=&i;                                      /*取 i 的地址赋值给 p。*/
   j=*p;                                      /*取指针 p 指向的变量值赋值给 j。*/
   a='w';                                     /*a 赋值为 w。*/
   q = &a;                                    /*取 a 的地址赋值给指针变量 q。*/
   b=*q;                                      /*取指针 q 指向的变量值赋值给 b。*/
   printf("%d  %d  %d \n",i ,j ,*p);          /*输出结果。*/
   printf("%c  %c  %c \n",a ,b ,*q);          /*输出结果。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 8.3.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
5  5  5
w  w  w
```

从结果可知，通过一个指向变量的指针，可以访问这个变量的值。在程序中，常常通过指针来访问一个变量的值。

4.2.4　指针使用实例

在程序中可以使用指针来访问变量的值。本节将讲述一个指针使用实例，通过指针所指向的变量比较两个整数的大小，然后从小到大输出两个整数。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。在终端中输入“vim”命令，然后按“Enter”键，打开 VIM。

② 在 VIM 中按“i”键，进行到插入模式，输入下面的代码。

```
#include <stdio.h>
void main()
{
   int i ,j;                                /*定义两个整型变量。*/
   int *p,*q,*temp;                         /*定义指向两个整型变量的指针变量。*/
   printf("please input the first number: \n");     /*提示输入。*/
   scanf("%d",&i);                                  /*输入一个数值。*/
   printf("please input the second number: \n") ;
   scanf("%d",&j);
   p=&i;                                    /*取 i 的地址赋值给指针 p。*/
   q=&j;                                    /*取 j 的地址赋值给指针 q。*/
   if(*p>*q)                                /*判断这两个指针指向值的大小。*/
   {                                        /*用一个中间指针变量交换两个指针。*/
     temp=p;
     p=q;
     q=temp;
   }
   printf("%d  %d\n",*p,*q);                /*输出结果。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 8.4.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序运行时，输出提示，这时输入一个数字 5，然后按“Enter”键。然后再输入一个数字 3，然后按“Enter”键。这时程序显示的结果如下所示。

```
3  5
```

这个程序是通过指针所指向的值来实现两个整数的排列。通过比较指针所指向变量的大小，然后用一个中间指针变量交换两个指针变量的值。

4.3 数组与指针

数组与指针的联系非常紧密。除了可以用下标访问数组元素以外，也可以用数组的指针访问数组变量。本节将讲解数组与指针的关系。

4.3.1 数组与指针的关系

定义一个数组的时候，就是定义这个数组的头指针，然后分配若干个存储单元。定义的数组名称是可以直接赋值给一个指针的，而这个指针可以指向这个数组的第一个元素。下面的程序可以说明数组与指针的这种关系。

```
#include <stdio.h>
void main()
{
   int *p;                          /*定义一个指向整型的指针。*/
   int a[3];                        /*定义一个整型数组。*/
   a[0]=10;                         /*对数组的变量赋值。*/
   a[1]=11;
   a[2]=12;
   p=a;                             /*将数组赋值给一个指针。*/
   printf("%d\n",*p);               /*输出这个指针指向的内容。*/
   p++;                             /*指针向后移动一个单元。*/
   printf("%d\n",*p);               /*输出指针指向的内容。*/
   p++;                             /*指针向后移动一个单元。*/
   printf("%d\n",*p);               /*输出指针指向的内容。*/
}
```

运行这个程序，结果如下所示。

```
10
11
12
```

从程序的结果可知，数组的定义就是返回了这个数组的指针。用指针来访问数组的元素与下标访问变量是等效的。

4.3.2 指针的算术运算

指针可以做加法和减法的算术运算，相当于指针的位置从这一位置向后或向前移动若干个单元。这里移动的单元，指的是可以存储相应的变量的内存空间，而不一定是这么多个字节。下面的实例可以说明指针算术运算的作用。

```
#include <stdio.h>
void main()
{
   int i;                                /*定义一个循环变量 i。*/
   int *p;                               /*定义一个指向整型变量的指针。*/
   int a[10];                            /*定义一个整型的数组。*/
   for(i=0;i<10;i++)                     /*变量 i 从 0 到 9 执行循环。*/
   {
```

```
        a[i]= i +10;                    /*对数组进行赋值。*/
    }
    p=a;                                /*将数组 a 的头指针赋值给 p。*/
    for(i=0;i<10;i++)                   /*进行 10 次循环。*/
    {
       printf("%d  ",*p);               /*输出指针 p 指向的值。*/
       p++;                             /*指针 p 向后移动一个单元。*/
    }
     printf("\n",*p);                   /*输出一个换行。*/
     p=p-4;                             /*指针向前移动 4 个单元。*/
     printf("%d  ",*p);                 /*输出指针 p 指向的值。*/
     p=p-3;                             /*指针向前移动 3 个单元。*/
     printf("%d  ",*p);                 /*输出指针 p 指向的值。*/
     p=p+5;                             /*指针向后移动 5 个单元。*/
     printf("%d  ",*p);                 /*输出指针 p 指向的值。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 8.6.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
10  11  12  13  14  15  16  17  18  19
16  13  18
```

4.3.3　字符数组与字符串

字符数组是一种特殊的数组。定义一个字符数组以后，这个字符数组会返回一个头指针。可以根据这个头指针来访问数组中的每一个字符。而 scanf()输入函数和 printf()输出函数可以输入或输出一个字符串。下面的实例是字符数组的使用。

```
#include <stdio.h>
void main()
{
   char a[30];                                  /*定义一个字符数组。*/
   char *p;                                     /*定义一个指向字符变量的指针。*/
   int i;                                       /*定义一个计数器变量。*/
   printf("please input a string:\n") ;         /*提示输出。*/
   scanf("%s",a) ;                              /*输入一个字符数组。*/
   printf("result:\n") ;
   printf("%s\n",a) ;                           /*输出一个字符数组。*/
   i=0;                                         /*循环计数器赋初值。*/
   while(a[i]!=NULL)                            /*判断相对应的字符是不是为空。*/
   {
```

```
        printf("%c",a[i]) ;                    /*输出这个字符。*/
        i++;                                   /*计数器自加。*/
      }
      printf("\n") ;                           /*输出一个换行。*/
      p=a;                                     /*数组的头指针赋值给指针 p。*/
      printf("%s\n",p) ;                       /*输出变量 p 的内容。*/
   while(*p!=NULL)                             /*判断 p 指向的内容是不是为 NULL,进行 while
                                                循环。*/
      {
        printf("%c",*p) ;                      /*输出指针 p 指向的一个字符。*/
        p++;                                   /*指针 p 指向下一个变量。*/
      }
   }
```

输入下面的命令，编译这个程序。

```
gcc 8.7.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序输出下面的提示，要求输入一个字符串。

```
please input a string:
```

这时输入下面的字符串，然后按“Enter”键。

```
asdfgh
```

程序的运行结果如下所示。

```
please input a string:
asdfgh
result:
asdfgh
asdfgh
asdfgh
asdfgh
```

4.4 二维数组与多维数组

如果数组的每一个元素是一个一维数组，则这个数组就是一个二维数组。在二维数组中，变量可以理解成行和列的关系。用同样的方法可以构造出多维数组。

4.4.1 二维数组的理解

二维数组可以理解成一个多行多列的表格，每一个单元格中存储了一个变量。可以同一维数组一样定义二维数组。下面的代码定义了一个 4 行 5 行的整型数组。

```
int a[4][5];
```

如表 4.3 所示，表示这个数组的行和列存储关系。

表 4.3　二维数组

行和列	0	1	2	3	4
0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
3	a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]

根据这个表格，可以用数组的下标访问每一个元素。下面的代码，是使用 for 循环对这个二维数组进行赋值与输出的实例。

```
#include <stdio.h>
void main()
{
    int i=10 , m , n ;                                  /*定义程序中的变量。*/
    int a[4][5];                                        /*定义一个 4 行 5 列的二维数组。*/
    for(m=0;m<4;m++)                                    /*用变量 m 从 0 到 3 进行行循环。*/
    {
       for(n=0;n<5;n++)                                 /*变量 n 从 0 到 4 进行列循环。*/
       {
          a[m][n]=i;                                    /*数组中变量赋值。*/
          i++;                                          /*i 自加。*/
       }
    }
    for(m=0;m<4;m++)                                    /*变量 m 的行循环。*/
    {
       for(n=0;n<5;n++)                                 /*变量 n 的列循环。*/
       {
         printf("a[%d][%d]=%d  ",m,n,a[m][n]);          /*输出变量名变量值。*/
       }
       printf("\n");                                    /*每行输出一个换行。*/
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 8.8.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
a[0][0]=10  a[0][1]=11  a[0][2]=12  a[0][3]=13  a[0][4]=14
a[1][0]=15  a[1][1]=16  a[1][2]=17  a[1][3]=18  a[1][4]=19
```

```
a[2][0]=20  a[2][1]=21  a[2][2]=22  a[2][3]=23  a[2][4]=24
a[3][0]=25  a[3][1]=26  a[3][2]=27  a[3][3]=28  a[3][4]=29
```

4.4.2 二维数组与指针

同一维数组一样，二维数组在定义时也是返回了一个指向第一个元素的指针。而指针向后移动存储单元，可以访问数组中的元素。下面的代码定义了一个二维数组，然后将二维数组的头指针赋值给一个指针变量。

```
int a[4][5];
int *p;
p=a;
```

根据数组与指针的关系，可以用如表 4.4 所示的指针变量来访问数组中的所有变量。

表 4.4 二维数组与指针

行和列	0	1	2	3	4
0	p	p+1	p+2	p+3	p+4
1	p+5+0	p+5+1	p+5+2	p+5+3	p+5+4
2	p+5+5+0	p+5+5+1	p+5+5+2	p+5+5+3	p+5+5+4
3	p+15+0	p+15+1	p+15+2	p+15+3	p+15+4

从表 4.4 可知，数组中变量的指针地址与首指针存在着下面的关系。

```
q=p+(行数*总列数)+列数
```

根据这个关系可以用指针的方法来访问一个数组。下面的程序使用了这种方法。

```
#include <stdio.h>
void main()
{
    int i=10 , m , n ;                              /*定义三个变量。*/
    int a[4][5];                                    /*定义一个 4 行 5 列的二维数组。*/
    int *p;                                         /*定义一个指针。*/
    for(m=0;m<4;m++)                                /*行的循环。*/
    {
       for(n=0;n<5;n++)                             /*列的循环。*/
       {
          a[m][n]=i;                                /*对数组变量进行赋值。*/
          i++;                                      /*i 自加。*/
       }
    }
    p=a;                                                /*把数组的头指针赋值给 p。*/
    for(m=0;m<4;m++)                                    /*行循环。*/
    {
       for(n=0;n<5;n++)                                 /*列循环。*/
       {
         printf("a[%d][%d]=%d  ",m,n,*(p+(5*m)+n));     /*用指针来访问数组的变量。*/
       }
       printf("\n");                                    /*每行输出一个换行。*/
    }
```

```
}
```

输入下面的命令，编译这个程序。

```
gcc 8.9.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
a[0][0]=10  a[0][1]=11  a[0][2]=12  a[0][3]=13  a[0][4]=14
a[1][0]=15  a[1][1]=16  a[1][2]=17  a[1][3]=18  a[1][4]=19
a[2][0]=20  a[2][1]=21  a[2][2]=22  a[2][3]=23  a[2][4]=24
a[3][0]=25  a[3][1]=26  a[3][2]=27  a[3][3]=28  a[3][4]=29
```

从程序的结果可知，用指针来访问数组与用下标访问数组的效果是一样的。

4.5　实例

数组可以在程序中实现数据存储，程序可以使用循环的方法对数组进行访问。本节将讲解数组与指针的编程使用实例。

4.5.1　学生成绩统计实例

本节编写一个成绩输入与统计程序。在程序中输入 6 个人的姓名、各科成绩，然后程序输出每个人的各科成绩、总成绩和平均成绩。程序的代码如下所示。

```
#include <stdio.h>
void main()
{
    char name[6][10];                          /*定义 6 个人的姓名，6 个字符数组。*/
    int score[6][5];                           /*定义一个二维数组存放成绩。*/
    int i;                                     /*定义一个计数器变量。*/
    for(i=0;i<6;i++)                           /*进行 6 次循环。*/
    {
      printf("please input the name:\n");          /*提示输入姓名。*/
      scanf("%s",name[i]);                         /*输入一个人的姓名。*/
      printf("score ,math:\n");
      scanf("%d",&score[i][0]);                    /*输入数学成绩。*/
      printf("score ,English:\n");
      scanf("%d",&score[i][1]);                    /*输入外语成绩。*/
      printf("score ,Chinese:\n");
      scanf("%d",&score[i][2]);                    /*输入汉语成绩。*/
      score[i][3] =  score[i][0]+score[i][1]+score[i][2];  /*求出这个人的总
分。*/
      score[i][4]= (int)(score[i][3] /3);          /*求出这个人的平均分。*/
    }
     printf("result:\n math  English Chinese  total  average :\n");
```

```
        for(i=0;i<6;i++)                                    /*循环输出成绩。*/
        {
          printf("%s: %d   %d    %d    %d   %d  \n",name[i],score[i][0],score[i][1],
score[i][2],
    score[i][3],score[i][4]);                               /*输出一个人各科的成绩。*/
        }
    }
```

输入下面的命令，编译这个程序。

```
gcc 8.10.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序运行时，在文本提示后面输入每个人的姓名和各科分数。程序显示的结果如下所示。

```
math  English Chinese  total  average :
jim: 45   76    87    208   69
tom: 56   76    45    177   59
bill: 56   76    87    219   73
lucy: 45   65    87    197   65
bod: 56   76    46    178   59
lily: 34   65    67    166   55
```

4.5.2 冒泡法排序实例

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。这种排序的算法可用如图 4-1 所示的来表示。

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。

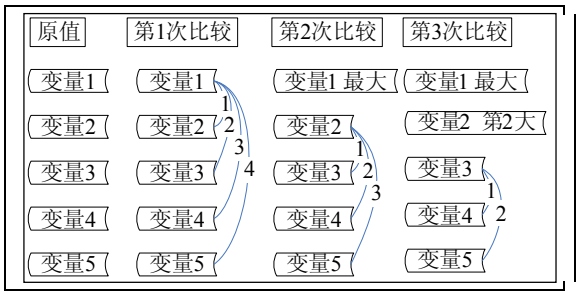

图 4-1　冒泡法排序的算法

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。

下面的程序是冒泡法排序的例子。在这个程序中需要注意两次循环比较。

```
#include <stdio.h>
main()
{
   int a[10];                          /*定义一个整型数组。*/
   int i,j,temp;                       /*定义循环变量和中间变量。*/
   for(i=0;i<10;i++)                   /*进行循环输入变量。*/
   {
      printf("please enter a number:\n");    /*输出提示。*/
      scanf("%d",&a[i]);               /*输入变量赋值给数组变量。*/
   }
   for(i=0;i<10;i++)                   /*进行 10 次循环。*/
   {
      for(j=i+1;j<10;j++)              /*循环比较剩余的变量。*/
      {
         if(a[i]<a[j])                 /*如果前面一个数比后面数小，交换两个数的值。*/
         {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
         }
      }
   }
   for(i=0;i<10;i++)                   /*循环输出排序以后的结果。*/
   {
      printf("%d  ",a[i]);
   }
}
```

编译并运行这一个程序，根据程序的提示输入 10 个数字，程序会对这些数字进行排序，

然后输出排序以后的结果。

4.5.3 统计字符串中字符

本节将讲解一个统计字符串中字符的例子。文本中的字符串是由大写字母、小写字母、标点符号等不同字符构成的。在程序中可以把文本存储在字符串中，然后分别统计出字符串中的各种字符个数然后输出。

为了识别这些字符，需要学习 ASCII 码。计算机中的所有字符，都是以 ASCII 码的形式保存在计算机中的。每一个 ASCII 码在内存中占据一个字节。ASCII 码与相对应的序号如图 4-2 所示。

Dec	Char	Code	Dec	Char	Dec	Char	Dec	Char
0		NUL	32		64	@	96	`
1		SOH	33	!	65	A	97	a
2		STX	34	"	66	B	98	b
3		ETX	35	#	67	C	99	c
4		EOT	36	$	68	D	100	d
5		ENQ	37	%	69	E	101	e
6		ACK	38	&	70	F	102	f
7		BEL	39	'	71	G	103	g
8		BS	40	(	72	H	104	h
9		HT	41	)	73	I	105	i
10		LF	42	*	74	J	106	j
11		VT	43	+	75	K	107	k
12		FF	44	,	76	L	108	l
13		CR	45	-	77	M	109	m
14		SO	46	.	78	N	110	n
15		SI	47	/	79	O	111	o
16		DLE	48	0	80	P	112	p
17		DC1	49	1	81	Q	113	q
18		DC2	50	2	82	R	114	r
19		DC3	51	3	83	S	115	s
20		DC4	52	4	84	T	116	t
21		NAK	53	5	85	U	117	u
22		SYN	54	6	86	V	118	v
23		ETB	55	7	87	W	119	w
24		CAN	56	8	88	X	120	x
25		EM	57	9	89	Y	121	y
26		SUB	58	:	90	Z	122	z
27		ESC	59	;	91	[	123	{
28		FS	60	<	92	\	124	|
29		GS	61	=	93	]	125	}
30	▲	RS	62	>	94	^	126	~
31	▼	US	63	?	95	_	127	⌂

图 4-2 ASCII 码表

字符与整型变量是等同的，这就很容易把字符与整型变量联系起来，通过序号的比较可以判断这些字符的种类。从图 4-2 可知，字符与序号的分布可以总结成下面的 4 类。

- 48-57：数字字符。
- 65-90：大写字母。
- 97-122：小写字母。
- 其他字符：特殊符号或标点符号。

要完成字符的统计，需要在程序中完成下面这些步骤。

① 定义一个字符数组，用来存储字符串。

② 定义相关的计数变量，变量要赋初值为 0。

③ 从键盘输入一个字符串，这一操作可以用 scanf()函数实现。
④ 用循环的方法访问字符串中的每一个字符。
⑤ 对每一个字符判断字符的种类，然后改变计数的值。
⑥ 用 printf()函数输出统计结果。

根据这些步骤编写的程序代码如下所示。

```
#include <stdio.h>
int main()
{
  char a[100];                              /*定义一个字符数组保存字符串。*/
  int i,uper,lower,marks,num;               /*定义计数的变量。*/
  i=0;                                      /*计数变量赋初值为 0。*/
  uper=0;
  lower=0;
  marks=0;
  num=0;
  printf("please input a string:\n") ;      /*提示输入信息。*/
  scanf("%s",a);                            /*输入一个字符串。*/
  while(a[i]!=NULL)                         /*循环访问字符串中的第一个字符。*/
  {
   if(a[i]<=57&&a[i]>=48)                   /*判断是否是数字。*/
   {
      num++;                                /*是数字则数字计数加 1。*/
   }
   else                                     /*其他的可能。*/
   {
      if(a[i]<=90&&a[i]>=65)                /*判断是否是大写字母。*/
      {
         uper++;                            /*是大写字母则大写字母的记数加 1。*/
      }
      else                                  /*其他的情况。*/
      {
         if(a[i]<=122&&a[i]>=97)            /*判断是否是小写字母。*/
         {
            lower++;                        /*是小写字母则小写字母的记数加 1。*/
         }
         else
         {
            marks++;                        /*其他的情况一定是特殊字符加 1。*/
         }
      }
   }
   i++;                                     /*字符总数加 1，进行下一次循环。*/
  }
  printf("total chars: %d \n",i) ;          /*输出总字符数。*/
  printf("uper chars: %d \n",uper) ;        /*输出大写字符数。*/
  printf("lower chars: %d \n",lower) ;      /*输出小写字符数。*/
  printf("number chars: %d \n",num) ;       /*输出数字字符数。*/
  printf("other chars: %d \n",marks) ;      /*输出其他字符数。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 8.12.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序提示输入一个字符串，这时输入下面的字符串，然后按“Enter”键。需要注意的是，字符串中不要有空格。

```
asdfghJKLMNQWERT123789123;',./ASDJjhjad
```

程序显示出的统计结果如下所示。

```
total chars: 39
uper chars: 14
lower chars: 11
number chars: 9
other chars: 5
```

4.5.4 小写字母转换成大写字母

本节将讲解大小写字母转换的程序。在一个数组中输入一个字符串，把字符串中的小写字母转换成大写字母，而原有的大写字母和特殊字符保存不变。

从图 4-2 可知，一个小写字母的 ASCII 码减去 32，就是相对应的大写字母。程序需要实现的主要步骤如下所示。

① 定义一个字符数组，用来存储字符串。

② 定义一个计数变量，赋初值为 0。

③ 从键盘输入一个字符串，这一操作可以用 scanf()函数实现。

④ 用循环的方法访问字符串中的每一个字符。判断这个字符是不是小写，如果是小写，则减去 32 转换成大写。

⑤ 用 printf()函数输出转换后的字符串。

根据这些步骤编写的程序代码如下所示。

```
#include <stdio.h>
int main()
{
  char a[100];                              /*定义一个字符数组保存字符串。*/
  int i;                                    /*定义一个计数器变量。*/
  i=0;                                      /*计数器变量赋初值为 0。*/
  printf("please input a string:\n") ;      /*提示输入一个字符串。*/
  scanf("%s",a);                            /*从键盘读取字符串。*/
  while(a[i]!=NULL)                         /*访问字符串中的每一个字符。*/
  {
     if(a[i]<=122&&a[i]>=97)                /*判断是不是小写字母。*/
```

```
        {
          a[i]=a[i]-32;                          /*是小写字母则ASCII码减去32转换为大写字
母。*/
        }
       i++;                                      /*计数器加1。*/
      }
      printf("result:  %s \n",a) ;               /*输出转换结果。*/
    }
```

输入下面的命令，编译这个程序。

```
gcc 8.13.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序提示输入一个字符串，这时输入下面的字符串，然后按“Enter”键。需要注意的是，字符串中不要有空格。

```
adajsdHAJDA12834;';'
```

程序会把字符串中的小写字母转换成大写字母，输出结果如下所示。

```
result:  ADAJSDHAJDA12834;';'
```

4.5.5　指针访问数组

根据本章所讲的数组和指针的关系，可知数组在定义时返回了一个头指针。用这个头指针可以访问数组中的第一个变量。指针的算术运算可以访问数组中所有的变量。上一节中的程序，也可以用指针来实现。在程序中，指针访问数组的方法如下所示。

（1）在程序中定义两个指向字符型变量的指针。

（2）字符数组的头指针分别赋值给这两个指针。一个指针用于循环访问字符数组中的每一个字符，另一个指针用于保存头指针的位置。

（3）指针自加表示指向数组中的下一个元素，可以用更改以后的指针来访问指向的字符。判断这个字符的 ASCII 码，如果是小写则减去 32 转换成大写。

根据这个思路编写的程序如下所示。

```
#include <stdio.h>
int main()
{
  char a[100];                                   /*定义一个数组保存字符串。*/
  char *p,*q;                                    /*定义两个指针。*/
  printf("please input a string:\n") ;           /*输出一个指示。*/
  scanf("%s",a);                                 /*输入一个字符串。*/
  p=a;                                           /*将字符串的头指针赋值给指针p。*/
  q=a;                                           /*将字符串的头指针赋值给指针q。*/
  while(*p!=NULL)                                /*访问指针p所在的一个字符。*/
```

```
    {
      if(*p<=122&&*p>=97)                    /*判断是否是小写。*/
      {
        *p= *p-32;                           /*如果是小写则减去 32 转换成大写。*/
      }
      *p++;                                  /*指针向后移动，指向下一个字符。*/
    }
    printf("result:  %s \n",q) ;             /*输出转换的结果。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 8.14.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序提示输入一个字符串，这时输入下面的字符串，然后按“Enter”键。需要注意的是，字符串中不要有空格。

```
ashdjYIUasd;'12
```

程序将小写字母转换成大写字母，输出的结果如下所示。

```
result:  ASHDJYIUASD;'12
```

4.6 常见问题

指针可以对内存进行读写操作，对指针错误的理解和应用可能造成内存数据的错误。本节将讲解指针操作时常见的错误。用指针读取数据时，不正确的访问可能读取不到需要的数据。当用指针进行写操作时，错误的操作可以破坏内存中其他程序的数据。

4.6.1 错误的写操作

在使用指针时，需要牢记，不可以随意向不确定的内存单元写入数据。这种写操作可能破坏内存中其他的数据，引起其他程序的错误。正确的方法是，一个指针先取得正确的内存地址，然后访问一个确定的内存单元。例如下面的程序就是错误地向一个未知的内存单元写入数据。

```
#include <stdio.h>              /*包含一个头文件。*/
main(void)                      /*主函数。*/

{

   int i;                       /*定义一个整型变量。*/
   int *p;                      /*定义一个指向整型变量的指针。*/
   i=10;
```

```
                                /*对整型变量赋值。*/
    *p=i;
                                /*对指针变量指向的值进行赋值。注意，这个操作是错误的。*/
    printf("%d",*p);            /*输出指针指向的值。*/
}
```

分析上面程序中的这一句话。

```
*p=i;
```

p 是新定义的一个指针，还没有对指针进行赋值。这个指针可能指向内存中一个未知的单元。但是程序中，对这个指针指向的地址赋值为 i，这样可能破坏内存中这个未知单元的数值，引起程序的错误。上面的程序，如果要实现指针的赋值，可以改写成下面的程序代码。

```
#include <stdio.h>                          /*包含一个头文件。*/
main(void)                                  /*主函数。*/

{

    int i,j;                                /*定义一个整型变量。*/
    int *p;                                 /*定义一个指向整型变量的指针。*/
    i=10;
                                            /*对整型变量赋值。*/
    p=&j;                                   /*将变量 j 取地址赋值给指针 p。*/
    *p=i;
                                            /*对指针变量指向的值进行赋值。*/
    printf("%d",*p);                        /*输出指针指向的值。*/
}
```

4.6.2　指针的错误赋值

在编程时，常常会发生指针错误赋值的问题。原因是对指针指向的变量进行赋值，但错误理解了指针操作符号，导致直接向这一个指针变量赋值。这种操作会对程序或系统造成影响。例如下面的代码错误地对一个指针进行赋值。

```
#include <stdio.h>                  /*包含一个头文件。*/
main(void)                          /*主函数。*/

{

    int i;                          /*定义一个整型变量。*/
    int *p;                         /*定义一个指向整型变量的指针。*/
    p=10;
                                    /*对指针变量直接赋值。注意，这个操作是错误的。*/
    printf("%d",*p);                /*输出指针指向的值。*/
}
```

在上面的程序中，本意是要对指针所指向的变量进行赋值，但是错误地理解了指针和变量的关系，错误地将一个数据直接赋值给一个指针变量。这个操作会导致程序的错误，正确的操作应该是将一个变量取地址，然后把一个数值赋值给这个指针所指向的变量。正确的代码如下所示。

```
#include <stdio.h>                    /*包含一个头文件。*/
main(void)                            /*主函数。*/

{

   int i;                             /*定义一个整型变量。*/
   int *p;                            /*定义一个指向整型变量的指针。*/
   p=&i;                              /*变量 i 取地址赋值给指针 p。*/
   *p=10;
                                      /*对指针变量直接赋值，这种赋值是正确的。*/
   printf("%d",*p);                   /*输出指针指向的值。*/
}
```

4.6.3 数组指针的越界错误

定义数组时会返回一个指向第一个变量的头指针。这个指针的加减运算可以向前或向后移动这个指针，进而访问数组中所有的变量。但移动指针时，如果不注意移动的次数和位置，会使指针指向数组以外的位置，使数组发生越界错误。例如下面的程序就是移动指针时没有考虑到移动的次数和数组的范围，使程序访问了数组以外的存储单元。

```
#include <stdio.h>                    /*包含一个头文件。*/
main(void)                            /*主函数。*/

{

   int i;                             /*定义一个整型变量。*/
   int *p;                            /*定义一个指向整型变量的指针。*/
   int a[5];                          /*定义一个整型数组。*/
   p=a;                               /*数组 a 的头指针赋值给指针 p。*/
   for(i=0;i<10;i++)
                                      /*for 循环，变量 i 从 0 到 10 执行 10 次循环体。*/
   {

      *p=i+10;
                              /*指针 p 指向的变量。*/
      p++;
                              /*指针 p 下一个变量。*/
   }

}
```

在这个程序中，for 循环会使指针 p 向后移动 10 次，并每次向指针指向的单元赋值。可是数组中只有 5 个变量，后 5 次的操作会对未知的内存区域赋值。这种向内存未知区域赋值的操作会使系统发生错误。正确的操作应该是指针移动的次数与数组中的变量个数相同。正确程序代码如下所示。

```
#include <stdio.h>                    /*包含一个头文件。*/
main(void)                            /*主函数。*/

{
```

```
    int i;                              /*定义一个整型变量。*/
    int *p;                             /*定义一个指向整型变量的指针。*/
    int a[5];                           /*定义一个整型数组。*/
    p=a;                                /*数组 a 的头指针赋值给指针 p。*/
    for(i=0;i<5;i++)
                                        /*for 循环，变量 i 从 0 到 4 执行 5 次循环体。*/
    {

       *p=i+10;
                                        /*指针 p 指向的变量。*/
       p++;
                                        /*指针 p 下一个变量。*/
    }

}
```

4.7　小结

数组与指针的知识是 C 程序中的重要内容，可以实现很多复杂的功能和算法。数组与指针这两个概念的理解是个难点，需要建立起直观形象的空间概念。在实际开发时各种对象与算法常常通过指针与数组建立起来的，需要使用各种复杂的指针和数组。在程序设计练习时需要注意数组与指针的应用技巧。

第 5 章 函 数

5.1 函数的理解

在程序中，各种功能都是通过函数实现的。在学习函数之前需要理解函数的作用与特点。一个函数可能有返回值，可能有不同的参数，完成不同的功能。本节将讲解这些函数的基本概念。

5.1.1 什么是函数

简单地说，函数就是把一个程序功能封装成一个整体。函数由类型名、函数名、参数列表、函数体等部分组成，一般形式如下所示。

```
类型名 函数名(参数列表)
{
   函数体;
}
```

函数这些部分的功能如下所示。

- 类型名：函数一般会返回一个数据，这个数据就是函数返回值。返回值的数据类型就是函数的数据类型。函数的类型可以是所有的数据类型。如果没有返回值，则声明函数的类型为 void。
- 函数名：函数的名称。函数名的规则与变量命令的规则是相同的，函数名需要有效。
- 参数列表：函数输入的参数，函数需要使用这些参数进行运算。
- 函数体：完成函数运算功能的程序。在函数体中可以调用其他的函数。

例如下面是一个求出两个数中最大数的函数。

```
int max(int x ,int y)                    /*定义一个函数。*/
{
   if(x>y)                               /*如果 x 大于 y 就返回 x。*/
   {
      return(x);
   }
   else                                  /*否则就返回 y。*/
   {
      return(y);
   }
}
```

这个函数可以求出两个数中最大的一个数，函数的各部分作用如下所示。

（1）int 是函数的类型名，表示这个函数可以返回一个整型变量。

（2）函数的名称是 max，用这个名称可以调用这个函数。

（3）参数列表是“int x ,int y”，表示这个函数有 x 和 y 两个整型参数，函数体可以调用这两个参数进行程序运算。

（4）函数体中判断两个参数的大小，用 return 语句返回较大的一个变量。

例如下面的程序，是用这个函数判断两个数的大小。

```
#include <stdio.h>
int main()
{
   int a,b,s;                              /*定义一个变量。*/
   printf("please input a:\n");            /*提示输入。*/
   scanf("%d",&a);                         /*输入一个变量。*/
   printf("please input b:\n");            /*提示输入。*/
   scanf("%d",&b);                         /*输入另一个变量。*/
   s=max(a,b);                             /*调用函数 max 求得较大的一个数。*/
   printf("the max number is %d\n",s);     /*输出结果。*/
}
int max(int x ,int y)                      /*定义一个函数。*/
{
   if(x>y)                                 /*如果 x 大于 y 就返回 x。*/
   {
      return(x);
   }
   else                                    /*否则就返回 y。*/
   {
      return(y);
   }
}
```

用下面的命令编译这段代码。

```
gcc 9.1.c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序提示输入数字，输入一个数字 5，再按“Enter”键。然后用同样的方法输入另一个数字 1。程序的运行结果如下所示。

```
the max number is 5
```

5.1.2　系统函数（库函数）与用户自定义的函数

按照函数的来源，可以将函数分为系统函数（库函数）与用户自定义函数。系统函数是编程平台提供的函数。用户自定义函数是用户自己按编程的需要编写的函数。这两种函数的作用与区别如下所示。

- 系统函数：是由系统提供，用户无须定义，也不必在程序中声明的函数。在程序需要使用库函数时，需在程序前包含该函数的头文件。在前面章节里反复使用的 printf、scanf 等函数都是库函数。
- 用户定义函数：用户按编程的需要写的函数。对于用户自定义函数，不仅要在程序中定义函数本身，而且在调用函数模块中还必须对该被调函数进行函数声明，然后才能使用。上一节的函数 max()就是一个用户自定义函数。

5.1.3 函数的返回值

所谓返回值，指的是函数运行以后会产生一个数据，这个数据用 return 语句返回给主程序。有些函数是没有返回值的，只执行了一个运行或操作过程。从这个角度可以把函数分为有返回值函数和无返回值函数两种。

1．有返回值函数

这种函数被调用执行以后，会向调用语句返回一个执行结果，这个结果就是返回值。这种函数在定义时，需要指定函数的类型名。调用函数时，需要有一个变量来接受函数的返回值。例如 5.1.1 节中的下列语句。

```
s=max(a,b);
```

函数 max 会返回一个整型的结果，这个结果需要赋值给变量 s。

2．无返回值函数

此类函数用于完成一定的处理过程，执行完成后不向调用者返回结果。由于函数没有返回值，用户在定义此类函数时需要指定返回为“空类型”。空类型的说明符为“void”。例如下面的自定义函数，就是用一个自定义函数输出 n 个指定字符。

```
#include <stdio.h>
void show(char a ,int x)                    /*定义一个无返回值函数。*/
{
   int i;                                   /*定义一个变量作为计数器。*/
   for(i=0;i<x;i++)                         /*for 循环输出字符。*/
   {
      printf("%c",a);                       /*输出这个字符。*/
   }
}

int main()                                  /*主函数。*/
{
   char c='a';                              /*定义一个变量 c，赋值为 a。*/
   show(c,6)  ;                             /*调用函数，将变量 c 输出 6 次。*/
   printf("\n") ;                           /*输出一个换行。*/
   show('*',9)  ;                           /*调用函数，将星号输出 9 次。*/
}
```

用下面的命令编译这段代码。

```
gcc 9.2.c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
aaaaaa
*********
```

5.1.4　无参函数

参数指的是调用语句向被调用函数传送的数据。从调用语句和被调用函数之间数据传送的角度，又可将函数分为无参函数和有参函数两种。

无参函数指的是函数在定义、函数声明及函数调用中均不带参数。调用语句和被调用函数之间不进行参数传送。此类函数通常用来完成一组指定的功能，可以返回一个参数。例如下面的程序中使用一个无参函数输出一行文本。

```
#include <stdio.h>

void hello()                                /*定义一个无参函数。*/
{
   printf("good morning .\n") ;             /*函数中输出一行文本。*/
}

int main()                                  /*主函数。*/
{
   hello();                                 /*调用函数输出信息，调用时没有参数。*/
   hello();                                 /*再次调用这个函数。*/
}
```

用下面的命令编译这段代码。

```
gcc 9.3.c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
good morning .
good morning .
```

5.1.5　有参函数

与无参函数相对应的是有参函数。有参函数在定义、调用、声明时都有参数。有参函数

也称为带参函数。在使用这些参数时，应该注意以下内容。

（1）在定义函数时，每一个参数都有一个具体的数据类型。

（2）在调用函数时，参数列表中的参数要与函数定义时的参数个数相同，所有的参数类型一一对应。所给的参数数值应该有效。如果调用时，所给的参数不合法，或者类型不匹配，或者个数不相同，程序就会发生错误。

函数定义时的参数被称为形式参数，只在函数体中有效。函数调用时的参数被称为实际参数，在主程序中是有效的变量。

5.1.6 函数参数实例

本节讲解一个函数使用实例。自定义函数有 4 个变量，前三个变量分别为整型，第 4 个变量为一个字符型。函数对三个变量进行从小到大排列，然后用给定的字符连接输出。前面的章节已经讲解过三个数字排序的问题。程序的代码如下所示。

```
#include <stdio.h>
void order(int a ,int b,int c,char s)  /*定义一个函数，有 4 个参数。*/
{
   int temp;                           /*函数中定义一个中间变量。*/
   if(a>b)                             /*如果 a 大于 b，则交换 a 与 b 的值。*/
   {
      temp=a;                          /*a 的值给一个中间变量。*/
      a=b;                             /*b 赋值给 a。*/
      b=temp;                          /*中间变量的值赋值给 b。*/
   }
   if(a>c)                             /*如果 a 大于 c，则交换 a 与 c 的值。*/
   {
       temp=a;
       a=c;
       c=temp;
   }
   if(b>c)                             /*如果 b 大于 c，则交换 b 与 c 的值。*/
   {
       temp=b;
       b=c;
       c=temp;
   }
   printf("%d%c%d%c%d",a, s, b, s, c); /*输出排序以后的值，并用参数中的字符 s 连接。*/
}

int main()
{
   order(5,4,7,'-');                   /*调用函数。*/
   printf("\n");                       /*输出一个换行。*/
   order(7,55,32,'<');
   printf("\n");
   order(9,3,6,'.');
}
```

用下面的命令编译这段代码。

```
gcc 9.4c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
4-5-7
7<32<55
3.6.9
```

自定义函数 order 的类型如下所示。

```
void order(int a ,int b,int c,char s)
```

这个函数没有返回值，有 4 个参数。其中前三个参数是整型，后一个参数是字符类型。所以需要用下面的语句来调用这个函数。调用时，参数列表中的参数需要与定义的参数列表相同。

```
order(5,4,7,'-');
order(7,55,32,'<');
order(9,3,6,'.');
```

5.2　自定义函数

在程序中需要把各种模块封装为自定义函数，在使用这个函数时，可以不考虑函数的内部执行过程。这样可以简化程序的设计，使代码和模块得到有效的重用。

5.2.1　自定义函数的编写

编写复杂程序时，需要将不同的功能分解为多个函数，这样可以简化程序的执行过程。在编写自定义函数时，需要考虑函数的参数与返回值。

例如要编写一个分数等级判断的函数。用户从键盘输入一个分数，程序调用一个函数来判断成绩的等级，并返回表示等级的字符。当输入数字 0 时，退出这个程序。需要用下面的方法来实现这个程序。

（1）需要用 while 循环语句，构成一个循环。

（2）输入一个数值，判断这个数值是不是 0。如果数值为 0，则中止循环。

（3）如果数值不为 0，则调用函数判断成绩。成绩作为函数调用的参数，返回一个字符。

（4）自定义函数可以用 if 语句的嵌套结构来实现分数等级的判断。

根据这个思路，编写出的程序如下所示。

```
#include <stdio.h>
char result(int x)                          /*定义一个分数评定函数。*/
```

```
{
   char s;                                  /*定义一个字符变量存储结果。*/
   if(x>=90)                                /*90 分以上的结果为 A。*/
   {
     s='A';
   }
   else                                     /*其他的值 80 分以上的结果为 B。*/
   {
      if(x>=80)
      {
         s='B';
      }
      else
      {
         if(x>=70)                          /*其他的值 70 分以上的结果为 C。*/
         {
            s='C';
         }
         else
         {
            if(x>=60)                       /*其他的值 60 分以上的结果为 D。*/
            {
               s='D';
            }
            else                            /*其他的值一定小于 60，结果为 E。*/
            {
               s='E';
            }
         }
      }
   }
   return(s);                               /*将变量 s 作为返回值返回。*/
}
int main()                                  /*主函数。*/
{
   char r;                                  /*定义一个存放结果的变量。*/
   int i=1 ;                                /*定义存放分数的变量，赋初值为 1。*/
   while(i!=0)                              /*i 的值不为 0 则提示输入和判断结果。*/
   {
     scanf("%d",&i);                        /*输入一个数值。*/
     r=result(i);                           /*调用函数判断等级，返回值赋值给 r。*/
     printf("%c\n",r);                      /*输出结果。*/
   }
}
```

输入下面的命令编译这段代码。

```
gcc 9.5c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

在程序中输入一个数字 55，然后按“Enter”键，程序会输出一个结果 E。输入相应的数值，程序能正确输出判断结果。输入数字 0，程序结束。

5.2.2　函数中调用函数

在自定义函数中，可以调用另外一个自定义函数。这样，就可以把复杂的程序功能分解为多个简单的函数。例如上一节中的程序，没有对输入值的有效性进行判断。可以进行分数评级之前，用一个函数判断输入的值是否合理，然后这个函数再调用分数评级函数 result。程序的代码如下所示。

```
#include <stdio.h>
char result(int x)                          /*定义一个分数评定函数。*/
{
   char s;                                  /*定义一个字符变量存储结果。*/
   if(x>=90)                                /*90 分以上的结果为 A。*/
   {
     s='A';
   }
   else                                     /*其他的值 80 分以上的结果为 B。*/
   {
      if(x>=80)
      {
         s='B';
      }
      else
      {
         if(x>=70)                          /*其他的值 70 分以上的结果为 C。*/
         {
            s='C';
         }
         else
         {
            if(x>=60)                       /*其他的值 60 分以上的结果为 D。*/
            {
               s='D';
            }
            else                            /*其他的值一定小于 60，结果为 E。*/
            {
               s='E';
            }
         }
      }
   }
   return(s);                               /*将变量 s 作为返回值返回。*/
}
void myresult(int x)                        /*定义一个有数值有效性判断功能的函数，无返
```

```
回值。*/
    {
       char s;                                  /*存放结果的变量。*/
       if(x>0&&x<=100)                          /*数值在这个范围内有效。*/
       {
          s=result(x);                          /*调用函数 result()判断分数等级。*/
          printf("%c\n",s);                     /*输出结果。*/
       }
       else                                     /*其他值的分数。*/
       {
          printf("error.\n",s);                 /*输出错误提示。*/
       }
    }
    int main()                                  /*主函数。*/
    {
       int i=1 ;                                /*定义存放分数的变量，赋初值为 1。*/
       while(i!=0)                              /*i 的值不为 0 则提示输入和判断结果。*/
       {
         scanf("%d",&i);                        /*输入一个数值。*/
         myresult(i);                           /*调用函数判断等级，返回值赋值给 r。*/
       }
    }
```

输入下面的命令编译这段代码。

```
gcc 9.6c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

在程序中输入一个数字 123，然后按“Enter”键，程序会输出提示“error.”。输入 1 至 100 的数值，程序会判断并显示出这个分数的级别。输入数值 0，程序结束运行。

5.2.3 函数的声明

在 C 程序中，如果需要调用后面的自定义函数，需要在程序最前面声明函数。如果未声明，程序就会发生错误，提示程序中找不到相应的函数。

函数的声明方法，是在程序的最前面，列出程序中可能调用的自定义函数。需要列出与函数定义相同的类型名、函数名、参数列表。下面的程序，是将上一节中的自定义函数写在后面，并在程序最前面声明函数。

```
    #include <stdio.h>
    void myresult(int x);                       /*声明程序中的函数 myresult。*/
    char result(int aax);                       /*声明函数 result。*/
                                                /*函数声明结束，下面是主函数。*/
    int main()                                  /*主函数将调用后面的函数。*/
    {
```

```
      int i=1 ;                                /*定义存放分数的变量，赋初值为 1。*/
      while(i!=0)                              /*i 的值不为 0 则提示输入和判断结果。*/
      {
        scanf("%d",&i);                        /*输入一个数值。*/
        myresult(i);                           /*调用函数判断等级，返回值赋值给 r。*/
      }
   }
   void myresult(int x)                        /*定义一个有数值有效性判断功能的函数，无返
回值。*/
   {
      char s;                                  /*存放结果的变量。*/
      if(x>0&&x<=100)                          /*数值在这个范围内有效。*/
      {
        s=result(x);                           /*调用函数 result()判断分数等级，调用的是
后面定义的函数。*/
        printf("%c\n",s);                      /*输出结果。*/
      }
      else                                     /*其他值的分数。*/
      {
        printf("error.\n",s);                  /*输出错误提示。*/
      }
   }
   char result(int x)                          /*定义一个分数评定函数。*/
   {
      char s;                                  /*定义一个字符变量存储结果。*/
      if(x>=90)                                /*90 分以上的结果为 A。*/
      {
       s='A';
      }
      else                                     /*其他的值 80 分以上的结果为 B。*/
      {
        if(x>=80)
        {
          s='B';
        }
        else
        {
          if(x>=70)                            /*其他的值 70 分以上的结果为 C。*/
          {
            s='C';
          }
          else
          {
            if(x>=60)                          /*其他的值 60 分以上的结果为 D。*/
            {
              s='D';
            }
            else                               /*其他的值一定小于 60，结果为 E。*/
            {
              s='E';
```

```
            }
          }
        }
    }
    return(s);                                  /*将变量 s 作为返回值返回。*/
}
```

这个程序与上一节中的程序运行结果是相同的。只是在主函数的后面编写自定义函数，在程序的最前面声明函数。

5.2.4 递归函数

所谓递归函数，指的是函数调用自身，用循环的方法实现运算。例如一个程序求出 100 以内的整数和，可以用下面的思想。

```
100 以内的整数和 = 100 + 99 以内的整数和;
99 以内的整数和 = 99 +98 以内的整数和;
……
2 以内的整数和 = 2 +以 1 以内的整数和;
1 以内的整数和为 1。
```

用这种思路可以用反复调用自身函数的方法求出 100 以内的整数和。这个程序的代码如下所示。

```
#include <stdio.h>
int addn(x)                                 /*定义一个函数，实现 0 到 x 之间整数的相加。*/
{
   int s;                                   /*定义一个变量保存结果。*/
   if(x==1)                                 /*结果 x 为 1，则结果为 1。*/
   {
      s=1;
   }
   else                                     /*x 不为 1 的情况。*/
   {
      s=x + addn(x-1);                      /*递归调用自身函数。*/
   }
   return(s);                               /*输出一个结果。*/
}

int main()
{
    int i=100;                              /*定义一个变量，赋值为 100。*/
    int j;                                  /*定义一个变量存储结果。*/
    j=addn(i);                              /*调用函数求出结果。*/
    printf("result: %d",j);                 /*输出结果。*/
}
```

输入下面的命令编译这段代码。

```
gcc 9.8c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的结果如下所示。

```
result: 5050
```

5.2.5　main 函数的参数 argc 与 argv

前面的例子中，main()函数都是没有参数的。但编写的程序常常需要从命令行中输入参数。例如，Linux 命令中复制文件的命令如下所示。

```
cp /root/1.txt /root/2.txt
```

“cp”后面的两个字符串就是这个程序的参数。在 main()函数中，默认有 argc 与 argv 两个参数。main 函数的原型如下所示。

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

main 函数是有一个返回值的，如果程序执行成功，则返回一个整型值 1。程序中常常省略这个返回值。argc 是一个整型变量，表示参数的个数。*argv[]是一个指针型数组，数组中的每一个指针指向一个参数的字符数组。下面的程序可以输出程序启动时后面的参数。

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   int i ;                                    /*定义一个计数器变量。*/
   for(i=0;i<argc;i++)                        /*用变量 i 进行 for 循环。*/
   {
      printf("%s\n",argv[i]);                 /*输出一个指针所指向的字符串。*/
   }
}
```

输入下面的命令编译这段代码。

```
gcc 9.9c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。在程序的后面加上若干个参数。

```
./a.out hello good evening
```

程序的结果如下所示。

```
hello
good
evening
```

5.3 函数与指针

函数的参数与返回值可以是指针。使用指针变量可以在函数中对其他函数的变量进行调用。利用数组的头指针，可以用指针移动的方法访问数组中的每个元素，所以可用指针作为参数使函数访问一个数组。本节将讲述函数中的指针应用。

5.3.1 值调用与引用调用

函数对参数的调用，有值调用和引用调用两种方法。前面的例子都是值调用。函数对这两种参数调用的处理方法是不同的。

- 值调用：进行函数调用时候，给形式参数分配内存空间，并把实际参数的值直接传递给形式参数。这一过程是参数值的直接传递过程。一旦形式参数获得了值，形式参数的变化对实际参数没有任何影响。
- 引用调用：当用指针或数组作为函数的参数时，可以用指针来访问实际参数的值。形式参数的变化将直接引起实际参数的变化。

> **注意**：调用语句中的参数是实际参数，被调用函数中的参数是形式参数。这一概念见本章 5.1.5 节所述。

例如在下面的程序中，形式参数的改变不会影响实际参数的值。

```
#include <stdio.h>
int aa(int x)                              /*定义一个函数，参数为 x。*/
{
   x = x*x;                                /*更改 x 的值。*/
   return(x);                              /*返回 x。*/
}
main()
{
   int x,y;                                /*定义两个整型变量 x 与 y。*/
   x=5;                                    /*x 赋值为 5。*/
   y=aa(x);                                /*调用函数，返回值赋值给 y。*/
   printf("X: %d Y: %d\n",x,y);            /*输出 x 与 y。*/
}
```

输入下面的命令编译这段代码。

```
gcc 9.10.c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的结果如下所示。

```
X: 5 Y: 25
```

从结果可知，函数中更改形式参数的值，并不会对实际参数的值造成影响。实际参数的值复制给形式参数以后，形式参数无法访问实际参数。

5.3.2　引用调用与指针

虽然函数调用时默认是值调用，但是传入的参数如果是指针，函数可以通过这个指针改变函数外部变量的值。参数的引用调用就是通过指针对外部变量的访问来实现的。

指针可以同普通变量一样，作为一个参数传递给一个函数。函数可以更改这一个指针和访问指针指向的变量。下面的程序是通过传入指针来更改函数以外变量的值。

```
#include <stdio.h>
void change(int *x,int *y);              /*声明一个函数。*/

main()                                   /*主函数。*/
{
   int m,n;                              /*定义两个变量。*/
   m=3;                                  /*对两个变量赋值。*/
   n=5;
   printf("m:%d  n:%d\n",m,n);           /*输出两个变量。*/
   change(&m,&n);                        /*用函数交换两个变量。*/
   printf("m:%d  n:%d\n",m,n);           /*输出两个变量。*/
   change(&m,&n);                        /*用函数交换两个变量。*/
   printf("m:%d  n:%d\n",m,n);           /*输出两个变量。*/
}

void change(int *x,int *y)               /*定义一个函数，指针作为函数的参数。*/
{
   int temp;                             /*定义一个中间变量。*/
   temp=*x;                              /*指针指向的值赋值给中间变量。*/
   *x=*y;                                /*指针 y 指向的值赋值给指针 x 指向的值。*/
   *y=temp;                              /*中间变量的值赋值给指针 y 指向的值。*/
}
```

输入下面的命令编译这段代码。

```
gcc 9.11.c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序输出的结果如下所示。

```
m:3  n:5
m:5  n:3
m:3  n:5
```

从结果可知，自定义函数通过参数中的指针访问指针所指向的变量，从而交换了两个变

量的值。这种指针访问变量的方法就是函数对参数的引用调用。

5.3.3 指针参数简单实例

本节将讲解一个指针参数实例，通过自定义函数对参数的引用调用对三个变量进行排序。函数有三个参数，分别为三个变量的指针。函数通过指针来访问主函数中的数值。程序的代码如下所示。

```
#include <stdio.h>
void order(int *x ,int *y,int *z);      /*声明一个函数。*/

main()                                  /*主函数。*/
{
   int a,b,c;                           /*定义三个变量。*/
   a=5;                                 /*对三个变量赋值。*/
   b=3;
   c=8;
   printf("%d %d %d\n",a,b,c);          /*输出三个变量。*/
   order(&a,&b,&c);                     /*调用函数对三个变量进行排序。*/
   printf("%d %d %d\n",a,b,c);          /*输出三个变量。*/
}

void order(int *x ,int *y,int *z)       /*自定义函数对三个变量进行排序。*/
{
   int temp;                            /*定义一个中间变量。*/
   if(*x > *y)                          /*对指针指向的值进行比较。*/
   {
      temp=*x;                          /*用一个中间变量存储指针变量 x 指向的值。*/
      *x=*y;                            /*指针 y 指向的值赋值给指针 x 指向的值。*/
      *y=temp;                          /*中间变量的值赋值给指针 y 指向的值。*/
   }
   if(*x > *z)                          /*同样的方法，对 x 和 z 两个指针指向的值进行
                                        排序。*/
   {
      temp=*x;
      *x=*z;
      *z=temp;
   }
   if(*y > *z)                          /*同样的方法，对 y 和 z 两个指针指向的值进行
                                        排序。*/
   {
      temp=*y;
      *y=*z;
      *z=temp;
   }
}
```

输入下面的命令编译这段代码。

```
gcc 9.12.c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序输出的结果如下所示。

```
5 3 8
3 5 8
```

与前面章节的三个数字进行排序不同的是这个程序调用一个函数，并将三个变量的指针作为参数，函数通过指针来访问程序中的变量。

5.3.4　自定义函数中指针使用实例

在 5.3.2 节中，用指针实现了两个变量的交换。在 5.3.3 节中，对两个变量进行排序时需要对两个变量使用指针访问进行交换。实际上，在自定义函数中可以用函数中的指针作为参数访问另外一个函数。本节中使用了与 5.3.2 节中相同的函数，代码如下所示。

```
#include <stdio.h>
void order(int *x ,int *y,int *z);          /*声明一个函数。*/
void change(int *x,int *y);                 /*声明一个函数。*/

main()                                      /*主函数。*/
{
   int a,b,c;                               /*定义三个变量。*/
   a=5;                                  /*对三个变量赋值。*/
   b=3;
   c=8;
   printf("%d %d %d\n",a,b,c);          /*输出三个变量。*/
   order(&a,&b,&c);                     /*调用函数对三个变量进行排序。*/
   printf("%d %d %d\n",a,b,c);          /*输出三个变量。*/
}

void order(int *x ,int *y,int *z)   /*自定义函数对三个变量进行排序。*/
{
   if(*x > *y)                          /*对指针 x 和 y 指向的值进行比较。*/
   {
     change(x,y);                       /*用函数 change 交换两个指针所指向的变量。*/
   }
   if(*x > *z)                          /*对指针 y 和 z 指向的值进行比较。*/
   {
      change(x,z);                      /*用函数 change 交换两个指针所指向的变量。*/
   }
   if(*y > *z)                          /*对指针 y 和 z 指向的值进行比较。*/
   {
     change(y,z);                       /*用函数 change 交换两个指针所指向的变量。*/
   }
}
```

```
void change(int *x,int *y)          /*定义一个函数，指针作为函数的参数。*/
{
   int temp;                        /*定义一个中间变量。*/
   temp=*x;                         /*指针指向的值赋值给中间变量。*/
   *x=*y;                           /*指针 y 指向的值赋值给指针 x 指向的值。*/
   *y=temp;                         /*中间变量的值赋值给指针 y 指向的值。*/
}
```

输入下面的命令编译这段代码。

```
gcc 9.13c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的功能与上一节的功能是相同的，实现对三个数的排序。在排序时使用了两次引用调用函数。程序输出的结果如下所示。

```
5 3 8
3 5 8
```

5.3.5 数组作为参数

数组在定义时会返回一个头指针，可以把这个头指针作为函数的参数，函数可以通过指针访问数组中所有的变量。例如下面的代码就是把数组的指针作为函数的参数，函数用指针来访问一个数组。

```
#include <stdio.h>
void myarray(int *a)            /*定义一个函数，用数组作为参数。*/
{
  int i;                        /*定义一个循环变量。*/
  for(i=0;i<10;i++)             /*for 循环，处理数组中的值。*/
   {
      *a=*a+15;                 /*指针指向的变量加上 15。*/
      a++;                      /*指针向后移动一个变量。*/
   }
}

int main()                      /*主函数。*/
{
   int i ;                      /*定义一个变量作为计数器。*/
   int m[10];                   /*定义一个数组。*/
   for(i=0;i<10;i++)            /*用 for 循环对数组赋值。*/
   {
      m[i]=i+10;                /*数组赋值。*/
   }
   for(i=0;i<10;i++)            /*for 循环输出结果。*/
```

```
    {
       printf("%d ",m[i]);          /*输出结果。*/
    }
    printf("\n");                   /*输出一个换行。*/
    myarray(m);                     /*把数组的头指针作为参数，访问数组中的每一个值。*/
    for(i=0;i<10;i++)               /*for 循环输出结果。*/
    {
       printf("%d ",m[i]);          /*输出结果。*/
    }
}
```

输入下面的命令编译这段代码。

```
gcc 9.14c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

在这个程序中，将数组的头指针作为函数的参数。函数通过这个指针来访问数组中的每一个变量，将每个变量的值加 15。程序的运行结果如下所示。

```
10 11 12 13 14 15 16 17 18 19
25 26 27 28 29 30 31 32 33 34
```

5.3.6 数组作为函数参数实例

数组作为函数参数，可以方便地访问函数外部的数组，这样函数就可以对函数外的多个变量发生作用。本节将讲解一个数组作为函数参数的实例，实现下面这些功能。

（1）定义三个字符串数组，前两个字符串数组分别赋值。

（2）自定义一个函数，将三个字符串数组的指针作为参数。函数将第二个数组添加到第一个数组后面，然后保存到第三个数组上。

（3）对数组的访问是通过数组的指针完成的。

程序的代码如下所示。

```
#include <stdio.h>
void stradd(char *a,char *b,char *c)/*定义一个连接字符串的函数,参数是 3 个指针。*/
{
   while(*a!=NULL)                    /*如果指针 a 对定的字符不是空字符则进行循环。*/
   {
      *c=*a;                          /*指针 a 指向的字符赋值给指针 c 指向的字符。*/
      c++;                            /*指针 c 向后移动一个元素。*/
      a++;                            /*指针 a 指向下一个变量。*/
   }
   while(*b!=NULL)                    /*如果指针 b 指向的变量不为空字符则进行循环。*/
   {
      *c=*b;                          /*指针 b 指向的字符赋值给指针 c 指向的字符。*/
      c++;                            /*指针 c 指向下一个字符。*/
      b++;                            /*指针 b 指向下一个字符。*/
   }
}

int main()
{
    char a[30]="abcdefg";             /*定义一个字符串数组，赋初值。*/
    char b[30]="HIJKLMN";             /*定义一个字符串数组 b，赋初值。*/
    char c[30]="";                    /*定义一个字值串数组，赋值为空。*/
    printf("a:  %s\n",a);             /*输出第一个字符串数组。*/
    printf("b:  %s\n",b);             /*输出第二个字符串数组。*/
    stradd(a,b,c);                    /*调用函数，连接两个字符串数组。*/
    printf("c:  %s\n",c);             /*输出字符串 c。*/
}
```

在这个程序的 stradd 函数中，是通过指针的移动和判断字符来实现字符串的连接的。下面的代码实现了字符串 a 复制到字符串 c 上。

```
   while(*a!=NULL)                    /*如果指针 a 对定的字符不是空字符则进行循环。*/
   {
      *c=*a;                          /*指针 a 指向的字符赋值给指针 c 指向的字符。*/
      c++;                            /*指针 c 向后移动一个元素。*/
      a++;                            /*指针 a 指向下一个变量。*/
   }
```

程序是用下面的方法来执行字符串的复制的。

（1）先判断指针 a 指向的字符是不是 NULL，如果不是 NULL，则进行 while 循环。

（2）将指针 a 当前的字符赋值给指针 c 当前的字符。在这个过程中是用 while 循环来逐个进行字符复制的。

（3）指针 c 指向后一个字符。

（4）指针 a 指向后一个字符。然后进行下次循环。结果指针 a 指向的字符为 NULL，表示数组 a 已经结束。

（5）用同样的步骤将字符串 b 上所有的字符复制到字符串 c 后面。

输入下面的命令编译这段代码。

```
gcc 9.15c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
a:  abcdefg
b:  HIJKLMN
c:  abcdefgHIJKLMN
```

5.4　返回值

函数执行一个操作过程以后一般需要返回一个处理结果，这个结果就是函数的返回值。函数可以用一个数值返回、用指针返回，也可以用指针直接访问程序中的变量。本节将讲解函数的返回值问题。

5.4.1　函数返回值的类型

函数的返回值根据数据类型和处理方式的不同，一般有下面的几种形式。函数对这几种类型返回值的处理是不同的。

- 无返回值：这种函数只是完成了一个处理过程。完成函数的运行以后程序会自动中止运行。无返回值的函数需要用 void 来声明。
- 返回一个具体值：函数返回运算的结果。函数执行的是一个运算，对参数进行处理以后将结果作为一个具体数据类型的变量值来返回。这类函数在定义时需要声明函数的数据类型。
- 用指针来访问主函数中的变量：在 5.3.3 节与 5.3.4 节讲解的内容中，函数可以通过指针来访问函数外部的变量。这种函数虽然没有返回值，但是对外部变量产生了影响。指针访问外部变量的方式，可以看作是一种特殊的返回值。
- 返回一个指针：指针是一种特殊的数据类型。函数中的指针可以作为返回值返回给主函数。这类函数需要用指针数据类型来声明。

- 返回结果到参数：如果参数中有一个指针，可以用这个指针直接访问这个指针所指向的变量。这种方式可以看作是将结果返回到参数中。

前三种情况在前面的节已经讲解。后面的节将重点讲解后面的两种返回类型。

5.4.2 函数返回指针

指针是一种特殊的数据类型，表示指向一个变量的内存地址。函数也可以返回一个指针，在主函数中可以用这个指针来访问所指向的变量。例如下面的程序是用一个函数来选择两个数中较大的一个数，返回较大数的指针，然后主函数通过这个指针输出结果。程序的代码如下所示。

```
#include <stdio.h>
int *max(int *i ,int *j)          /*定义一个函数求两个数的最大值，返回一个指针。*/
{
    int *p;                       /*定义一个指针，存储结果。*/
    if(*i>*j)                     /*比较两个指针所指向变量的大小。*/
    {
       p=i;                       /*指针 i 赋值给指针 p。*/
    }
    else
    {
       p=j;                       /*另一种情况是指针 j 赋值给指针 p。*/
    }
    return(p);                    /*返回指针 p。*/
}

int main()                        /*程序的主函数。*/
{
   int i ,j ;                     /*定义两个变量。*/
   int *q;                        /*定义一个指向整型变量的指针。*/
   i=5;                           /*变量赋初值。*/
   j=3;
   q=max(&i,&j);                  /*取两个数的指针作为参数，求两个数的较大值。
                                  函数会返回较大数的指针。*/
   printf("max = %d" ,*q);        /*输出指针 q 指向的变量值。*/
}
```

输入下面的命令编译这段代码。

```
gcc 9.16c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
5
```

5.4.3　函数返回指针实例

本节将讲解一个函数返回指针实例。自定义一个函数连接两个字符串，两个字符串的指针作为函数的参数，函数连接两个字符串以后返回连接以后字符串的指针。程序的代码如下所示。

```
    #include <stdio.h>
    char *stradd(char *a,char *b)   /*定义一个连接字符串函数,参数是两个指针，返回一个指针。*/
    {
       char *p;                     /*定义一个指针 p。*/
       p=a;                         /*把指针 p 赋值给指针 a。*/
       while(*a!=NULL)              /*如果指针 a 对定的字符不是空字符则进行循环。*/
       {
          a++;                      /*指针 a 指向下一个变量。*/
       }
       while(*b!=NULL)              /*如果指针 b 指向的变量不为空字符则进行循环。*/
       {
          *a=*b;                    /*指针 b 指向的字符赋值给指针 c 指向的字符。*/
          b++;                      /*指针 b 指向下一个字符。*/
          a++;                      /*指针 a 指向下一个字符。*/
       }
       return(p);                   /*返回指针 p，也就是返回原来的指针 a。*/
    }

    int main()
    {
        char a[30]="abcdefg";       /*定义一个字符串数组 a，赋初值。*/
        char b[30]="HIJKLMN";       /*定义一个字符串数组 b，赋初值。*/
        char *c;                    /*定义一个指向字符的指针 c。*/
        printf("a:  %s\n",a);       /*输出第一个字符串数组。*/
        printf("b:  %s\n",b);       /*输出第二个字符串数组。*/
        c= stradd(a,b);             /*调用函数，连接两个字符串数组，然后返回一个指针。*/
        printf("c:  %s\n",c);       /*输出这个指针所指向的字符数组。*/
    }
```

输入下面的命令编译这段代码。

```
gcc 9.17c
```

然后对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

在这个程序中，函数把字符串 b 添加到字符串 a 上，实际上返回的是 a 的指针。程序的运行结果如下所示。

```
a:  abcdefg
b:  HIJKLMN
c:  abcdefgHIJKLMN
```

5.4.4 函数返回结果到参数

如果函数的参数是指针，那么函数可以访问这个指针所指向的变量的值，函数就可以把结果返回给参数中的指针变量。如果函数的参数不是指针，函数是不能把结果返回给参数的。例如下面的代码是实现两个数的加法，将结果返回给参数的指针。

```
#include <stdio.h>
void  add(int a,int b,int *c)   /*定义一个自定义函数，参数是两个整型变量和一个指针。*/
{
   *c=a+b;                      /*将相加的结果赋值给参数中指针指向的值。*/
}

int main()                      /*程序的主函数。*/
{
   int a,b,c;                   /*定义三个整型变量。*/
   a=5;                         /*变量赋值。*/
   b=8;
   add(a,b,&c);                 /*调用函数求和，c 取地址作为参数。*/
   printf("reslut is %d",c);    /*输出 c 的值。*/
}
```

5.5 库函数

在 C 程序的编译器中有大量的库函数。库函数可以完成复杂的程序功能。用户在编程时可以直接调用系统的库函数。本章将介绍库函数的调用方法。

5.5.1 库函数的种类

库函数是按照不同的功能封装在多个头文件中的。用户在需要调用库函数时需要在程序的最前面用 include 语句包含相关的头文件。库函数的常用功能与种类如下所示。

- 字符类型分类函数：对字符按照字母、数字、控制字符、分隔符、大小写字母等方面进行分类和测试。
- 转换函数：用于字符或字符串的转换。可以在字符变量和各类数字变量之间进行转换，也可以在大、小写之间进行数据类型转换。
- 目录路径函数：用于文件目录和路径操作，可以对文件或目录进行创建、删除、移动等操作。
- 诊断函数：用于系统内部错误检测，完成系统管理、系统诊断的功能。
- 图形函数：用于屏幕管理和绘制各种图形，这些函数可以用不同颜色设计出图形效果。
- 输入输出函数：用各种设备的输入输出功能，实现用户与这些设备的交互与管理。
- 接口函数：用于系统、BIOS 和硬件的接口，完成设备的控制与管理。
- 字符串函数：用于字符串操作和处理，完成字符串的复制、转换、全并等功能。

- 内存管理函数：用于内存管理，程序可以用这些功能申请或释放内存。在数据结构中常常使用这些内存管理函数。
- 数学函数：用于数学函数计算。例如开方、乘方、指数、对数等数学运算，是通过调用数学函数来完成的。
- 日期和时间函数：用于日期、时间转换操作。程序调用系统时间，进行时间有关的操作需要调用时间函数。
- 进程控制函数：在 C 程序中，可以使用进程控制函数对 Linux 的进程、线程、管道进行管理。

5.5.2　库函数包含文件

要使用系统库函数需要在程序的最前面包含相应的头文件。系统函数是这些头文件定义的，可以查看这些头文件。

（1）单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。

（2）在终端中输入下面的命令，进入到系统存储头文件的文件夹。

```
cd /usr/include
```

（3）输入“ls”命令，查看这个目录下的文件。可以发现这个文件夹下有很多.h 头文件。下面是其中的一部分头文件。

```
gcrypt-module.h     monetary.h      ppmcmap.h       xf86mm.h
gdbm.h              mp.h            ppmfloyd.h      xlocale.h
gdcache.h           mqueue.h        ppm.h           zconf.h
gdfontg.h           ncurses.h       pr29.h          zlib.h
gdfontl.h           netdb.h         printf.h        gcrypt.h
gdfontmb.h          newt.h          profile.h       gcrypt.h
```

（4）输入下面的命令，查看一个头文件“head.h”。这个头文件定义了与系统时间有关的函数。

```
vim time.h
```

（5）查看头文件的代码，可以发现文件定义了一些系统变量与系统函数。下面代码是这个文件的一部分，定义了一个表示时间的结构体。文件对这个结构体的用法和参数作了详细的说明，在编程时，可以查看这些帮助信息。

```
struct tm
{
  int tm_sec;                   /* Seconds.     [0-60] (1 leap second) */
  int tm_min;                   /* Minutes.     [0-59] */
  int tm_hour;                  /* Hours.       [0-23] */
  int tm_mday;                  /* Day.        [1-31] */
  int tm_mon;                   /* Month.       [0-11] */
  int tm_year;                  /* Year - 1900.  */
  int tm_wday;                  /* Day of week. [0-6] */
```

```
  int tm_yday;                  /* Days in year.[0-365] */
  int tm_isdst;                 /* DST.        [-1/0/1]*/
  long int tm_gmtoff;           /* Seconds east of UTC.  */
  long int __tm_gmtoff;         /* Seconds east of UTC.  */
  __const char *__tm_zone;      /* Timezone abbreviation.  */
}
```

5.5.3 头文件使用实例

编程中有很多功能需要调用系统头文件，以便使用头文件中的函数。本节讲解一个头文件使用的实例，实现系统时间的调用。程序的代码如下所示。

```
#include <stdio.h>          /*包含标准输入输出的头文件。*/
#include <time.h>           /*包含时间头文件。*/
main(void)                  /*主函数。*/
{
    struct tm *ptr;         /*定义一个指针，这个数据类型在头文件 time.h 里面定义。*/
    time_t lt;              /*定义一个变量，这个变量在头文件 time.h 里面定义。*/
    lt =time(NULL);         /*取得当前的系统时间。*/
    ptr=gmtime(&lt);        /*格式化这个时间。*/
    printf(asctime(ptr));   /*输出这个时间。*/
    printf(ctime(&lt));     /*以另一种形式输出当前的时间。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 9.19.c
```

输入下面的命令，对编译后的程序添加可执行命令。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
Thu Dec 20 15:47:19 2007
Thu Dec 20 15:47:19 2007
```

5.6　小结

本章讲述了 C 程序中的函数操作。学习难点是自定义函数的理解和使用。函数可以把一个功能或模块封装起来，程序可以方便地使用这个已经封装的功能。在自定义函数中，需要注意函数的参数与返回值。指针与数组作为一种特殊的数据类型可以作为函数的参数和返回值，这一知识比较难以理解，读者在学习时需要根据实例理解函数中指针和数组的作用。

第 6 章　字符与字符串处理

6.1　字符测试函数介绍

所谓字符测试，是指对一个字符进行大小写是否可以打印、是否可以显示、是否是数字等方面进行判断。C 程序提供了丰富的字符测试函数，这些函数是头文件“ctype.h”定义的，使用这些函数之前需要包含这个头文件。

6.1.1　数字或字母测试函数 isalnum

函数 isalnum 的作用，是检查参数 c 是否为英文字母或阿拉伯数字。若参数 c 是一个字母或数字，则返回真值，否则返回值为假。这个函数包含于头文件“ctype.h”中，使用方法如下所示。

```
int isalnum (int c)
```

参数 c 是一个字符变量。但在 C 程序中，字符变量等同于这个变量所对应的 ASCII 码的值，所以参数也可以是一个 ASCII 码值的整型数值。下面是这个函数的使用实例，测试一个字符数组中所有的字符，如果是字母或数字则输出结果。

```
#include <stdio.h>
#include <ctype.h>                          /*包含头文件 ctype.h。*/
main()
{
  char s[]="12as056;^*&";                   /*定义一个字符串数组。*/
  int i;                                    /*定义一个整型变量作为循环的计数器。*/
  for (i=0;s[i]!=NULL;i++ )                 /*for 循环，判断 s[i]是否为空作为循环条件。*/
  {
    if (isalnum(s[i]))                      /*判断当前的字符是不是一个字母或数字。*/
    {
       printf("%c is a number or character.\n",s[i]);      /*输出结果。*/
    }
  }
}
```

输入下面的命令编译这段代码。

```
gcc 10.1.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序会用循环的方法，对字符数组中的每一个字符进行测试，如果是字母或数字，则输出结果。程序的运行结果如下所示。

```
1 is an number or character.
2 is an number or character.
a is an number or character.
s is an number or character.
0 is an number or character.
5 is an number or character.
6 is an number or character.
```

6.1.2　字母测试函数 isalpha

函数 isalpha 可以测试一个字符是不是英文字母。这个函数的使用方法如下所示。

```
int isalpha (int c)
```

函数的参数 c 表示一个字母或表示一个字母的 ASCII 码值。如果这个参数是一个英文字母，则返回真值，否则返回值为假。这里所说的英文字母，指的是 26 个大写字母和 26 个小写字母，而不包括其他的任何字符。下面的程序是对一个字符数组中的每一个字符进行测试，如果是字母则输出结果。

```
#include <stdio.h>
#include <ctype.h>                          /*包含头文件 ctype.h。*/
main()
{
  char s[]="12as056;^*&";                   /*定义一个字符串数组。*/
  int i;                                    /*定义一个整型变量作为循环的计数器。*/
  for (i=0;s[i]!=NULL;i++ )                 /*for 循环，判断 s[i]是否为空作为循环
条件。*/
  {
    if (isalpha(s[i]))                      /*判断当前的字符是不是字母。*/
    {
 printf("%c is a character.\n",s[i]);       /*输出结果。*/
    }
  }
}
```

输入下面的命令编译这段代码。

```
gcc 10.2.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序会测试字符数组中的每一个字符，如果是字母就输出结果，运行结果如下所示。

```
a is a character.
s is a character.
Q is a character.
W is a character.
```

6.1.3 可打印字符测试函数 isgraph

所谓可打印字符，指的是这个字符可以在屏幕上显示，或是可以在打印机中打印出这个字符。ASCII 码中的有些字符，是用于格式控制或特殊作用，是不可以打印的。可打印字符包括字母、数字、标点符号、键盘上可打印符号等。

函数 isgraph 的作用是判断一个字符是否是可打印字符，使用方法如下所示。

```
int isgraph (int c)
```

参数 c 表示一个字符，或表示一个字符的 ASCII 码值。如果参数是一个可打印字符，则返回一个真值，否则返回值为假。下面的代码是使用这个函数判断字符可否打印的实例。

```
#include <stdio.h>
#include <ctype.h>                              /*包含头文件 ctype.h。*/
main()
{
  char s[]="12 0Q\n*&";                         /*定义一个字符串数组。*/
  int i;                                        /*定义一个整型变量作为循环的计数器。*/
  for (i=0;s[i]!=NULL;i++ )                     /*for 循环，判断 s[i]是否为空作为循环
条件。*/
  {
    if (isgraph(s[i]))                          /*判断当前字符是否是可打印字符。*/
    {
       printf("%c is a printable character.\n",s[i]);   /*如果是可打印字符则输
出结果。*/
    }
  }
}
```

输入下面的命令编译这段代码。

```
gcc 10.3.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序会循环测试字符数组中的每一个字符，如果是可打印字符就输出结果，运行结果如下所示。

```
1 is a printable character character.
2 is a printable character character.
0 is a printable character character.
Q is a printable character character.
```

```
* is a printable character character.
& is a printable character character.
```

6.1.4　大小写字母测试函数 islower 和 isupper

函数 islower 用于测试一个字符是不是小写字符，isupper 用于测试一个字符是不是大写字符。这两个函数的使用方法如下所示。

```
int islower（int c）
int islower（int c）
```

在函数 islower 中，参数 c 表示一个字符，如果这个参数是一个小写字母，函数就返回真值，否则返回值为假。函数 isupper 的用法与 islower 相似。下面是一个判断字符大小写的实例，判断一个字符数组中有哪些小写字母与大写字母。

```
#include <stdio.h>
#include <ctype.h>                          /*包含一个头文件。*/
main()
{
  char s[]="12asSHDqw^i&*";                 /*定义一个字符串数组。*/
  int i;                                    /*定义一个变量作为循环计数器。*/
  for (i=0;s[i]!=NULL;i++ )                 /*如果数组中当前的字符存在则进行循环。*/
  {
    if (islower(s[i]))                      /*如果是小写字母则输出一个结果。*/
    {
      printf("%c is a islower character.\n",s[i]); /*输出结果。*/
    }
    if (isupper(s[i]))                                /*如果是大写字母则输出结果。*/
    {
      printf("%c is a upper character.\n",s[i]);   /*输出结果。*/
    }
  }
}
```

输入下面的命令编译这段代码。

```
gcc 10.4.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序会判断字符数组中的每一个字符，然后输出结果，运行结果如下所示。

```
a is a islower character.
 s is a islower character.
S is a upper character.
H is a upper character.
D is a upper character.
q is a islower character.
```

```
w is a islower character.
```

6.1.5 数字测试函数 isxdigit

函数 isxdigit 可以测试一个字母是不是 0 到 9 之间的阿拉伯数字。这个函数的使用方法如下所示。

```
int isdigit(int c)
```

这个函数的参数 c 表示一个字符，或者表示 ASCII 码表中的一个编号。函数对这个字符进行判断，如果是一个阿拉伯数字则返回一个真值，否则返回值为假。下面是一个 isxdigit 函数的使用实例，判断一个字符数组中的字符，如果是数字则输出结果。程序的代码如下所示。

```
#include <ctype.h>                                  /*包含 ctype.h 头文件。*/
#include <stdio.h>
main()                                              /*程序的主函数。*/
{
   char s[]="123asd0ASD$%^";                        /*定义一个字符串数组。*/
   int i;                                           /*定义一个变量作为循环计数器。*/
   for(i=0;s[i]!=0;i++)                             /*for 循环，当前的有没有字符作为
循环条件。*/
   {
      if(isdigit(s[i]))                             /*判断当前字符是不是一个数字。*/
      {
         printf("%c is a number.\n",s[i]);          /*输出判断结果。*/
      }
   }
}
```

输入下面的命令编译这段代码。

```
gcc 10.5.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序可以判断出数组中的数字，结果如下所示。

```
1 is a number.
2 is a number.
3 is a number.
0 is a number.
```

6.1.6 符号测试函数 ispunct

函数 ispunct 可以测试一个字符是否为标点符号或特殊符号。这个函数的使用方法如下所示。

```
int ispunct(int c)
```

函数的参数 c 表示需要测试的字符，或表示 ASCII 码表中用来表示这个字符的编号。函数对这个字符进行测试，如果是一个标点符号或特殊字符，则返回一个真值，否则返回值为假。下面是这个函数的使用实例，对一个字符数组中的标点符号或特殊字符进行测试并输出。

```
#include <ctype.h>                                  /*包含 ctype.h 头文件。*/
#include <stdio.h>
main()                                              /*程序的主函数。*/
{
   char s[]="123a,.;sd0ASD$^";                      /*定义一个字符串数组。*/
   int i;                                           /*定义一个变量作为循环计数器。*/
   for(i=0;s[i]!=0;i++)                             /*for 循环，当前的有没有字符作为
                                                    循环条件。*/
   {
      if(ispunct(s[i]))                             /*判断当前的字符是否是一个符号。*/
      {
      printf("%c is a is a punct.\n",s[i]);         /*如果是符号则输出结果。*/
      }
   }
}
```

输入下面的命令编译这段代码。

```
gcc 10.6.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序可以判断出字符数组中的符号，结果如下所示。

```
, is a is a punct.
. is a is a punct.
; is a is a punct.
$ is a is a punct.
^ is a is a punct.
```

6.1.7 其他字符测试函数

除了上面讲解的这些字符测试函数以外，还有空格测试、可否打印字符测试、控制字符测试等函数。这些函数的使用方法和上面这些函数的使用方法是相似的。如果需要对程序中的字符进行类型测试，需要在程序的最前面包含头文件“ctype.h”。这些函数的功能与使用方法如表 6.1 所示。

表 6.1 字符测试函数

函 数 名	作 用	使用方法
isalnum	测试字符是否为英文或数字	int isalnum (int c)
isalpha	测试字符是否为英文字母	int isalpha (int c)
isascii	isascii（测试字符是否为 ASCII 码字符）	int isascii(int c)
iscntrl	测试字符是否为 ASCII 码的控制字符	int iscntrl(int c)
isdigit	测试字符是否为阿拉伯数字	int isdigit(int c)
isgraph	测试字符是否为可打印字符	int isgraph (int c)
islower	测试字符是否为小写字母	int islower (int c)
isprint	测试字符是否为可打印字符	int isprint (int c)
isspace	测试字符是否为空格	int isspace (int c)
ispunct	测试字符是否为标点符号或特殊符号	int ispunct (int c)
isupper	测试字符是否为大写字母	int isupper (int c)
isxdigit	测试字符是否为十六进制字符	int isxdigit (int c)

6.2 字符测试函数综合实例

在程序中常常需要判断输入的内容是否有效，这就需要使用字符测试函数。本节将讲解两个字符测试使用的实例。第一个实例对字符串中的各类字符进行统计，第二个实例判断输入的电话号码或姓名是否有效。

6.2.1 统计字符串中各类字符的个数

本节讲解一个字符测试实例。程序提示用户从键盘输入一串字符，这一串字符保存在一个字符数组中，然后程序统计字符中大写字母、小写字母、数字、符号的个数，然后输出结果。程序需要用循环的方法来访问字符数组中的每一个字符，然后判断当前字符的类别。编写的代码如下所示。

```
#include <ctype.h>                                  /*包含 ctype.h 头文件。*/
#include <stdio.h>
main()                                              /*程序的主函数。*/
{
   char s[100];                                     /*定义一个字符数组。*/
   int i,c_num,c_lower,c_upper,c_mark ;             /*定义几个计数变量。*/
   c_num=0;                                         /*对计数变量赋初值为 0。*/
   c_lower=0;
   c_upper=0;
```

```
    c_mark=0;
    printf("please input a string:\n");          /*输出一个提示信息。*/
    scanf("%s",s);                               /*从键盘输入一个字符串。*/
    for(i=0;s[i]!=NULL;i++)                      /*for 循环，访问字符数组中的所有字符。*/
    {
      if(isdigit(s[i]))                          /*判断当前字符是不是一个数字。*/
      {
        c_num++;                                 /*数字字符的计数变量加 1。*/
      }
      if(islower(s[i]))                          /*判断当前字符是不是一个小写字母。*/
      {
        c_lower++;                               /*小写字母的计数加 1。*/
      }
      if(isupper(s[i]))                          /*判断当前字符是不是一个大写字母。*/
      {
        c_upper++;                               /*大写字母的计数加 1。*/
      }
      if(ispunct(s[i]))                          /*判断当前字符是不是一个特殊符号。*/
      {
        c_mark++;                                /*符号的计数加 1。*/
      }
    }
    printf("upper: %d\n",c_upper);               /*输出大写字母的统计结果。*/
    printf("lower: %d\n",c_lower);               /*输出小写字母的统计结果。*/
    printf("mark: %d\n",c_mark);                 /*输出符号的统计结果。*/
    printf("number: %d\n",c_num);                /*输出数字的统计结果。*/
    printf("total: %d\n",i);                     /*输出字符的总数。*/
}
```

输入下面的命令编译这段代码。

```
gcc 10.7.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序输出下面的语句，提示输入一串字符。

```
please input a string:
```

这时输入下面的字符，需要注意的是字符之间不可以有空格。然后按“Enter”键。

```
asd^*2749#$%YIDHioprt
```

程序对输入字符进行统计，显示的结果如下所示。

```
upper: 4
lower: 8
mark: 5
```

```
number: 4
total: 21
```

6.2.2 判断电话号码与姓名是否正确

本节讲解一个字符测试编程实例。在程序中常常需要测试输入的数据是否正确，对数据的测试常常是使用字符测试功能来实现的。需要构造一个函数，用循环的方法测试字符串中的每一个字符，然后返回一个测试结果。在这个实例中，构造两个函数，分别测试输入的姓名与电话号码是否正确，然后输出结果。要测试姓名与电话号码，需要考虑以下几个方面的内容。

（1）姓名的长度，至少需要三个字符，一般不会超过 20 个字符。

（2）姓名只可以由字母构成，不能出现符号与数字。

（3）电话号码的长度，最少需要三位，最多不超过 13 位。

（4）电话号码中所有的字符必须为数字，出现其他的字符立即返回错误。

根据这些要求编写的程序代码如下所示。

```
#include<ctype.h>                /*包含 ctype.h 头文件。*/
#include<stdio.h>

int istel(char *p)               /*自定义电话号码测试函数，参数为数组的头指针。*/
{
   int i ;                       /*定义一个变量作为计数器。*/
   i=0 ;                         /*计数器赋初值为 1。*/
   while(*p!=NULL)               /*判断当前指针所指向的字符是不是 NULL。*/
   {
     if(!isdigit(*p))            /*判断指针所指向的字符是不是数字，然后用!取反。*/
     {
        return 0;                /*如果不是一个数字则返回 0。*/
     }
     i++;                        /*计数加 1。*/
     if(i>13)                    /*如果字符总数大于 13 则返回 0。*/
     {
        return 0;                /*如果字符总数大于 13 则返回 0。*/
     }
     p++;                        /*指针向后移动一个字符。*/
   }
   if(i<3)                       /*如果字符的总数小于 3 则返回 0。*/
   {
      return 0;                  /*如果字符的总数小于 3 则返回 0。*/
   }
   return 1;                     /*如果能执行到这一步，说明是一个电话号号。*/
}

int isname(char *p)              /*自定义姓名测试函数，参数为姓名数组的头指针。*/
{
   int i ;                           /*定义一个整型变量作为计数器。*/
   i=0 ;                             /*计数器赋初值为 0。*/
   while(*p!=NULL)                   /*如果当前指向的字符不为 NULL 则向下循环。*/
```

```
    {
      if(!isalpha(*p))                    /*测试当前的字符是不是字母，然后用!取反。*/
      {
         return 0;                        /*如果当前指向的字符不是字母则返回 0。*/
      }
      i++;                                /*指针指向下一个字符。*/
      if(i>20)                            /*如果字符的总数大于 20，则返回 0。*/
      {
         return 0;                        /*如果字符的总数大于 20，则返回 0。*/
      }
      p++;                                /*指针指向下一个字符。*/
    }
    if(i<3)                               /*如果字符的总数小于 3，则返回 0。*/
    {
       return 0;                          /*如果字符的总数小于 3，则返回 0。*/
    }
    return 1;                             /*如果能执行到这一步，说明测试正确，返回 1。*/
}

main()                                    /*程序的主函数。*/
{
    char name[50];                        /*定义一个字符数组存储姓名。*/
    char tel[50];                         /*定义一个字符数组，存储电话。*/
    char *q;                              /*定义一个指向字符的指针。*/

    printf("please input a telephone number:\n");    /*输出一行提示。*/
    scanf("%s",tel);                      /*从键盘输入一行文本，赋值给数组 tel。*/
    q=tel;                                /*把数组的头指针赋值给指针 q。*/
    if(istel(q))                          /*把指针作参数，调用函数判断是不是一个电话号码。*/
    {
       printf("%s is a telephoe number.\n ",tel);    /*如果为真，输出结果。*/
    }
    else
    {
       printf("%s is not  a telephoe number.\n ",tel); /*如果为假，输出错误提示。*/
    }

    printf("please input a name:\n");                 /*提示输入一个姓名。*/
    scanf("%s",name);                     /*从键盘读取一行文本，赋值给数组 name。*/
    q=name;                               /*把数组的头指针赋值给指针 q。*/
    if(isname(q))                         /*把指针作参数，调用函数判断是不是一个姓名。*/
    {
       printf("%s is a name.\n ",name);  /*如果判断结果为真值，则输出结果正确。*/
    }
    else
    {
       printf("%s is not  a name.\n ",name);     /*另一种情况，输出结果错误。*/
    }
}
```

输入下面的命令编译这段代码。

```
gcc 10.8.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序输出下面的语句，提示输入一行电话号码。

```
please input a telephone number:
```

这时输入下面的字符，然后按“Enter”键。

```
234456a
```

程序判断这行文本是不是一个电话号码，输出的结果如下所示。因为电话号码中的字符只能是数字，所以判断结果为错误。

```
234567a is not  a telephoe number.
```

用同样的方法可以对姓名进行测试。

6.3 字符串转换

所谓字符串的转换，指的是把字符串转换成整型、浮点型等数据类型，或者进行大小写转换。本节将讲解常用的字符串转换操作与相关函数的使用。

6.3.1 C 程序中的字符串

在 C 程序中，并没有字符串这一数据类型。这里所说的字符串，实际上是一个字符数组。字符串通常是保存在一个字符数组中的，可以用这个数组的名称或这个数组的头指针定义一个字符串。可以用下面的方法来定义一个字符串。

```
char a[50];
char b[];
```

用后面一种方法定义字符串，没有设置数组的长度，这种数组的长度是可以根据需要改变的。可以用下面的方法对字符串进行赋值。

```
char a[50]="asdfgh";
char b[]="ASDFG";
```

也可以直接定义一个指向字符型的指针，对这个指针赋值一个字符串，这种方法如下所示。

```
char *a="asdfg";
char *b;
*b="ASDFG";
```

但是下面这种方法先定义一个数组，然后用另一个语句赋值为字符串，这种方法是错误的。

```
char a[50];
a="asdfg";
```

如果需要在数组定义以后，在后面的程序中进行赋值，可以用键盘输入的方法，代码如下所示。

```
char a[50];
scanf("%s",a);
```

6.3.2　字符串转换成浮点型函数 atof

函数 atof 的作用是将一个字符串转换成一个浮点型变量。函数的使用方法如下所示。

```
double atof(char *nptr);
```

函数的参数 nptr 表示一个字符串，函数可以把字符串转换成一个浮点型数，然后返回。在处理字符串时，跳过前面的空格，遇上数字或正负符号才开始做转换，一直到字符串的结尾。如果字符串中有字母或其他符号，函数会去除这个字符和这个字符以后的内容。下面是这个函数的使用实例，需要注意的是字符串中字母的处理情况。

```
#include <stdlib.h>                                  /*包含 stdlib.h 头文件。*/
#include <stdio.h>
int main()
{
   char a[8]="-123.45";                              /*定义字符串然后赋值。*/
   char b[6]="23.456";
   char c[6]="-1234";
   char d[6]="123asd";                               /*字符串中含有字母。*/
   char e[6]="12a12";                                /*字符串中含有字母。*/
   float x,y,z,m,n ;                                 /*定义几个浮点型变量。*/
   x=atof(a);                                        /*用 atof 函数转换成浮点型。*/
   y=atof(b);
   z=atof(c);
   m=atof(d);                                        /*注意结果中字母的处理。*/
   n=atof(e);
   printf("string: %s  float: %f",a,x);              /*输出字符串和转换结果。*/
   printf("string: %s  float: %f",b,y);
   printf("string: %s  float: %f",c,z);
   printf("string: %s  float: %f",d,m);
   printf("string: %s  float: %f",e,n);
}
```

输入下面的命令编译这段代码。

```
gcc 10.9.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
string: -123.45  float: -123.449997
string: 23.456-1234  float: 23.455999
string: -1234  float: -1234.000000
string: 123asd12a12  float: 123.000000
string: 12a12  float: 12.000000
```

6.3.3 字符串转换成整型函数 atoi

函数 atoi 的作用是将一个字符串转换成一个整型数，使用方法如下所示。

```
int atoi(char *nptr);
```

参数 nptr 是一个字符串的指针。函数会检测这个字符串，如果字符串前面有空格则跳过空格。从第一个字符直到字符串结束，如果字符串中有其他字符，则去除这个字符与这个字符以后的内容。如果字符串中有小数点，则去除小数点以后的内容（包括小数点）。下面是使用 atoi 函数将字符串转换成整型变量的实例。

```
#include <stdlib.h>                          /*包含 stdlib.h 头文件。*/
#include <stdio.h>
int main()
{
   char a[8]="-1234";                        /*定义需要转换的字符串。*/
   char b[6]="123";
   char c[8]="-12.24";
   char d[6]="0.135";
   int x,y,z,n ;                             /*定义 4 个整型变量，存储转换后的结果。*/
   x=atoi(a);                                /*用 atoi 函数将字符串转换成整型。*/
   y=atoi(b);
   z=atoi(c);
   n=atoi(d);
   printf("string: %s  int: %d\n",a,x);          /*输出字符串和转换的结果。*/
   printf("string: %s  int: %d\n",b,y);
   printf("string: %s  int: %d\n",c,z);
   printf("string: %s  int: %d\n",d,n);
}
```

输入下面的命令编译这段代码。

```
gcc 10.10.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
string: -1234  int: -1234
string: 123  int: 123
```

```
string: -12.24  int: -12
string: 0.135  int: 0
```

6.3.4　字符串转换成长整型函数 atol

函数 atol 的作用是将一个字符串转换成长整型数，使用方法如下所示。

```
long atol(char *nptr);
```

参数 nprt 是一个字符串的指针，函数 atol 会把这个字符串转换成一个长整型数返回。字符串前面的 0 会跳过，字符串前面有字母时会中止转换返回 0。娄字后面的字符串会省略不计下面是使用函数 atol 将一个字符串转换成长整型数的实例。

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
   char a[8]="123567";                    /*定义需要转换的字符串并且赋值。*/
   char b[8]="-45645";                    /*字符串中可以有负号。*/
   char c[8]="5645.234";                  /*需要注意小数点的处理方式。*/
   char d[8]="0.135";                     /*需要注意小数点的处理方式。*/
   char e[8]="5675asd";                   /*需要注意字母的处理方式。*/
   long x,y,z,n,m ;                       /*定义几个长整型变量存储转换结果。*/
   x=atol(a);                             /*用 atol 函数将字符串转换成长整型数。*/
   y=atol(b);
   z=atol(c);
   n=atol(d);
   m=atol(e);
   printf("string: %s  long: %ld\n",a,x);   /*输出字符串和转换以后的长整型数。*/
   printf("string: %s  long: %ld\n",b,y);
   printf("string: %s  long: %ld\n",c,z);
   printf("string: %s  long: %ld\n",d,n);
   printf("string: %s  long: %ld\n",e,m);
}
```

输入下面的命令编译这段代码。

```
gcc 10.11.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
string: 123567  long: 123567
string: -45645  long: -45645
string: 5645.2340.135  long: 5645
string: 0.135  long: 0
string: 5675asd  long: 5675
```

6.3.5 将浮点型数转换成字符串函数 ecvt

函数 ecvt 可以将一个浮点型数转换成一个字符串，这个函数的使用方法如下所示。

```
char *ecvt(double number,int ndigits,int *decpt,int *sign)
```

这个函数参数与返回值的作用如下所示。

- number：是一个 double 型的浮点数，函数需要对这个浮点数进行转换。
- ndigits：在浮点数中从左向右取的位数。
- decpt：是一个整型数的指针，显示浮点数中小数点在第几位。
- sign：是一个整型数的指针。代表数值的正与负，如果为正则返回 0，否则返回 1。
- 函数的返回值是一个指向字符串的指针。下面的代码是使用函数 ecvt 将浮点型数转换成字符串的实例。

```
#include <stdlib.h>                          /*包含头文件 stdlib.h。*/
#include <stdio.h>
int main()
{
   float a,b ;                               /*定义两个浮点型数。*/
   char  *p;                                 /*定义一个指向浮点数的指针。*/
   int po,sign;                              /*定义两个整型变量用于结果的返回。*/
   a=123.45;                                 /*需要转换的浮点数。*/
   b=-2345.754;
   p=ecvt(a,4,&po,&sign);                    /*转换成字符串。*/
   printf("float:%f  string :%s  howmany:4 dot:%d sign:%d\n",a,p,po,sign);
   p=ecvt(b,4,&po,&sign);                    /*转换成字符串。*/
   printf("float:%f  string :%s  howmany:4 dot:%d sign:%d\n",b,p,po,sign);
}
```

输入下面的命令编译这段代码。

```
gcc 10.12.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。howmany 表示在浮点型数中取的位数，dot 表示浮点数中小数点出现在第几位的后面，sign 表示浮点数正负。

```
float:123.449997  string :1234  howmany:4 dot:3 sign:0
float:-2345.753906  string :2346  howmany:4 dot:4 sign:1
```

6.3.6 字母的大小写转换函数 tolower 和 toupper

函数 tolower 可以把一个大写字母转换成小写字母，函数 toupper 可以把一个小写字母转换成大写字母。这两个函数的使用方法如下所示。

```
int tolower(int c)
```

```
int toupper(int c)
```

参数 c 表示需要进行转换的字母。函数 tolower 可以把一个大写字母转换成小写字母。如果这个字母是小写字母或其他符号，则不进行转换直接返回，返回值是一个字符。下面的代码是用这两个函数进行大小写转换的实例。

```
#include <stdio.h>
#include <stdlib.h>                    /*包含 stdlib.h 头文件。*/
main()
{
   char a[20]="asFG$%";                /*定义一个字符串并且赋值。*/
   int i;                              /*定义一个整型变量作为计数器。*/
   char c;                             /*定义一个字符变量存储结果。*/
   printf("tolower:\n");               /*输出提示。*/
   for(i=0;a[i]!=NULL;i++)             /*for 循环访问数组中的每一个字符。*/
   {
      c=tolower(a[i]);                 /*将当前的字符转换成小写。*/
      printf("%c%c\n",a[i],c);         /*输出结果。*/
   }
   printf("toupper:\n");               /*输出提示。*/
   for(i=0;a[i]!=NULL;i++)             /*用 for 循环访问字符串中的每一个字符。*/
   {
      c=toupper(a[i]);                 /*将当前字符转换成大写。*/
      printf("%c%c\n",a[i],c);         /*输出结果。*/
   }
}
```

输入下面的命令编译这段代码。

```
gcc 10.13.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
tolower:
  a  a
  s  s
  F  f
  G  g
  $  $
  %  %
toupper:
  a  A
  s  S
  F  F
  G  G
  $  $
```

% %

6.3.7 其他字符串转换函数

除了上面的字符串转换函数以外，还有二进制转换、十六进制转换等函数。这些函数的使用方法和上面这些函数的使用方法是相似的。如果需要对程序中的字符串与其他类型进行转换，需要在程序的最前面包含头文件“stdlib.h”。这些函数的功能与使用方法如表 6.2 所示。

表 6.2 字符串转换函数

函　数	作　用	使用方法
atof	将字符串转换成浮点型数	double atof(const char *nptr)
atoi	将字符串转换成整型数	int atoi(const char *nptr)
atol	将字符串转换成长整型数	long atol(const char *nptr)
gcvt	将浮点型数转换为字符串	char *gcvt(double number，size_t ndigits，char *buf)
ecvt	将浮点型数转换为字符串	char *ecvt(double number，size_t ndigits，char *buf)
fcvt	将浮点型数转换为字符串	char *fcvt(double number，size_t ndigits，char *buf)
strtod	将字符串转换成浮点数	double strtod(const char *nptr,char **endptr)
strtol	将字符串转换成长整型数	long int strtol(const char *nptr,char **endptr,int base)
strtoul	将字符串转换成无符号长整型数	unsigned long int strtoul(const char *nptr,char **endptr,int base)
toascii	将整型数转换成合法的 ASCII 码字符	int toascii(int c)
tolower	将大写字母转换成小写字母	int tolower(int c)
toupper	将小写字母转换成大写字母	int toupper(int c)

6.4 字符串比较

字符串处理指的是对字符串进行比较、复制、连接、查找、替换等操作。这些字符串处理操作是通过字符串处理函数实现的，这些函数是头文件“string. h”定义和声明的。本章将讲解这些函数的使用，学习的重点是对内存和地址的理解。

本节将讲解字符串比较的各种操作。字符串比较指的是比较两个字符串的大小、是否相同等操作。常用函数有 bcmp, memcmp, strcmp, strncasecmp 等。

6.4.1　字符串比较函数 bcmp

函数 bcmp 用来比较两个字符串的前 n 个字节是否相同。如果相同则返回 0 值，否则返回非 0 值。函数的使用方法如下所示。

```
int bcmp ( const void *s1,const void * s2,int n);
```

> **注意**：返回的 0 值作为 if 判断的条件时，相当于假，非 0 值为真。

参数 s1 与 s2 表示需要进行内容比较的两个字符串，n 表示需要比较的两个字符串前 n 个字符。下面是这个函数的使用实例。程序中先定义两个字符串数组，然后比较前几个字节是否相同并且输出结果。

```
#include <stdio.h>
#include <string.h>                          /*包含 string.h 头文件。*/
main()
{
   char a[20]="asdfgQWERT!@$%";              /*定义两个字符串并且赋值。*/
   char b[20]="asdfgqwERT!@$%";
   if(bcmp(a,b,5)==0)                        /*比较这两个字符串的前 5 个字符，前 5
                                               个字符是相同的。*/
   {
      printf("same.\n");                     /*结果为真则输出相同。*/
   }
   else
   {
      printf("not same.\n");                 /*结果为假则输出不同。*/
   }
   if(bcmp(a,b,7)==0)                        /*比较两个字符串的前 7 个字符，前 7 个字
                                               符是相同的。*/
   {
      printf("same.\n");                     /*如果结果为真则输出相同。*/
   }
   else
   {
      printf("not same.\n");                 /*结果为假则输出不同。*/
   }
}
```

在这个程序中需要注意函数 bcmp 的返回值。像这种进行判断或比较的函数，返回值如果是真或假，可以直接作为判断语句的条件。如果返回值是数字，则需要根据返回值的情况，把返回值与某一数值进行比较，然后作为判断语句的条件。bcmp 判断结果为相同时，返回值是 0，如果直接作为 if 语句的条件，则表示为假，所以需要同 0 进行比较。

输入下面的命令编译这段代码。

```
gcc 10.14.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
same.
not same.
```

6.4.2 字符串大小比较函数 memcmp

函数 memcmp 用来比较两个字符串的大小是否相同，并且返回第一个不相同字符的差值。函数的使用方法如下所示。

```
int memcmp (const void *s1,const void *s2,size_t n)
```

参数 s1 与 s2 表示需要进行比较的两个字符串，n 表示需要进行比较的前 n 个字符。这里的字符串大小比较是以 ASCII 码表上的顺序来决定的。memcmp 函数首先将字符串 s1 第一个字符值减去 s2 第一个字符的值，若差为 0 则继续比较下个字符，若差值不为 0 则返回这个差值。例如字符串“sA”与“sb”，第一个字母相同，则比较下一个字母。字符“A”(65)和“b”(98)的差值是-33，函数则返回-33。如果两个字符串的前 n 个字母相同，函数返回 0，字符串 s1 小于 s2 则返回小于 0 的值。s1 大于 s2 则返回大于 0 的值。下面是使用这个函数进行字符串比较的实例。

```
#include <stdio.h>
#include <string.h>                          /*包含 string.h 头文件。*/
main()
{
   char a[20]="asdfgQWERT";                  /*定义两个字符串并且赋值。*/
   char b[20]="asdfgqwERT";
   int i,j;                                  /*定义两个整型变量存储比较结果。*/
   i=memcmp(a,b,5);                          /*比较两个字符串的前 5 个字符。*/
   j=memcmp(a,b,7);                          /*比较两个字符串的前 7 个字符。*/
   printf("memcpy %s , %s ,5 charactor ,result : %d \n",a,b,i);    /*输出结果。*/
   printf("memcpy %s , %s ,7 charactor ,result : %d \n",a,b,j);
}
```

输入下面的命令编译这段代码。

```
gcc 10.15.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。两个字符串中第一个不相同的字母是 Q 和 q，Q 比 q 的 ASCII 码值小 32。

```
memcpy asdfgQWERT , asdfgqwERT ,5 charactor ,result : 0
```

```
memcpy asdfgQWERT , asdfgqwERT ,7 charactor ,result : -32
```

6.4.3　忽略大小写比较字符串函数 strncasecmp

函数 strncasecmp 可以忽略大小写比较两个字符串，这个函数的使用方法如下所示。

```
定义函数  int strncasecmp(const char *s1,const char *s2,size_t n)
```

此函数的参数与作用和上一节中的 memcmp 函数是相同的，不同点是比较字母时忽略了字母的大小写差异。这个函数的使用实例如下所示。

```
#include <stdio.h>
#include <string.h>                              /*包含 string.h 头文件。*/
main()
{
   char a[20]="asdfgQWERT";                      /*定义两个字符串并且赋值。*/
   char b[20]="asdfgqwFRT";
   int i,j,k;                                    /*定义几个变量存储比较结果。*/
   i=strncasecmp(a,b,5);                         /*忽略大小写比较前 5 个字符。*/
   j=strncasecmp(a,b,7);                         /*忽略大小写比较前 7 个字符。*/
   k=strncasecmp(a,b,8);                         /*忽略大小写比较前 8 个字符。*/
   printf("strncasecmp %s , %s ,5 charactor ,result : %d \n",a,b,i);
                                                 /*输出结果。*/
   printf("strncasecmp %s , %s ,7 charactor ,result : %d \n",a,b,j);
   printf("strncasecmp %s , %s ,8 charactor ,result : %d \n",a,b,j);
}
```

输入下面的命令编译这段代码。

```
gcc 10.17.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。在忽略大小写的情况下，前 7 个字符是相同的，第 8 个字符 E 与 F 的差值为–1。

```
strncasecmp asdfgQWERT , asdfgqwFRT ,5 charactor ,result : 0
strncasecmp asdfgQWERT , asdfgqwFRT ,7 charactor ,result : 0
strncasecmp asdfgQWERT , asdfgqwFRT ,7 charactor ,result : -1
```

6.5 字符串复制

字符串复制指的是将一个字符串的全部或其中的若干字符，写入到另一个字符串中。本节将讲述字符串复制的相关操作，可以进行字符串复制的函数主要有 bcopy, memccpy, strcpy, strncpy 等。

6.5.1 字符串复制函数 bcopy

函数 bcopy 的作用是将一个字符串的前 n 个字符复制到另一个字符串中，使用方法如下所示。

```
void bcopy ( const void *src,void *dest ,int n);
```

函数的参数 src 表示需要复制字符的字符串，dest 表示复制到的字符串，n 表示需要在字符串中复制的字符数目。函数会改变字符串 dest 的值，没有返回值。下面是这个函数的使用实例。

```
#include <stdio.h>
#include <string.h>                          /*包含 string.h 头文件。*/
main()
{
   char a[20]="asdfgh";                      /*定义两个字符串并且赋值。*/
   char b[20]="ijklmn";
   printf("1 :%s \n %s\n",a,b);              /*输出原有的字符串。*/
   bcopy(a,b,4);                             /*第一个字符串的前 4 个字符复制到
                                             第二个字符串中。*/
   printf("2 :%s \n %s\n",a,b);              /*输出复制以后的结果。*/
}
```

输入下面的命令编译这段代码。

```
gcc 10.17.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。函数 bcopy 会把第一个字符串中的前 4 个字母复制到第二个字符串的前 4 个字母。第二个字符串中的前 4 个字母将会被替换。

```
1 :asdfgh
ijklmn
2 :asdfgh
asdfmn
```

6.5.2　字符串复制函数 memccpy

函数 memccpy 可以将一个字符串中的前 n 个字节复制到另一个字符串中。与函数 bcopy 不同的是 memccpy 可以检查字符串里是不是有某一个字符。该函数的使用方法如下所示。

```
void * memccpy(void *dest, const void * src, int c,size_t n)
```

函数参数中的 src 和 dest 分别表示源字符串与目标字符串，与函数 bcopy 的参数相反。dest 是复制的目录字符串的指针，src 需要复制的字符串指针。c 表示需要在字符串 dest 中查找赋值为 c 的字符。如果查找到这个字符，则返回下一个字符的指针。n 表示需要在字符串 src 中复制的字符的个数。下面的程序是使用这个函数进行字符串复制的实例。

```
#include <stdio.h>
#include <string.h>                          /*包含 string.h 头文件。*/
main()
{
   char a[20]="asdfgh";                      /*定义两个字符串并且赋值。*/
   char b[20]="ijklmn";
   char *s;                                  /*定义一个指向字符的指针。*/
   printf("%s\n%s\n",a,b);                   /*输出两个源字符串。*/
   s=memccpy(a,b,'k',3);                     /*进行字符串替换，查找赋值为 k 的字符。
返回赋值为 k 的下一个指针。*/
   printf("%c\n",*s);                        /*输出返回指针指向的字符。*/
   printf("%s\n%s\n",a,b);                   /*输出替换以后的字符串和源字符串。*/
}
```

输入下面的命令编译这段代码。

```
gcc 10.18.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
asdfgh
ijklmn
f
ijkfgh
ijklmn
```

6.5.3　字符串复制函数 strcpy

函数 strcpy 可以将一个字符串复制到另一个字符串，函数的使用方法如下所示。

```
char *strcpy(char *dest,const char *src)
```

在参数列表中，dest 是复制字符串的目标指针，src 是源字符串指针。函数将返回字符串

dest 的指针。使用这个函数时，需要注意字符串 dest 需要有足够的空间来存储字符串 src，否则将会发生溢出错误。使用这个函数进行字符串复制的实例如下所示。

```
#include <stdio.h>
#include <string.h>                              /*包含 string.h 头文件。*/
main()
{
   char a[20]="asdfgh";                          /*定义一个字符串进行赋值。*/
   char b[20];                                   /*定义一个字符串。*/
   char *p;                                      /*定义一个指向字符的指针。*/
   p=strcpy(b,a);                                /*将字符串 a 复制到字符串 b 中。*/
   printf("%s\n",a);                             /*输出源字符串 a。*/
   printf("%s\n",b);                             /*输出复制后的字符串 b。*/
   printf("%s\n",p);                             /*输出指针 p 指向的内容。*/
}
```

输入下面的命令编译这段代码。

```
gcc 10.19.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
asdfgh
asdfgh
asdfgh
```

6.5.4 字符串复制函数 strncpy

函数 strncpy 可以将一个字符串中的若干个字符复制到另一个字符串中，该函数的使用方法如下所示。

```
char * strncpy(char *dest,const char *src,size_t n);
```

在参数列表中，dest 是目标字符串的指针，src 是需要复制的字符串的指针，n 是需要在字符串 src 中复制的字符个数。返回值是字符串 dest 的头指针。下面是使用 strncpy 进行字符串复制的实例。

```
#include <stdio.h>
#include <string.h>                  /*包含 string.h 头文件。*/
main()
{
   char a[20]="asdfghijk";           /*定义一个字符串并且赋值。*/
   char b[20]="";                    /*定义一个空字符串。*/
   char *p;                          /*定义一个指向字符的指针。*/
   p=strncpy(b,a,5);                 /*将字符串 a 的内容取 5 个字符复制到字符串 b 中。*/
   printf("%s\n",a);                 /*输出字符串 a。*/
```

```
    printf("%s\n",b);                    /*输出字符串 b。*/
    printf("%s\n",p);                    /*输出指针 p 指向的字符串。*/
}
```

输入下面的命令编译这段代码。

```
gcc 10.20.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
asdfghijk
asdfg
asdfg
```

6.6　字符串的清理与填充

字符串的清理指的是删除一个字符串的部分内容，将所有要清理的字节写为 NULL。字符串填充指的是把一个字符串的部分字节写为某一个字符。本节将讲述字符串的这两种操作。

6.6.1　字符串清理函数 bzero

函数 bzero 的主要作用是将字符串中的部分字节写为 0，即写入 NULL 值。函数的使用方法如下所示。

```
void bzero(void *s,int n)
```

参数列表中，s 是一个字符串的头指针，n 表示需要在字符串中清理前 n 个字符。函数没有返回址，操作时已经改变了字符串 s 的值。下面是 bzero 的使用实例。

```
#include <stdio.h>
#include <string.h>                      /*包含 string.h 头文件。*/
main()
{
   int i;                                /*定义一个循环变量 i。*/
   char a[20]="asdfghijk";               /*定义一个字符串，赋初值。*/
   bzero(a,3);                           /*用 bzero 函数将字符串 a 的前三个字符清除。*/
   for(i=0;i<6;i++)                      /*用 for 循环执行 6 次输出。*/
   {
      printf("%c\n",a[i]);               /*输出数组中当前的字符。*/
   }
}
```

输入下面的命令编译这段代码。

```
gcc 10.21.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。前三个字符已经被清空，没有输出内容，后三次输出分别为 f、g、h 三个字符。

```
f
g
h
```

6.6.2 字符串填充函数 memset

函数 memset 的作用是将一个字符的前 n 个字符填充为某一个字符。该函数的使用方法如下所示。

```
void * memset (void *s ,int c, size_t n)
```

参数列表中，s 指的是需要处理的字符串头指针，c 表示需要写入的字符，n 表示从字符串第一个字符开始需要填充的字符数。函数没有返回值。下面是这个函数的使用实例。

```
#include <stdio.h>
#include <string.h>                          /*包含 string.h 头文件。*/
main()
{
   char a[20]="asdfghijk";                   /*定义一个字符串，赋初值。*/
   printf("%s\n",a);                         /*输出这个字符串。*/
   memset(a,'W',3);                          /*将字符串中的前三个字符替换成 W。*/
   printf("%s\n",a);                         /*输出替换以后的字符。*/
   memset(a,'x',6);                          /*将字符串中的前 6 个字符替换成 x。*/
   printf("%s\n",a);                         /*输出替换以后的字符。*/
}
```

输入下面的命令编译这段代码。

```
gcc 10.22.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
asdfghijk
WWWfghijk
xxxxxxijk
```

6.7　字符串查找

字符串查找指的是在一个字符串中检索某一个字符或字符串。如果发现这个字符或字符串则返回这一个字符或字符串的指针。本节将讲解与字符串查找相关的 index、rindex、strchr、strrchr 等函数。

6.7.1　字符查找函数 index 与 rindex

函数 index 用来在字符串中找出需要查找字符第一次的出现位置，然后将该字符地址返回。rindex 的使用方法与 index 相似，但作用是找出字符串中最后一次某字符的出现位置。这两个函数的使用方法如下所示。

```
char *index( const char *s, int c);
char *rindex( const char *s, int c);
```

参数列表中，s 是一个字符串的头指针，c 表示需要在字符串 s 里面查找的字符。返回值是一个指针，指向找到的这个字符的地址。下面是这两个函数的使用实例。

```
#include <stdio.h>
#include <string.h>                  /*包含 string.h 头文件。*/
main()
{
   char *p,*q;                       /*定义两个指向字符的指针。*/
   char a[20]="asdfghasdf";          /*定义一个字符串并且赋值。*/
   p=index(a,'g');                   /*在字符串中查找字符 g 第一次出现的位置。*/
   q=index(a,'d');                   /*在字符串中查找字符 d 最后一次出现的位置。*/
   printf("%c\n",*p);                /*输出指针 p 指向的内容。*/
   printf("%c\n",*q);                /*输出指针 q 指向的内容。*/
}
```

输入下面的命令编译这段代码。

```
gcc 10.23.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
g
d
```

6.7.2　字符查找函数 memchr

函数 memchr 的作用是在一个字符串的前 n 个字符中查找某一个字符，返回这个字符的指针地址。函数的使用方法如下所示。

```
void * memchr(const void *s,int c,size_t n)
```

参数列表中，s 表示需要查找的字符串，c 表示需要查找的字符，n 表示在字符串 s 中的前 n 个字符里查找。如果找到了这个字符，则会返回这个字符的指针，如果没有这个字符，则返回 0。下面是函数 memchr 的使用实例。

```
#include <stdio.h>
#include <string.h>                    /*包含 string.h 头文件。*/
main()
{
   char *p;                            /*定义一个指针字符的指针。*/
   char a[20]="asdfghasdf";            /*定义一个字符串并且赋值。*/
   p=memchr(a,'f',10);                 /*在字符串 a 的前 10 个字符中查找字符 f。*/
   printf("%c\n",*p);                  /*输出指针 p 指向的字符。*/
}
```

输入下面的命令编译这段代码。

```
gcc 10.24.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。找到字符 f 的地址，在这个地址上的字符一定是 f。

```
f
```

6.7.3 字符查找函数 strchr 与 strrchr

函数 strchr 的作用是在一个字符串中查找某一个字符第一次出现的位置。函数 strrchr 的作用是在一个字符串中查找某一个字符最后一次出现的位置。这两个函数的使用方法如下所示。

```
char * strchr (const char *s,int c)
char * strrchr (const char *s,int c)
```

参数列表中，s 是需要查找的字符串头指针，c 表示需要查找的字符。如果查找到这个字符，将返回这个字符指针，如果没有找到这个字符则返回 0。下面是这两个函数的使用实例。

```
#include <stdio.h>
#include <string.h>                    /*包含 string.h 头文件。*/
main()
{
   char *p,*q;                         /*定义两个指向字符的指针。*/
   char a[20]="asdfghasdf";            /*定义一个字符串并且赋初值。*/
   p=strchr(a,'s');                    /*在字符串 a 中查找 s 第一次出现的位置。*/
   q=strrchr(a,'d');                   /*在字符串 a 中查找 d 最后一次出现的位置。*/
   printf("%c\n",*p);                  /*输出指针 p 指向的字符。*/
   printf("%c\n",*q);                  /*输出指针 q 指向的字符。*/
}
```

输入下面的命令编译这段代码。

```
gcc 10.25.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
s
d
```

6.8　字符串的连接与分割

字符串的连接指的是将一个字符串添加到另一个字符串的后面。字符串的分割指的是把一个字符串按照一定的标记切分成多个字符串。本节将讲解字符串的这两种操作。

6.8.1　字符串连接函数 strcat

函数 strcat 的作用是将一个字符串连接到另一个字符串后面。下面是这个函数的使用方法。

```
char *strcat (char *dest,const char *src)
```

参数列表中，dest 和 src 是两个字符串的头指针。函数会把字符串 src 的内容添加到字符串 dest 上面，并返回指针 dest。需要注意的是，字符串 dest 需要有足够的空间来存储字符串 src 的数据。下面是这个函数的使用实例。

```
#include <stdio.h>
#include <string.h>                    /*包含 string.h 头文件。*/
main()
{
   char *p;                            /*定义一个指向字符的指针。*/
   char a[20]="asdfg";                 /*定义一个字符串并且赋初值。*/
   char b[20]="hjklm" ;
   printf("%s\n",a);                   /*输出原来的字符串。*/
   printf("%s\n",b);
   p=strcat(a,b);                      /*将字符串 b 连接到字符串 a 后面。*/
   printf("%s\n",a);                   /*输出连接以后的字符串。*/
   printf("%s\n",b);
   printf("%s\n",p);                   /*输出指针 p 指向的字符串。*/
}
```

输入下面的命令编译这段代码。

```
gcc 10.26.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
asdfg
hjklm
asdfghjklm
hjklm
asdfghjklm
```

6.8.2 字符串分割函数 strtok

函数 strtok 的作用是将字符串分割成多个字符串。函数的使用方法如下所示。

```
char * strtok(char *s,const char *delim);
```

参数列表中，s 表示需要分割的字符串，delim 表示分割标记的字符串。在第一次调用时，strtok 在参数 s 字符串中发现参数 delim 的分割字符时，将该字符改为 NULL 字符，然后返回更改以后的字符串。再次调用时，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。下面是这个函数的使用实例，在一个字符串中以 a 为标记，将字符串分割为多个字符串然后输出。

```
#include <stdio.h>
#include <string.h>                          /*包含 string.h 头文件。*/
main()
{
   char *p;                                  /*定义一个指向字符的指针。*/
   char a[20]="qweaQWEa^*&aIOP";             /*定义一个字符串并且赋初值。*/
   char s[]="a";                             /*定义分割字符串的标记。*/
   printf("%s\n",a);                         /*输出源字符串。*/
   p=strtok(a,s);                            /*调用 strtok 函数分割字符串。*/
   printf("%s\n",p);                         /*输出指针 p 指向的字符串。*/
   while(p=strtok(NULL,s))                   /*用 while 循环来循环输出字符串。*/
   {
      printf("%s\n",p);                      /*每次循环输出字符串的结果。*/
   }
}
```

输入下面的命令编译这段代码。

```
gcc 10.27.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。函数 strtok 将一个长字符串分割为多个短字符串分多次输出。

```
qweaQWEa^*&aIOP
qwe
QWE
^*&
IOP
```

6.9　其他字符串函数

C 程序有丰富的字符串处理函数。除了前面几节讲述的字符串操作函数以外，本节还将讲解 strlen, strspn 等函数，这两个函数的作用是对字符串的长度和字符进行统计。

6.9.1　字符串长度函数 strlen

函数 strlen 的作用是返回字符串的长度，也就是字符串里一共有多少个字符。函数的使用方法如下所示。

```
size_t strlen (const char *s);
```

参数列表中，s 表示一个字符串，函数返回一个表示这个字符串长度的整型数。下面是这个函数的使用实例。

```
#include <stdio.h>
#include <string.h>                    /*包含 string.h 头文件。*/
main()
{
   char a[20]="qweaQWEa^*&aIOP";       /*定义两个字符串分别赋值。*/
   char b[]="qwer";
   int i,j;                            /*定义两个整型变量分别存储结果。*/
   i=strlen(a);                        /*用 strlen 函数求出字符串 a 的长度。*/
   j=strlen(b);                        /*用 strlen 函数求出字符串 b 的长度。*/
   printf("%d\n",i);                   /*输出结果。*/
   printf("%d\n",j);
}
```

输入下面的命令编译这段代码。

```
gcc 10.28.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。函数 strlen 分别求出了这两个字符串的长度。

```
15
4
```

6.9.2 允许出现字符查找函数 strspn

函数 strspn 的作用是返回一个字符串中首次不包含在指定字符串内容中的字符的位置。这个函数的使用方法如下所示。

```
size_t strspn (const char *s,const char * accept);
```

参数列表中，s 表示需要查找的字符串，accept 表示一个字符串，包含了字符串 s 中允许出现的字符。函数 strspn 在字符串 s 中查找第一次没有在 accept 字符串中出现的字符。查找到这个字符后，返回表示这个字符位置的数字。也可以理解为，返回值为 n，则字符串中前面 n 个字符都可以在字符串 accept 里面找到。下面是这个函数的使用实例。

```
#include <stdio.h>
#include <string.h>          /*包含 string.h 头文件。*/
main()
{
   char a[]="C is a useful language for Linux OS.I'll study it hard.";
                             /*定义一个字符串。*/
   char *s="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ ";
                             /*定义允许出现字符的字符串，最后一个字符是一个空格。*/
   int i;                    /*定义一个整型数存放结果。*/
   i=strspn(a,s);            /*用 strspn 函数进行查找。*/
   printf("%d",i);           /*输出结果。*/
}
```

输入下面的命令编译这段代码。

```
gcc 10.29.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。在字符串 a 中出现而没有在字符串 s 中出现的第一个字符是“.”，位置是第 35 个字符。

```
35
```

6.9.3 不允许出现字符查找函数 strcspn

函数 strcspn 的作用，与上一节中的 strspn 的作用相似，不同点是查找出一个字符串中第一次不允许出现的某个字符的位置。这个函数的使用方法如下所示。

```
size_t strcspn ( const char *s,const char * reject)
```

参数列表中，s 表示需要查找的字符串，reject 表示一个字符串。在字符串 s 中不得出现字符串 reject 里面的任意一个字符，如果在字符串 s 中查找到字符串 reject 里面的任意一个字符，将返回这个字符的位置。下面的程序是这个函数的使用实例。

```
    #include <stdio.h>
    #include <string.h>                                  /*包含 string.h 头文件。*/
    main()
    {
       char a[]="C is a useful language for Linux OS.I started to learn it 3 years
ago .I'll study it hard.";
                                                         /*定义一个字符串并且赋值。*/
       printf("%d\n",strcspn(a," "));                    /*查找第一次出现空格的位置。*/
       printf("%d\n",strcspn(a,"'."));                   /*查找第一次出现句点的位置。*/
       printf("%d\n",strcspn(a,"aeiou"));                /*查找第一次出现 aeiou 中某一个字
母的位置。*/
       printf("%d\n",strcspn(a,"AEIOU"));                /*查找第一次出现 AEIOU 中某一个字
母的位置。*/
       printf("%d\n",strcspn(a,"1234567890"));           /*查找第一次出现数字的位置。*/
    }
```

输入下面的命令编译这段代码。

```
gcc 10.30.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
1
35
2
33
58
```

6.10　小结

本章讲述了 C 程序中的字符测试、字符串转换、字符串操作三方面的知识。在 C 程序中常常需要使用这些知识对字符和字符串进行测试和操作。这些知识的使用方法和要点如下所示。

- 字符测试：如果需要对某一个字符进行类型测试，需要用到字符测试函数。字符测试的函数主要有 isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint 等。这些函数的作用都是对一个字符进行类型测试，返回结果都是表示真假的 1 或 0。使用这些函数时需要注意这些函数的作用。
- 字符串转换：本章所讲述的字符串转换，主要是字符串到整型、浮点型、长整型的转换，整型、浮点型、长整型到字符串的转换。常用的函数有 atof, atoi, atol, gcvt, strtod, strtol, strtoul, toascii, tolower 等。在使用这些函数时，需要注意函数的参数和函数对各种参数的处理机制。

- 字符串处理：本章讲述了字符串连接、分割、复制、查找、比较等字符串操作。主要的函数有 bcopy, bzero, index, strcat, strchr, strncat, strncpy, strrchr, strtok 等。这些函数的参数比较复杂，使用时需要注意各个参数的含义和作用。

第 7 章　结构体

7.1　结构体的操作

结构体是一种自定义数据类型，有定义、新建、访问等基本操作。本节将讲解结构体的这些操作。在进行本节的学习时需要正确理解结构体的作用和特点。

7.1.1　结构体的理解

可以用一个实例来理解什么是结构体。假设用下面的变量来描述一个学生的信息。

```
char name[15];
int age;
char address[50];
int height;
```

如果有多个学生，这些数据就无法体现出各自的联系和归属关系。因为每个学生都有相同类别的信息，可以把这些信息封装成一个整体，作为一个新的数据类型。可以用下面的新数据类型来描述一个学生。

```
学生数据类型{
char name[15];
int age;
char address[50];
int height;
}
```

当用这个类型定义一个学生后，这个学生就有了这个类型里面所有的数据。学生的各个变量与这个学生产生了一对一的联系。从这方面来说，结构体的作用是把若干个简单的数据类型封装成一个整体，作为一个新的数据类型。这个数据类型可以和普通数据类型一样参与程序的运算。

7.1.2　结构体的定义

所谓结构体的定义，是指列出一个结构体中所包括的多个变量和数据类型，建立一个新的数据类型。结构体的定义方法如下所示。

```
struct 结构名
{
  类型 1  变量名 1;
  类型 2  变量名 2;
  ...
```

```
};
```

结构名是新定义的数据类型的名称，里面变量名被称为这个结构体的成员。例如对上一结中的学生定义一个结构体，代码如下所示。

```
struct student                          /*定义一个结构体 student。*/
{
   char name[15];                       /*姓名。*/
   int age;                             /*年龄。*/
   char address[50];                    /*住址。*/
   int height;                          /*身高。*/
}
```

如果把这个结构体作为一个新的数据类型，可以用这个数据类型声明新的变量。用结构体 student 声明几个学生，代码如下所示。

```
struct  student A;
struct  student Jim,Lily;
```

也可以用下面这种方法直接在定义结构体时声明两个结构变量。

```
struct student                          /*定义一个结构体 student。*/
{
   char name[15];                       /*姓名。*/
   int age;                             /*年龄。*/
   char address[50];                    /*住址。*/
   int height;                          /*身高。*/
}Jim,Lily;
```

结构体可以作为一个普通的数据类型，于是可以用结构体声明一个数组。例如下面的代码就是用上面定义的结构体 student 声明 10 个学生的数组。

```
student stu[10];
```

在一个结构体中，可以引用另外一个结构体构成更复杂的数据类型。假设需要定义一个班级的结构体，每个班有一个班主任和 50 个学生。班主任需要保存姓名，50 个学生可以用 student 结构体定义一个结构体数组，代码如下所示。

```
struct class
{
  char teacher[20];                     /*定义一个字符串，表示班主任的姓名。*/
  student stu[50];                      /*这是一个结构体数组，表示 50 个学生。*/
}
```

可以用这个班级结构体来定义一个班，代码如下所示。

```
class cla1;
```

这样，班级就是一个数据类型。定义了一个 cla1 班级以后，可以访问这个班下面的老师、每一个学生的变量。例如下面的代码使用这种方法定义学生与班级的结构体。在程序中需要注意结构体数组的使用。

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

```
struct student                                          /*定义一个结构体。*/
{
   char name[15];                                       /*姓名。*/
   int age;                                             /*年龄。*/
   char address[50];                                    /*住址。*/
   int height;                                          /*身高。 */
};

struct class                                            /*定义一个班结构体。*/
{
   char teacher[20];                                    /*班主任姓名。*/
   struct student stu[50];                              /*用结构体定义 50 个学生。*/
};

main()                                                  /*程序的主函数。*/
{
    struct class A;                                     /*用学生结构体声明一个班。*/
}
```

7.1.3　结构体的访问

结构体是自定义的数据类型，因此结构体变量也可以和其他类型的变量一样赋值、运算。不同的是结构体本身并不能作为一个变量进行访问，结构体变量以成员作为基本变量。结构体成员的表示方式如下所示。

```
结构变量.成员名
```

如果将“结构变量.成员名”看成一个整体，则这个整体的数据类型与结构中该结构成员的数据类型相同，可以同前面所讲的基本变量一样的访问。例如下面的代码是结构体的使用实例，先定义一个学生结构体 student，然后用这个结构体定义两个学生变量 Jim 和 Lily，并且赋值输出。

```
#include <stdio.h>

struct student                                          /*定义一个学生结构体。*/
{
   char name[20];                                       /*姓名。*/
   int age;                                             /*年龄。*/
   char sex ;                                           /*性别。*/
   int height;                                          /*身高。*/
}

main()
{
    struct student Jim,Lily;                /*用结构体声明两个学生。*/
    Jim.age=13;                             /*对 Jim 的年龄赋值。*/
    strcpy(Jim.name,"Jim Green");           /*用 strcpy 函数对 Jim 的姓名进行赋值。*/
    Jim.sex='m';                            /*对 Jim 的性别进行赋值。*/
    Jim.height=168;                         /*对 Jim 的身高进行赋值。*/
```

```
    Lily.age=13;
    strcpy(Lily.name,"Lily Kate");
    Lily.sex='f';
    Lily.height=174;

    printf("Jim\n");                              /*输出姓名。*/
    printf(" Name:  %s\n",Jim.name);              /*输出姓名。*/
    printf(" Age:   %d\n",Jim.age);               /*输出年龄。*/
    printf(" Sex:   %c:\n",Jim.sex);              /*输出性别。*/
    printf(" Height:%d:\n",Jim.height );          /*输出身高。*/

    printf("Lily\n");                             /*输出 Lily 的信息。*/
    printf(" Name:  %s\n",Lily.name);
    printf(" Age:   %d\n",Lily.age);
    printf(" Sex:   %c\n",Lily.sex);
    printf(" Height:%d\n",Lily.height );
}
```

输入下面的命令，编译这个程序。

```
gcc 11.1.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
Jim
 Name:  Jim Green
 Age:   13
 Sex:   m:
 Height:168:
Lily
 Name:  Lily Kate
 Age:   13
 Sex:   f
 Height:174
```

7.1.4 结构体数组

所谓结构体数组，指的是把结构体作为一种基本的数据类型，用这种数据类型定义一个数组。结构体数组的定义和基本数据类型的数组定义方法是相同的。在定义结构体数组之前需要定义一个结构体。结构体数组在访问时需要和数组一样用下标表明使用哪一个变量，然后再访问这个结构变量的成员。下面的代码是一个结构体数组的定义与使用的实例。

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

```
struct student                                  /*定义一个结构体。*/
{
   char name[15];                               /*姓名。*/
   int age;                                     /*年龄。*/
   char address[50];                            /*住址。*/
   int height;                                  /*身高。 */
};

main()                                          /*主程序。*/
{
  struct student stu[3];                        /*定义一个结构体变量表示三个学生。*/
  stu[1].age=15;                                /*第二个学生的年龄为 15。*/
  stu[2].height=167;                            /*第三个学生的身高为 167。*/
  stu[0].sex='m';                               /*第一个学生的性别为 m。*/
}
```

7.1.5　结构体使用实例

在程序中使用结构体以后，可以使程序复杂的数据关系构成逻辑联系，使用这种联系可以方便地访问一组数据。本小节将讲解一个结构体使用实例。先定义一个表示学生信息的结构体，然后用这个结构体定义数组存储若干个学生的信息。从键盘输入学生信息，然后用循环的方法输出学生的信息。要实现这个程序需要在程序中完成下面这些操作。

（1）定义一个用来表示学生的结构体。

（2）定义一个结构体数组，用来存放若干学生的信息。

（3）定义一个字符变量。用户输入“y”或“Y”时，继续输入学生数据。用户输入“n”或“N”时，结束输入信息。

（4）定义一个整型变量，用来存储当前学生的序号。

（5）用 while 语句完成输入的循环。

（6）用 for 语句循环输出学生的信息。

根据上面的思路编写的程序代码如下所示。

```
#include <stdio.h>
#include <stdlib.h>              /*因为要用 strcpy 函数，所以要包含头文件 stdlib.h。*/

struct student                   /*定义一个结构体。*/
{
   char name[20];                /*姓名。*/
   int age;                      /*年龄。*/
   int sex;                      /*性别。*/
   int height;                                  /*身高。 */
};

main()
{
  struct student stu[50];                       /*定义一个结构体，表示有 50 个学生。*/
  int i=0;                                      /*输入学生的计数。*/
  int j ;                                       /*输出的计数。*/
  char s;                                       /*用户输入数据。*/
```

```
    char name[20];                                  /*姓名。*/
    while(1)                                        /*进入一个 while 循环。*/
    {
      printf("input a student?\n?Y/N:");            /*提示用户输入选择。*/
      scanf("%c",&s);                               /*用户输入一个字符。*/

      if(s=='N'||s=='n')                            /*输入 N 或 n 则中断循环。*/
      {
        break;                                      /*中止当前循环。*/
      }

      if(s!='n'&&s!='N'&&s!='y'&&s!='Y')            /*如果不是 N、n、Y、y 这几个字母则进入
下次循环。*/
      {
        printf("error.\n");                         /*输出提示。*/
        continue;                                   /*进入下一次循环。*/
      }

      printf("please input age:\n");                /*提示输入年龄。*/
      scanf("%d",&stu[i].age);                      /*输入年龄。*/
      printf("please input height:\n");             /*提示身高。*/
      scanf("%d",&stu[i].height);                   /*输入身高。*/
      printf("please input sex:\n");                /*提示性别。*/
      scanf("%d",&stu[i].sex);                      /*输入性别。*/
      printf("please input name:\n");               /*提示姓名。*/
      scanf("%s",name);                             /*输入姓名到 name 字符串。*/
      strcpy(stu[i].name,name) ;                    /*将字符串复制到学生姓名上。*/
      i++;                                          /*计数器自加。*/
    }

    for(j=0;j<i;j++)                                /*for 循环输出。*/
    {
     printf("Student[%d]\n",j);                     /*提示第几个学生。*/
     printf("  Name  :%s  \n",stu[j].name);         /*输出姓名。*/
     printf("  Age   :%d  \n",stu[j].age);          /*输出年龄。*/
     printf("  Sex   :%d  \n",stu[j].sex);          /*输出性别。*/
     printf("  Height:%d  \n",stu[j].height);/*输出身高。*/
    }
  }
```

输入下面的命令，编译这个程序。

```
gcc 11.4.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序输出下面的提示，询问是不是输入一个学生。

```
input a student?
?Y/N:
```

这时输入一个“Y”，然后在各项提示的后面输入学生的信息。输入完一个学生的信息后，输入“Y”继续输入下一个学生的信息，输入“N”则停止输入信息，然后循环输出已经输入的内容。显示的结果如下所示。

```
Student[0]
  Name  :Jim
  Age   :16
  Sex   :1
  Height:168
Student[1]
  Name  :Lily
  Age   :18
  Sex   :0
  Height:175
Student[2]
  Name  :Bush
  Age   :15
  Sex   :1
  Height:178
Student[3]
  Name  :Lucy
  Age   :19
  Sex   :0
  Height:175
```

7.2　结构体与指针

结构体是一种特殊的数据类型，那么就可以定义一个指向结构体的指针。这种指向结构体的指针就是结构体指针。结构体指针和普通指针一样，可以访问结构体的成员，可以作为函数的参数和返回值。本节将讲解结构体指针的操作。

7.2.1　结构体指针的定义

所谓结构体指针的定义，指的是定义一个指针，这个指针指向一个结构体类型的变量，用这个指针可以访问一个结构中的成员。结构体指针的定义和普通指针的定义方法是相同的，在变量名前面加一个星号即可。下面的代码是利用前面的学生结构体定义一个结构体指针。

```
struct student                                  /*定义一个结构体。*/
{
   char name[20];                               /*姓名。*/
   int age;                                     /*年龄。*/
   int sex;                                     /*性别。*/
   int height;                                  /*身高。 */
};
```

```
struct student *stu;                                    /*定义一个结构体指针 stu。*/
```

7.2.2 结构体指针的访问

所谓结构体指针的访问，是指用结构体指针访问这个指针所指向的变量和成员，对这些变量和成员进行赋值或读取的操作。有两种方法访问结构体指针，可以用下面的方法来理解。

（1）假设使用上一节定义的结构体定义一个结构体指针，代码如下所示。

```
struct student *stu;
```

（2）用星号（*）可以访问这一个结构体，代码如下所示。

```
*p;
```

（3）既然*p 是一个结构体，则可以使用这个结构体直接访问成员变量，代码如下所示。

```
(*p).age = 15;
(*p).sex = 0 ;
```

（4）(*p).age 这个成员变量可以写成另一种形式，用一个箭头表示指针所指向的变量，代码如下所示。这些代码和上面的代码是等价的。

```
p->age=15;
p->sex=0;
```

根据所述的结构体变量的访问方法可以在程序中用结构体指针来访问一个结构体。下面是指针访问结构体的实例。

```
#include <stdio.h>
#include <stdlib.h>          /*因为要用 strcpy 函数，所以要包含头文件 stdlib.h。*/

struct student               /*定义一个结构体。*/
{
   char name[20];            /*姓名。*/
   int age;                  /*年龄。*/
   int sex;                  /*性别。*/
   int height;               /*身高。 */
};

main()
{
   struct student stu;       /*定义一个结构体变量。*/
   struct student *p;                           /*定义一个指向结构体变量的指针。*/

   p=&stu;                                      /*结构体取地址赋值给结构体指针。*/
   stu.age=15;                                  /*访问结构体的成员，对年龄赋值。*/
   strcpy(stu.name,"Jim");                      /*用 strcpy 函数对姓名赋值。*/

   (*p).sex =1;                                 /*用结构体指针访问性别成员变量。*/
   (*p).height=168;                             /*用结构体指针访问身高成员变量。*/

   printf("Name  : %s \n",stu.name);            /*输出这些成员变量的值。*/
   printf("Age   : %d \n",stu.age);
```

```
    printf("Sex   : %d \n",stu.sex);
    printf("Height: %d \n\n",stu.height);

    p->height=169;                              /*用箭头方法访问成员变量。*/
    p->sex=0;

    printf("Name  : %s \n",stu.name);           /*输出结构体的成员。*/
    printf("Age   : %d \n",stu.age);
    printf("Sex   : %d \n",stu.sex);
    printf("Height: %d \n\n",stu.height);

    printf("Name  : %s \n",(*p).name);          /*用指针访问结构体变量。*/
    printf("Age   : %d \n",(*p).age);
    printf("Sex   : %d \n",p->sex);             /*用箭头符号代替结构体指针访问结构体
                                                成员。*/
    printf("Height: %d \n\n",p->height);
}
```

输入下面的命令，编译这个程序。

```
gcc 11.5.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。从结果可知，结构体指针对成员的访问与结构体直接访问成员的作用是相同的，箭头符号和指针星号对结构体成员的访问作用是相同的。

```
Name  : Jim
Age   : 15
Sex   : 1
Height: 168

Name  : Jim
Age   : 15
Sex   : 0
Height: 169

Name  : Jim
Age   : 15
Sex   : 0
Height: 169
```

7.2.3　结构体作为函数的参数

结构体作为一个特殊的数据类型是可以作为函数的参数的。使用结构体变量作为函数参数时，需要在参数列表中声明这个函数的参数是一个结构体。在函数中可以直接调用这个结

构体和成员变量。调用函数时需要将结构体变量作为函数的参数。下面的程序是结构体作为函数参数的实例，实现一个学生信息的输出功能。

```
    #include <stdio.h>
    #include <stdlib.h>                          /*因为要用 strcpy 函数，所以要包含头文
件 stdlib.h。*/

    struct student                               /*定义一个结构体。*/
    {
       char name[20];                            /*姓名。*/
       int age;                                  /*年龄。*/
       int sex;                                  /*性别。*/
       int height;                               /*身高。 */
    };

    void showstu(struct student s)               /*定义一个自定义函数，结构体变量作为参
                                                 数。*/
    {
       printf("A student:\n");                   /*输出信息。*/
       printf("  Name  : %s \n",s.name);         /*访问和输出参数中结构体的变量成员。*/
       printf("  Age   : %d \n",s.age);          /*访问和输出参数中结构体的变量成员年
                                                 龄。*/
       printf("  Sex   : %d \n",s.sex);          /*访问和输出参数中结构体的变量成员性
                                                 别。*/
       printf("  Height: %d \n\n",s.height);     /*访问和输出参数中结构体的变量成员身
                                                 高。*/
    }

    void main()                                  /*主函数。*/
    {
       struct student stu1,stu2;                 /*定义两个结构体变量，表示两个学生。*/

       stu1.age=17;                              /*对第一个结构体变量的成员进行赋值。*/
       stu1.sex=1;
       stu1.height=176;
       strcpy(stu1.name,"Jim");                  /*strcpy 函数对字符串进行赋值。*/
       stu2.age=23;                              /*对第二个结构体变量的成员进行赋值。*/
       stu2.sex=0;
       stu2.height=171;
       strcpy(stu2.name,"Lily");                 /*strcpy 函数对字符串进行赋值。*/

       showstu(stu1);                            /*调用函数输出结构体 stu1 的信息。*/
       showstu(stu2);                            /*调用函数输出结构体 stu2 的信息。*/
    }
```

输入下面的命令，编译这个程序。

```
    gcc 11.6.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
    chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
A student:
  Name  : Jim
  Age   : 17
  Sex   : 1
  Height: 176
A student:
  Name  : Lily
  Age   : 23
  Sex   : 0
  Height: 171
```

7.2.4　结构体指针作为函数的参数

既然结构体可以作为函数的参数，结构体指针可以访问一个结构体变量，那么结构体指针也可以作为函数的参数。当结构体指针作为函数的参数时，这个指针可以访问结构体的成员，对成员进行赋值或读取操作。下面的程序是使用结构体指针作为函数参数的实例。在程序中自定义一个函数，结构体指针作为这个函数的参数，函数输出这个指针的信息。

```
#include <stdio.h>
#include <stdlib.h>                          /*因为要用 strcpy 函数，所以要包含头文
                                               件 stdlib.h。*/

struct student                               /*定义一个结构体。*/
{
   char name[20];                            /*姓名。*/
   int age;                                  /*年龄。*/
   int sex;                                  /*性别。*/
   int height;                               /*身高。 */
};

void showstu(struct student *p)              /*定义一个自定义函数，结构体指针作为参
                                               数。*/
{
   printf("A student:\n");                   /*输出信息。*/
   printf("  Name  : %s \n",s.name);         /*访问和输出参数中结构体指针指向的成
                                               员。*/
   printf("  Age   : %d \n",s.age);          /*访问和输出参数中结构体指针指向的成
                                               员年龄。*/
   printf("  Sex   : %d \n",s.sex);          /*访问和输出参数中结构体指针指向的成
                                               员性别。*/
   printf("  Height: %d \n\n",s.height);     /*访问和输出参数中结构体指针指向的成
                                               员身高。*/
}

void main()                                  /*主函数。*/
```

```
{
   struct student stu1,stu2;                    /*定义两个结构体变量，表示两个学生。*/
   struct student *p1,*p2;                      /*定义两个结构体指针，指向学生结构体。*/

   p1=&stu1;                                    /*第一个学生取地址各个领域给指针 p1。*/
   p2=&stu2;                                    /*第二个学生取地址各个领域给指针 p2。*/
   stu1.age=17;                                 /*对第一个结构体变量的成员进行赋值。*/
   stu1.sex=1;
   stu1.height=176;
   strcpy(stu1.name,"Jim");                     /*strcpy 函数对字符串进行赋值。*/
   stu2.age=23;                                 /*对第二个结构体变量的成员进行赋值。*/
   stu2.sex=0;
   stu2.height=171;
   strcpy(stu2.name,"Lily");                    /*strcpy 函数对字符串进行赋值。*/

   showstu(stu1);                               /*调用函数输出指针 p1 的信息。*/
   showstu(stu2);                               /*调用函数输出指针 p2 的信息。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 11.7.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

在程序的提示信息后面输出学生信息，然后程序显示的结果如下所示。

```
A student:
  Name  : Jim
  Age   : 17
  Sex   : 1
  Height: 176
A student:
  Name  : Lily
  Age   : 23
  Sex   : 0
  Height: 171
```

7.2.5 结构体作为函数的返回值

自定义函数中的结构体可以作为返回值返回给主函数。结构体返回值的处理方法与普通变量的处理方法是相同的。在主函数中，需要有一个结构体来接收函数返回的结构体。下面的代码是自定义一个函数，函数要求用户输入一个用户的信息，然后将这个学生以结构体的形式返回到主函数中。主函数调用 7.2.3 节中的显示学生信息函数输出这个学生。程序代码如下所示。

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

```
    #include <stdlib.h>                          /*因为要用 strcpy 函数，所以要包含头文件
stdlib.h。*/

    struct student                               /*定义一个结构体。*/
    {
       char name[20];                            /*姓名。*/
       int age;                                  /*年龄。*/
       int sex;                                  /*性别。*/
       int height;                               /*身高。*/
    };

    void showstu(struct student s)               /*定义一个自定义函数，结构体变量作为参数。*/
    {
       printf("A student:\n");                   /*输出信息。*/
       printf("  Name  : %s \n",s.name);         /*访问和输出参数中结构体的变量成员。*/
       printf("  Age   : %d \n",s.age);          /*访问和输出参数中结构体的变量成员年龄。*/
       printf("  Sex   : %d \n",s.sex);          /*访问和输出参数中结构体的变量成员性别。*/
       printf("  Height: %d \n\n",s.height);     /*访问和输出参数中结构体的变量成员身高。*/
    }

    struct student getstu(void)                  /*定义一个函数，结构体作为函数的返回值。*/
    {
        struct student stu;                      /*定义一个结构体。*/
        char name[20];                           /*定义一个字符串，存储姓名。*/
        printf("please input age:\n");           /*提示输入年龄。*/
        scanf("%d",&stu.age);                    /*输入年龄。*/
        printf("please input height:\n");        /*提示身高。*/
        scanf("%d",&stu.height);                 /*输入身高。*/
        printf("please input sex:\n");           /*提示性别。*/
        scanf("%d",&stu.sex);                    /*输入性别。*/
        printf("please input name:\n");          /*提示姓名。*/
        scanf("%s",name);                        /*输入姓名到 name 字符串。*/
        strcpy(stu.name,name) ;                  /*将字符串复制到学生姓名上。*/
        return(stu);                                 /*返回这个结构体。*/
    }

    void main()                                      /*主函数。*/
    {
       struct student stu1,stu2,stu3;                /*定义三个结构体，表示三个学生。*/
       stu1=getstu();                                /*用函数 getstu()输入 stu1 的信息。*/
       stu2=getstu();                                /*用函数 getstu()输入 stu2 的信息。*/
       stu3=getstu();                                /*用函数 getstu()输入 stu3 的信息。*/
       showstu(stu1);                 /*结构体作为参数，调用 showstu 输出 stu1 的信息。*/
       showstu(stu2);                 /*结构体作为参数，调用 showstu 输出 stu2 的信息。*/
       showstu(stu3);                 /*结构体作为参数，调用 showstu 输出 stu3 的信息。*/
    }
```

输入下面的命令，编译这个程序。

```
    gcc 11.8.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

在程序的提示信息后面输出学生信息，然后程序显示的结果如下所示。

```
A student:
  Name  : Jim
  Age   : 17
  Sex   : 1
  Height: 178
A student:
  Name  : Lily
  Age   : 21
  Sex   : 1
  Height: 174
A student:
  Name  : Lucy
  Age   : 22
  Sex   : 0
  Height: 171
```

7.2.6 结构体指针作为函数的返回值

既然结构体可以作为函数的返回值，那么结构体指针作为一种特殊的数据类型也可以作为函数的返回值。当一个函数需要用结构体指针作为返回值时，需要在函数中输入一个指针，函数对指针进行处理，然后返回这个指针。在函数中可以用这个指针来访问指针指向的结构体。下面是一个使用结构体指针作函数返回值的实例。函数利用这个指针输入学生的信息，然后返回该学生结构体的指针，程序通个这个指针取值调用学生显示函数来输出学生的信息。

```
#include <stdio.h>
#include <stdlib.h>                    /*因为要用 strcpy 函数，所以要包含头文件
stdlib.h。*/

struct student                         /*定义一个结构体。*/
{
   char name[20];                      /*姓名。*/
   int age;                            /*年龄。*/
   int sex;                            /*性别。*/
   int height;                         /*身高。 */
};

void showstu(struct student s)         /*定义一个自定义函数，结构体变量作为参数。*/
{
   printf("A student:\n");             /*输出信息。*/
   printf("  Name  : %s \n",s.name);   /*访问和输出参数中结构体的变量成员。*/
   printf("  Age   : %d \n",s.age);    /*访问和输出参数中结构体的变量成员年龄。*/
```

```
   printf("  Sex   : %d \n",s.sex);      /*访问和输出参数中结构体的变量成员性别。*/
   printf("  Height: %d \n\n",s.height);         /*访问和输出参数中结构体的变量成
                                                 员身高。*/
}

struct student *getstu(struct student  *s)      /*自定义一个函数，*/
{
    char name[20];                              /*定义一个字符串，存储姓名。*/
    printf("please input age:\n");              /*提示输入姓名。*/
    scanf("%d",&(*s).age);                      /*用指针访问结构体的变量。*/
    printf("please input height:\n");           /*提示输入身高。*/
    scanf("%d",&(*s).height);                   /*用指针访问结构体变量的成员。*/
    printf("please input sex:\n");              /*提示输入性别。*/
    scanf("%d",&(*s).sex);
    printf("please input name:\n");             /*提示输入姓名。*/
    scanf("%s",name);                           /*输入一个字符串，赋值给 name。*/
    strcpy(s->name,name) ;                      /*用 strcpy 函数把字符串复制到结
                                                 构体 name 成员上。*/
    return(s);                                  /*返回这个指针。*/
}

void main()                                     /*主函数。*/
{
   struct student s1,s2,s3;                     /*用结构体定义三个学生。*/
   struct student *stu1,*stu2,*stu3;            /*定义三个结构体类型的指针。*/
   stu1=getstu(&s1);                    /*对第一个学生取地址作为参数，调用函数。*/
   stu2=getstu(&s2);                    /*对第二个学生取地址作为参数，调用函数。*/
   stu3=getstu(&s3);
   showstu(*stu1);                      /*用第一个指针取结构体变量作为参数显示。*/
   showstu(*stu2);                      /*用第二个指针取结构体变量作为参数显示。*/
   showstu(*stu3);                      /*用第三个指针取结构体变量作为参数显示。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 11.9.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

在程序的提示信息后面输出学生信息，程序显示的结果如下所示。

```
A student:
  Name  : Jim
  Age   : 17
  Sex   : 1
  Height: 178
A student:
```

```
    Name  : Lily
    Age   : 21
    Sex   : 0
    Height: 172
  A student:
    Name  : Brucs
    Age   : 23
    Sex   : 0
    Height: 177
```

7.3 结构体实例

结构体在程序中有着非常重要的运用。在复杂的程序中需要定义结构体作为新的数据类型，用结构体数据类型作为函数的参数或返回值。本节将讲解一个结构运用实例，在程序中用结构体进行各种数据的存储。

7.3.1 程序的需求分析

本节将讲解一个学生管理程序。在程序中用结构体进行学生数据的存储，结构体作为函数的参数与返回值。程序需要完成下面的功能。

（1）菜单选择功能。进入程序以后，显示一个菜单，用户在键盘上输入一个数字选择相应的功能。

（2）学生添加功能。用户选择学生添加菜单项以后，可以在文字提示下输入一个学生的信息。

（3）学生姓名列表功能。这个功能是列出所有学生的编号和姓名。

（4）所有学生详细列表功能。按输入的顺序列出所有的数据。

（5）按姓名查询功能。输入一个学生的姓名时，可以查询出这个学生的数据。

（6）按年龄查询功能。输入一个年龄段以后，显示这个年龄段所有的学生信息。

（7）按性别查询功能。输入一个性别以后，显示这个性别所有的学生信息。

（8）按身高查询功能。输入一个身高区域以后，显示这个身高区域中所有学生的信息。

（9）删除学生功能。输入一个学生的姓名，在结构体数组中删除这个学生。

（10）程序退出功能。选择这一个选项以后，结束程序。

7.3.2 程序中的函数

程序中的每个功能是分成不同的函数来完成的，每一个函数根据需要完成一定的功能，这样就可以把一个复杂的程序分为不同的函数模块。各个功能不同的函数在主函数的调用下统一地完成一个复杂的功能。下面是这个程序中的各个自定义函数。

- 显示一个学生的函数，参数是一个结构体。

```
void showstu(struct student s)
```

- 删除一个学生的函数，没有参数。

```
void delestu();
```

- 菜单函数，完成菜单的显示和选择。

```
int menu();
```

- 读取学生函数，完成一个学生信息的输入，返回一个结构体。

```
struct student getstu(void);
```

- 按学生姓名查找函数。

```
void selectbyname();
```

- 按年龄查找函数。

```
void selectbyage();
```

- 按性别查找函数。

```
void selectbysex();
```

- 按身高查找函数。

```
void selectbyheight();
```

- 主函数，完成程序中各个函数的组织和调用。

```
int  main()
```

7.3.3　程序中的结构体与全局变量

所谓全局变量，指的是程序中所有的函数都可以访问的变量。可以把变量定义语句写在程序在最前面，这样在所有的函数中都可以调用这些变量。在本程序中学生的结构体数组与学生人数的计数都是全局变量。结构体变量是这样定义的。

```
int i;
struct student stu[100];
```

7.3.4　头文件和函数声明

本节以后的多个节讲解的程序是不同的模块。本程序在光盘中的目录是“/源文件/11/7.10.c”。C 程序中的代码可以分解为多个函数，在学习时把这些函数写在一个程序中即可运行。

在程序的最前面需要包含程序中需要使用的头文件。对于程序中自定义的函数，当前面的函数调用后面的函数时可能找不到这个函数，需要在程序的最前面声明这些函数。包含文件和声明函数的代码如下所示。

```
#include <stdio.h>
#include <stdlib.h>                                  /*包含 stdlib.h 头文件。*/
void delestu();                                      /*声明程序中所有的自定义变量。*/
int menu();
struct student getstu(void);
void selectbyname();
void selectbyage();
void selectbysex();
```

```
void selectbyheight();
void showstu(struct student s);
```

7.3.5 定义结构体和全局变量

声明变量以后需要在程序中定义表示学生的结构体，然后用这个结构体类型定义一个结构体数组，保存若干个学生信息，需要有一个变量 i 来保存当前一共有多少个学生。这些变量是在函数以外定义的，所有的函数都可以访问。这一模块的代码如下所示。

```
struct student                          /*定义一个结构体。*/
{
   char name[20];                       /*姓名。*/
   int age;                             /*年龄。*/
   int sex;                             /*性别。*/
   int height;                          /*身高。 */
};
int i;                                  /*进行一个全局变量，用于学生的计数。*/
struct student stu[100];                /*定义一个结构体数组，最多可以存储 100 个学生。*/
```

7.3.6 显示学生信息的函数

这个函数的功能是显示一个学生的详细信息。函数的参数是一个学生类型的结构体。函数通过参数中的结构体访问结构体的成员，输出详细信息。函数完成输出以后没有返回值。这一模块的代码如下所示。

```
void showstu(struct student s)          /*定义一个自定义函数，结构体变量作为参数。*/
{
   printf("A student:\n");              /*输出信息。*/
   printf("  Name  : %s \n",s.name);    /*访问和输出参数中结构体的变量成员。*/
   printf("  Age   : %d \n",s.age);     /*访问和输出参数中结构体的变量成员年龄。*/
   printf("  Sex   : %d \n",s.sex);     /*访问和输出参数中结构体的变量成员性别。*/
   printf("  Height: %d \n\n",s.height); /*访问和输出参数中结构体的变量成员身高。*/
}
```

7.3.7 程序的选择菜单

这个函数的功能是显示学生管理程序所有的功能作为菜单。用户从键盘输入一个编号，如果这个编号有效则返回这个编号，如果编号为 9 则退出程序，如果编号无效则显示错误再次显示菜单。函数的执行流程如图 7-1 所示。

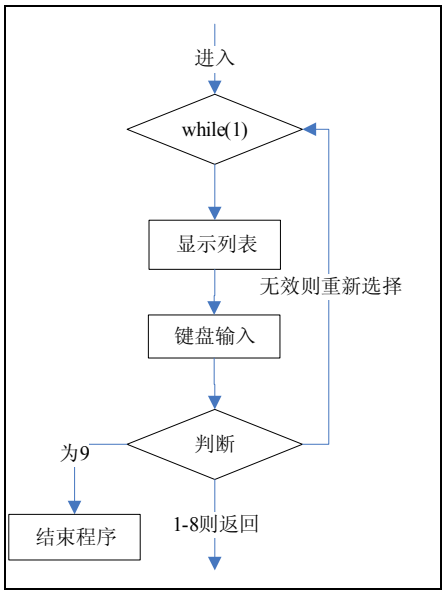

图 7-1　menu 函数的执行流程

这个函数的代码如下所示。

```
int menu()
{
   int i;                                  /*定义变量，这个就是只有这个函数有效。*/
   i=0;                                    /*赋初值。*/
   while(1)                                /*进入一个循环。*/
   {
      printf("Please select a menu:\n");                    /*提示输入。*/
      printf("      1: add a student.\n");                  /*添加学生选项。*/
      printf("      2: list the name all the student.\n");
                                                            /*显示所有学生选项。*/
      printf("      3: list informations of all the student.\n");
                                                            /*显示学生详细信息选项。*/
      printf("      4: select a student by name .\n");      /*按姓名查找选项。*/
      printf("      5: select students  by age .\n");       /*按年龄查找。*/
      printf("      6: select students  by sex .\n");       /*按性别查找。*/
      printf("      7: select students  by height .\n");    /*按身高查找。*/
      printf("      8: delete a student .\n");              /*删除一个学生。*/
      printf("      9: exit .\n");                          /*退出。*/
      scanf("%d",&i);                                       /*输入一个变量。*/
      if(i==9)                                              /*如果输入为 9 则退出程序。*/
      {
         printf("Byebye.\n");                               /*输出信息。*/
         exit(1);                                           /*用 exit 函数退出程序。*/
      }
      if(i<1 || i>9)                                        /*输入的数字不正确。*/
      {
         printf("error.\n");                                /*提示错误。*/
         continue;                                          /*进行下一次选择。*/
      }
```

```
        else
        {
           return(i);                                    /*输入正确则返回这个变量。*/
        }
      }
   }
```

7.3.8 学生信息输入函数

这个函数的功能是提示用户从键盘输入一个学生的信息，然后把这个学生信息的结构体返回。这个函数没有输入参数，代码如下所示。需要注意的是，学生姓名的输入是先把学生姓名输入到另一个字符串上，然后用 strcpy 函数把这个字符串复制到结构体的姓名成员变量中。

```
   struct student getstu(void)                          /*定义一个函数，结构体作为函数的
返回值。*/
   {
       struct student stu;                              /*定义一个结构体。*/
       char name[20];                                   /*定义一个字符串，存储姓名。*/
       printf("please input age:\n");                   /*提示输入年龄。*/
       scanf("%d",&stu.age);                            /*输入年龄。*/
       printf("please input height:\n");                /*提示身高。*/
       scanf("%d",&stu.height);                         /*输入身高。*/
       printf("please input sex:\n");                   /*提示性别。*/
       scanf("%d",&stu.sex);                            /*输入性别。*/
       printf("please input name:\n");                  /*提示姓名。*/
       scanf("%s",name);                                /*输入姓名到 name 字符串。*/
       strcpy(stu.name,name) ;                          /*将字符串复制到学生姓名上。*/
       return(stu);                                     /*返回这个结构体。*/
   }
```

7.3.9　按姓名查找函数

这个函数的功能是按照学生的姓名查找出一个学生的详细信息。函数中需要输入一个学生的姓名，然后用 strcmp 函数比较这个字符串与学生结构体数组中的学生姓名是否相同，如果姓名相同则调用 showstu 函数输出这个学生的信息。

```
    void selectbyname()
    {
       int j,n;                                  /*定义两个计数器变量。*/
       char name[20];                            /*定义一个字符串。*/
       n=0;                                      /*计数变量 n 赋值为 0。*/
       printf("please input a name;\n");         /*提示输入姓名。*/
       scanf("%s",name);                         /*输入一个姓名。*/
       for(j=0;j<i;j++)                          /*for 循环，访问学生数组中所有的学生。*/
       {
         if(strcmp(stu[j].name,name))            /*比较当前输入的姓名与数组中的学生姓名是否
相同。*/
         {
            showstu(stu[j]);                     /*相同则调用 showstu 函数输出这个学生。*/
            n++;                                 /*然后计数自加。*/
         }
       }
       if(n==0)                                  /*如果输出学生的个数为 0。*/
       {
          printf("there is no such a student.\n");      /*提示没有这个学生。*/
       }
    }
```

7.3.10　删除学生函数

这个函数的作用是删除数组中的一个学生。实现方法是从键盘中输入一个学生的姓名，然后用 for 循环访问学生数组中所有的学生，把这个姓名与学生数组中的比较，如果相同则结束外循环。

然后再次执行内循环把下一个学生赋值给这个学生，相当于删除了当前的学生，后面所有的学生都向前移动了一个位置，最后将学生的计数减 1。这一功能的代码如下所示。

```
    void delestu()
    {
       int j,n;                                        /*定义两个计数器变量。*/
       char name[20];                                  /*定义一个数组。*/
       n=0;                                            /*定义一个变量作为计数器。*/
       printf("delete a student:\n");                  /*提示信息。*/
       printf("please input a name:\n");               /*提示输入学生的姓名。*/
       scanf("%s",name);                               /*从键盘输入学生的姓名。*/
       for(j=0;j<i;j++)                                /*for 循环访问学生数组中的变量。*/
       {
         if(strcmp(stu[j].name,name))    /*将学生数组中的姓名与当前输入的姓名进行比较。*/
         {
             for( ;j<i-1;j++)                /*如果姓名相同，则执行这个循环，*/
```

```
            {
               stu[j]=stu[j+1];          /*将下一个学生赋值给当前的学生。*/
            }
            n++;                         /*计数加 1。*/
            i--;                         /*学生计数减 1。*/
            break;                       /*结束当前循环。*/
         }
      }
      if(n==0)                           /*如果输出学生的个数为 0。*/
      {
         printf("there is no such a student.\n");      /*提示没有这个学生。*/
      }
   }
```

7.3.11 按年龄查找函数

这个函数的功能是提示用户输入一个年龄区域。程序用 for 循环的方法查找出这个年龄段中所有的学生，然后列表显示出这些学生的详细信息。函数没有参数和返回值，代码如下所示。

```
   void selectbyage()
   {
      int j,k,t,n;                                          /*定义函数中的变量。*/
      n=0;                                                  /*定义一个计数变量。*/
      printf("please input a top age:\n");                  /*提示输出一个年龄。*/
      scanf("%d",&j);                                       /*输入年龄。*/
      printf("please input a bottom age:\n");
      scanf("%d",&k);
      for(t=0;t<i;t++)                                      /*进行循环，查找所有的学生。*/
      {
        if(stu[t].age<=j &&stu[t].age>k)                    /*数组中学生的年龄与当前输入
的两个年龄作比较。*/
        {
           showstu(stu[t]);                                 /*显示这个学生。*/
           n++;                                             /*计数变量加 1。*/
        }
      }
      if(n==0)                                              /*如果计数变量为 0。*/
      {
         printf("there is no such a student.\n");         /*显示没有要查找的学生。*/
      }
   }
```

7.3.12　按身高查找函数

这个函数的作用是提示用户输入一个身高区域。程序用循环的方法把学生数组中的年龄与这个年龄进行比较，如果学生的年龄在这个年龄区间则输出这个学生的详细信息。函数没有参数和返回值，代码如下所示。

```
void selectbyheight()
{
   int j,k,t,n;                                        /*定义函数中的变量。*/
   n=0;                                                /*定义一个计数器。*/
   printf("please input a top height:\n");             /*输入一个较大值。*/
   scanf("%d",&j);                                     /*从键盘读取一个变量。*/
   printf("please input a bottom  height:\n");         /*提示输入一个较小值。*/
   scanf("%d",&k);                                     /*从键盘读取一个变量。*/
   for(t=0;t<i;t++)                                 /*用 for 循环访问数组中的每一个变量。*/
   {
     if(stu[t].height<=j &&stu[t].height>k)         /*把输入的年龄和当前的年龄做比较。*/
     {
        showstu(stu[t]);                            /*如果在输入的年龄段则输出这个学生。*/
        n++;                                        /*计数变量加 1。*/
     }
   }
   if(n==0)                                         /*如果计数变量为 0。*/
   {
      printf("there is no such a student.\n");      /*输出没有要查找的学生。*/
   }
}
```

7.3.13　按性别查找函数

这个函数的功能是提示输入一个性别。程序用循环来访问学生数组中的每一个学生，把输入的性别与学生数组的性别进行比较，如果相同则输出这个学生的信息。函数没有参数和返回值，代码如下所示。

```
void selectbysex()
{
   int j,t,n;                                   /*定义函数中的变量。*/
   n=0;
   printf("please input the sex:\n");           /*提示输入性别。*/
   scanf("%d",&j);                              /*输入性别。*/
   for(t=0;t<i;t++)                             /*用 for 循环访问学生数组中的每一个结构体。*/
   {
     if(stu[t].sex==j)                          /*结构体的成员变量与输入的性别作比较。*/
     {
        showstu(stu[t]);                        /*显示这个学生。*/
        n++;                                    /*计数变量自加。*/
     }
   }
   if(n==0)                                                /*如果计数变量为 0。*/
   {
```

```
        printf("there is no such a student.\n");      /*提示没有这个学生。*/
    }
}
```

7.3.14 程序的主函数

程序主函数的功能是调用上面这些自定义的函数和模块，使这些功能能够成为一个统一的整体。函数的主要内容是用 while 语句进入一个循环，然后用 menu 要求用户输入一个菜单选项，根据不同的返回结果调用不同的函数执行不同的功能。在 menu 函数中，输入 9 时会结束这个程序的运行。主函数的流程如图 7-2 所示。

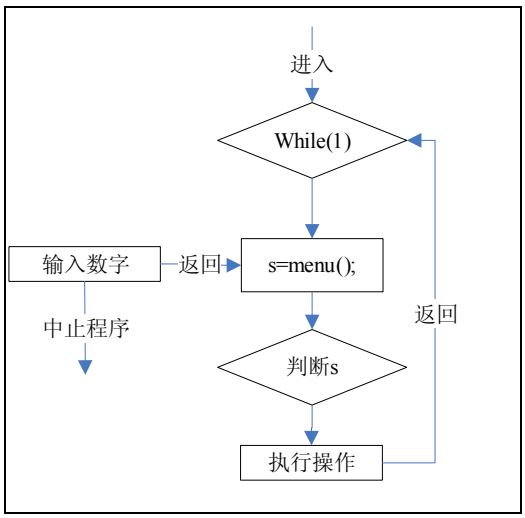

图 7-2 主函数的执行流程图

根据上面的程序流程图编写出的主函数代码如下所示。

```
int  main()                          /*程序的主函数。*/
{
   int s,j;                          /*定义局部变量。*/
   i=0;                              /*对全局变量 i 赋初值为 0。*/
   while(1)                          /*进入一个 while 循环。*/
   {
      s=menu();                      /*从菜单返回一个选项。*/
      if(s==1)                       /*如果选项为 1，则调用学生输入函数，添加一个学生。*/
      {
         stu[i]=getstu();            /*添加一个学生。*/
         i++;                        /*学生计数加 1。*/
      }
      if(s==2)                       /*选项为 2。*/
      {
         printf("All the students:\n");              /*列出所有学生。*/
         for(j=0;j<i;j++)                            /*for 循环访问所有的学生。*/
            {
               printf("  %d:  %s\n",j,stu[j].name);  /*输出学生的姓名与编号。*/
            }
```

```
        }
        if(s==3)                                        /*选项为 3。*/
        {
           for(j=0;j<i;j++)                             /*循环访问所有的学生。*/
           {
              showstu(stu[j]);                          /*显示一个学生的详细信息。*/
           }
        }
        if(s==4)                                        /*选项为 4。*/
        {
           selectbyname();                              /*按姓名查找一个学生。*/
        }
        if(s==5)                                        /*如果选项为 5。*/
        {
           selectbyage();                               /*按年龄查找。*/
        }
        if(s==6)                                        /*选项为 6。*/
        {
           selectbysex();                               /*按性别查查。*/
        }
        if(s==7)                                        /*选项为 7。*/
        {
           selectbyheight();                            /*按身高查找。*/
        }
        if(s==8)                                        /*选项为 8。*/
        {
           delestu();                                   /*删除一个学生。*/
        }
     }
}
```

7.3.15　程序的运行和调试

本节讲解上面编写程序的运行和调试。输入下面的命令，编译这个程序。

输入下面的命令，编译这个程序。

```
gcc 11.4.c -o stu
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x stu
```

下面对这个程序进行运行和测试。

① 输入下面的命令，运行这个程序。

```
./stu
```

② 程序的运行结果如下所示。程序显示出一个菜单，提示用户输入一个选项。

```
Please select a menu:
      1: add a student.
      2: list the name all the student.
```

```
        3: list informations of all the student.
        4: select a student by name .
        5: select students  by age .
        6: select students  by sex .
        7: select students  by height .
        8: delete a student .
        9: exit .
```

③ 这时输入“1”，添加一个学生。然后再次用同样的方法一共输入三个学生。

④ 输入“2”，显示所有的学生。结果如下所示。

```
All the students:
  0:  Tom
  1:  Lily
  2:  Lucy
```

⑤ 输入“3”，显示所有学生的详细信息，结果如下所示。

```
A student:
  Name  : Tom
  Age   : 19
  Sex   : 1
  Height: 176

A student:
  Name  : Lily
  Age   : 17
  Sex   : 0
  Height: 166

A student:
  Name  : Lucy
  Age   : 23
  Sex   : 0
  Height: 179
```

⑥ 输入“4”，按姓名查找一个学生。程序输出下面的提示。

```
please input a name;
```

⑦ 这时输入一个姓名“Lily”。程序的显示结果如下所示。

```
A student:
  Name  : Lily
  Age   : 17
  Sex   : 0
  Height: 166
```

⑧ 输入“7”，按身高查找学生。程序输入下面的提示，要求输入一个上限值。

```
please input a top height:
```

⑨ 这时输入“170”。然后在另一个提示后面输入一个下限值“160”。程序会查找出身

高在“160”与“170”之间所有学生，显示结果如下所示。

```
A student:
  Name  : Lily
  Age   : 17
  Sex   : 0
  Height: 166
```

⑩ 输入“8”，删除一个学生。在提示后面输入一个学生的姓名“Lily”，程序会删除这个学生。

⑪ 然后输入“3”，显示学生的详细信息。显示的结果如下所示，学生“Lily”的信息已经被删除。

```
A student:
  Name  : Tom
  Age   : 19
  Sex   : 1
  Height: 176

A student:
  Name  : Lucy
  Age   : 23
  Sex   : 0
  Height: 179
```

（12）输入“9”，退出这个程序。

7.4　小结

本章讲解了 C 程序的结构体操作。结构体是对现有数据类型的扩充，可以把有实际含义的事物定义为一个结构体数据类型。程序中使用结构体数据类型可以方便地处理有实际含义的数据。本章的内容中，结构体指针和与结构体相关的函数是个难点，要掌握用结构体指针访问结构成员变量的方法，需要仔细理解结构体指针与结构体函数的作用。本章中的最后一个实例是结构体的一个综合运用，处理了复杂的数据与功能，建议学习时完成这个程序的编写。

第 8 章　时间函数

8.1　常用时间函数

在调用系统时间处理时间问题时，需要使用时间函数。本节将讲解 time, gmtime, ctime, asctime, mktime 等常用时间函数。

8.1.1　返回时间函数 time

函数 time 可以返回一个时间值。该函数的使用方法如下所示。

```
time_t time(time_t *t);
```

time 函数会返回从公元 1970 年 1 月 1 日的 UTC 时间的 0 时 0 分 0 秒算起到现在所经过的秒数。参数 t 是一个指针，即使不是一个空指针，函数也会将返回值存到 t 指针所指的内存单元中。time_t 是“time.h”头文件中定义的一个数据类型，表示一个时间的秒数，相当于一个长整型变量。如果 t 是一个空指针，函数会返回一个 time_t 型长整型数。

> **注意**：UTC（Universal Time Coordinated）指的是协调世界时，相当于格林威治平均时，和英国时间是相同的。中国北京时间比 UTC 时间早 8 小时。

下面是使用 time 函数返回当前时间的秒数的实例。

```
#include <stdio.h>
#include <time.h>                    /*包含“time.h”头文件。*/

int main()
{
  time_t s;                          /*定义一个 time_t 型时间变量。*/
  s= time((time_t*)NULL);            /*取当前的时间，参数是一个空指针。*/
  printf("Now  :%ld\n",s);           /*输出时间。*/
}
```

输入下面的命令编译这个程序。

```
gcc 12.1.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
Now  :1198607050
```

再次运行这个程序，结果如下所示。

```
Now  :1198607162
```

从结果可知，第二次运行程序经过了 112 秒。

下面的代码是在 time 函数的参数中使用一个指针，用这个指针接收 time 函数返回的结果。

```
#include <stdio.h>
#include <time.h>                /*包含“time.h”头文件。*/

int main()
{
  time_t *p;                     /*定义一个指向 time_t 类型变量的指针。*/
  time(p);                       /*取时间，参数是指针 p，返回结果到指针的内存单元。*/
  printf("Now  :%ld\n",*p);      /*输出时间。*/
}
```

输入下面的命令编译这个程序。

```
gcc 12.2.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。可见使用指针接收时间结果的作用与返回时间方法是相同的。

```
Now  :1198607309
```

8.1.2　取当前时间函数 gmtime

函数 gmtime 的作用是将 time_t 表示秒数的时间转换人可以理解的时间。这个函数的使用方法如下所示。

```
struct tm *gmtime(time_t *timep);
```

从上面的使用方法可知，函数的参数是一个表示当前时间秒数的指针。返回值是一个 tm 类型的结构体指针。tm 结构体是在“time.h”头文件中定义的，定义方法和成员如下所示。

```
struct tm
{
   int tm_sec;
   int tm_min;
   int tm_hour;
   int tm_mday;
   int tm_mon;
   int tm_year;
```

```
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

这些成员表示的含义和范围如下所示。

- int tm_sec：代表当前秒数，正常范围是 0~59。
- int tm_min：代表当前分钟数，正常范围是 0~59。
- int tm_hour：从午夜算起的小时数，范围是 0~23。
- int tm_mday：当前月份的日数，范围是 1~31。
- int tm_mon：代表当前月份，从一月算起，范围是 0~11。
- int tm_year：从 1900 年算起至今的年数。
- int tm_wday：一星期的日数，从星期一算起，范围是 0~6。
- int tm_yday：从本年 1 月 1 日算起至今的天数，范围为 0~365。
- int tm_isdst：是不是使用了夏令时。如果为 1 表示使用了夏令时，为 0 则表示没有使用夏令时。

> **注意**：夏令时间指的是某些国家在夏季将时间调整若干个小时。我国过去曾使用过夏令时，现在已经终止。计算机中可以设置夏令时。

需要注意的是，这里的时间返回的是 UTC 时间，即英国零时区时间。如果计算机中使用了 UTC 时间并且设置了时间，则结果与计算机上显示的时间有一些差异。

```
#include <stdio.h>
#include <time.h>                              /*包含“time.h”头文件。*/

main(){
   time_t timep;                               /*定义一个 time_t 型变量。*/
   struct tm *p;                               /*定义一个 tm 型结构体指针。*/
   time(&timep);                               /*取当前时间，返回到 timep 的值中。*/
   p=gmtime(&timep);                           /*取当前时间，返回到结构体指针 p 上。*/
   printf("Year  :%d\n",1900+p->tm_year);      /*输出年。*/
   printf("Month :%d\n",1+p->tm_mon);          /*输出月。*/
   printf("Day   :%d\n",p->tm_mday);           /*日。*/
   printf("Hour  :%d\n",p->tm_hour);           /*小时。*/
   printf("Minute:%d\n",p->tm_min);            /*分。*/
   printf("Second:%d\n",p->tm_sec);            /*秒。*/
   printf("Weekday:%d\n",p->tm_wday);          /*星期几。*/
   printf("Days  :%d\n",p->tm_yday);           /*一年的第几天。*/
   printf("Isdst :%d\n",p->tm_isdst);          /*是否使用了夏令时。*/
}
```

需要注意的是，tm 结构体中的年是从 1900 年到现在的第几年，所以在显示时需要加 1900。月份的返回值是 0 至 11，所以加 1 以后才可以表示当前的月份。

输入下面的命令编译这个程序。

```
gcc 12.3.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。时间这样显示时，可以表示为可以理解的时间。表示成秒数的时间是无法被识别的。

```
Year  :2007
Month :12
Day   :25
Hour  :18
Minute:47
Second:52
Weekday:2
Days  :358
Isdst :0
```

再次运行这个程序，结果如下所示。可见这个时间的返回值是系统时间，是随时变化的。

```
Year  :2007
Month :12
Day   :25
Hour  :18
Minute:54
Second:49
Weekday:2
Days  :358
Isdst :0
```

8.1.3　字符串格式时间函数 ctime

函数 ctime 的作用是将一个时间返回成一个可以识别的字符串格式。这个函数的使用方法如下所示。

```
char *ctime(time_t *timep);
```

从上面的使用方法可知，函数 ctime 的参数 timep 是一个指向 timep_t 类型的指针。函数会把这个指针转换成一个字符串，然后返回这个字符串的头指针。这里返回的时间已经转换成本地时区的时间，与计算机上显示的时间相同。字符串的显示格式为“Feb Jun 14 12:56:08 1999”这种形式。下面是这个函数的使用实例。

```
#include <stdio.h>
#include <time.h>                /*包含“time.h”头文件。*/
#include <stdlib.h>              /*包含“stdlib.h”头文件。*/

int main()
{
  time_t *p;                     /*定义一个指向 time_t 类型变量的指针。*/
```

```
  char s[30];                    /*定义一个字符串 s。*/
  time(p);                       /*取时间，参数是指针 p，返回结果到指针的内存单元。*/
  strcpy(s,ctime(p)) ;           /*将 ctime 返回的结果复制到字符串 s 上。*/
  printf("%s\n",s);              /*输出字符串 s。*/
}
```

输入下面的命令编译这个程序。

```
gcc 12.4.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
Tue Dec 25 14:03:42 2007
```

8.1.4 字符串格式时间函数 asctime

函数 asctime 的作用是将一个 tm 格式的时间转换为一个字符串格式。这个函数的使用方法如下所示。

```
char * asctime(struct tm * timeptr);
```

函数的参数是一个 tm 格式的时间结构体指针，返回值是一个字符串。与函数 ctime 不同的是，ctime 的参数是一个表示秒数的时间指针。返回的字符串格式与 ctime 的返回格式是相同的。下面是 asctime 的使用实例。

```
#include <stdio.h>
#include <time.h>                /*包含“time.h”头文件。*/
#include <stdlib.h>              /*包含“stdlib.h”头文件。*/

int main()
{
  time_t *p;                     /*定义一个指向 time_t 类型变量的指针。*/
  struct gm *q;                  /*定义一个 gm 类型的指针。*/
  char s[30];                    /*定义一个字符串 s。*/
  time(p);                       /*取时间，参数是指针 p，返回结果到指针的内存单元。*/
  q=gmtime(p);                   /*用 gmtime 函数返回一个 gm 格式的时间指针。*/
  strcpy(s,asctime(q)) ;         /*将 asctime 返回的结果复制到字符串 s 上。*/
  printf("%s\n",s);              /*输出字符串 s。*/
}
```

输入下面的命令编译这个程序。

```
gcc 12.5.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
Tue Dec 25 19:24:43 2007
```

8.1.5　取得当地时间函数 localtime

函数 localtime 的作用是返回 tm 格式的当地时间。与 gmtime 函数不同的是，gmtime 函数返回的是一个 UTC 时间。localtime 时间的使用方法如下所示。

```
struct tm *localtime(time_t * timep);
```

从上面的使用方法可知，localtime 函数的参数是一个 time_t 型时间的指针，返回值是一个 tm 型的结构体指针。这个函数的使用实例如下所示。

```
#include <stdio.h>
#include <time.h>                              /*包含“time.h”头文件。*/

main(){
   time_t timep;                               /*定义一个 time_t 型变量。*/
   struct tm *p;                               /*定义一个 tm 型结构体指针。*/
   time(&timep);                               /*取当前时间，返回到 timep 的值中。*/
   p=localtime(&timep);                        /*取本地时间，返回到结构体指针 p 上。*/
   printf("Year  :%d\n",1900+p->tm_year);      /*输出年。*/
   printf("Month :%d\n",1+p->tm_mon);          /*输出月。*/
   printf("Day   :%d\n",p->tm_mday);           /*日。*/
   printf("Hour  :%d\n",p->tm_hour);           /*小时。*/
   printf("Minute:%d\n",p->tm_min);            /*分。*/
   printf("Second:%d\n",p->tm_sec);            /*秒。*/
   printf("Weekday:%d\n",p->tm_wday);          /*星期几。*/
   printf("Days  :%d\n",p->tm_yday);           /*一年的第几天。*/
   printf("Isdst :%d\n",p->tm_isdst);          /*是否使用了夏令时。*/
}
```

输入下面的命令编译这个程序。

```
gcc 12.6.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。这次显示时间的结果是转换成本地时区以后的时间，和计算机上显示的时间是相同的。

```
Year  :2007
Month :12
```

```
Day   :25
Hour  :14
Minute:29
Second:14
Weekday:2
Days  :358
Isdst :0
```

8.1.6 将时间转换成秒数函数 mktime

函数 mktime 的作用是将一个 tm 结构类型的时间转换成秒数时间，与 gmtime 的作用相反。该函数的使用方法如下所示。

```
time_t mktime(tm * timeptr);
```

从上面的使用方法可知，函数的参数是一个 tm 类型的指针，返回一个 time_t 类型的数字表示当前的秒数。下面是这个函数的使用实例。先取得当前时间的秒数，然后用 ctime 函数转化为字符串输出。把这个时间转换成 tm 结构体时间以后，再用 mktime 转换成 time_t 格式的时间，然后再用 ctime 将时间转化为字符串输出。程序的代码如下所示。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>
#include <time.h>                          /*程序的包含文件。*/

main()
{
    time_t t;                              /*定义一个 time_t 型的时间。*/
    struct tm *p;                          /*定义一个 tm 型的时间指针。*/
    char s[30];                            /*定义一个字符串。*/
    time(&t);                              /*取当前时间。*/
    strcpy(s,ctime(&t));                   /*将时间转换成字符串。*/
    printf("%s\n",s);                      /*输出时间。*/
    p=gmtime(&t);                          /*将时间转换成 tm 格式的结构体。*/
    t=mktime(p);                           /*将时间转换成 time_t 型的时间。*/
    strcpy(s,ctime(&t));                   /*将时间转换成字符串。*/
    printf("%s\n",s);                      /*输出时间。*/
}
```

输入下面的命令编译这个程序。

```
gcc 12.7.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序输出的结果是相同的。两次输出的是同一个时间值，第二次输出时经过了三次转换。

```
Wed Dec 26 08:41:07 2007
Wed Dec 26 08:41:07 2007
```

8.1.7　取得当前的时间函数 gettimeofday

前面所讲到的时间函数只能把时间精确到秒。如果对时间的处理精度为微秒级，需要使用函数 gettimeofday。一微秒等于百万分之一秒。这个函数的使用方法如下所示。

```
int gettimeofday ( struct timeval * tv , struct timezone * tz )
```

这个函数的参数是两个结构体指针。这两个结构体的定义如下所示。

```
struct timeval
{
   long tv_sec;
   long tv_usec;
};
```

结构体成员的含义如下所示。

- tv_sec：当前时间的秒数。
- tv_usec：当前时间的微秒数。

```
struct timezone{
   int tz_minuteswest;
   int tz_dsttime;
};
```

结构体成员的含义如下所示。

- tz_minuteswest：与 UTC 时间相差的分钟数。
- tz_dsttime：与夏令时间相差的分钟数。

函数 gettimeofday 会把当前时间的这些参数返回到这两个结构体指针上。如果处理成功，则返回真值 1，否则返回 0。这个函数的使用实例如下所示。

```
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include <time.h>                              /*包含头文件 time.h。*/

main()
{
   struct timeval tv;                          /*定义一个 timeval 型的结构体。*/
   struct timezone tz;                         /*定义一个 timezone 型的结构体。*/
   gettimeofday (&tv , &tz);                   /*取得当前时间。两个结构体的指针作为参
                                               数。*/
   printf("tv_sec       : %d\n", tv.tv_sec);             /*输出秒。*/
   printf("tv_usec      : %d\n",tv.tv_usec);             /*输出微秒。*/
   printf("tz_minuteswest: %d\n", tz.tz_minuteswest);    /*输出 UTC 时间相差的分
                                                         钟数。*/
   printf("tz_dsttime    : %d\n",tz.tz_dsttime);         /*输出与夏令时间相差的
                                                         分钟数。*/
```

```
}
```

输入下面的命令编译这个程序。

```
gcc 12.8.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
tv_sec       : 1198594465
tv_usec      : 462591
tz_minuteswest: 0
tz_dsttime    : 0
```

再次运行这个程序，结果如下所示。秒数与微秒数已经改变。

```
tv_sec       : 1198598472
tv_usec      : 726880
tz_minuteswest: 0
tz_dsttime    : 0
```

8.1.8 设置当前时间函数 settimeofday

函数 settimeofday 的作用是设置当前的系统时间。只有以 root 用户登录以后才有权限进行这个操作。该函数的使用方法如下所示。

```
int settimeofday (struct timeval *tv, struct timezone *tz);
```

这个函数的参数是 timeval 类型的结构体指针和 timezone 类型的结构体指针。这两个结构体类型的定义在上一节中已经讲述。下面是这个函数的使用实例，取当前的时间输出，然后将当前的时间向前调整 4000 秒。

```
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include <time.h>                                  /*程序的包含文件。*/

main()
{
   struct timeval tv;                              /*定义一个 timeval 类型的结构体。*/
   struct timezone tz;                             /*定义一个 timezone 型的结构体。*/
   gettimeofday (&tv , &tz);                       /*取当前的时间。*/
   printf("tv_sec       : %d\n", tv.tv_sec);              /*输出秒。*/
   printf("tv_usec      : %d\n",tv.tv_usec);              /*输出微秒。*/
   printf("tz_minuteswest: %d\n", tz.tz_minuteswest);  /*与 UTC 时间相差的分钟
                                                          数。*/
   printf("tz_dsttime    : %d\n",tz.tz_dsttime);         /*夏令时间相差的分钟数。
```

```
                                                                    */
    tv.tv_sec=tv.tv_sec- 4000;                                      /*当前的时间减去 4000
                                                                    秒。*/
    settimeofday (&tv , &tz);                                       /*设置当前时间。*/
}
```

输入下面的命令编译这个程序。

```
gcc 12.9.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示，这个时间是当前时间。程序同时将计算机的时间向前设置了 4000 秒，可以看到计算机的时间已经改变。

```
tv_sec       : 1198599321
tv_usec      : 264551
tz_minuteswest: 0
tz_dsttime    : 0
```

8.2　时间函数使用实例

本节将讲解几个时间函数使用实例。在处理时间时需要用到上一节所讲的各种时间函数，在进行计时的程序中需要用到 gettimeofday 函数，这个函数可以返回精确到微秒的时间值。

8.2.1 运行程序所需要的时间

计算机运行一个程序时需要占用一定的时间。用时间函数可以对程序开始和结束的时间进行计时，程序结束时可以处理这两个时间，显示出结果。因为计算机运行程序的时间常常很短，因此需要用 gettimeofday 函数来取得精确到微秒的时间值。程序的代码如下所示。

```
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include <time.h>                          /*包含头文件。*/

main()
{
   struct timeval tv,tv2;                  /*定义两个 timeval 类型的结构体。*/
   struct timezone tz;                     /*定义一个 timzone 型的结构体。*/
   long sec,usec,i;                        /*定义三个长整型数。*/

   gettimeofday (&tv , &tz);               /*取得当前时间。*/
   for(i=0;i<100000000;i++)                /*程序进行多次循环。*/
   {
       ;;;;;;;;;;;;;;;;;;;                /*执行一段空语句。分号表示空语句。*/
   }
   gettimeofday (&tv2, &tz);               /*取得结束时的时间。*/
   sec=tv2.tv_sec-tv.tv_sec;               /*用减法求出秒数。*/
   usec=tv2.tv_usec-tv.tv_usec;            /*用减法求得微秒数。*/
   printf("%ld\n",sec*1000000 + usec);     /*把秒转换成微秒，再与微秒相加，输出结果。
                                           */
}
```

输入下面的命令编译这个程序。

```
gcc 12.10.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。说明进行这些循环运算所占用的时间是 0.64 秒。

```
647834
```

8.2.2 两次输入之间的时间间隔

本节的实例对两次的键盘输入时间进行计时，然后求出两次输入的时间间隔。这个程序的计时方法与上一节程序的计时方法是相同的。输出时将时间转换成秒和微秒两部分。

```
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
```

```
    #include <time.h>                              /*包含头文件。*/

    main()
    {
       struct timeval tv,tv2;                      /*定义两个 timeval 型的结构体。*/
       struct timezone tz;                         /*定义一个 timezone 型的结构体。*/
       long sec,usec,i;                            /*定义三个长整型数。*/

       printf("please input a number:\n");         /*提示输入。*/
       scanf("%d",&i);                             /*输入。*/
       gettimeofday (&tv , &tz);                   /*取当前的时间。*/
       printf("please input a number:\n");         /*第二次提示输入。*/
       scanf("%d",&i);                             /*输入。*/
       gettimeofday (&tv2 , &tz);                  /*取当前的时间。*/

       sec=tv2.tv_sec-tv.tv_sec;                   /*用减法求得秒数。*/
       usec=tv2.tv_usec-tv.tv_usec;                /*用减法求得微秒数。*/
       printf("%dSEC  %ldUSEC\n",(sec*1000000  +  usec)/1000000,(sec*1000000  +
usec)%1000000);
    }
```

输入下面的命令编译这个程序。

```
gcc 12.11.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

在程序的提示后面两次输入数字，显示的结果如下所示。表明这两次输入数字之间的间隔是 1.71 秒。

```
1SEC 710287USEC
```

8.2.3　设置系统时间

用 settimeofday 函数可以设置计算机的时间。可以从键盘输入时间构造出一个 tm 结构体类型的时间，然后用 mktime 函数转换成一个 time_t 型的时间。把这个时间值构造成 timeval 结构体时间的值，可以用这个值设置系统时间。程序的代码如下所示。

```
    #include <stdio.h>
    #include <sys/time.h>
    #include <unistd.h>
    #include <time.h>                    /*包含文件。*/

    main()
    {
        time_t t;                        /*定义一个 time_t 型的时间。*/
        struct tm *p;                    /*定义一个 tm 型的结构体时间指针。*/
```

```
        int i;                              /*定义一个整型变量。*/
        struct timeval tv;                  /*定义一个 timeval 型的结构体。*/
        struct timezone tz;                 /*定义一个 timezone 型的结构体。*/

        time(&p);                           /*用 time 函数取当前的时间。*/
        p=gmtime(&t);                       /*用 gmtime 函数取出 tm 结构体时间。*/
        printf("Change your time:\n");      /*提示。*/
        printf("Year:\n");                  /*提示输入年。*/
        scanf("%d",&i);                     /*输入。*/
        (*p).tm_year=i-1900;                /*把年的时间减去 1900，赋值给结构体中的年时间。
*/
        printf("Month:\n");                 /*输入月。*/
        scanf("%d",&i);                     /*输入。*/
        (*p).tm_mon=i-1;                    /*月份减一赋值给结构体中的月时间。*/
        printf("Date:\n");                  /*输入日。*/
        scanf("%d",&i);
        (*p).tm_mday=i;
        printf("Hour:\n");                  /*输入小时。*/
        scanf("%d",&i);
        (*p).tm_hour=i;
        printf("Minute:\n");                /*输入分。*/
        scanf("%d",&i);
        (*p).tm_min=i;
        printf("Second:\n");                /*输入秒。*/
        scanf("%d",&i);
        (*p).tm_sec=i;

        t=mktime(p);                        /*把输入的时间结构体构造成 time_型的时间。*/
        tz.tz_minuteswest=0;                /*构造 timezone 型结构体。*/
        tz.tz_dsttime=0;
        tv.tv_sec=t;                        /*构造 timeval 结构体中的秒。*/
        tv.tv_usec=0;                       /*构造微秒。*/
        settimeofday (&tv , &tz);           /*设置时间。*/
        printf("Done!\n");                  /*输出提示。*/
    }
```

输入下面的命令编译这个程序。

```
gcc 12.12.c
```

然后输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
Change your time:
Year:
```

这时输入相应的时间，完成年、月、日、时、分、秒的输入以后，程序会把计算机的时间更改为当前输入的时间。

8.3　小结

本章讲解了 C 函数中的时间函数。在本章的学习中，time_t 时间类型与 tm 时间结构体的理解是个难点。相关的函数都需要使用这两种时间或指针作为参数。其中 time, gmtime, ctime, asctime, gettimeofday 这几个函数需要重点掌握。最后通过三个实例演示了时间函数的使用方法与技巧。

第 9 章　目录与文件

9.1　文件操作的权限

所谓权限，指的是文件系统为了进行安全管理需要在对文件操作时进行用户身份认证。合法的用户可以进行操作，而没有权限的用户不能进行文件操作。用 C 程序进行目录与文件操作时，需要设置目录的权限。本节将讲解权限的表示方法。

前面章节讲解的 Linux 命令中，可以用 chmod 命令更改文件的权限。例如下面的命令是对一个文件添加可执行权限。

```
chmod +x a.out
```

在 C 编程中，需要用三个八进制数字来表示文件的权限。例如 777 表示这个用户、同组成员、其他成员对这个文件都有执行、写、读的权限。

三个数字表示的文件权限如下所示。

- 第一个数字表示本用户的权限，相当于 User 的权限。
- 第二个数字表示同组用户的权限，相当于 Group 的权限。
- 第三个数字表示其他用户的权限，相当于 Other 的权限。

数字不同的值代表不同的权限，数字的含义如下所示。

- 4 表示可读权限，相当于 r 权限。
- 2 表示可写权限，相当于 w 权限。
- 1 表示可执行权限，相当于 x 权限。

如果有多种权限，可把几个权限加起来。例如下面不同的权限组合。

- 7 等于 4+2+1，表示这个文件有可读、可写、可执行的权限。
- 5 等于 4+1，表示这个文件有可读、可执行的权限。
- 6 等于 4+2，表示这个文件有可读、可写权限，但是不可以执行。

如果对一个文件设置的权限是 764 表示的权限含义如下所示。

- 7 等于 4+2+1，表示本用户可读、可写、可执行。
- 6 等于 4+2，表示同组用户可读、可写、不可执行。
- 4 含义是其他用户只可读、不可写、不可执行。

chmod 命令也可以使用这些数字作为参数对一个文件更改权限。例如下面的命令就是把一个文件的权限设置为 766。

```
chmod 766 a.out
```

9.2　错误处理与错误号

在进行文件操作时，用户可能会遇到权限不足、找不到文件等错误，这时需要在程序中设置错误捕捉语句并显示错误。错误捕捉和错误输出是应用错误号和 strerror 函数来实现的。

9.2.1　错误定义的理解

Linux 系统已经把所有的错误定义成为不同的错误号和错误常数，程序如果发生了异常，会返回这一个错误的常数。这个常数可以显示为整型数字，也可以用 strerror 函数来显示为已经定义的错误信息。

可以打开包含文件来查看这些错误号的错误信息。在终端中输入下面的命令，打开错误定义文件。

```
vim /usr/include/asm-generic/errno-base.h
```

显示的错误定义代码如下所示。

```
#ifndef _ASM_GENERIC_ERRNO_BASE_H
#define _ASM_GENERIC_ERRNO_BASE_H
#define EPERM        1  /* Operation not permitted */
#define ENOENT       2  /* No such file or directory */
#define ESRCH        3  /* No such process */
#define EINTR        4  /* Interrupted system call */
#define EIO          5  /* I/O error */
#define ENXIO        6  /* No such device or address */
#define E2BIG        7  /* Argument list too long */
#define ENOEXEC     8   /* Exec format error */
#define EBADF       9   /* Bad file number */
#define ECHILD      10  /* No child processes */
#define EAGAIN      11  /* Try again */
#define ENOMEM      12  /* Out of memory */
#define EACCES      13  /* Permission denied */
#define EFAULT      14  /* Bad address */
#define ENOTBLK     15  /* Block device required */
#define EBUSY       16  /* Device or resource busy */
#define EEXIST      17  /* File exists */
#define EXDEV       18  /* Cross-device link */
#define ENODEV      19  /* No such device */
#define ENOTDIR     20  /* Not a directory */
#define EISDIR      21  /* Is a directory */
#define EINVAL      22  /* Invalid argument */
#define ENFILE      23  /* File table overflow */
#define EMFILE      24  /* Too many open files */
#define ENOTTY      25  /* Not a typewriter */
#define ETXTBSY     26  /* Text file busy */
#define EFBIG       27  /* File too large */
#define ENOSPC      28  /* No space left on device */
#define ESPIPE      29  /* Illegal seek */
```

```
#define EROFS       30  /* Read-only file system */
#define EMLINK      31  /* Too many links */
#define EPIPE       32  /* Broken pipe */
#define EDOM        33  /* Math argument out of domain of func */
#define ERANGE      34  /* Math result not representable */
#endif
```

还有一个错误定义文件，可以用下面的命令打开查看这个文件。

```
vim /usr/include/asm-generic/errno.h
```

从这两个文件可知，Linux 系统一共定义 131 种错误常数。用这些错误常数可以返回程序的错误信息。

9.2.2 用错误常数显示错误信息

程序错误返回的错误常数是不容易理解的，需要转换成有效的识别语句。函数 strerror 可以把一个错误常数转换成一个错误提示语句。例如下面的实例是把 ENOENT, EIO, EEXIST 三个错误的常数和错误信息显示出来。

```
#include <stdio.h>
#include <string.h>
#include <errno.h>                          /*包含错误处理头文件。*/
int main(void)
{
   printf("ENOENT:\n");                     /*输出提示。*/
   char *mesg = strerror(ENOENT);           /*将错误号转换成一个字符串。*/
   printf("  Errno  :%d\n",ENOENT);         /*输出错误号。*/
   printf("  Message:%s\n",mesg);           /*输出错误信息。*/

   printf("EIO  :\n");                      /*EIO 错误。*/
   char *mesg1 = strerror(EIO);
   printf("  Errno  :%d\n",EIO);
   printf("  Message:%s\n",mesg1);

   printf("EEXIST   :\n");                  /*文件重名错误。*/
   char *mesg2 = strerror(EEXIST);
   printf("  Errno  :%d\n",EEXIST);
   printf("  Message:%s\n",mesg2);
}
```

这三个错误的含义如下所示。

- ENOENT：没有相关的文件或文件夹。
- EIO：I/O 出现错误。
- EEXIST：文件重名。

> 注意：I/O 指的是计算机的输入和输出，包括一切操作、程序或设备与计算机之间发生的数据流通。最常见的 I/O 设备有打印机、硬盘、键盘和鼠标。

输入下面的命令，编译这个程序。

```
gcc 13.err.1.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
ENOENT:
  Errno  :2
  Message:No such file or directory
EIO    :
  Errno  :5
  Message:Input/output error
EEXIST  :
  Errno  :17
  Message:File exists
```

9.2.3　用错误序号显示错误信息

函数 strerror 可以把一个错误常数返回成一个表示错误信息的字符串。这个函数的使用方法如下所示。

```
char *strerror(int errnum);
```

函数的参数是一个表示错误信息的整型数，返回值是一个字符串。上一节程序中的错误常数实际上也是一个整型数。下面的实例是输出前 15 个序号的错误信息。

```
#include <stdio.h>
#include <string.h>
#include <errno.h>                                  /*包含错误处理头文件。*/
int main(void)
{
   int i;                                           /*定义一个循环变量 i。*/
   for(i=1;i<=15;i++)                               /*循环输出。*/
   {
      printf("Errno:%d  ",i);                       /*输出错误号。*/
      printf("Message:%s\n",strerror(i));           /*输出错误信息。*/
   }
}
```

输入下面的命令，编译这个程序。

```
gcc 13.err.2.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
Errno:1  Message:Operation not permitted
Errno:2  Message:No such file or directory
Errno:3  Message:No such process
Errno:4  Message:Interrupted system call
Errno:5  Message:Input/output error
Errno:6  Message:No such device or address
Errno:7  Message:Argument list too long
Errno:8  Message:Exec format error
Errno:9  Message:Bad file descriptor
Errno:10  Message:No child processes
Errno:11  Message:Resource temporarily unavailable
Errno:12  Message:Cannot allocate memory
Errno:13  Message:Permission denied
Errno:14  Message:Bad address
Errno:15  Message:Block device required
```

9.3 创建与删除目录

在 Linux 系统中，目录就是一个文件夹，文件可以存放在目录中。目录是一种特殊的文件，需要对目录设置权限。本节将讲解 C 程序的目录操作。

9.3.1 创建目录函数 mkdir

函数 mkdir 可以在硬盘中建立一个目录，相当于 mkdir 命令。但与 mkdir 命令不同的是，这里的操作是用 C 语言的函数完成目录创建的。函数的使用方法如下所示。

```
int mkdit(char *pathname,mode_t mode);
```

在参数列表中，pathname 是一个字符型指针，表示需要创建的目录路径，mode 是表示权限的八进制数字。如果目录创建成功，则返回整型数 0，否则返回整型数-1。要使用这个函数需要在程序中包含“sys/types.h”与“sys/stat.h”两个头文件。

在创建目录时，可能有权限、目录名、硬盘空间等问题，可能返回的错误常数如下所示。需要在程序中捕捉这些错误并输出错误信息。

- EPERM：目录中有不合规则的名字。
- EEXIST：参数 pathname 所指的目录已存在。
- EFAULT：pathname 指向了非法的地址。
- EACCESS：权限不足，不允许创建目录。
- ENAMETOOLONG：参数 pathname 太长。
- ENOENT：所指的上级目录不存在。
- ENOTDIR：参数 pathname 不是目录。
- ENOMEM：核心内存不足。
- EROFS：欲创建的目录在只读文件系统内。

- ELOOP：参数 pathname 有多个符合的链接。
- ENOSPC：磁盘空间不足。

下面的实例是在目录/root 下面创建一个 tmp11 目录。程序中设置一个错误号变量，用于捕捉程序发生的异常，如果创建不成功则输出这个错误。

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>                          /*包含头文件。*/
#include <errno.h>                             /*包含头文件处理程序的错误。*/
main()
{
   extern int errno;                           /*设置一个错误。*/
   char *path="/root/tmp11";                   /*定义一个字符串表示需要创建的目录。*/

   if(mkdir(path,0766)==0)                     /*创建一个目录。*/
     /*需要注意这里的权限设置参数，第一个 0 表示这里的是八进制数，766 的含义如本章第一节所
     述。*/
     /*如果目录创建成功，则会返回 0，返回值与 0 进行比较。*/
   {
      printf("created the directory %s.\n",path);       /*输出目录创建成功。*/
   }
   else                                        /*如果不成功则输出提示信息。*/
   {
      printf("cant't creat the directory %s.\n",path); /*输出信息。*/
      printf("errno: %d\n",errno);             /*输出错误号。*/
      printf("ERR  : %s\n",strerror(errno));/*输出错误信息。*/
   }
}
```

输入下面的命令，编译这个程序。

```
gcc 13.1.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示，表明已经创建了这个目录。可以在终端中输入“ls /root”命令查看这个目录。

```
created the directory /root/tmp11
```

当再次运行这个程序时，需要创建的目录与上一次创建的目录重名，则不能创建目录，显示的结果如下所示。这时返回了一个错误号，输出了错误信息为“File exists”。

```
cant't creat the directory /root/tmp11.
errno: 17
ERR: File exists
```

9.3.2 删除目录函数 rmdir

rmdir 函数的作用是删除一个目录。该函数的使用方法如下所示。

```
int rmdir(char *pathname);
```

参数 pathname 是需要删除的目录字符串的头指针。如果删除成功则返回一个整型 0，否则返回-1。可以设置一个 errno 来捕获发生的错误。下面是这个函数可能发生的错误。

- EACCESS：权限不足，不允许创建目录。
- EBUSY:系统繁忙，没有删除。
- EFAULT：pathname 指向了非法的地址。
- EINVAL：有这个目录，但是不能删除。
- ELOOP：参数 pathname 有多个符合的链接。
- ENAMETOOLONG：给出的参数太长。
- ENOENT：所指向的上级目录不存在。
- ENOMEM：核心内存不足。
- ENOTDIR：不是一个目录。
- EROFS：指向的目录在一个只读目录里。

下面是使用这个函数来删除一个目录的实例，用来删除上一节创建的目录/root/tmp11。

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>                                  /*包含头文件。*/
main()
{
   extern int errno;                                /*设置错误编号。*/
   char *path="/root/tmp11";                        /*设置需要删除的目录。*/

   if(rmdir(path)==0)                               /*删除文件，返回值与 0 比较。*/
   {
      printf("deleted the directory %s.\n",path);       /*显示删除成功。*/
   }
   else                                             /*如果删除不成功。*/
   {
     printf("cant't delete the directory %s.\n",path); /*显示删除错误。*/
     printf("errno: %d\n",errno);                   /*显示错误号。*/
     printf("ERR  : %s\n",strerror(errno));         /*显示错误信息。*/
   }
}
```

输入下面的命令，编译这个程序。

```
gcc 13.2.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示，表明已经删除了这个目录。

```
deleted the directory /root/tmp11.
```

再次运行这个程序，因为没有目录/root/tmp11，所以显示错误信息。程序的运行结果如下所示。

```
cant't delete the directory /root/tmp11.
errno: 2
ERR  : No such file or directory
```

9.4　文件的创建与删除

所谓创建文件，是指在一个目录中建立一个空文件，可供其他程序的写入操作。删除文件指从磁盘中删除无用的文件。本节将讲解文件的建立与删除操作。

9.4.1　创建文件函数 creat

函数 creat 的作用是在目录中建立一个空文件，该函数的使用方法如下所示。

```
int creat(char * pathname, mode_t mode);
```

函数的参数 pathname 表示需要建立文件的文件名和目录名，mode 表示这个文件的权限。文件权限的设置见本章第一节所述。文件创建成功时返回创建文件的编号，否则返回-1。

使用这个函数时，需要在程序的前面包含下面三个头文件。

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
```

如果创建文件不成功，可用 errno 捕获错误编号然后输出。creat 函数可能发生的错误如下所示。

- EEXIST：参数 pathname 所指的文件已存在。
- EACCESS：参数 pathname 所指定的文件不符合所要求测试的权限。
- EROFS：欲打开写入权限的文件存在于只读文件系统内。
- EFAULT：参数 pathname 指针超出可存取的内存空间。
- EINVAL：参数 mode 不正确。
- ENAMETOOLONG：参数 pathname 太长。
- ENOTDIR：参数 pathname 为一目录。
- ENOMEM：核心内存不足。
- ELOOP：参数 pathname 有过多符号链接问题。
- EMFILE：已达到进程可同时打开的文件数上限。

下面的实例是使用 creat 函数在/root 目录下面建立一个文件 tmp.txt。

```
#include <stdio.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>                              /*包含头文件。*/
main()
{
   extern int errno;                            /*定义错误号。*/
   char *path="/root/tmp.txt";

   if(creat(path,0766)==-1)                     /*用 creat 函数创建一个文件。*/
   {
     printf("cant't create the file %s.\n",path);       /*不能创建文件。*/
     printf("errno: %d\n",errno);               /*显示错误号。*/
     printf("ERR  : %s\n",strerror(errno));     /*显示错误信息。*/
   }
   else                                         /*另一种情况。*/
   {
     printf("created file %s.\n",path);         /*显示创建成功。*/
   }
}
```

输入下面的命令，编译这个程序。

```
gcc 13.3.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
created file /root/tmp.txt .
```

与创建目录不同的是，当再次运行这个程序时也能创建同名的文件。这时新创建的文件会覆盖以前的文件。

9.4.2 删除文件函数 remove

函数 remove 的作用是删除一个文件。这个函数的使用方法如下所示。

```
int remove(char *pathname);
```

参数 pathname 是一个字符型指针，表示需要删除的目录。文件删除成功则返回 0，否则返回-1。要使用这个函数需要在程序的最前面包含下面的头文件。

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

函数 pathname 在删除文件时，可能发生权限不足或目录不存在的问题。可能产生的错误如下所示，需要在程序中设置 errno 来捕获错误信息。

- EACCESS：权限不足，不允许创建目录。

- EBUSY:系统繁忙，没有删除。
- EFAULT：pathname 指向了非法的地址。
- EINVAL：有这个目录，但是不能删除。
- ELOOP：参数 pathname 有多个符合的链接。
- ENAMETOOLONG：给出的参数太长。
- ENOENT：所指向的上级目录不存在。
- ENOMEM：核心内存不足。
- ENOTDIR：不是一个目录。
- EROFS：指向的目录在一个只读目录里。

下面是用 remove 函数删除上一节创建的文件/root/tmp.txt。

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>                              /*包含头文件。*/
main()
{
   extern int errno;                            /*定义一个错误号。*/
   char *path="/root/tmp.txt";                  /*定义要删除的文件名。*/

   if(remove(path)==0)                          /*删除文件。*/
   {
      printf("Deleted file %s.\n",path);        /*显示删除成功。*/
   }
   else                                         /*另一种情况。*/
   {
      printf("cant't delete the file %s.\n",path);      /*显示删除失败。*/
      printf("errno: %d\n",errno);              /*显示错误号。*/
      printf("ERR  : %s\n",strerror(errno)); /*显示错误信息。*/
   }
}
```

输入下面的命令，编译这个程序。

```
gcc 13.4.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。显示已经删除了这个文件。

```
Deleted file /root/tmp.txt.
```

当再次运行这个程序时，显示的结果如下所示。因为需要删除的目录在上一次程序运行

时已经删除，所以找不到要删除的文件。

```
cant't delete the file /root/tmp.txt.
errno: 2
ERR  : No such file or directory
```

9.4.3 建立临时文件函数 mkstemp

所谓临时文件，指的是程序运行时为了存储中间数据而建立的文件。计算机重启时，这些文件会自动删除。如果在程序运行时需要把文件短时间地写到磁盘上，可以使用 mkstemp 函数创建一个临时文件。mkstemp 函数的使用方法如下所示。

```
int mkstemp(char *template);
```

参数 template 表示需要建立临时文件的文件名字符串。文件名字符串中最后六个字符必须是 XXXXXX。mkstemp 函数会以可读写模式和 0600 权限来打开该文件，如果文件不存在则会建立这个文件，返回值是打开文件的编号，如果文件建立不成功则返回-1。

该函数可能发生的错误如下所示。可以用 errno 来捕获错误信息。

- EEXIST：文件同名错误。
- EINVAL：参数 template 字符串最后六个字符非 XXXXXX。

需要注意的是，参数 template 所指的文件名称字符串必须声明为数组，用下面这种声明数组的方法声明。

```
char template[] =”template-XXXXXX”;
```

使用下面这种声明字符串的方法声明的 template 是不能运行的。

```
char *template = “template-XXXXXX”;
```

下面是使用 mkstemp 函数建立文件的实例。

```
#include <stdio.h>
#include <stdlib.h>                                    /*包含头文件。*/
main()
{
   extern int errno;                                   /*设置一个错误号。*/
   char path[]="mytemp-XXXXXX";                        /*定义文件名。*/

   if(mkstemp(path)!=-1)                               /*建立一个临时文件。*/
   {
      printf("created temp file %s.\n",path);          /*显示文件已经建立。*/
   }
   else                                                /*另一种情况。*/
   {
      printf("cant't create temp file %s.\n",path); /*显示不能创建临时文件。*/
      printf("errno: %d\n",errno);                     /*显示错误号。*/
      printf("ERR  : %s\n",strerror(errno));           /*显示错误信息。*/
   }
}
```

输入下面的命令，编译这个程序。

```
gcc 13.5.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。显示已经建立了一个临时文件 mytemp-USfaDc。

```
created temp file mytemp-USfaDc.
```

再次运行这个程序，显示的结果如下所示。显示已经建立了一个临时文件 mytemp-hL1La9。系统会自动生成一个不同名的文件名来建立临时文件。

```
created temp file mytemp-hL1La9.
```

在终端中输入“ls”命令，显示的结果如下所示。可以在当前文件夹中查看到这两个临时文件。

```
13.1.c  13.3.c  13.eer.1.o  a.out  mytemp-hL1La9  tmp.c
13.2.c  13.4.c  13.eer.2.c  a.txt  mytemp-USfaDc  tmp.o
```

9.5　文件的打开与关闭

文件的打开指的是从磁盘中找到一个文件，返回一个整型的打开文件顺序编号。打开的文件处于可读、可写状态。文件的关闭指的是释放打开的文件，使文件处于不可读写的状态。

9.5.1　打开文件函数 open

函数 open 的作用是打开一个文件，使文件处于可读写的状态。这个函数的使用方法如下所示。

```
int open(char *pathname, int flags);
int open(char *pathname,int flags, mode_t mode);
```

9.5.2 文件打开方式的设置

在上一节所示函数的参数列表中，pathname 表示要打开文件的路径字符串。参数 flags 是系统定义的一些整型常数，表示文件的打开方式。Flags 的可选值如下所示。

- O_RDONLY：以只读方式打开文件。
- O_WRONLY：以只写方式打开文件。
- O_RDWR：以可读写方式打开文件。

上述三种旗标是互斥的，也就是不可同时使用，但可与下列的旗标利用“|”运算符组合。

- O_CREAT：若要打开的文件不存在则自动建立该文件。
- O_EXCL：如果 O_CREAT 已被设置，此指令会去检查文件是否存在。文件若不存在则建立该文件，否则将导致打开文件错误。此外，若 O_CREAT 与 O_EXCL 同时设置时，如果要打开的文件为一个链接，则会打开失败。
- O_NOCTTY：如果要打开的文件为终端机设备时，则不会将该终端机当成进程控制终端机。
- O_TRUNC：若文件存在并且以可写的方式打开时，此标志会清空文件。这样原来的文件内容会丢失。
- O_APPEND：以附加的文件打开文件。当读写文件时会从文件尾开始向后移动，写入的数据会以附加的方式加入到文件后面。
- O_NONBLOCK：以不可阻断的方式打开文件，也就是无论文件有无数据读取或等待操作，都会立即打开文件。
- O_NDELAY：O_NONBLOCK。
- O_SYNC：以同步的方式打开文件，所有的文件操作不写入到缓存。
- O_NOFOLLOW：如果参数 pathname 所指的文件为一符号链接，则会打开失败。
- O_DIRECTORY：如果参数 pathname 所指的文件的目录不存在，则打开文件失败。

9.5.3 打开文件的权限

打开文件时，如果没有这个文件则会自动新建一个文件。在新建文件时需要设置新建文件权限。系统为参数 mode 定义了下面这些常数，可以直接使用这些常数来设置文件的权限。这些权限设置只有在建立新文件时才会有效。

- S_IRWXU：00700 权限，该文件所有者具有可读、可写、可执行的权限。
- S_IRUSR 或 S_IREAD：00400 权限，该文件所有者具有可读取的权限。
- S_IWUSR 或 S_IWRITE：00200 权限，该文件所有者具有可写入的权限。
- S_IXUSR 或 S_IEXEC：00100 权限，该文件所有者具有可执行的权限。
- S_IRWXG：00070 权限，该文件用户组具有可读、可写、可执行的权限。
- S_IRGRP：00040 权限，该文件用户组具有可读的权限。
- S_IWGRP：00020 权限，该文件用户组具有可写入的权限。
- S_IXGRP：00010 权限，该文件用户组具有可执行的权限。
- S_IRWXO：00007 权限，其他用户具有可读、可写、可执行的权限。

- S_IROTH：00004 权限，其他用户具有可读的权限。
- S_IWOTH：00002 权限，其他用户具有可写入的权限。
- S_IXOTH：00001 权限，其他用户具有可执行的权限。

9.5.4　文件打开实例

本节讲解一个文件打开实例，用 open 函数打开一个文件。调用这个函数时，如果正确地打开了这个文件则返回这个文件的打开编号，如果打开失败则返回 0。使用这个函数之前需要在程序的最前面包含下面这些头文件。

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
```

open 函数可能发生下面这些错误。可用 errno 捕获打开文件时发生的错误。

- EEXIST：参数 pathname 所指的文件已存在，却使用了 O_CREAT 和 O_EXCL 旗标。
- EACCESS：参数 pathname 所指的文件没有打开权限。
- EROFS：欲写入权限的文件存在于只读文件系统内。
- EFAULT：参数 pathname 指针超出可存取内存空间。
- EINVAL：参数 mode 不正确。
- ENAMETOOLONG：参数 pathname 太长。
- ENOTDIR：参数 pathname 不在一个目录中。
- ENOMEM：核心内存不足。
- ELOOP：参数 pathname 有过多符号链接问题。
- EIO：I/O 存取错误。

下面的程序是使用 open 函数打开一个文件的实例。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>                          /*包含头文件。*/
main()
{
   int fd,fd1;                              /*定义打开文件的编号。*/
   char path[]="/root/txt1.txt";            /*定义需要打开的文件。*/
   extern int errno;                        /*定义错误号。*/
   fd=open(path,O_WRONLY,0766);             /*打开文件，只读，不能自动新建。*/
   if(fd!=-1)                               /*如果返回值不为-1。*/
    {
      printf("opened file %s  .\n",path);   /*显示已经打开文件。*/
    }
   else                                         /*另一种情况。*/
    {
      printf("cant't open file %s.\n",path);    /*显示不能打开文件。*/
      printf("errno: %d\n",errno);              /*显示错误号。*/
```

```
        printf("ERR  : %s\n",strerror(errno));     /*显示错误信息。*/
      }

    fd1=open(path,O_WRONLY|O_CREAT,0766);    /*打开文件，如果没有文件则自动新建。*/
    if(fd1!=-1)                              /*如果返回的文件号不为-1。*/
      {
        printf("opened file %s  .\n",path);    /*显示文件新建成功。*/
      }
    else                                     /*另一种情况。*/
      {
        printf("cant't open file %s.\n",path);/*显示不能打开文件。*/
        printf("errno: %d\n",errno);          /*显示错误号。*/
        printf("ERR  : %s\n",strerror(errno)); /*显示错误信息。*/
      }
  }
```

输入下面的命令，编译这个程序。

```
gcc 13.6.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。第一次是以只读的方式打开文件，没有这个文件时显示打开错误。第二次在 Flags 参数中加入了 O_CREAT，表示没有文件时新建这个文件，显示文件打开成功。

```
cant't open file /root/txt1.txt.
errno: 2
ERR  : No such file or directory
opened file /root/txt1.txt  .
```

输入“cd ~”命令进入到用户的根目录，然后输入“ls”命令，可以发现目录中有一个文件 txt1.txt。

9.5.5 关闭文件函数 close

函数 close 的作用是关闭一个已经打开的文件。使用完文件后需要使用 close 函数关闭该文件，这个操作会让数据写回磁盘，并释放该文件所占用的资源。该函数的使用方法如下所示。

```
int close(int fd);
```

参数 fd 是用 open 函数打开文件时返回的打开序号。如果成功关闭文件则返回 0，发生错误则返回-1。在进程结束时，系统会自动关闭已打开的文件，但仍建议在程序中关闭文件，并检查返回值是否正确。在关闭文件时，可能返回 EBADF 错误，表示需要关闭的文件号不存在。

在使用这个函数时，需要在程序的前面包含下面的头文件。

```
#include<unistd.h>
```

下面是一个实例，用 open 函数打开一个文件以后，再用 close 函数关闭这个文件。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>                              /*包含头文件。*/
main()
{
   int fd;                                      /*定义一个文件编号。*/
   char path[]="/root/txt1.txt";                /*定义需要打开的文件名。*/
   extern int errno;                            /*定义一个错误号。*/

   fd=open(path,O_WRONLY|O_CREAT,0766);         /*打开文件，如果没有这个文件则新建。*/
   if(fd!=-1)                                   /*打开成功。*/
   {
      printf("opened file %s  .\n",path);       /*显示打开成功。*/
   }
   else                                         /*打开不成功。*/
   {
      printf("cant't open file %s.\n",path);/*显示打开不成功。*/
      printf("errno: %d\n",errno);              /*显示错误号。*/
      printf("ERR  : %s\n",strerror(errno)); /*显示错误信息。*/
   }

   if(close(fd)==0)                             /*关闭打开的文件。*/
   {
      printf("closed.\n");                      /*显示关闭成功。*/
   }
   else                                         /*另一种情况。*/
   {
      printf("close file %s error.\n",path);/*显示关闭不成功。*/
      printf("errno: %d\n",errno);              /*显示错误号。*/
      printf("ERR  : %s\n",strerror(errno)); /*显示错误信息。*/
   }

    if(close(1156)==0)                          /*关闭一个不存在的文件号。*/
   {
      printf("closed.\n");                      /*如果关闭成功则显示。*/
   }
   else
   {
      printf("close file %s error.\n",path);    /*关闭不成功则显示错误。*/
      printf("errno: %d\n",errno);              /*显示错误号。*/
      printf("ERR  : %s\n",strerror(errno));    /*显示错误信息。*/
   }
}
```

输入下面的命令，编译这个程序。

```
gcc 13.7.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。第一次关闭的是已经打开的一个文件，可以正常关闭。第二次关闭的是一个不存在的文件号，显示没有这个文件。

```
opened file /root/txt1.txt  .
closed.
close file /root/txt1.txt error.
errno: 9
ERR  : Bad file descriptor
```

9.6 文件读写

文件读写指的是从文件中读出信息或将信息写入到文件中。文件读取是使用 read 函数来实现的，文件写入是通过 write 函数来实现的。在进行文件写入的操作时，只是在文件的缓冲区中操作，可能没有立即写入到文件中。需要使用 sync 或 fsynx 函数将缓冲区中的数据写入到文件中。

9.6.1 在文件中写字符串函数 write

函数 write 可以把一个字符串写入到一个已经打开的文件中。这个函数的使用方法如下所示。

```
ssize_t write (int fd,void * buf,size_t count);
```

在参数列表中，fd 是已经打开文件的文件号，buf 是需要写入的字符串，count 是一个整型数，表示需要写入的字符个数。size_t 是一个相当于整型的数据类型，表示需要写入的字节的数目。将字符串写入文件以后，文件的写位置也会随之移动。

如果写入成功，write 函数会返回实际写入的字节数。发生错误发生时则返回-1，可以用 errno 来捕获发生的错误。可能发生的错误如下所示。

- EINTR：此操作被其他操作中断。
- EAGAIN：当前打开文件是不可写的方式打开的，不能写入文件。
- EADF：参数 fd 不是有效的文件编号，或该文件已关闭。

下面是这个函数的使用实例。用 open 函数打开一个文件，将一个字符串写入到这个文件中，然后关闭文件。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>                                  /*包含头文件。*/
main()
{
   int fd;                                          /*定义打开的文件号。*/
   char path[]="/root/txt1.txt";                    /*定义要写入文件的路径。*/
   char s[]="hello ,Linux.\nI've leart C program for two weeks.\n";
                                                    /*定义要写入的字符串。*/
   extern int errno;                                /*使用错误号。*/

   fd=open(path,O_WRONLY|O_CREAT);                  /*打开文件。*/
   if(fd!=-1)                                       /*打开文件是否成功。*/
   {
      printf("opened file %s  .\n",path);           /*文件打开成功。*/
   }
   else
   {
      printf("cant't open file %s.\n",path);        /*文件打开不成功。*/
       printf("errno: %d\n",errno);                 /*显示错误号。*/
       printf("ERR  : %s\n",strerror(errno));       /*显示错误信息。*/
   }

   write(fd,s,sizeof(s));                           /*将内容写入到文件。*/
   close(fd);                                       /*关闭文件。*/
   printf("Done");                                  /*显示信息。*/
}
```

在函数中，sizeof 可以返回字符串 s 的长度。输入下面的命令，编译这个程序。

```
gcc 13.8.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
opened file /root/txt1.txt .
Done
```

这时在终端中输入下面的命令，查看写入文件的信息。

```
vim  /root/txt1.txt
```

vim 显示的/root/txt1.txt 文件内容如下所示。

```
hello ,Linux.
I've leart C program for two weeks.
```

9.6.2 读取文件函数 read

函数 read 可以从一个打开的文件中读取字符串。这个函数的使用方法如下所示。

```
ssize_t read(int fd,void *buf ,size_t count);
```

在参数列表中，fd 表示已经打开文件的编号。buf 是一个空指针，读取的内容会返回到这个指针指向的字符串。count 表示需要读取字符的个数。返回值表示读取到的字符的个数。如果返回值为 0，表示已经到达文件末尾或文件中没有内容可供读取。在读文件时，文件的读位置会随着读取到的字节移动。

当有错误发生时，返回值为-1，可以用 errno 来捕获错误编号。可能返回的错误如下所示。

- EINTR：此操作被其他操作中断。
- EAGAIN：当前打开文件是不可写的方式打开的，不能写入文件。
- EADF：参数 fd 不是有效的文件编号，或该文件已关闭。

下面是使用 read 函数进行文件读取的实例，可以读取上一节中写入文件的内容，然后显示出读取的字符串和实际读取到的字符数。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>                              /*包含头文件。*/

main()
{
   int fd;                                      /*定义文件号。*/
   char path[]="/root/txt1.txt";                /*定义打开的文件名。*/
   int size;
   char s[100];                                 /*定义一个字符串。*/
   extern int errno;                            /*使用错误号。*/

   fd=open(path,O_RDONLY);                      /*打开文件。*/
   if(fd!=-1)                                   /*打开成功。*/
   {
      printf("opened file %s  .\n",path);
   }
   else
   {
      printf("cant't open file %s.\n",path);    /*打开不成功。*/
      printf("errno: %d\n",errno);              /*显示错误号。*/
      printf("ERR  : %s\n",strerror(errno));    /*显示错误信息。*/
   }

   size=read(fd,s,sizeof(s));                   /*读取文件内容。*/
   close(fd);                                   /*关闭文件。*/
   printf("%s\n",s);                            /*输出。*/
   printf("%d\n",size);                         /*输出返回的字节数。*/
}
```

入下面的命令，编译这个程序。

```
gcc 13.9.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。程序显示了文件打开成功和上一节写入的字符串。最后一行显示了实际读取到的字符数。

```
opened file /root/txt1.txt  .
hello ,Linux.
I've leart C program for two weeks.
51
```

9.6.3　文件读写位置的移动

每一个已打开的文件都有一个读写位置。当打开文件时通常读写位置是指向文件开头，若是以附加的方式打开文件，则读写位置会指向文件末尾。read 或 write 函数读或写文件时，读写位置会随读写的字节相应移动。可以用 lseek 函数在文件内容中的位置上面移动，这样就可以在文件中不同的位置进行上读写。这个函数的使用方法如下所示。

```
off_t lseek(int fd,off_t offset ,int whence);
```

在参数列表中，fd 表示用 open 函数打开的文件返回编号，参数 offset 表示根据参数 whence 来移动读写位置的位移数，whence 表示系统定义的常量，可能有下面这些赋值。

- SEEK_SET：参数 offset 即为新的读写位置。
- SEEK_CUR：以目前的读写位置往后增加 offset 个位移量。
- SEEK_END：将读写位置指向文件末尾后再增加 offset 个位移量。

当 whence 值为 SEEK_CUR 或 SEEK_END 时，参数 offet 允许负值的出现。这时表示在当前位置或文件末尾位置上向前移动若干个字节。下面是一些常用的文件位置移动方式。

- lseek（int fd,0,SEEK_SET）：将读写位置移到文件开头。
- lseek（int fd，0,SEEK_END）：将读写位置移到文件结尾。
- lseek（int fd，0,SEEK_CUR）：取得当前的文件位置。

函数调用成功时返回这个文件当前的读写位置，即距文件开头多少个字节。若有错误则返回-1，errno 会存放错误号。这个函数可能产生的错误如下所示。

- EBADF：传入的参数不是一个已经打开的文件。
- EINVAL：给入的 whence 参数不合理。
- EOVERFLOW：给入的移动参数导致文件头指针指向了文件开头以前，产生了溢出错误。

要使用这个函数，需要在程序的前面包含下面的头文件。

- #include<sys/types.h>

- #include<unistd.h>

下面是 lseek 函数的使用实例。打开上一节所使用的文本文件，在文件读取之前，用 lseek 函数移动文件的读写位置。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>                                  /*包含头文件。*/
main()
{
   int fd;                                          /*定义文件号。*/
   char path[]="/root/txt1.txt";                    /*定义文件路径。*/
   int size;
   char s[100]="";                                  /*定义一个字符串。*/
   extern int errno;                                /*使用错误号。*/

   fd=open(path,O_RDONLY);                          /*打开文件。*/
   if(fd!=-1)                                       /*打开成功。*/
   {
     printf("opened file %s  .\n",path);
   }
   else
   {
     printf("cant't open file %s.\n",path);         /*打开失败。*/
      printf("errno: %d\n",errno);                  /*显示错误号。*/
      printf("ERR  : %s\n",strerror(errno));        /*显示错误信息。*/
   }

   size=read(fd,s,3);                               /*读取三个字符。*/
   printf("%d  :",size);                            /*输出返回的字符数。*/
   printf("%s\n",s);                                /*输出三个读取的字符。*/

   size=read(fd,s,3) ;                              /*读取后面的三个字符。*/
   printf("%d  :",size);                            /*输出返回的字符数。*/
   printf("%s\n",s);                                /*输出三个读取的字符。*/

   lseek(fd,8,SEEK_SET);                            /*移动到第 8 个字符。*/
   size=read(fd,s,3);                               /*读取后面的三个字符。*/
   printf("%d  :",size);                            /*输出返回的字符数。*/
   printf("%s\n",s);                                /*输出三个读取的字符。*/

   lseek(fd,0,SEEK_SET);                            /*移动到第 0 个字符。*/
   size=read(fd,s,3);                               /*读取后面的三个字符。*/
   printf("%d  :",size);                            /*输出返回的字符数。*/
   printf("%s\n",s);                                /*输出三个读取的字符。*/

   close(fd);                                       /*关闭文件。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 13.10.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。需要注意的是，文件读写时空格也是一个字符，但是输出时看不到这个字符。

```
3  :hel
3  :lo
3  :inu
3  :hel
```

9.6.4　将缓冲区数据写入到磁盘函数 sync

所谓缓冲区，是 Linux 系统对文件的一种处理方式。在对文件进行写操作时，并没有立即把数据写入到磁盘，而是把数据写入到缓冲区中。如果需要把数据立即写入到磁盘，可以用 sync 函数。用这个函数强制写入缓冲区数据的好处是保证数据有同步。这个函数的使用方法如下所示。

```
int sync(void)
```

这个函数会对当前程序打开的所有文件进行处理，将缓冲区中的内容写入到文件。函数没有参数，返回值为 0。这个函数一般不会产生错误。要使用这个函数，需要在程序中包含下面的头文件。

```
#include<unistd.h>
```

下面是这个函数的使用实例。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>                                          /*包含头文件。*/

main()
{
   int fd;                                                  /*定义文件号。*/
   char path[]="/root/txt1.txt";                            /*要打开的文件。*/
   char s[]="hello ,Linux.\nI've leart C program for two weeks.\n";    /* 要
写入的字符串。*/
   extern int errno;                                        /*定义错误号。*/

   fd=open(path,O_WRONLY|O_CREAT);                          /*打开文件。*/
```

```
    if(fd!=-1)                                          /*文件打开成功。*/
    {
      printf("opened file %s  .\n",path);
    }
    else
    {
      printf("cant't open file %s.\n",path);            /*文件打开失败。*/
       printf("errno: %d\n",errno);                     /*显示错误号。*/
       printf("ERR  : %s\n",strerror(errno));           /*显示错误信息。*/
    }
    write(fd,s,sizeof(s));                              /*写入文件。*/
    sync();                                             /*将缓冲区数据写入磁盘。*/
    printf("sync function done. \n");                   /*输出信息。*/
    close(fd);                                          /*关闭文件。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 13.11.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
opened file /root/txt1.txt  .
sync function done.
```

9.6.5 将缓冲区数据写入到磁盘函数 fsync

函数 fsync 的作用是将缓冲区的数据写入到磁盘。与 sync 函数不同的是，这个函数可以指定打开文件的编号，执行以后会返回一个值。这个函数的使用方法如下所示。

```
int fsync(int fd);
```

参数 fd 是 open 函数打开文件时返回的编号。函数如果执行成功则返回 0，否则返回-1。在使用这个函数时，需要在文件前面包含下面的头文件。

```
#include<unistd.h>
```

函数可能发生下面这些错误，可以用 errno 捕获错误。

- EBADF：参数 fd 不是一个正确的文件打开编号或者文件不能写入。
- EIO：发生了 I/O 错误。前面的章节已经讲述过 I / O 错误。
- EROFS 或 EINVAL：fd 是一个特殊的文件，不能够写入内容。

下面程序是 fsync 函数的使用实例。在文件中写入内容以后，用 fsync 函数将文件写入到缓冲区。

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

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>                                  /*包含头文件。*/

main()
{
   int fd;                                          /*定义文件号。*/
   char path[]="/root/txt1.txt";                    /*要打开的文件。*/
   char s[]="hello ,Linux.\nI've leart C program for two weeks.\n";    /* 要
写入的字符串。*/
   extern int errno;                                /*定义错误号。*/

   fd=open(path,O_WRONLY|O_CREAT);                  /*打开文件。*/
   if(fd!=-1)                                       /*文件打开成功。*/
   {
      printf("opened file %s  .\n",path);
   }
   else
   {
      printf("cant't open file %s.\n",path);        /*文件打开失败。*/
       printf("errno: %d\n",errno);                 /*显示错误号。*/
       printf("ERR  : %s\n",strerror(errno));       /*显示错误信息。*/
   }

   write(fd,s,sizeof(s));                           /*写入文件。*/
   if(fsync(fd)==0)                                 /*将缓冲区数据写入磁盘。*/
   {
      printf("fsync function done.\n");             /*写入成功。*/
   }
   else
   {
      printf("fsync function failed. \n");          /*写入失败。*/
   }
   close(fd);                                       /*关闭文件。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 9.12.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
opened file /root/txt1.txt  .
```

```
fsync function done.
```

9.7 文件锁定

所谓的文件锁定，指的是以独占的方式打开文件。一个程序打开文件以后，其他的程序不能读取或写入文件。文件锁定有利于文件内容的一致性。本节将讲解文件的锁定问题。

9.7.1 文件锁定的理解

当多个程序同时打开同一个文件时，可能导致文件内容不一致的情况发生。例如一个文件中的数据是一个账户金额。用户打开这个文件读取数据，进行处理以后将结果写入到文件。如果文件没有进行锁定时，可以发生下面这种错误。

（1）假设文件中的数据为 10000。用户 A 打开文件读取这个数据，但是还没有及时的写入。

（2）这时用户 B 读取这个数据，将结果加 10000，然后将结果 20000 写入到这个文件。

（3）用户 A 的处理时间较长，比用户 B 早读入数据，但后写入数据。A 将数据加 1000 得到 11000，然后将数据写入到文件。

这时就发生了一个错误，用户 B 的数据丢失。为了阻止这种错误需要在打开文件时进行文件锁定，正确的做法如下所示。

（1）假设文件中的数据为 10000。用户 A 打开文件读取这个数据，但是还不能及时的写入，于是将文件加一把锁，不允许别的用户访问。

（2）用户 B 访问这个文件时，文件已经加锁，无法访问。于是等待用户 A 完成数据访问。

（3）用户 A 完成数据处理以后，将信息写入到文件，这时解除对文件的锁定。

（4）这时用户 B 可以对文件进行访问。访问的同时对文件添加一个锁定。在解除锁定以前，其他的用户不能访问这个文件。

9.7.2 文件锁定函数 flock

在访问一个文件时，可以用 flock 函数对文件进行锁定，防止其他用户同时访问这个文件发生数据不一致的情况。这个函数的使用方法如下所示。

```
int flock(int fd,int operation);
```

在参数列表中，fd 是 open 函数打开文件时返回的打开序号。operation 是系统定义的一些整型常量，可能的取值和含义如下所示。

- LOCK_SH：建立共享锁定，其他的程序可以同时访问这一个文件。多个程序可同时对同一个文件建立共享锁定。
- LOCK_EX：建立互斥锁定，其他用户不能同时访问这一个文件。一个文件同时只有一个互斥锁定。单一文件不能同时建立共享锁定与互斥锁定。
- LOCK_UN：解除文件锁定状态。
- LOCK_NB：无法建立锁定时，此操作可不被阻断，马上返回进程。通常与 LOCK_SH 或 LOCK_EX 做 OR（|）组合。

当文件锁定成功时会返回 0，否则返回-1。可以用 errno 来捕获发生的错误。这个函数可能发生的错误如下所示。

- EBADF：df 参数不是一个已经打开的文件。
- EINTR：在进行操作时，这个操作被其他的程序或信号所中断。
- EINVAL：给入的参数不合法。
- ENOLCK：核心内存溢出错误。
- EWOULDBLOCK：这个文件已经被其他程序建立互斥锁定。

下面的程序是 flock 函数的使用实例。打开文件以后，对文件建立一个锁定。当其他程序打开这个文件时，则无法打开锁定的文件。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>                              /*包含头文件。*/

main()
{
   int fd,i;                                    /*定义变量。*/
   char path[]="/root/txt1.txt";                /*要访问的文件。*/
   extern int errno;                            /*定义错误号。*/

   fd=open(path,O_WRONLY|O_CREAT);              /*打开文件。*/
   if(fd!=-1)                                   /*打开成功。*/
   {
      printf("opened file %s  .\n",path);
      printf("please input a number to lock the file.\n"); /*输入一个数字。*/
      scanf("%d",&i);                           /*输入。*/
      if(flock(fd,LOCK_EX)==0)                  /*锁定文件。*/
      {
        printf("the file was locked.\n");           /*输出信息。*/
       }
       else
       {
          printf("the file was not locked.\n");     /*文件锁定失败。*/
       }
      printf("please input a number to unlock the file.\n");    /*提示输入。*/
      scanf("%d",&i);                                   /*输入。*/
      if(flock(fd,LOCK_UN)==0)                          /*解除文件锁定。*/
      {
        printf("the file was unlocked.\n");         /*输出解除锁定成功。*/
       }
       else
       {
          printf("the file was not unlocked.\n");   /*输出解除锁定失败。*/
       }
      close(fd);                                    /*关闭文件。*/
```

```
    }
    else
    {
       printf("cant't open file %s.\n",path);          /*不能打开文件的情况。*/
        printf("errno: %d\n",errno);                   /*显示错误号。*/
        printf("ERR  : %s\n",strerror(errno));         /*显示错误信息。*/
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 13.13.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。显示输入一个数字锁定文件。输入一个数字以后，显示文件锁定。这时输入同一个数字可以解除文件锁定。

```
opened file /root/txt1.txt  .
please input a number to lock the file.
```

9.7.3 文件锁定函数 fcntl

函数 fcntl 的作用是对一个文件进行锁定。与 flock 函数不同的是，fcntl 可以设定对文件的某一部分进行锁定。该函数的使用方法如下所示。

```
int fcntl(int fd , int cmd);
int fcntl(int fd,int cmd,long arg);
int fcntl(int fd,int cmd,struct flock *lock);
```

这个函数的参数列表有多种形式。在编译时会自动与参数进行匹配。参数 fd 是 open 函数打开文件时返回的文件编号，cmd 是系统定义的一些整型常量，用来设置文件的打开方式，有下面几种可选方式。

- F_DUPFD：用来查找大于或等于参数 arg 的最小且仍未使用的文件编号，并且复制参数 fd 的文件编号，执行成功则返回新复制的文件编号。
- F_GETFD：取得 close-on-exec 标志。若此标志的参数 arg 的 FD_CLOEXEC 位为 0，代表在调用 exec()相关函数时文件将不会关闭。
- F_SETFD：设置 close-on-exec 旗标。该旗标以参数 arg 的 FD_CLOEXEC 位决定。
- F_GETFL：取得文件描述词状态旗标，此旗标为 open 的参数返回序号。
- F_SETFL： 设置文件描述词状态旗标，参数 arg 为新旗标，但只允许 O_APPEND、O_NONBLOCK 和 O_ASYNC 位的改变，其他位的改变将不受影响。
- F_GETLK：取得文件锁定的状态。
- F_SETLK：设置文件锁定的状态。此时 flcok 结构的 l_type 值必须是 F_RDLCK、

F_WRLCK 或 F_UNLCK。如果无法建立锁定则返回-1，错误代码为 EACCES 或 EAGAIN。

- F_SETLKW 或 F_SETLK：这两个参数的作用相同，无法建立锁定时，此调用会一直等到锁定动作成功为止。

参数 lock 指针为 flock 类型的结构体指针，用来设置锁定文件的一个区域。flock 结构体的定义方式如下所示。

```
struct flock
{
   short int l_type;         /*文件的锁定状态。*/
   short int l_whence;       /*设定 l_start 位置。*/
   off_t l_start;            /*锁定区域的开头位置，从这个地方开始锁定文件的内容。*/
   off_t l_len;              /*锁定文件区域的大小。*/
   pid_t l_pid;              /*锁定文件的进程。*/
};
```

l_type 有下面三种可选的参数。

- F_RDLCK：建立一个供读取用的锁定。
- F_WRLCK：建立一个供写入用的锁定。
- F_UNLCK：删除之前建立的锁定方式。

l_whence 有下面的三种可选设置方式。

- SEEK_SET：以文件开头为锁定的起始位置。
- SEEK_CUR：以文件当前的读写位置为锁定的起始位置。
- SEEK_END：以文件结尾为锁定的起始位置。

fcntl 函数执行成功则返回 0，失败则返回-1。这个函数可能发生下面的错误，可以用 errno 来捕获发生的错误。

- EACCES 或 EAGAIN：文件已经被锁定，没有权限再进行锁定操作。
- EAGAIN：文件已经被另外一个进程占用。
- EBADF：df 参数无效或文件已经关闭。
- EDEADLK：这个文件被死锁。
- EFAULT：文件处于只读的分区内。
- EINTR：操作被其他的程序或信号中断。
- EINVAL：函数所给的参数不合法。
- EMFILE：已经超过了最多可以打开的文件数。
- ENOLCK：已经建立了太多的锁定方式。
- EPERM：在以追加方式打开的文件上错误的进行锁定。

在使用这个函数时，需要在程序的最前面包含下面的头文件。

```
#include <unistd.h>
#include <fcntl.h>
```

9.7.4 文件锁定函数 fcntl 使用实例

本节讲解 fcntl 函数的使用实例。打开一个文件后，用 fcntl 函数来锁定文件的一部分字节。在程序中需要定义一个 flock 类型的结构体，然后设置这个结构体各个成员的变量。用这个结构体的指针作为 fcntl 函数的参数。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>                                  /*包含头文件。*/

main()
{
   int fd;                                          /*定义打开的文件名。*/
   char path[]="/root/txt1.txt";                    /*定义要打开的文件名字符串。*/
   struct flock fl;                                 /*定义一个 flock 结构体。*/
   char s[]="hello ,Linux.\nI've leart C program for two weeks.\n";
   extern int errno;                                /*定义一个错误号。*/

   fd=open(path,O_WRONLY|O_CREAT);                  /*打开文件。*/
   if(fd!=-1)                                       /*文件打开成功。*/
   {
      printf("opened file %s  .\n",path);           /*输出文件打开成功。*/
   }
   else
   {
      printf("cant't open file %s.\n",path);        /*文件打开失败。*/
       printf("errno: %d\n",errno);                 /*显示错误号。*/
       printf("ERR  : %s\n",strerror(errno));       /*显示错误信息。*/
   }

   fl.l_type=F_RDLCK;                               /*构造 fl 结构体的内容。*/
   fl.l_whence=SEEK_SET;                            /*文件位置方式。*/
   fl.l_start=2;                                    /*从第二个字节开始。*/
   fl.l_len=10;                                     /*一共锁定 10 个字节。*/
   fl.l_pid=15;                                     /*进程号。*/

   if(fcntl(fd,F_SETLKW,&fl)==0)                    /*锁定文件的这个区域。*/
   {
      printf("some string of the file was locked.\n");  /*显示锁定成功。*/
   }
   else
   {
       printf("locked error.\n");                   /*锁定不成功则显示错误。*/
       printf("errno: %d\n",errno);                 /*显示错误号。*/
       printf("ERR  : %s\n",strerror(errno));       /*显示错误信息。*/
    }
    close(fd);                                      /*关闭文件。*/
```

```
}
```

输入下面的命令，编译这个程序。

```
gcc 13.14.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示，表示文件从第二个字节开始以后的 10 个字节已经被锁定。

```
some string of the file was locked.
```

9.8　文件的移动与复制

文件的移动指的是把文件中一个目录中转移到另一个目录中，C 程序提供了方便的文件移动函数。文件的复制指的是将文件作一个备份，C 程序没有提供文件复制函数。需要新建一个文件，从原文件中读取内容写入到新文件中。本节将讲解文件的移动与复制操作。

9.8.1　文件的移动函数 rename

在 Linux 系统中，移动文件有两种方式。一种方式是在同一个分区中移动文件，这种文件移动方式相当于把文件进行重命名。另一种方式是在不同分区之间移动文件。本节只讲前一种文件移动方式。

在同一个分区中移动文件可以用 rename 函数。该函数的使用方式如下所示。

```
int rename(char *oldpath, char *newpath);
```

在参数列表中，oldpath 表示原文件的路径，newpath 表示文件的新路径。rename 函数可以把文件从原路径移动到新路径中。如果文件移动成功将返回 0，不成功返回-1。文件移运不成功时，可能产生下面这些错误。可以用 errno 捕获程序中的错误。

- EACCES：文件的目录不可写或没有可写权限。
- EBUSY：文件被其他程序占用，处于繁忙状态。
- EFAULT：新文件或旧文件处于不可访问的目录中。
- EINVAL：文件名指向的目录不存在。
- EISDIR：旧文件是一个目录，或新文件是一个目录。
- ELOOP：新文件或旧文件有太多的文件或链接相匹配。
- EMLINK：文件目录中的文件已经到了最大数目录。
- ENAMETOOLONG：文件名太长。
- ENOENT：旧文件或新文件不存在。
- ENOMEM：内核内存不足。
- ENOSPC：磁盘的空间不足。

- ENOTDIR：新文件或旧文件指向的目录不存在。
- EROFS：新文件处于只读分区内。
- EXDEV：新文件和旧文件不是在同一个类型的文件分区内。

使用这个函数时，需要在程序的最前面包含下面的头文件。

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

9.8.2 rename 函数使用实例

本节讲述一个实例，使用 rename 函数移动一个文件。将/root/a.txt 文件移到到/tmp 目录中，文件名为 b.txt。在使用这个函数时，需要注意对错误信息的处理。

```
#include <stdio.h>
#include <errno.h>                                  /*包含头文件。*/

main()
{
   char path[]="/root/a.txt";                       /*原文件。*/
   char path1[]="/root/a11.txt";                    /*目录文件。*/
   char newpath[]="/tmp/b.txt";                     /*一个不存在的文件。*/
   extern int errno;                                /*错误号。*/
   if(rename(path,newpath)==0)                      /*移动文件。*/
   {
      printf("the file %s was moved to %s .\n",path,newpath);   /*移动成功。*/
   }
   else
   {
      printf("can't move the file %s .\n",path);    /*显示移动失败。*/
       printf("errno: %d\n",errno);                 /*显示错误号。*/
       printf("ERR  : %s\n",strerror(errno));       /*显示错误信息。*/
   }

   if(rename(path1,newpath)==0)                     /*移动一个不存在的文件。*/
   {
      printf("the file %s was moved to %s .\n",path1,newpath); /*显示结果。*/
   }
   else
   {
      printf("can't move the file %s .\n",path1);   /*显示移动不成功能。*/
       printf("errno: %d\n",errno);                 /*显示错误号。*/
       printf("ERR  : %s\n",strerror(errno));       /*显示错误信息。*/
   }
}
```

输入下面的命令，编译这个程序。

```
gcc 13.15.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。第一次是移动一个已经存在的文件，显示文件移动成功。第二次是移动一个不存在的文件，显示文件移动失败。显示的错误信息是没有这个文件。

```
the file /root/a.txt was moved to /tmp/b.txt .
can't move the file /root/a11.txt .
errno: 2
ERR  : No such file or directory
```

在终端中输入下面的命令，查看/tmp 文件夹下有没有移动的文件。

```
ls /tmp/b.txt
```

显示的结果如下所示。表明/tmp 文件夹下面有程序中移动的文件。

```
/tmp/b.txt
```

9.8.3　文件复制实例

在 C 程序中，没有直接复制一个文件的函数。如果需要复制一个文件，可以分别打开源文件和目标文件。依次从源文件中读取一定长度的内容，然后写入到目标文件中。下面的程序是使用这种方法进行文件复制的实例。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>                              /*包含头文件。*/

main()
{
   int fd,fd2,size,i=1;
   char path[]="/root/s.txt";                   /*要打开的文件。*/
   char newpath[]="/root/a2.txt";               /*复制的目录文件。*/
   char buf[100];                               /*定义一个字符串。*/
   extern int errno;                            /*定义一个错误号。*/

   fd=open(path,O_RDONLY);                      /*打开源文件。*/
   fd2=open(newpath,O_WRONLY|O_CREAT);          /*打开目标文件。*/
   if(fd!=-1)
   {
      printf("opened file %s  .\n",path);       /*源文件打开成功。*/
   }
   else
   {
      printf("cant't open file %s.\n",path);    /*源文件打开失败。*/
       printf("errno: %d\n",errno);             /*显示错误号。*/
       printf("ERR  : %s\n",strerror(errno));   /*显示错误信息。*/
```

```
    }
    for(;i!=0;)
    {
        i=read(fd,buf,sizeof(buf));          /*从源文件中读取内容。*/
        printf("%d",i);                      /*输出读取的字节个数。*/
        printf("%s",buf);                    /*输出内容。*/
        if(i==-1)                            /*到达末尾则结束。*/
        {
           break;                            /*结束循环。*/
        }
        else
        {
           write(fd2,buf,sizeof(buf));       /*将读取的内容写入到目标文件中。*/
        }
    }
    close(fd);                               /*关闭文件。*/
    close(fd2);                              /*关闭文件。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 13.15.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。显示文件已经复制。

```
opened file /root/s.txt  .
file was copied.
```

在终端中输入下面的命令，可以查看到已经复制的文件。

```
ls /root
```

9.9 文件实例：电话本程序

在程序中处理的数据需要记录到文件中才能够长期保存。本章所讲的文件操作可以用来进行各种文件的读写操作。本节将讲述一个文件使用实例，设计一个电话号码管理程序，电话号码的内容使用文件读写函数记录到文件中。在本章的学习中，需要重点理解文件的读写操作。

9.9.1 程序功能分析

在设计一个程序之前，需要对程序的各种功能进行规划，安排程序的各个模块和实现方式。本节将对这个程序的需要和实现方法进行大体分析。

一个电话号码管理程序需要对电话号码进行添加、删除、保存、打开等操作。这个程序需要完成下面这些功能。

（1）进行程序以后，需要有一个选择菜单。

（2）需要添加电话号码，实现数据的输入。

（3）需要删除电话号码，实现数据的管理。

（4）需要进行数据列表，显示所有的电话号码信息。

（5）需要根据一个姓名进行查找，查出这个用户的电话。

（6）要有文件写入功能，把信息保存到文件上。

（7）要有文件读取功能，从文件中读取以前保存的数据。

9.9.2　程序的函数

这个程序实现了复杂的功能，需要把这些功能写成不同的函数模块。这样可以把一个复杂的程序拆分成多个独立的简单模块，然后用主函数把这些程序组织到一起。这个程序需要编写下面这些模块。

- 菜单函数，完成菜单的显示和选择。

```
int menu();
```

- 显示电话函数，参数是一个结构体，显示结构体的信息。

```
void shownum(struct telnum t)
```

- 添加电话函数，完成一个电话号码的添加，返回一个结构体。

```
struct telnum addnum()
```

- 查找函数，用户输入一个姓名，查找到这个用户的电话。

```
void selectbyname()
```

- 删除电话号码函数，用户输入一个姓名，删除这个用户的电话号码。

```
void delenum()
```

- 保存信息功能，将所有电话号码保存到文件上。

```
void savetofile()
```

导入电话号码，从文件中读取以前的文件信息。

```
void loadfromfile()
```

主函数，完成各个函数的调用。

```
main()
```

9.9.3　包含文件

在这个程序中，需要进行字符串比较、字符串复制、错误处理、文件读写等操作。这些操作需要包含相应的头文件。程序的第一部分是包含需要使用的头文件，代码如下所示。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>                                  /*包含相关的头文件。*/
```

9.9.4 数据的定义

程序中的电话号码和姓名之间存在着一对一的对应关系，需要定义一个结构体来体现出这种对应关系。电话号码和计数变量是全局变量，程序中所有的函数需要访问这些数据。程序的这一部分需要定义一个结构体，定义程序的全局变量，代码如下所示。

```
struct telnum
{
   char name[7];                                    /*姓名。*/
   char tel[13];                                    /*电话号码。*/
};                                                  /*定义保存一个电话号码的结构体。*/
struct telnum num[100];                             /*全局变量，一个结构体数组。*/
int i;                                              /*全局变量，当前记录的条数。*/
```

9.9.5 菜单函数

程序中有很多功能，需要把这些功能用一个菜单供用户选择。每一个菜单有一个编号，用户从键盘输入相对应的编号。函数对输入的编号进行判断，如果输入有效则返回这个选项，如果输入的选项为 0 则退出这个程序，如果输入的选项无效则显示错误要求用户重新选择。这个函数的运行流程如图 9-1 所示。

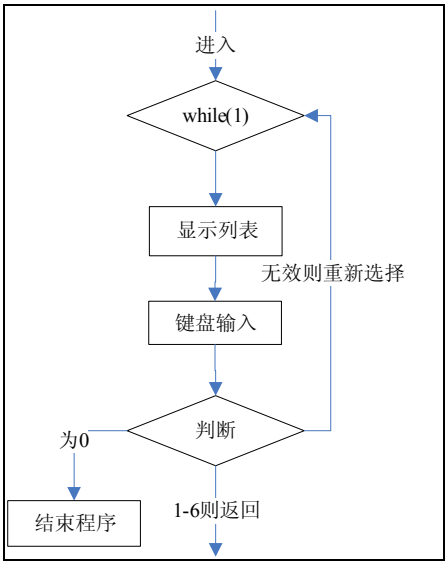

图 9-1 菜单选择的流程图

根据这个菜单选择的流程图编写的程序代码如下所示。

```
int menu()
{
   int i;                                          /*选择的数字。*/
   i=0;                                            /*赋初值为 0。*/
   while(1)
   {
      printf("Please select a menu:\n");           /*提示输入。*/
      printf("      1: add a number.\n");          /*选项。*/
      printf("      2: all the number.\n");
      printf("      3: select a number by name.\n");
      printf("      4: delete a number .\n");
      printf("      5: save to file .\n");
      printf("      6: load numbers from file .\n");
      printf("      0: exit .\n");
      scanf("%d",&i);                              /*输入信息。*/
      if(i==0)                                     /*如果为 0 则退出。*/
      {
         printf("Byebye.\n");
         exit(1);                                  /*退出程序。*/
      }
      if(i<1 || i>6)                               /*不在正确的范围则显示错误。*/
      {
         printf("error.\n");
         continue;                                     /*下次循环。*/
      }
      else
      {
         return(i);                                    /*正常范围则返回这个值。*/
      }
   }
}
```

9.9.6　显示电话信息函数

这个函数的作用是显示一条电话号码的信息，参数是一个电话信息结构体。函数访问这个结构体的成员变量，输出这些信息。完成数据的输出以后，自动结束程序，没有返回值。函数的代码如下所示。

```
void shownum(struct telnum t)
{
    printf("Name   :%s\n",t.name);                 /*显示姓名。*/
    printf("    tel:%s\n",t.tel);                  /*显示电话。*/
}
```

9.9.7　添加电话号码函数

这个函数的作用是提示用户输入电话和姓名信息，然后返回一个电话号码结构体。程序中需要注意的是，这里使用 strcpy 函数将输入的字符串信息复制到结构体的成员中。这个函

数没有参数，返回值是一个结构体。

```
struct telnum addnum()
{
    struct telnum numtmp;                          /*定义一个电话号码结构体变量。*/
    char na[7],tel[13];                            /*定义两个数组。*/
    printf("add a telephone number:\n");           /*提示信息。*/
    printf("please input the name:\n");
    scanf("%s",na);                                /*输入姓名。*/
    printf("please input the num:\n");             /*输入电话号码。*/
    scanf("%s",tel);
    strcpy(numtmp.name,na);                        /*变量复制到结构体的成员上。*/
    strcpy(numtmp.tel,tel);
    return(numtmp);                                /*返回这个结构体。*/
}
```

9.9.8 按姓名查找函数

这个函数的作用是提示用户输入一个姓名并查找此用户的电话号码。函数用 for 循环语句访问结构体数组中的每一个姓名。将每个姓名与当前输入的姓名做比较，如果相同则输出这条记录。程序中使用了一个计数器，如果没有输出任何记录则输出错误提示。

```
void selectbyname()
{
    char na[20];                                   /*定义一个字符串。*/
    int n,j;                                       /*计数变量。*/
    n=0;
    printf("select a number by name:\n");          /*输入姓名。*/
    printf("please input a name :\n");
    scanf("%s",na);                                /*输入。*/
    for(j=0;j<i;j++)                               /*循环访问结构体。*/
    {
        if(strcmp(num[j].name,na)==0)              /*比较输入的变量与结构体中姓名是否相
                                                   同。*/
        {
            shownum(num[j]);                       /*相同则输出。*/
            n++;                                   /*计数加 1。*/
        }
    }
    if(n==0)
    {
        printf("no such a name");                  /*如果 n 为 0 就显示没有这一条记录。*/
    }
}
```

9.9.9 删除电话号码函数

这个函数的作用是从电话号码数组中删除一条电话号码信息。实例的方法是让用户输入的姓名与数组中的每一个姓名进行比较。如果有一条记录中的姓名与输入姓名相同，则将数

组中这条记录后面的所有记录向前移动一个位置。这样便覆盖了需要删除的电话号码。同时，需要将电话号码总数减 1。程序的代码如下所示。

```
void delenum()
{
    char na[20];                                /*定义一个字符串。*/
    int j,n;                                    /*定义计数变量。*/
    n=0;                                        /*计数变量赋初值为 0。*/
    printf("delete a num by name:\n");          /*提示。*/
    printf("please input a name :\n");          /*提示。*/
    scanf("%s",na);                             /*输入一个姓名。*/
    for(j=0;j<i;j++)                            /*循环访问结构体数组中的每一个姓名。*/
    {
        if(strcmp(num[j].name,na)==0)           /*比较输入的姓名与结构体数组中的每一
                                                  个姓名。*/
        {
            n++;                                /*相同则计数加 1。*/
            for(;j<i;j++)                       /*把后面的变量向前移动。*/
           {
              num[j]=num[j+1];
           }
           i--;                                 /*总数减 1。*/
            break;                              /*结束循环。*/
        }
    }
    if(n==0)                                    /*如果计数为 0 表示没有这个姓名。*/
    {
        printf("no such a name");
    }
}
```

9.9.10　保存到文件函数

这个函数的作用是将用户输入的所有电话号码保存到一个文本文件中，函数没有参数和返回值。需要保存的内容都已保存到全局变量中，这个函数可以通过访问全局变量的内容将需要保存的内容保存到一个文本文件中。函数用“open”函数打开一个文件，如果没有文件会自动创建这个文件，然后使用 for 循环访问全局变量数组中的每一个结构体成员，将这些信息依次写入到文件中。函数的代码如下所示。

```
void savetofile()
{
   int j,fd;                                    /*定义变量。*/
   char file[]="/root/tel.txt";                 /*定义文件名。*/
   extern int errno;                            /*设置错误号。*/
   fd=open(file,O_WRONLY|O_CREAT);              /*打开文件。*/
   if(fd!=-1)                                   /*打开正常。*/
   {
     printf("opened file %s  .\n",file);        /*显示正常。*/
   }
```

```
    else                                          /*不能打开文件就显示错误。*/
    {
       printf("cant't open file %s.\n",file);
        printf("errno: %d\n",errno);
        printf("ERR  : %s\n",strerror(errno));
    }
    for(j=0;j<i;j++)                              /*循环访问结构体数组，写入变量。*/
    {
        printf(" %d %s\n",j,num[j].name);         /*显示姓名。*/
       write(fd,num[j].name,7);                   /*保存姓名。*/
       write(fd,num[j].tel,13);                   /*保存电话。*/
    }
    printf("saved.\n");                           /*输出信息。*/
    close(fd);                                    /*关闭文件。*/
}
```

9.9.11 从文件导入信息函数

本函数的作用是从以前保存的文件中读取电话号码和姓名信息，并将这些信息保存到全局变量的数组中。在导入信息时，依次读取若干个字符。如果读取的字符个数不为 0，则在结构体数组中添加一条记录。函数的代码如下所示。

```
void loadfromfile()
{
    int j=0,fd,t;                                 /*定义变量。*/
    i=0;
    char na[7];                                   /*定义姓名。*/
    char tel[13];                                 /*定义电话。*/
    char file[]="/root/tel.txt";                  /*定义文件名。*/
    extern int errno;                             /*设置错误号。*/
    fd=open(file,O_WRONLY|O_CREAT);               /*打开文件。*/
    if(fd!=-1)                                    /*文件打开成功。*/
    {
      printf("opened file %s  .\n",file);
    }
    else                                          /*文件打开失败。*/
    {
      printf("cant't open file %s.\n",file);
       printf("errno: %d\n",errno);
       printf("ERR  : %s\n",strerror(errno));
    }
    while((t=read(fd,na,7))!=0&&t!=-1)             /*读取文件。*/
    {
        strcpy(num[i].name,na);                   /*复制字符串到结构体中。*/
        read(fd,tel,13);                          /*读取电话。*/
        strcpy(num[i].tel,tel);                   /*复制字符串到结构体中。*/
        i++;                                      /*计数加 1。*/
     }
    close(fd);                                    /*关闭文件。*/
```

```
}
```

9.9.12　主函数

主函数的作用是将上面编写的这些函数用一定的关系组织起来，统一完成一个程序功能。在程序开始时，进入一个 while 循环，在循环中反复调用 menu 函数来进行功能选择，然后根据返回的数值来调用相关的子函数。主函数的流程可用图 9-2 来表示。

根据图中的程序运行流程编写的主函数如下所示。

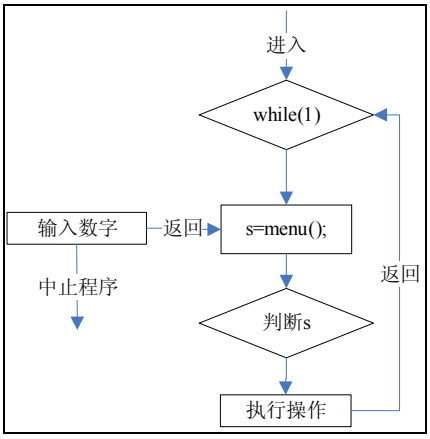

图 9-2　程序主函数的流程图

```
main()
{
    int s,j;                                    /*定义变量。*/
    printf("- - - - Telphone Notebook. - - - \n"); /*输出提示。*/

    while(1)                                    /*进入一个死循环。*/
    {
       s=menu();                                /*显示菜单。*/
       if(s==1)                                 /*分别判断 s 的值，调用不同的函数。*/
      {
          num[i]=addnum();                      /*添加一个电话号码。*/
          i++;                                  /*总数加 1。*/
       }
       if(s==2)                                 /*显示所有记录。*/
       {
          for(j=0;j<i;j++)                      /*循环。*/
          {
             shownum(num[j]);                   /*显示所有记录，结构体作为参数。*/
           }
        }
        if(s==3)                                /*按姓名查找。*/
        {
          selectbyname();
        }
        if(s==4)                                /*删除一个记录。*/
        {
          delenum();
        }
       if(s==5)                                 /*保存到文件。*/
       {
          savetofile();
        }
        if(s==6)                                /*从文件中导入。*/
        {
```

```
            loadfromfile();
          }
      }
  }
```

9.9.13　程序的运行

输入下面的命令，编译这个程序。

```
gcc 13.2.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。程序最开始显示一个选择菜单，要求用户选择一个功能。

```
- - - - Telphone Notebook. - - -
Please select a menu:
      1: add a number.
      2: all the number.
      3: select a number by name.
      4: delete a number .
      5: save to file .
      6: load numbers from file .
      0: exit .
```

用户输入“1”，然后按“Enter”键，提示用户输入姓名和电话。用同样的方法输入三个姓名和电话。这时选择“2”，显示所有的电话，结果如下所示。

```
Name   :jim
   tel:65455676
Name   :Lily
   tel:65746367
Name   :Lucy
   tel:98347834
```

用户选择“4”，然后输入一个姓名，可以删除已经添加的姓名和电话。如果没有这个姓名，程序会显示没有这条记录。

用户选择“5”，程序会把输入的信息保存到文件“/root/tel.txt”中。这时输入下面的命令，查看保存的信息。

```
vim /root/tel.txt
```

VIM 显示的文本信息如下所示。

```
jim65455676Lily65746367Lucy98347834
```

当程序再次启动时，输入“6”，可以从这个文件导入上次保存的信息。输入“0”可以退

出这个程序。

9.10 小结

本章讲解了目录和文件的基本操作。其中需要掌握的知识包括目录的创建与删除、文件的创建与删除、文件的移动与复制、文件的锁定与权限、文件的缓冲区操作和文件的读写。其中文件的读写是本章的难点，需要做大量的编程练习。最后的综合实例讲解了文件在程序中的运用方法，难点是将程序处理的信息保存到文件。文件内容的具体操作，将在下一章中讲到。

第 10 章　文件 I/O

10.1　文件的打开与关闭

打开文件是指在硬盘中找到这个文件，使这个文件处于被调用状态。进行文件读写之前需要进行文件打开操作，文件访问结束以后需要关闭文件。

10.1.1　文件打开函数 fopen

函数 fopen 的作用是打开一个文件，这个函数的使用方法如下所示。

```
FILE * fopen(char * path,char * mode);
```

在参数列表中，path 表示需要打开的文件名字符串。Mode 表示文件打开形态的字符串，这个参数的可能内容如下所示。

- r：打开只读文件，该文件必须存在。
- r+：打开可读写的文件，该文件必须存在。
- w：打开只写文件。若文件存在则文件长度清为零，即该文件内容全部删除。若文件不存在则建立该文件。
- w+：打开可读写文件。若文件存在则文件长度清为零，即该文件内容全部删除。若文件不存在则建立该文件。
- a：以追加的方式打开只写文件。若文件不存在，则会建立该文件。如果文件存在，写入的数据会被加到文件末尾，文件原来的内容保持不变。
- a+：以追加方式打开可读写的文件。若文件不存在，则会建立该文件。如果文件存在，写入的数据会被加到文件末尾。文件原来的内容保持不变。

上述的打开状态描述字符串都可以再加一个 b 字符，如 rb、w+b 或 ab+等组合。加入 b 字符用来告诉函数库打开的文件是二进制文件，而非纯文本文件。

在使用这个文件时，需要包含下面的头文件。

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

如果文件被正常打开，会返回一个 FILE 类型的文件指针。打开失败则返回的内容为 NULL，可用 errno 来捕获所发生的错误。可能返回的错误编号如下所示。

EINVAL：需要的文件或文件的目录不存在。

用 fopen 函数打开一个文件以后，一般会进行读写处理。如果文件打开发生错误，则后面的文件操作都会发生错误，所以需要对文件的打开状态进行判断并及时进行处理。下面是使用 fopen 函数打开文件的实例。

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

main()
{
   FILE * fp;
   extern int errno;
   char file[]="/root/a1.txt";

   fp=fopen(file,"r");
   if(fp==NULL)
   {
      printf("cant't open file %s.\n",file);
      printf("errno: %d\n",errno);
      printf("ERR  : %s\n",strerror(errno));
      return;
    }
    else
    {
        printf("%s was opened.\n",file);
    }
   fclose(fp);

   fp=fopen("/root/a.sh","r");
   if(fp==NULL)
   {
      printf("cant't open file %s.\n",file);
      printf("errno: %d\n",errno);
      printf("ERR  : %s\n",strerror(errno));
    }
    else
    {
        printf("file was opened.\n");
    }
   fclose(fp);

   fp=fopen("/root/a.sh","a+");
   if(fp==NULL)
   {
      printf("cant't open file %s.\n",file);
      printf("errno: %d\n",errno);
      printf("ERR  : %s\n",strerror(errno));
      return;
    }
    else
    {
        printf("file was created.\n");
    }
   fclose(fp);
}
```

输入下面的命令，编译这个程序。

```
gcc 14.1.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。第一次是打开一个已经存在的文件，可以正常打开。第二次是以只读的方式打开一个不存在的文件，发生错误。第三次是用 a+的方式打开一个文件，没有这个文件时会自动新建这个文件。

```
/root/a1.txt  was opened.
cant't open file /root/a.sh.
errno: 2
ERR  : No such file or directory
/root/a.sh  was opened.
```

10.1.2　文件打开函数 fdopen

文件 fdopen 的作用是将 open 函数打开文件时返回的文件打开编号转换成文件指针返回。open 函数打开文件的方法如上一章所述，打开文件以后会返回一个整型的编号。fdopen 函数的使用方法如下所示。

```
FILE * fdopen(int fildes,const char *mode);
```

在参数列表中，fildes 表示 open 函数打开文件以后返回的编号。mode 是一个字符串指针，用来表示文件打开的方式。字符串的内容如上一节中 fopen 函数中的 mode 参数。mode 的参数要与 open 函数打开文件的读写参数相同。如果文件打开成功，则返回这个文件的指针，否则返回一个空指针 NULL。函数可能返回下面的错误信息，可以用 errno 捕获文件打开时的错误。

EINVAL：需要的文件或文件的目录不存在。

用 fdopen 函数打开文件的实例如下所示。

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main()
{
    int fd;
   char path[]="/root/txt1.txt";
   FILE *fp;
   extern int errno;
```

```
    fd=open(path,O_WRONLY|O_CREAT,0766);
    if(fd!=-1)
    {
      printf("opened file %s  .\n",path);
    }
    else
    {
      printf("cant't open file %s.\n",path);
       printf("errno: %d\n",errno);
       printf("ERR  : %s\n",strerror(errno));
    }
    fp =fdopen(fd,"r");
}
```

输入下面的命令，编译这个程序。

```
gcc 14.2.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
opened file /root/txt1.txt
```

10.1.3 打开文件函数 freopen

函数 freopen 的作用是将文件指针以前打开的文件关闭，然后重新打开一个文件。这个函数的使用方法如下所示。

```
FILE * freopen(char * path,char * mode,FILE * stream);
```

在参数列表中，path 表示需要打开文件的字符串，stream 表示原有的文件指针，mode 表示文件的打开方式。调用这个函数时会关闭 stream 指针所打开的文件，然后打开 path 所代表的文件，然后返回一个文件指针。

如果文件打开成功，则返回打开文件的指针，否则返回 NULL，可以用 error 来捕获所发生的错误。下面的实例使用 freopen 函数打开一个文件。

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

main()
{
  FILE *fp;
  char path[]="/root/txt1.txt";
  extern int errno;

  fp=freopen("/root/txt1.txt","r",fp);
```

```
  if(fp==NULL)
  {
      printf("cant't open file %s.\n",path);
      printf("errno: %d\n",errno);
      printf("ERR  : %s\n",strerror(errno));
      return;
   }
   else
   {
      printf("%s was opened.\n",path);
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 14.3.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
/root/txt1.txt was opened.
```

10.1.4　关闭文件函数 fclose

函数 fclose 的作用是关闭已经打开的文件指针。在打开和访问文件以后，需要及时地关闭打开的文件，以释放系统资源。这个函数的使用方法如下所示。

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

参数 stream 表示已经打开的文件指针。如果文件关闭成功则返回整型的 0，如果失败可用 errno 来捕获所发生的错误。下面的实例使用这个函数关闭已经打开的文件。

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

main()
{
   FILE * fp;
   extern int errno;
   char file[]="/root/a1.txt";
   fp=fopen(file,"r");
   if(fp==NULL)
   {
      printf("can't open file %s.\n",file);
      printf("errno: %d\n",errno);
      printf("ERR  : %s\n",strerror(errno));
      return;
```

```
    }
    else
    {
        printf("%s was opened.\n",file);
    }
  if(fclose(fp)==0)
  {
    printf("file was closed.\n");
    }
    else
    {
      printf("can't close file .\n");
      printf("errno: %d\n",errno);
      printf("ERR  : %s\n",strerror(errno));
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 14.4.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示，表示打开文件以后又成功关闭文件。

```
/root/a1.txt was opened.
file was closed.
```

10.2 文件的读写

所谓文件的读写，指的是向已经打开的文件中写入数据，或者从文件中读取数据。本节将讲解文件读写操作，这些操作是通过文件读写函数实现的。

10.2.1 字符写入函数 putc 与 fputc

函数 putc 的作用是将一个字符写入到文本文件中。fputc 的作用与使用方法与 putc 的完全相同。这两个函数的使用方法如下所示。

```
int  putc(int c,FILE *stream);
int fputc(int c,FILE *stream);
```

在参数列表中，c 表示需要写入的字符，stream 表示已经打开的文件指针。如果写入成功，则返回这个字符，返回 EOF 表示写入失败。下面是这个函数的使用实例。

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

main()
```

```
{
   FILE * fp;
   extern int errno;
   char file[]="/root/a1.txt";
   char txt[5]="hello";
   int i=0;
   fp=fopen(file,"a+");
   if(fp==NULL)
   {
      printf("cant't open file %s.\n",file);
      printf("errno: %d\n",errno);
      printf("ERR  : %s\n",strerror(errno));
      return;
    }
    else
    {
        printf("%s was opened.\n",file);
    }


   for(i=0;i<5;i++)
   {
      putc(txt[i],fp);
      printf("%c",txt[i]);
    }
    fclose(fp);
}
```

输入下面的命令，编译这个程序。

```
gcc 14.5.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示，表明已经向这个文件中写入字符。

```
/root/a1.txt was opened.
hello
```

输入下面的命令，查看文件/root/a1.txt 的内容。

```
vim /root/a1.txt
```

可以查看到文件/root/a1.txt 中有下面的文本内容。

```
hello
```

10.2.2　向文件中写入字符串函数 fputs

函数 fputs 的作用是将一个字符串写入到文件中。这个函数的使用方法如下所示。

```
int fputs(char * s,FILE * stream);
```

在参数列表中，s 表示需要写入的字符串，stream 表示已经打开的文件指针。如果写入成功，则会返回实际写入字符的个数，写入不成功时会返回 EOF。下面是这个函数的使用实例。

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

main()
{
   FILE * fp;
   extern int errno;
   char file[]="/root/a1.txt";
   char *txt="hello";
   int i;
   fp=fopen(file,"a+");
   if(fp==NULL)
   {
      printf("cant't open file %s.\n",file);
      printf("errno: %d\n",errno);
      printf("ERR  : %s\n",strerror(errno));
      return;
    }
    else
    {
        printf("%s was opened.\n",file);
    }

    i=fputs(txt,fp);
    {
        printf("%d char was written.\n",i);
    }
    close(fp);
}
```

输入下面的命令，编译这个程序。

```
gcc 14.6.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。程序打开了文件以后，向文件/root/a1.txt 中写入了字符串。

```
/root/a1.txt was opened.
```

输入下面的命令，查看文件/root/a1.txt 的内容。

```
vim /root/a1.txt
```

可以查看到文件/root/a1.txt 中有下面的文本内容。

```
hello
```

10.2.3　数据写入函数 fwrite

函数 fwrite 的作用是将一条或多条数据写入到已经打开的文件中。写入的数据可以是整型、字符、字符串、结构体等内容，这些数据需要存储在一段连续的内存上。这个函数的使用方法如下所示。

```
size_t fwrite(void * ptr,size_t size,size_t nmemb,FILE * stream);
```

在参数列表中，ptr 表示需要写入数据的地址，size 表示每一条数据的字节数，nmemb 表示需要写入多少条数据，stream 表示已经打开的文件指针。如果写入成功，则返回实际写入的数据的字节数，否则返回 EOF。下面是这个函数的使用实例。在程序中定义一个结构体数组，将已经赋值的结构体数组保存到已经打开的文件中。

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
struct stu

{
    char name[10];
    int age;
};

main()
{
   struct stu mystu[3];
   FILE * fp;
   extern int errno;
   char file[]="/root/a1.txt";
   int i;

   strcpy(mystu[0].name,"Jim");
   mystu[0].age=14;
   strcpy(mystu[0].name,"Jam");
   mystu[1].age=14;
   strcpy(mystu[0].name,"Lily");
   mystu[2].age=19;

   fp=fopen(file,"a+");
   if(fp==NULL)
   {
      printf("cant't open file %s.\n",file);
      printf("errno: %d\n",errno);
      printf("ERR  : %s\n",strerror(errno));
      return;
    }
    else
    {
```

```
            printf("%s was opened.\n",file);
        }

        i=fwrite(mystu,sizeof(mystu),3,fp);
        printf("%d bit was written.\n",i);
        close(fp);
    }
```

输入下面的命令，编译这个程序。

```
gcc 14.7.c
```

输入下面的命令，对编译的程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。表明这个程序向文件中写入了 48 个字节。

```
48 bit was written.
```

输入下面的命令，查看文件/root/a1.txt 的内容。

```
vim /root/a1.txt
```

文件/root/a1.txt 中的内容如下所示。

10.3 小结

本章讲述了文件 I/O 操作，重点内容是文件的打开关闭和读写操作。在使用文件打开函数时，需要注意打开方式和权限的设置。使用 putc, fputc, fputs, fwite 等文件读写函数时，需要注意这些函数作用和不同参数的含义。在编程时，可以把程序的结果记录到文件中，实现程序数据的长期保存。

第 11 章　网络编程

11.1　网络编程的基本概念

不同的程序进行网络通信时，是通过 IP 地址和套接字来访问一个主机的。在学习网络编程之前，需要理解网络的一些概念和术语。

11.1.1　IP 地址

IP 地址的作用是标识计算机的网卡地址，每一台计算机都有一个 IP 地址。在程序中是通过 IP 地址来访问一台计算机的。本节将讲述 IP 地址的一些知识。IP 地址是用来标识全球计算机地址的一种符号，就比如一个手机的号码，使用这个地址可以访问一个计算机。作为计算机的统一标识，IP 地址需要有以下特点。

IP 地址具有统一的格式。IP 地址是 32 位长度的二进制数值，存储空间是 4 个字节。这 4 个字节的二进制字符值可以表示一台计算机。

IP 地址可以使用点分十进制来表示。二进制的数值是不便于记忆的，可以把每个字节用一个整数来表示。例如 11000000 10101000 00000001 00000110 是一台计算机的 IP 地址，转换成点分十进制便是 192.168.1.1。

在同一个网络中，IP 地址是唯一的。因为需要根据 IP 地址来访问一台计算机，所以在可以访问的范围以内，每一台计算机的 IP 地址是唯一的。

在终端中输入下面的命令可以查看自己计算机的 IP 信息。

```
ifconfig
```

终端中显示的 IP 信息与网卡信息如下所示。

```
eth0      Link encap:Ethernet  HWaddr 00:0F:EA:45:4E:51
          inet addr:192.168.1.100  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::20f:eaff:fe45:4e51/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6332 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4484 errors:0 dropped:0 overruns:0 carrier:0
          collisions:56 txqueuelen:1000
          RX bytes:6710445 (6.3 MiB)  TX bytes:531858 (519.3 KiB)
          Interrupt:18
```

11.1.2　端口

所谓端口，是指计算机中为了标识在计算机中访问网络的不同程序而设的编号。每一个程序在访问网络时都会分配一个标识符，程序在访问网络或接受访问时，会用这个标识符表

示这一网络数据属于这个程序。这里的端口并非网卡接线的端口，而是不同程序的逻辑编号，并不是实际存在的。

端口号是一个 16 位的无符号整数，对应的十进制取值范围是 0~65535。不同编号范围的端口有不同的作用。低于 256 的端口是系统保留端口号，主要用于系统进程通信。例如网站的 WWW 服务使用的是 80 号端口，FTP 服务使用的是 21 号端口。不在这一范围的端口号是自由端口号，在编程时可以调用这些端口号。

11.1.3　域名

域名是用来代替 IP 地址来标识计算机的一种直观名称。例如百度网站的 IP 地址是 202.108.22.43，这个 IP 地址没有任何逻辑含义，是不便于记忆的。而 www.baidu.com 是一个便于记录的名称，用于代替这个 IP 地址。在访问计算机时，可以用这个域名来代替 IP 地址。

在 C 语言编程时，有时需要用域名来访问一个计算机，这时要将域名转换成相应的 IP 地址。在终端中，可以用 ping 命令来查看一个域名所对应的 IP 地址。例如可以输入下面的命令来查看百度的 IP 地址。

```
ping www.baidu.com
```

终端中显示的结果如下所示。表明百度的 IP 地址是 202.108.22.43。

```
PING www.a.shifen.com (202.108.22.43) 56(84) bytes of data.
64 bytes from xd-22-43-a8.bta.net.cn (202.108.22.43): icmp_seq=2 ttl=58
time=33.9 ms
```

11.1.4　TCP 与 UDP

TCP 与 UDP 是两种不同的网络传输方式。两个不同计算机中的程序，使用 IP 地址和端口，要使用一种约定的方法进行数据传输。TCP 与 UDP 就是网络中的两种数据传输约定，主要的区别是进行数据传输时是否进行连接。

- TCP：TCP 是一种面向连接的网络传输方式。这种方式可以理解为打电话。计算机 A 先呼叫计算机 B，计算机 B 接受连接后发出确认信息，计算机 A 收到确认信息以后发送信息，计算机 B 完成数据接收以后发送完毕信息，这时再关闭数据连接。所以 TCP 是面向连接的可靠的信息传输方式。这种方式是可靠的，缺点是传输过程复杂，需要占用较多的网络资源。
- UDP：UDP 是一种不面向连接的传输方式。可以简章理解成邮寄信件。将信件封装放入邮筒以后，不再参预邮件的传送过程。使用 UDP 传送信息时，不建立连接，直接把信息发送到网络上，由网络完成信息的传送。信息传递完成以后也不发送确认信息。这种传输方式是不可靠的，但是有很好的传输效率。对传输可靠性要求不高时，可以选择使用这种传输方式。

11.2　套接字

套接字（Socket）的本义是插座，在网络中用来描述计算机中不同程序与其他计算机程序

的通信方式。本节将讲述套接字的含义与数据类型。

11.2.1 什么是套接字

程序访问网络进行数据通信时，TCP 和 UDP 会遇到同时为多个应用程序并发进行通信的问题。多个 TCP 连接或多个应用程序进程可能需要通过同一个 TCP 协议端口传输数据。为了区别不同的应用程序的进程和连接，需要使用应用程序与 TCP / IP 协议交互的套接字(Socket)的接口。

区分不同应用程序进程间的网络通信和连接，主要使用 3 个参数。通信的目的 IP 地址、使用的传输层协议（TCP 或 UDP）和使用的端口号。在编程时，就是使用这三个参数来构成一个套接字。这个套接字相当于一个接口，可以进行不同计算机程序的信息传输。

11.2.2 套接字相关的数据类型

C 程序进行套接字编程时，常会使用到 sockaddr 数据类型或 sockaddr_in 数据类型。这两种数据类型是系统中定义的结构体，用于保存套接字的信息。

sockaddr 用来保存一个套接字，定义方法如下所示。

```
struct sockaddr
{
   unsigned short int sa_family;
   char sa_data[14];
};
```

在这个结构体中，成员的含义如下所示。

- sa_family：指定通信的地址类型。如果是 TCP/IP 通信，则该值为 AF_INET。
- sa_data：最多使用 14 个字符长度，用来保存 IP 地址和端口信息。
- sockaddr_in 的功能与 soockaddr 相同，也是用来保存一个套接字的信息。不同的是将 IP 地址与端口分开为不同的成员。这个结构体的定义方法如下所示。

```
struct socketaddr_in
{
   unsigned short int sin_family;
   uint16_t sin_port;
   struct in_addr sin_addr;
   unsigned char sin_zero[8];
};
```

这个结构体的成员与作用如下所示。

- sin_family：与 sockaddr 结构体中的 sa_family 相同。
- sin_port：套接字使用的端口号。
- sin_addr：需要访问的 IP 地址。
- sin_zero：未使用的字段，填充为 0。

在这一结构体中，in_addr 也是一个结构体，定义方法如下所示。作用是保存一个 IP 地址。

```
struct in_addr
```

```
{
   uint32_t s_addr;
};
```

这两个套接字结构体的保存内容是相同的。可以用数据类型转换的方法来转换这两个类型的变量。

11.2.3　套接字类型

套接字类型指的是在网络通信中不同的数据传输方式。例如 UDP 和 TCP 就是两种不同的套接字类型。常用的套接字类型有下面 3 种。

- 流套接字（SOCK_STREAM）：流套接字使用了面向连接的可靠的数据通信方式，即 TCP（The Transmission Control Protocol）协议。这种服务可以实现无差错、无重复发送、并按顺序接收。
- 数据报套接字（Raw Sockets）：数据报套接字使用了不面向连接的数据传输方式，即 UDP（User Datagram Protocol）协议。这种协议在数据发送出去以后，就完成通信的任务。然后，完全依靠网络来完成数据传输。网络不能完全保证数据正确传输，也不能保证数据按顺序接收。这种套接字不能保证数据传输的可靠性，可能在通信中出现数据丢失的情况，需要在程序中作出相应的处理。
- 原始套接字（SOCK_RAW）：前面讲述的两种套接字是系统定义的，所有的信息都需要按照这种方式进行封装。原始套接字是没有经过处理的 IP 数据包，可以根据自己程序的要求进行封装。如果要访问其他的协议，需要使用原始套接字来构造相应的协议。

11.3　域名与 IP 地址

在网络编程时，知道域名是不能直接访问一个主机的，需要转换成相应的 IP 地址。但是有时在程序中也需要将一个 IP 地址转换成一个域名。本节将讲解 C 程序中的 IP 地址与域名的转换问题。

11.3.1　用域名取得主机的 IP 地址

域名是为了便于记忆来代替 IP 地址访问网络的方法。在使用域名访问网络时，需要将这个域名转换成相对应的 IP 地址。用域名返回地址的函数是 gethostbyname。这个函数的使用方法如下所示。

```
struct hostent *gethostbyname(const char *name);
```

在参数列表中，name 表示一个域名的字符串。函数会把这个域名转换成一个主机地址结构体返回。结构体 hostent 的定义方法如下所示。

```
struct hostent
{
   char  *h_name;
   char **h_aliases;
   int    h_addrtype;
```

```
    int    h_length;
    char **h_addr_list;
}
```

这个结构体成员含义如下所示。

- h_name：正式的主机名称。
- h_aliases：这个主机的别名。
- h_addrtype：主机名的类型。
- h_length：地址的长度。
- addr_list：从域名服务器取得的主机地址。

在解析域名时，可能发生没有这个域名或域名服务器的错误。可能返回的错误信息如下所示，可以用 error 来捕获错误编号。

- HOST_NOT_FOUND：主机没有找到。
- NO_ADDRESS or NO_DATA：没有 IP 地址或没有数据。
- NO_RECOVERY：域名服务器发生错误。
- TRY_AGAIN：请稍候再重试。

下面的实例是解析域名 www.163.com 的 IP 地址，并且显示出相关信息的实例。

```
#include <stdio.h>
#include <sys/socket.h>
#include <netdb.h>                                  /*包含相关的头文件。*/

int   main(int argc,char *argv[])
{
   struct hostent *host;                            /*定义 hostent 结构体指针。*/
   char hostname[]="www.163.com";                   /*域名。*/
   char hostname2[]="www.1d5r6f.com";               /*一个不存在的域名。*/
   struct in_addr in;                               /*结构体。*/
   struct sockaddr_in addr_in;
   extern int h_errno;                              /*错误号。*/

   if((host=gethostbyname(hostname))!=NULL)         /*取得主机地址。*/
   {
      memcpy(&addr_in.sin_addr.s_addr,host->h_addr,4);      /*复制地址。*/
      in.s_addr=addr_in.sin_addr.s_addr;
      printf("Domain name: %s \n",hostname);                /*输出主机名。*/
      printf("IP length:    %d\n",host->h_length);          /*地址的长度。*/
      printf("Type:    %d\n",host->h_addrtype);             /*地址的种类。*/
      printf("IP         : %s \n",inet_ntoa(in));          /*IP 地址。*/
    }
    else                                                   /*出错的处理。*/
    {
        printf("Domain name: %s \n",hostname);
        printf("error: %d\n",h_errno);
         printf("%s\n",hstrerror(h_errno));
   }

   if((host=gethostbyname(hostname2))!=NULL)        /*解析一个不存在的地址。*/
```

```
    {
       memcpy(&addr_in.sin_addr.s_addr,host->h_addr,4);
       in.s_addr=addr_in.sin_addr.s_addr;
       printf("Domain name: %s \n",hostname2);
       printf("IP         : %s \n",inet_ntoa(in));
       printf("IP length:    %d\n",host->h_length);
       printf("Type:    %d\n",host->h_addrtype);
    }
    else                                                    /*输出错误信息。*/
    {
        printf("Domain name: %s \n",hostname2);
        printf("error: %d\n",h_errno);
         printf("%s\n",hstrerror(h_errno));
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 11.1.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。程序解析了第一个域名的 IP，并显示了 IP 地址和地址长度。第 2 个域名不存在，显示出错误信息。

```
Domain name: www.163.com
IP length:    4
Type:    2
IP         : 202.108.9.31
Domain name: www.1d5r6f.com
error: 1
Unknown host
```

11.3.2　用 IP 地址返回域名

用一个 IP 地址可以查询到这个 IP 的域名，需要使用的函数是 gethostbyaddr。这个函数的使用方法如下所示。

```
struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);
```

在参数列表中，addr 表示一个保存了 IP 地址的字符串，Len 表示这个 IP 地址的长度，type 的值一般为 AF_INET。函数的返回值是 hostent 类型的指针，这一指针的定义和上一节的相同。如果转换失败，则返回 null 指针。下面的实例使用了 gethostbyaddr 函数查找一个 IP 所对应的域名，其中使用的 IP 地址是央视国际的网站。

```
#include <stdio.h>
#include <sys/socket.h>
```

```
#include <netdb.h>                                              /*包含相关的头文件。*/

int  main(int argc,char *argv[])
{
   struct hostent *host;                                        /*主机名结构体。*/
   char addr[]="202.108.249.216";                               /*定义 IP 地址。*/
   struct in_addr in;                                           /*地址结构体。*/
   struct sockaddr_in addr_in;
   extern int h_errno;

   if((host=gethostbyaddr(addr,sizeof(addr),AF_INET))!=(struct hostent *)NULL)
   {
      memcpy(&addr_in.sin_addr.s_addr,host->h_addr,4); /*输出主机名的信息。*/
      in.s_addr=addr_in.sin_addr.s_addr;
      printf("Domain name: %s \n",host->h_name);
      printf("IP length:    %d\n",host->h_length);
      printf("Type:    %d\n",host->h_addrtype);
      printf("IP         : %s \n",inet_ntoa(in));
    }
    else                                                        /*出错的处理。*/
    {
        printf("error: %d\n",h_errno);
         printf("%s\n",hstrerror(h_errno));
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 11.2.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
Domain name: www.cctv.com
IP length:    4
Type:    2
IP         : 202.108.249.216
```

11.4 网络协议

所谓网络协议，是指不同的计算机、不同的操作系统在进行网络通信时的统一约定。在进行网络编程时，需要遵守这些约定。本节将讲述编程中如何取得协议信息。

11.4.1　由协议名取得协议数据

每一种协议，如 TCP、UDP 等，是协议的名称。在编程时，需要用这些协议名称取得这些协议的数据。函数 getprotobyname 的作用是按名称取得一个协议的数据。这个函数的使用方法如下所示。

```
struct protoent *getprotobyname(char *name);
```

参数 name 表示一个协议名称字符串。返回值是一个 protoent 的结构体指针。结构体 protoent 的定义方法如下所示。

```
struct protoent
{
   char  *p_name;
   char **p_aliases;
   int    p_proto;
}
```

这个结构体的成员含义如下所示。

- p_name：协议的名称。
- p_aliases：协议的别名。
- p_proto：协议的序号。

使用这个函数之前，需要在程序中包含下面的头文件。

```
#include <netdb.h>
```

下面的实例使用了这个函数取得几种常用协议的数据。

```
#include <stdio.h>
#include <netdb.h>                                  /*包含相关的头文件。*/

int  main()
{
    struct protoent *pro;                           /*定义一个表示协议的结构体。*/
    pro=getprotobyname("icmp");                     /*取得 icmp 协议的信息。*/
    printf("protocol name : %s\n",pro->p_name);          /*名称。*/
    printf("protocol number : %d\n",pro->p_proto);       /*协议号。*/
    printf("protocol alias: %s\n",pro->p_aliases[0]);    /*别名。*/
    pro=getprotobyname("udp");                           /*取得 udp 协议的信息。*/
    printf("protocol name : %s\n",pro->p_name);
    printf("protocol number : %d\n",pro->p_proto);
    printf("protocol alias: %s\n",pro->p_aliases[0]);
    pro=getprotobyname("tcp");                           /*取得 tcp 协议的信息。*/
    printf("protocol name : %s\n",pro->p_name);
    printf("protocol number : %d\n",pro->p_proto);
    printf("protocol alias: %s\n",pro->p_aliases[0]);
}
```

输入下面的命令，编译这个程序。

```
gcc 11.3.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
protocol name : icmp
protocol number : 1
protocol alias: ICMP
protocol name : udp
protocol number : 17
protocol alias: UDP
protocol name : tcp
protocol number : 6
protocol alias: TCP
```

11.4.2 由协议编号取得协议信息

知道了一个协议的编号，可以用 getprotobynumber 函数来取得这个协议的信息。这个函数的使用方法如下所示。

```
struct protoent *getprotobynumber(int proto);
```

在参数列表中，proto 表示一个协议的编号。函数会返回这一个协议的结构体指针 protoent。结构体 protoent 的定义方法与上一节相同。在使用这个函数以前，需要在程序的最前面包含下面的头文件。

```
#include <netdb.h>
```

下面的实例使用了 getprotobynumber 函数取得 0~4 号协议的信息。

```
#include <stdio.h>
#include <netdb.h>                              /*包含相关的头文件。*/

int  main()
{
   struct protoent *pro;                        /*定义一个表示协议的结构体指针。*/
   int i;

   for(i=0;i<5;i++)                             /*循环。*/
   {
   pro=getprotobynumber(i);                     /*取得协议信息。*/
   printf("protocol name : %s\n",pro->p_name);          /*协议名。*/
   printf("protocol number : %d\n",pro->p_proto);       /*协议号。*/
   printf("protocol alias: %s\n",pro->p_aliases[0]);    /*别名。*/
   }
}
```

输入下面的命令，编译这个程序。

```
gcc 11.4.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
protocol name : ip
protocol number : 0
protocol alias: IP
protocol name : icmp
protocol number : 1
protocol alias: ICMP
protocol name : igmp
protocol number : 2
protocol alias: IGMP
protocol name : ggp
protocol number : 3
protocol alias: GGP
protocol name : ipencap
protocol number : 4
protocol alias: IP-ENCAP
```

11.4.3　取得系统支持的所有协议

函数 getprotoent 的作用是取得系统中所有可以支持的协议。这个函数的使用方法如下所示。

```
struct protoent *getprotoent(void);
```

这个函数没有参数，每次调用时依次返回一个系统支持的协议。返回的类型是 protoent 结构体指针。protoent 结构体指针如 11.4.1 节所述。到达末尾时会返回 NULL 指针。

系统中支持协议的类型是记录在/etc/protocols 文件中的。可以输入下面的命令查看这个文件中支持协议的内容。

```
vim /etc/protocols
```

下面的实例使用了 getprotoent 函数来查看系统支持协议的类型。在程序中，把返回的值是否为 NULL 作为循环的条件，然后输出这些协议的信息。

```
#include <stdio.h>
#include <netdb.h>                                  /*包含头文件。*/

int  main()
{
   struct protoent *pro;                            /*定义一个表示协议的结构体。*/
   while(pro=getprotoent())                         /*循环取得一个协议。*/
   {printf("protocol name : %s  ",pro->p_name);
```

```
    printf("protocol number : %d  ",pro->p_proto);
    printf("protocol alias: %s\n",pro->p_aliases[0]);
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 11.5.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
protocol name : ip  protocol number : 0  protocol alias: IP
protocol name : hopopt  protocol number : 0  protocol alias: HOPOPT
protocol name : icmp  protocol number : 1  protocol alias: ICMP
protocol name : igmp  protocol number : 2  protocol alias: IGMP
protocol name : ggp  protocol number : 3  protocol alias: GGP
protocol name : ipencap  protocol number : 4  protocol alias: IP-ENCAP
protocol name : st  protocol number : 5  protocol alias: ST
protocol name : tcp  protocol number : 6  protocol alias: TCP
protocol name : cbt  protocol number : 7  protocol alias: CBT
protocol name : egp  protocol number : 8  protocol alias: EGP
protocol name : igp  protocol number : 9  protocol alias: IGP
protocol name : bbn-rcc  protocol number : 10  protocol alias: BBN-RCC-MON
protocol name : nvp  protocol number : 11  protocol alias: NVP-II
protocol name : pup  protocol number : 12  protocol alias: PUP
protocol name : argus  protocol number : 13  protocol alias: ARGUS
protocol name : emcon  protocol number : 14  protocol alias: EMCON
protocol name : xnet  protocol number : 15  protocol alias: XNET
protocol name : chaos  protocol number : 16  protocol alias: CHAOS
protocol name : udp  protocol number : 17  protocol alias: UDP
protocol name : mux  protocol number : 18  protocol alias: MUX
protocol name : dcn  protocol number : 19  protocol alias: DCN-MEAS
protocol name : hmp  protocol number : 20  protocol alias: HMP
```

11.5　网络服务

所谓网络服务，指的是网络上的计算机通过运行程序为其他的计算机提供信息或运算的功能。例如打开一个网页是访问了网站服务器上的服务，从服务器下载了网页文件。本节将讲解 C 程序中的网络服务类型。

11.5.1　取得系统支持的网络服务

函数 servent 的作用是取得系统所支持的服务。这个函数的使用方法如下所示。

```
struct servent *getservent(void);
```

这个函数没有参数，返回值是一个 servent 类型的结构体指针。结构体 servent 的定义方法如下所示。

```
struct servent
{
   char  *s_name;
   char **s_aliases;
   int    s_port;
   char  *s_proto;
}
```

这个结构体的成员，含义如下所示。

- s_name：这个服务的名称。
- s_aliases：这个服务可能的别名。
- s_port：这个服务可能的端口。
- s_proto：这个服务使用的协议。

每次调用这个服务时，会返回一个系统支持的服务。如果已经到达最后一个结果则会返回 NULL。可以输入下面的命令，查看系统文件列表中支持的服务。

```
vim /etc/services
```

下面的实例使用了 getservent 函数来返回系统中所有的服务。需要注意的是输出的服务端口需要用 ntohs 函数进行转换。

```
#include <stdio.h>
#include <netdb.h>                                  /*包含相关的头文件。*/

int  main()
{
    struct servent *ser;                            /*定义一个表示服务的结构体指针。*/
    while( ser=getservent())                        /*取得一个服务。*/
    {
       printf("name : %s  ",ser->s_name);           /*服务名。*/
       printf("port : %d  ",ntohs(ser->s_port));/*服务端口。*/
       printf("protocol:%s  ",ser->s_proto);            /*协议。*/
       printf("alias: %s\n",ser->s_aliases[0]);         /*别名。*/
```

```
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 11.6.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

下面是程序运行结果的一部分。while 循环多次调用 getservent 函数返回了系统支持的服务。

```
name : tcpmux  port : 1  protocol:tcp  alias: (null)
name : tcpmux  port : 1  protocol:udp  alias: (null)
name : rje  port : 5  protocol:tcp  alias: (null)
name : rje  port : 5  protocol:udp  alias: (null)
name : echo  port : 7  protocol:tcp  alias: (null)
name : echo  port : 7  protocol:udp  alias: (null)
name : discard  port : 9  protocol:tcp  alias: sink
name : discard  port : 9  protocol:udp  alias: sink
name : systat  port : 11  protocol:tcp  alias: users
name : systat  port : 11  protocol:udp  alias: users
name : daytime  port : 13  protocol:tcp  alias: (null)
name : daytime  port : 13  protocol:udp  alias: (null)
name : qotd  port : 17  protocol:tcp  alias: quote
name : qotd  port : 17  protocol:udp  alias: quote
name : msp  port : 18  protocol:tcp  alias: (null)
name : msp  port : 18  protocol:udp  alias: (null)
name : chargen  port : 19  protocol:tcp  alias: ttytst
name : chargen  port : 19  protocol:udp  alias: ttytst
name : ftp-data  port : 20  protocol:tcp  alias: (null)
name : ftp-data  port : 20  protocol:udp  alias: (null)
name : ftp  port : 21  protocol:tcp  alias: (null)
```

11.5.2 用名称取得系统所支持的服务

函数 getservbyname 可以用一个服务的名称来取得一个服务。这个函数的使用方式如下所示。

```
struct servent *getservbyname(char *name, char *proto);
```

在参数列表中，name 表示一个服务的名称，proto 表示某服务使用的协议。函数 getservbyname 可以查出参数所对应的系统服务返回一个 servent 指针。servent 的定义方法如下所示。如果没有查询到这个服务，会返回 NULL 指针。在使用这个函数之前，需要在程序的最前面包含下面的头文件。

```
#include <netdb.h>
```

下面的实例使用了 getservbyname 函数通过名称来取得一个系统服务。

```
#include <stdio.h>
#include <netdb.h>                                  /*包含相关的头文件。*/

int  main()
{
    struct servent *ser;                            /*定义一个表示服务的结构体指针。*/

    if( ser=getservbyname("ftp","tcp"))             /*获取 FTP 服务的信息。*/
    {
      printf("name : %s\n",ser->s_name);            /*服务名称。*/
      printf("port : %d  ",ntohs(ser->s_port));     /*服务端口号。*/
      printf("protocol:%s\n",ser->s_proto);         /*服务协议。*/
      printf("alias: %s\n",ser->s_aliases[0]);      /*别名。*/
    }
    else
    {
      printf("there is no such a service.\n");      /*错误处理。*/
    }

    if( ser=getservbyname("http","tcp"))            /*取得 http 服务的信息。*/
    {
      printf("name : %s\n",ser->s_name);            /*输出信息。*/
      printf("port : %d  ",ntohs(ser->s_port));     /*端口号。*/
      printf("protocol:%s\n",ser->s_proto);         /*协议。*/
      printf("alias: %s\n",ser->s_aliases[0]);      /*别名。*/
    }
    else
    {
      printf("there is no such a service.\n");      /*出错处理。*/
    }

    if( ser=getservbyname("asdasd","tcp"))          /*取得一个不存在的服务。*/
    {
      printf("name : %s\n",ser->s_name);            /*输出服务的信息。*/
      printf("port : %d  ",ntohs(ser->s_port));
      printf("protocol:%s\n",ser->s_proto);
      printf("alias: %s\n",ser->s_aliases[0]);
    }
    else
    {
     printf("there is no such a service.\n");      /*出错处理。*/
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 11.7.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
name : ftp
port : 21  protocol:tcp
alias: (null)
name : http
port : 80  protocol:tcp
alias: www
there is no such a service.
```

11.5.3 由端口取得服务名称

函数 getservbyport 可以从一个端口取得一个服务的信息。这个函数的使用方法如下所示。

```
struct servent *getservbyport(int port, char *proto);
```

在参数列表中，port 表示一个端口的编号。需要注意的是这个端口号需要用 ntohs 函数进行转换。Proto 表示一个协议的字符串。函数会返回这个端口服务的 servent 类型的指针。servent 的结构体定义方法如 11.5.1 节所示。

下面的实例使用了 getservbyport 函数将一个端口号转换成一个服务类型。

```
#include <stdio.h>
#include <netdb.h>                                  /*包含相关的头文件。*/

int  main()
{
   struct servent *ser;                             /*定义一个表示服务的结构体指针。*/

   if( ser=getservbyport(htons(23),"tcp"))          /*取得 23 号端口服务的信息。*/
   {
      printf("name : %s\n",ser->s_name);            /*输出服务的信息。*/
      printf("port : %d\n",ntohs(ser->s_port));
      printf("protocol:%s\n",ser->s_proto);
      printf("alias: %s\n",ser->s_aliases[0]);
   }
   else                                             /*没有这一个服务时的处理。*/
   {
      printf("there is no such a service.\n");
   }

   if( ser=getservbyport(htons(80),"tcp"))          /*取得 80 号端口的服务。*/
   {
      printf("name : %s\n",ser->s_name);                /*名称。*/
      printf("port : %d\n",ntohs(ser->s_port));         /*端口号。*/
      printf("protocol:%s\n",ser->s_proto);             /*协议名称。*/
      printf("alias: %s\n",ser->s_aliases[0]);          /*别名。*/
```

```
    }
    else
    {
       printf("there is no such a service.\n");      /*出错处理。*/
    }

    if( ser=getservbyport(htons(61111),"tcp"))       /*取得一个未知端口的服务。*/
    {
       printf("name : %s\n",ser->s_name);            /*输出服务的信息。*/
       printf("port : %d\n",ntohs(ser->s_port));
       printf("protocol:%s\n",ser->s_proto);
       printf("alias: %s\n",ser->s_aliases[0]);
    }
    else
    {
       printf("there is no such a service.\n");      /*出错处理。*/
    }
 }
```

输入下面的命令，编译这个程序。

```
gcc 11.8.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
name : telnet
port : 23
protocol:tcp
alias: (null)
name : http
port : 80
protocol:tcp
alias: www
there is no such a service.
```

11.6　网络 IP 地址的转换

网络 IP 地址本是用 32 位二进制来表示的，为了记忆的方便可以用点分十进制来表示 IP 地址。同时，网络 IP 地址在网络传输与计算机内部的字符存储的方式是不同的，需要用相关函数将网络 IP 地址进行转换。

11.6.1　将网络地址转换成长整型

函数 inet_addr 可以将一个网络 IP 地址转换成一个十进制长整型数。这个函数的使用方法

如下所示。

```
long inet_addr(char *cp);
```

函数的参数 cp 表示一个 IP 地址字符串。函数会将这个 IP 地址转换成为一个长整型数。使用这个函数之前，需要在程序中包含下面的头文件。

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

下面的实例使用了这个函数将一个 IP 地址转换成一个长整型数进行输出。

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>                          /*包含相关的头文件。*/

int  main()
{
    char cp[]="192.168.1.1";                    /*IP 地址。*/
    printf("%ld\n",inet_addr(cp));              /*转换成长整型 IP 地址。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 11.9.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
16885952
```

11.6.2 将长整型 IP 地址转换成网络地址

函数 inet_ntoa 可以将一个整型 IP 地址转换成一个点分十进制网络 IP 地址。这个函数的使用方法如下所示。

```
char *inet_ntoa(struct in_addr in);
```

函数的参数 in 是一个 in_addr 类型的结构体。这个结构体的定义方法如下所示。

```
struct in_addr
{
  uint32_t s_addr;
};
```

结构体只有一个成员，s_addr 是一个长整型数，用来存储一个长整型的 IP 地址。函数

inet_ntoa 会将这个 IP 地址换成一个字符串返回。使用这个函数之前，需要在程序的最前面包含下面的头文件。

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

下面的程序是这个函数的实例。将上一节 IP 地址转换生成的长整型数转换成为一个 IP 地址。

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>                                  /*包含相关的头文件。*/

int  main()
{
   struct in_addr ip;                                   /*定义一个地址结构体。*/
   ip.s_addr=16885952;                                  /*长整型的 IP 地址。*/
   printf("%s\n",inet_ntoa(ip));                        /*转换以后输出。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 11.10.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
192.168.1.1
```

11.6.3　主机字符顺序与网络字符顺序的转换

计算机中的字符与网络中的字符的存储顺序是不同的。计算机中的整型数与网络中的整型数进行交换时，需要用相关的函数进行转换。如果将计算机中的长整型 IP 地址转换成网络字符顺序的整型 IP 地址，使用 htonl 函数。这些函数如下所示。

- uint32_t htonl(uint32_t hostlong);
- uint16_t htons(uint16_t hostshort);
- uint32_t ntohl(uint32_t netlong);
- uint16_t ntohs(uint16_t netshort);

这些函数的使用如下所示。

- htonl：将计算机中的 32 位长整型数转换成网络字符顺序的 32 位长整型数。
- htons：将计算机中的 16 位整型数转换成网络字符顺序的 16 位整型数。
- ntohl：将网络字符顺序的 32 位长整型数转换成计算机中的 32 位长整型数。

- ntohs：将网络字符顺序的 16 位整型数转换成计算机中的 16 位整型数。

这些函数的参数都表示需要转换的整型数，函数把这些整型数转换以后返回。使用这个函数之前，需要在程序的最前面包含下面的头文件。

```
#include <arpa/inet.h>
```

下面的实例使用了这函数进行相关的数值转换。

```
#include <stdio.h>
#include <arpa/inet.h>

int  main()
{
   long local;
   int port;
   local =123456;
   port=1024;
   printf("net: %d\n",htonl(local));                    /*转换成网络字节顺序。*/
   printf("net: %d\n",htons(port));
   printf("local: %d\n",ntohl(htonl(local)));           /*转换成本地字节顺序。*/
   printf("local: %d\n",ntohs(htons(port)));
}
```

输入下面的命令，编译这个程序。

```
gcc 11.11.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

- net: 1088553216
- net: 4
- local: 123456
- local: 1024

11.7　错误处理

网络程序中可能出现网络故障、域名无法解析、主机不响应等错误，因此在编程时需要对这些错误进行处理。本节将讲述程序中的错误处理方法。

11.7.1　herror 函数显示错误

函数 herror 可以显示上一个网络函数发生的错误。这个函数的使用方法如下所示。

```
void herror(const char *s);
```

这个函数的参数 s 表示一个字符串。在调用这个函数时，会先输出这个字符串，然后在输出错误信息。输出的错误信息是上一个与网络相关的函数发生的错误，所以这个函数可以在程序中随意使用。使用这个函数时，需要在程序的最前面包含下面的头文件。

```
#include <netdb.h>
```

下面的实例使用 herror 函数输出网络程序中的错误。

```
#include <stdio.h>
#include <netdb.h>                              /*包含头文件。*/

int  main()
{
    char err[]="err:";
    herror(err);                                /*输出错误。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 11.12.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。因为程序中没有调用网络函数，程序输出的结果是没有错误。

```
err:Resolver Error 0 (no error)
```

11.7.2　捕获错误编号

网络程序中，可以使用下面的语句来捕获发生错误的编号。

```
extern int h_errno;
```

捕获这个错误编号以后，可以用 hstrerror 函数来输出这个错误信息。这个函数的使用方法如下所示。

```
char *hstrerror(int err);
```

这个函数的参数 err 表示已经捕获的错误编程，函数会返回这个编号所对应的错误信息字符串。在使用这个函数时，需要在程序的最前面包含下面的头文件。

```
#include <netdb.h>
```

下面的实例使用这个函数输出 0 号到 5 号的网络错误编号的错误信息。

```
#include <stdio.h>
#include <netdb.h>                                    /*包含相关的头文件。*/

int  main()
{
    int i;
    for(i=0;i<6;i++)
    {
       printf("%d : %s \n",i,hstrerror(i));          /*按序号输出一个错误。*/
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 11.13.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

- 0 : Resolver Error 0 (no error)
- 1 : Unknown host
- 2 : Host name lookup failure
- 3 : Unknown server error
- 4 : No address associated with name
- 5 : Unknown resolver error

11.8 小结

本章主要讲述了 IP 地址、域名、网络套接字、端口、服务等概念和相关的处理，其中套接字的理解和网络编程中常用的结构体是本章的重点。在编程中，常常需要使用到服务、地址、主机等结构体，需要理解这些结构体的作用与使用方法。最后一节的错误处理是网络编程时通用的错误处理方法，用这些错误处理函数可以方便地进行程序跟踪和调试。

第 12 章　无连接的套接字通信

12.1　socket 套接字

所谓 socket 套接字，指的是在网络通信前建立的通信接口。进行网络连接前，需要向系统注册申请一个新的 socket，然后使用这个 socket 进行网络连接。本章将讲解 socket 的建立与访问操作。

12.1.1　建立 socket

在进行网络连接前，需要用 socket 函数向系统申请一个通信端口。这个函数的使用方法如下所示。

```
int socket(int domain, int type, int protocol);
```

在参数表表中，domain 指定使用何种的地址类型，可能的值是下面这些系统常量。

- PF_UNIX,PF_LOCAL,AF_UNIX,AF_LOCAL：这些是 UNIX 进程通信协议。
- PF_INET,AF_INET：Ipv4 网络协议。
- PF_INET6,AF_INET6：Ipv6 网络协议。
- PF_IPX,AF_IPX：IPX-Novell 协议。
- PF_NETLINK,AF_NETLINK：核心用户接口装置。
- PF_X25,AF_X25、ITU-T X.25：ISO-8208 协议。
- PF_AX25,AF_AX25：业余无线 AX.25 协议。
- PF_ATMPVC,AF_ATMPVC：存取原始 ATM PVCs。
- PF_APPLETALK,AF_APPLETALK：DDP 网络协议。
- PF_PACKET,AF_PACKET：初级封包接口。

type 参数的作用是设置通信的协议类型，可能的取值如下所示。

- SOCK_STREAM：提供面向连接的稳定数据传输，即 TCP 协议。
- OOB：在所有数据传送前必须使用 connect()来建立连线状态。
- SOCK_DGRAM：使用不连续不可靠的数据包连接。
- SOCK_SEQPACKET：提供连续可靠的数据包连接。
- SOCK_RAW：提供原始网络协议存取。
- SOCK_RDM：提供可靠的数据包连接。
- SOCK_PACKET：与网络驱动程序直接通信。

参数 protocol 用来指定 socket 所使用的传输协议编号。这一参数通常不具体设置，一般

设置为 0 即可。

在使用这个函数建立套接字前，需要在程序的最前面包含下面的头文件。

```
#include <sys/types.h>
#include <sys/socket.h>
```

如果建立套接字成功，则返回这个套接字的编号。如果不成功，则返回-1。这个函数可能发生的错误如下所示。

- EPROTONOSUPPORT：参数 domain 指定的类型不支持参数 type 或 protocol 指定的协议。
- ENFILE：核心内存不足，无法建立新的 socket 结构。
- EMFILE：进程文件表溢出，无法再建立新的套接字。
- EACCESS ：权限不足，无法建立 type 或 protocol 指定的协议。
- ENOBUFS、ENOMEM：内存不足。
- EINVAL：参数不合法。

下面的实例使用了 socket 函数建立一个套接字。

```
#include<sys/types.h>
#include<sys/socket.h>
#include<stdio.h>                                   /*包含头文件。*/

int  main()
{
  int s;
  if((s = socket(AF_INET,SOCK_STREAM,0))<0)         /*建立一个套接字。*/
  {
    perror("connect");                              /*输出错误信息。*/
    exit(1);                                        /*出错则退出程序。*/
  }
  else                                              /*成功则输出相关信息。*/
  {
      printf("a socket was created.\n");
      printf("socket number:%d\n",s);
   }

  if((s = socket(AF_INET,123,0))<0)                 /*用错误的参数建立一个套接字。*/
  {
    perror("connect");                              /*输出错误信息。*/
    exit(1);
  }
  else                                              /*建立成功的情况。*/
  {
      printf("a socket was created.\n");
      printf("socket number:%d\n",s);
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 16.14.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
a socket was created.
socket number:3
connect: Invalid argument
```

12.1.2　取得 socket 状态

函数 getsockopt 可以取得一个 socket 的参数。这个函数的使用方法如下所示。

```
int getsockopt(int s,int level,int optname,void* optval,socklen_t* optlen);
```

在参数列表中，s 表示已经建立 socket 的编号，level 代表需要设置的网络层，一般设成 SOL_SOCKET 来表示 socket 层，参数 optname 表示需要获取的选项，可以设置成下面这些值。

- SO_DEBUG：打开或关闭排错模式。
- SO_REUSEADDR：允许在 bind 函数中本地 IP 地址可重复使用。
- SO_TYPE：返回 socket 形态。
- SO_ERROR：返回 socket 已发生的错误原因。
- SO_DONTROUTE：送出的数据包不要利用路由设备来传输。
- SO_BROADCAST：使用广播方式传送。
- SO_SNDBUF：设置送出的暂存区大小。
- SO_RCVBUF：设置接收的暂存区大小。
- SO_KEEPALIVE：定期确定连线是否已终止。
- SO_OOBINLINE：当接收到 OOB 数据时会马上送至标准输入设备。
- SO_LINGER：确保数据可以安全可靠传送出去。

参数 optval 是取得的某个参数的返回值指针，程序的返回值会保存在这个指针指向的变量中，参数 optlen 表示 optval 的内存长度。函数如果执行成功则返回 0，反之返回-1。这个函数可能发生下面这些错误。

- EBADF：参数 s 不是合法的 socket 代码。
- ENOTSOCK：参数 s 为一打开文件的编号，而不是一个 socket。
- ENOPROTOOPT：参数 optname 指定的选项不正确。
- EFAULT：参数 optval 指针指向的内存空间无法读取。

在使用这个函数之前，需要在程序中的最前面包含下面的头文件。

```
#include<sys/types.h>
#include<sys/socket.h>
```

下面的程序是用 getsockopt 函数来读取一个 socket 的参数的实例。

```
#include<sys/types.h>
#include<sys/socket.h>
#include<stdio.h>                                    /*包含头文件。*/

int  main()
{
  int s;                                             /*定义相关的变量。*/
  int val,len ;
  len= sizeof(int);                                  /*长度。*/

  if((s = socket(AF_INET,SOCK_STREAM,0))<0)          /*建立一个套接字。*/
  {
    perror("connect");
    exit(1);
  }
  else                                               /*建立成功。*/
  {
      printf("a socket was created.\n");
      printf("socket number:%d\n",s);
   }

   getsockopt(s,SOL_SOCKET,SO_TYPE,&val,&len);  /*取得套接字的一个信息。*/
   perror("socket:");                                /*输出错误。*/
   printf("optval = %d\n",val);                      /*输出结果。*/
   getsockopt(100,SOL_SOCKET,SO_TYPE,&val,&len);/*用错误的方法取得一个信息。*/
   perror("socket:");
}
```

输入下面的命令，编译这个程序。

```
gcc 16.16.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

- a socket was created.
- socket number:3
- socket:: Success
- optval = 1
- socket:: Bad file descriptor

12.1.3　设置 socket 状态

函数 setsockopt 可以设置一个 socket 的状态，这个函数的使用方法如下所示。

```
    int setsockopt(int s,int level,int optname,const void * optval,socklent *
optlen);
```

在参数列表中，s 表示已经打开的 socket。参数 level 代表欲设置的网络层，一般设成 SOL_SOCKET 以存取 socket 层，参数 optname,optval,optlen 的含义与 12.1.2 节中的这些参数作用相同。

如果函数设置 socket 成功则返回 0，若有错误则返回-1。这个函数可能发生下面列出的错误，可以用 errno 捕获已经发生的错误。

- EBADF：参数 s 不是合法的 socket 代码。
- ENOTSOCK：参数 s 为一打开文件的编号，而不是一个 socket。
- ENOPROTOOPT：参数 optname 指定的选项不正确。
- EFAULT：参数 optval 指针指向的内存空间无法读取。

使用这个函数前，需要在程序的最前面包含下面的头文件。

```
    #include <sys/types.h>
    #include <sys/socket.h>
```

下面的实例使用了 setsockopt 函数来设置一个 socket 的状态。参数 optname 设置成 SO_TYPE，表示设置这个 socket 的状态，设置的值为 1。

```
    #include<sys/types.h>
    #include<sys/socket.h>
    #include<stdio.h>                                          /*包含头文件。*/

    int  main()
    {
      int s;                                                   /*定义相关的变量。*/
      int val=1,len,i ;
      len= sizeof(int);

      if((s = socket(AF_INET,SOCK_STREAM,0))<0)                /*建立一个套接字。*/
      {
        perror("connect");
        exit(1);
      }
      else                                                     /*套接字建立成功。*/
      {
          printf("a socket was created.\n");
          printf("socket number:%d\n",s);
       }

       i=setsockopt(s,SOL_SOCKET,SO_TYPE,&val,len);            /*设置套接字的一个参数。*/
       if("i==0")                                              /*判断结果情况。*/
```

```
    {
       printf("set socket ok.\n.");
     }
     else
    {
       printf("set socket error.\n.");
    }

    setsockopt (100,SOL_SOCKET,SO_TYPE,&val,&len);   /*用错误的方法设置一个套接
                                                      字。*/
    perror("socket");                                 /*输出错误。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 16.16.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

- a socket was created.
- socket number:3
- set socket ok.
- socket: Bad file descriptor

12.2 无连接的套接字通信

所谓无连接的套接字通信，指的是使用 UDP 协议进行信息传输。使用这种协议进行通信时，两个计算机之前没有建立连接的过程。需要处理的内容只是把信息发送到另一个计算机。这种通信的方式比较简单。本节将讲述这种无连接的 UDP 通信。

12.2.1 工作流程

无套接字的通信不需要建立起客户机与服务器之间的连接，因此在程序中没有建立连接的过程。进行通信之前，需要建立网络套接字。服务器需要绑定一个端口，在这个端口上监听接收到的信息。客户机需要设置远程 IP 和端口，需要传递的信息需要发送到这个 IP 和端口上。客户机和服务器的交互过程可用图 12-1 来表示。

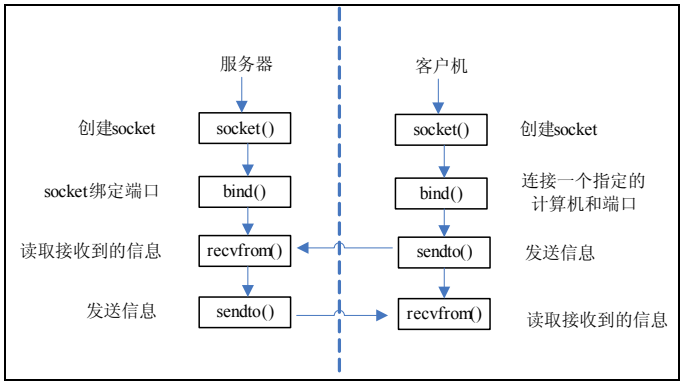

图 12-1　无连接的套接字通信

12.2.1　信息发送函数 sendto

函数 sendto 可以通过一个已经建立的套接字，将一段信息发送到另一个程序的套接字中。这个函数的使用方法如下所示。

```
    int sendto ( int s ,  void * msg, int len, unsigned int flags,struct sockaddr
* to , int tolen ) ;
```

在参数列表中，s 表示已经建立好的 socket（在使用 UDP 协议时，不需要进行计算机连接的操作），msg 表示需要发送的字符串，len 表示发送字符串的长度，参数 flags 一般设 0，其他可能的数值如下所示。

- MSG_OOB：传送的数据以 out-of-band 送出。
- MSG_DONTROUTE：取消路由表查询。
- MSG_DONTWAIT：设置为不可阻断传输。
- MSG_NOSIGNAL：此传输不愿被 SIGPIPE 信号中断。

参数 sockaddr 是一个表示套接字的结构体。这个结构体的定义如下所示。

这个结构体的定义方法如下所示。

```
    struct socketaddr_in
    {
       unsigned short int sin_family;
       uint16_t sin_port;
       struct in_addr sin_addr;
       unsigned char sin_zero[8];
    };
```

这个结构体的成员与作用如下所示。

- sin_family：与 sockaddr 结构体中的 sa_family 相同。
- sin_port：套接字使用的端口号。
- sin_addr：需要访问的 IP 地址。
- sin_zero：未使用的字段，填充为 0。

在这一结构体中，in_addr 也是一个结构体，定义方法如下所示。作用是保存一个 IP 地址。

```
struct in_addr
{
  uint32_t s_addr;
};
```

在程序中，需要设置这个结构体每一个成员的值。参数 tolen 是 sockaddr 结构体的长度，这一长度可用 sizeof 函数来取得。这个结构体会把 socket 用指定的 socket 传送给对方主机。如果传送成功，则返回传送字符的个数。传送失败则返回-1。传送失败时，错误原因保存在 errno 中，可能发生的错误如下所示。

- EBADF：参数 s 不是一个正常的 socket。
- EFAULT：参数中的指针指向无法读取的内存空间。
- WNOTSOCK：s 为一文件描述词，而不是一个 socket。
- EINTR：被其他信号所中断。
- EAGAIN：此动作会令进程阻断。
- ENOBUFS：系统的缓冲内存不足。
- EINVAL：传给系统调用的参数不正确。

使用 sendto 函数发送信息之前，需要在程序中包含下面的头文件。

```
#include < sys/types.h >
#include < sys/socket.h >
```

12.2.2 信息接收函数 recvfrom

函数 recvfrom 可以从一个 socket 中接收其他主机发送到来的信息。这个函数的使用方法如下所示。

```
int recvfrom(int s,void *buf,int len,unsigned int flags ,sockaddr *from ,int
*fromlen);
```

在参数列表中，s 表示一个已经建立一的网络套接字，buf 表示接收到的信息保存到的内存地址，Len 表示可以保存信息的 buf 内存长度，flags 一般设 0，其他可能的设置如下所示。

- MSG_OOB：接收以 out-of-band 送出的数据。
- MSG_PEEK：返回来的数据不从系统内删除，如果再调用 recv()会返回相同的数据内容。
- MSG_WAITALL：强迫接收到 len 大小的数据后才能返回，除非有错误或信号产生。
- MSG_NOSIGNAL：此操作不愿被 SIGPIPE 信号中断。

参数 from 表示 IP 地址与端口等信息。Fromlen 表示 sockaddr 的长度，这个值可以用 sizeof 函数来取得。

这个函数调用成功，则返回接收到的字符数。失败则返回-1，错误原因存于 errno 中。可能发生的错误如下所示。

- EBADF：参数 s 不是一个正确的 socket。
- EFAULT：参数中的指针可能指向了无法读取的内存空间。
- ENOTSOCK：参数 s 是文件描述词，而不是一个 socket。

- EINTR：被其他信号所中断。
- EAGAIN：此动作会令进程阻断。
- ENOBUFS：系统的缓冲内存不足。
- ENOMEM：核心内存不足。
- EINVAL：传给系统调用的参数不正确。

使用 recvfrom 函数接收数据前，需要在程序的最前面包含下面的头文件。

```
#include <sys/types.h>
#include <sys/socket.h>
```

12.3　无连接的套接字通信实例

上一节讲述了无连接的套接字通信原理和相关函数，本节将使用这些函数编写无连接的 UDP 套接字信息程序。在信息传输时，需要有服务器端与客户端。服务器处于监听状态，接收到客户端发送的信息以后会返回一定的信息。客户端会主动向服务器发送信息。

12.3.1　无连接套接字通信客户端

本节将讲述无连接套接字通信的客户端。这个程序的主要内容是，建立一个套接字，然后从键盘读取一个字符串用 sendto 函数将这个字符串发送到服务器，然后接收服务器发送回的信息。

在实际编程时，客户机与服务器是在同一个计算机上的。可以使用 127.0.0.1 这个 IP 地址表示本地计算机，这样和访问远程计算机的效果是相同的。

```
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>                          /*包含相关的头文件。*/

#define REMOTEPORT 4567                          /*定义表示端口号的常量。*/
#define REMOTEIP "127.0.0.1"                     /*定义表示表 IP 的常量。*/

int   main(int argc,char *argv[])
{
   int s,len;                                    /*定义相关的变量。*/
   struct sockaddr_in addr;
   int addr_len;
   char msg[256];                                /*定义一个数组发送与接收数据。*/
   int i=0;

   if (( s= socket(AF_INET, SOCK_DGRAM, 0) )<0)    /*建立一个 socket。*/
   {
      perror("error");                              /*输出错误。*/
```

```
        exit(1);
    }
    else
    {
        printf("socket created .\n");        /*输出提示信息。*/
        printf("socked id: %d \n",s);
        printf("remote ip: %s \n",REMOTEIP);
        printf("remote port: %d \n",REMOTEPORT);
    }

    len=sizeof(struct sockaddr_in);          /*长度。*/
    bzero(&addr,sizeof(addr));               /*空间地址结构体所在的内存空间。*/
    addr.sin_family=AF_INET;                 /*填充地址与端口的信息。*/
    addr.sin_port=htons(REMOTEPORT);         /*端口。*/
    addr.sin_addr.s_addr=inet_addr(REMOTEIP);

    while (1)                                /*循环。*/
    {
        bzero(msg,sizeof(msg));              /*清空 msg 所在的内存。*/
        len = read(STDIN_FILENO,msg,sizeof(msg));   /*接收信息。*/
        sendto(s,msg,len,0,&addr,addr_len);         /*发送信息。*/
        printf("\nInput message: %s \n",msg);       /*输出结果。*/
        len= recvfrom (s,msg,sizeof(msg),0,&addr,&addr_len);
                                                    /*这是接收到的信息。*/
        printf("%d :",i);                           /*输出计数。*/
        i++;                                        /*计数自加。*/
        printf("Received message: %s \n",msg);      /*这是服务器返回的信息。*/
    }
}
```

输入下面的命令，编译这个程序。需要加-o 参数指定一个输出文件名。

```
gcc 16.17.c -o udpcli
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x udpcli
```

12.3.2 无连接套接字通信服务器

本节将讲述无连接套接字通信的服务器。网络传输的功能需要计算机上的多个程序才能完成数据通信。上一节编写程序发送的信息，需要有一个服务器程序来接收和处理。

在服务器程序中，主要是使用 recvfrom 函数从套接字中接收传入的信息。对信息进行处理以后，用 sendto 函数发回一段信息。服务器程序的代码如下所示。

```
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>                  /*包含相关的头文件。*/
```

```
#define LOCALPORT 4567                  /*定义一个端口号。*/

int   main(int argc,char *argv[])
{
    int mysock,len;                     /*定义相关的变量。*/
    struct sockaddr_in addr;
    int i=0;
    char msg[256];                      /*定义一个数组，保存发送与接收的信息。*/
    int addr_len;

    if (( mysock= socket(AF_INET, SOCK_DGRAM, 0) )<0)
                                        /*建立一个连接。*/
    {
       perror("error");                 /*输出错误。*/
       exit(1);
    }
    else                                /*socket 建立成功则提示信息。*/
    {
        printf("socket created .\n");
        printf("socked id: %d \n",mysock);
     }

    addr_len=sizeof(struct sockaddr_in);    /*长度。*/
    bzero(&addr,sizeof(addr));              /*清空地址所在的内存。*/
    addr.sin_family=AF_INET;                /*填充地址结构体。*/
    addr.sin_port=htons(LOCALPORT);
    addr.sin_addr.s_addr=htonl(INADDR_ANY);

    if(bind(mysock,&addr,sizeof(addr))<0)   /*在 socket 上面绑定端口号与 IP。*/
    {
       perror("connect");                   /*输出错误信息。*/
       exit(1);
    }
    else
    {
        printf("bind ok.\n");               /*绑定成功则输出信息。*/
        printf("local port :%d \n",LOCALPORT);
    }

    while (1)                               /*进入一个循环。*/
    {
       bzero(msg,sizeof(msg));              /*清空 msg 所在的内存。*/
       len= recvfrom (mysock,msg,sizeof(msg),0,&addr,&addr_len);
                                            /*接收到信息。*/
       printf("%d :",i);                    /*输出计数。*/
       i++;                                                 /*计数自加。*/
       printf("message from : %s \n",inet_ntoa(addr.sin_addr));
                                                            /*输出 IP 地址。*/
       printf("  message length : %d \n",len);             /*输出长度信息。*/
       printf("  message : %s \n\n",msg);                  /*输出信息。*/
```

```
            sendto(mysock,msg,len,0,&addr,addr_len);          /*字符串返回给客户端*/
        }
    }
```

输入下面的命令，编译这个程序。需要加-o 参数指定一个输出文件名。

```
gcc 16.18.c -o udpser
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x udpser
```

12.3.3 测试 UDP 通信程序

编写出的网络通信程序，并不能单一测试一个功能。在进行测试时，需要使用客户端发送数据，查看服务器是否正常接收和处理了数据，然后再查看客户端是否接收到服务器发送回的数据。本章将对前面两节编写的程序进行测试。

① 输入下面的命令，打开 UDP 通信的服务器端。

```
 ./udpser
```

② 程序的运行结果如下所示。显示结果表明已经建立起一个 socket，监听的本地端口为 4567。

```
socket created .
socked id: 3
bind ok.
local port :4567
```

③ 打开另一个终端。在终端中输入下面的命令，打开客户端程序。

```
./udpser
```

④ 程序显示的结果如下所示。表明已经建立了一个 socket，远程的 IP 是 127.0.0.1，实际上就是本地计算机。远程端口是 4567。

```
socket created .
socked id: 3
remote ip: 127.0.0.1
remote port: 4567
```

⑤ 输入下面的文本，向服务器发送一个信息。

```
hello
```

⑥ 程序再次显示这个字符串，然后将这个字符串发送到服务器。服务器接收这段信息后返回同样的信息，结果如下所示。

```
Input message: hello
0 :Received message: hello
```

⑦ 在服务器的终端中，显示的结果如下所示。

```
0 :message from : 127.0.0.1
```

```
  message length : 6
  message : hello
```

⑧ 再在客户端的终端中输入下面的信息。

```
good morning .
```

⑨ 客户端显示的结果如下所示，表明第二次接收到信息。

```
Input message: good morning .
1 :Received message: good morning .
```

⑩ 在服务端查看传递的信息，结果如下所示。

```
1 :message from : 127.0.0.1
  message length : 15
  message : good morning .
```

⑪ 上面的测试结果表明，这两个程序可以正常完成两个计算机之间的字符串传输。在本地计算机测试正常以后，在广域网中运行一般是正常的。这两个程序可以作为不同计算机之间的聊天工具。

⑫ 这个程序进入了 while 循环以后，就一直进行信息发送和接收的操作，不能结束循环，可以按“Ctrl”+“C”组合键关闭这两个程序。

12.4　小结

本章讲述了网络通信套接字和无连接的套接字通信。在学习时，需要重点理解套接字的含义和套接字的基本操作。无连接的套接字通信是一种简单的网络通信方式，编程的重点是信息的发送与接收。本章的实例，是无连接的套接字通信基本方式，可以实现两个计算机之间的文字信息传输，读者可以在这两个程序的基础上编写出文本模式下的聊天软件。

第 13 章　面向连接的套接字通信

13.1　面向连接的套接字通信工作流程

为了实现服务器与客户机的通信，服务器和客户机都必须建立套接字。服务器与客户机的工作原理可以用下面的过程来描述。

① 服务器先用 socket 函数来建立一个套接字，用这个套接字完成通信的监听。

② 用 bind 函数来绑定一个端口号和 IP 地址。因为本地计算机可能有多个网址和 IP，每一个 IP 和端口有多个端口。需要指定一个 IP 和端口进行监听。

③ 服务器调用 listen 函数，使服务器的这个端口和 IP 处于监听状态，等待客户机的连接。

④ 客户机用 socket 函数建立一个套接字，设定远程 IP 和端口。

⑤ 客户机调用 connect 函数连接远程计算机指定的端口。

⑥ 服务器用 accept 函数来接受远程计算机的连接，建立起与客户机之间的通信。

⑦ 建立连接以后，客户机用 write 函数向 socket 中写入数据。也可以用 read 函数读取服务器发送来的数据。

⑧ 服务器用 read 函数读取客户机发送来的数据，也可以用 write 函数来发送数据。

⑨ 完成通信以后，用 close 函数关闭 socket 连接。

客户机与服务器建立面向连接的套接字进行通信，请求与响应过程可用图 13-1 来表示。

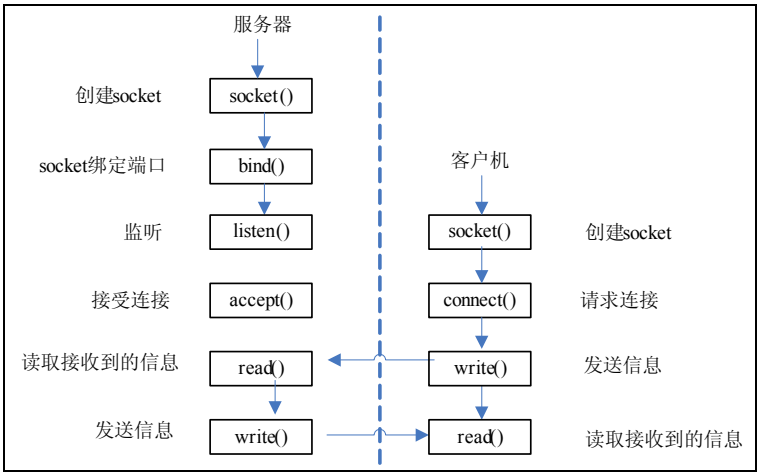

图 13-1　面向连接的套接字通信工作流程

13.2　绑定端口

绑定端口指的是将套接字与指定的端口相连。用 socket 函数建立起一个套接字以后，需要用 bind 函数在这个套接字上面绑定一个端口。本节将讲解套接字的绑定端口操作。

13.2.1　绑定端口函数 bind

函数 bind 可以将一个端口绑定到一个已经建立的 socket 上，这个函数的使用方法如下所示。

```
int bind(int sockfd,struct sockaddr * my_addr,int addrlen);
```

参数列表中，sockfd 表示已经建立的 socket 编号，sockaddr 是一个指向 sockaddr 结构体类型的指针。sockaddr 的定义方法如下所示。

```
struct sockaddr
{
   unsigned short int sa_family;
   char sa_data[14];
};
```

这个结构体的成员含义如下所示。

sa_family：为调用 socket()时的 domain 参数，即 AF_xxxx 值。

sa_data：最多使用 14 个字符长度，含有 IP 地址与端口的信息。

如果建立 socket 时使用的是 AF_INET 参数，则 socketaddr_in 结构体的定义方法如下所示。

```
struct socketaddr_in
{
   unsigned short int sin_family;
   uint16_t sin_port;
   struct in_addr sin_addr;
   unsigned char sin_zero[8];
};
```

结构体的成员 in_addr 也是一个结构体，定义方式如下所示。

```
struct in_addr
{
   uint32_t s_addr;
};
```

在这些结构体中，成员变量的作用与含义如下所示。

- sin_family：即为 sa_family，为调用 socket()时的 domain 参数。
- sin_port：使用的端口号。
- sin_addr.s_addr：IP 地址。
- sin_zero：未使用的字段，填充为 0。

参数 addrlen 表示 my_addr 的长度，可以用 sizeof 函数来取得。函数可以把指定的 IP 与端口绑定到已经建立的 socket 上面。

如果绑定成功，则返回 0，失败则返回-1。函数可能发生下面的错误，可以用 error 捕获发生的错误。

- EBADF：参数 sockfd 不是一个合法的 socket。
- EACCESS：权限不足。
- ENOTSOCK：参数 sockfd 是文件描述词，而不是 socket。

在使用这个函数前，需要在程序的最前面包含下面的头文件。

```
#include<sys/types.h>
#include<sys/socket.h>
```

13.2.2 bind 函数绑定端口实例

下面的实例使用了 bind 函数在一个打开的 socket 上面绑定 IP 与端口。绑定的端口是 5678，IP 为 INADDR_ANY，表示本地计算机的默认 IP 地址。

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>                                          /*包含头文件。*/

#define PORT 5678                                           /*定义一个表示端口的常量。*/

main()
{
   int sockfd,newsockfd, fd;                                /*定义相关的变量。*/
   struct sockaddr_in addr;
   int addr_len = sizeof(struct sockaddr_in);
   fd_set myreadfds;
   char msgbuffer[256];                                     /*定义一个字符数组存储发送和
                                                            接收到的信息。*/

   if ((sockfd = socket(AF_INET,SOCK_STREAM,0))<0) /*建立一个 socket。*/
   {
      perror("socket");
      exit(1);
   }
   else                                                     /*建立成功。*/
   {
      printf("socket created .\n");
      printf("socked id: %d \n",sockfd);
   }

   bzero(&addr,sizeof(addr));                        /*清空表示地址的结构体变量。*/
   addr.sin_family =AF_INET;
   addr.sin_port = htons(PORT);                      /*设置 addr 的成员。*/
   addr.sin_addr.s_addr = htonl(INADDR_ANY);

   if(bind(sockfd,&addr,sizeof(addr))<0)                /*bind 函数绑定端口。*/
```

```
    {
       perror("connect");
       exit(1);
    }
    else                                                    /*失败的情况。*/
    {
        printf("connected.\n");
        printf("local port:%d\n",PORT)    ;
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 17.19.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
socket created .
socked id: 3
connected.
local port:5678
```

13.3　监听与连接

所谓监听，指的是 socket 的端口处于等待状态，如果客户端有连接请求，这个端口会接受这个连接。连接指的是客户端向服务端发送一个通信申请，服务端会响应这个请求。本节将讲述 socket 的监听与连接操作。

13.3.1　等待监听函数 listen

服务器必须等待客户端的连接请求，listen 函数用于实现监听等待功能。这个函数的使用方法如下所示。

```
int listen(int s,int backlog);
```

在参数列表中，s 表示已经建立的 socket，backlog 表示能同时处理的最大连接请求，如果超过这个数目，客户端将会接收到 ECONNREFUSED 拒绝连接的错误。需要注意的是，listen 并未真正的接受连接，只是设置 socket 的状态为 listen 模式，真正接受客户端连接的是 accept 函数。通常情况下，listen 函数会在 socket, bind 函数之后调用，然后才会调用 accept 函数。

listen 函数只适用 SOCK_STREAM 或 SOCK_SEQPACKET 的 socket 类型。如果 socket 为 AF_INET 则参数 backlog 最大值可设至 128，即最多可以同时接受 128 个客户端的请求。

如果调用成功，则函数的返回值为 0，失败返回-1。函数可能发生如下所示的错误，可以

用 errno 来捕获发生的错误。

- EBADF：参数 sockfd 不是一个合法的 socket。
- EACCESS：权限不足。
- EOPNOTSUPP：指定的 socket 不支持 listen 模式。

在使用这个函数前，需要在程序的最前面包含下面的头文件。

```
#include<sys/socket.h>
```

13.3.2 listen 函数使用实例

本节将讲述一个 listen 函数使用实例。在程序中，先建立一个 socket，然后用 bind 函数在这个 socket 上面绑定端口与 IP，然后用 listen 函数设置这个 socket 进行监听。程序的代码如下所示。

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>                                  /*包含头文件。*/

#define PORT 5678                                   /*定义一个表示端口号的常量。*/
#define MAX 10                                      /*最多的连接客户端数。*/

main()
{
   int sockfd,newsockfd,is_connected[MAX],fd;   /*定义相关的变量。*/
   struct sockaddr_in addr;
   int addr_len = sizeof(struct sockaddr_in);
   fd_set myreadfds;
   char msgbuffer[256];
   char msg[] ="This is the message from server.Connected.\n";

   if ((sockfd = socket(AF_INET,SOCK_STREAM,0))<0) /*建立一个 socket。*/
   {
      perror("socket");
      exit(1);
   }
   else                                             /*socket 建立成功。*/
   {
      printf("socket created .\n");
      printf("socked id: %d \n",sockfd);
   }

   bzero(&addr,sizeof(addr));                       /*清空 addr 所在的内存。*/
   addr.sin_family =AF_INET;                        /*设置 addr 的相关信息。*/
   addr.sin_port = htons(PORT);
   addr.sin_addr.s_addr = htonl(INADDR_ANY);

   if(bind(sockfd,&addr,sizeof(addr))<0)            /*绑定端口。*/
```

```
    {
       perror("connect");
       exit(1);
    }
    else                                                    /*端口绑定成功。*/
    {
        printf("connected.\n");
        printf("local port:%d\n",PORT)    ;
    }

    if(listen(sockfd,3)<0)                                  /*监听一个端口。*/
    {
       perror("listen");                                    /*错误则输出提示。*/
       exit(1);
    }
    else                                                    /*输出结果。*/
    {
       printf("listenning......\n");
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 17.20.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
socket created .
socked id: 3
connected.
local port:5678
listenning......
```

13.3.3　接受连接函数 accept

服务器处于监听状态时，如果获得客户机的请求会将这个请求放在等待队列中，当系统空闲时将处理客户机的连接请求。接受连接请求的函数是 accept，这个函数的使用方法如下所示。

```
int accept(int s,struct sockaddr * addr,int * addrlen);
```

在参数列表中，s 表示处于监听状态的 socket，addr 是一个 sockaddr 结构体类型的指针，系统会把远程主机的这些信息保存到这个结构体指针上，addrlen 表示 sockaddr 的内存长度，可以用 sizeof 函数来取得。

当 accept 函数接受一个连接时，会返回一个新的 socket 编号。以后的数据传输与读取就是通过这个新的 socket 编号来处理。原来参数中的 socket 可以继续使用。接受连接以后，远程主机的地址和端口信息将会保存到 addr 所指的结构体内。如果处理失败，返回值为-1。函数可能产生下面的错误，可以用 error 来捕获发生的错误。

- EBADF：参数 s 不是一个合法的 socket 代码。
- EFAULT：参数 addr 指针指向无法存取的内存空间。
- ENOTSOCK 参数 s 为一文件描述词，而不是一个 socket。
- EOPNOTSUPP：指定的 socket 不是 SOCK_STREAM。
- EPERM：防火墙拒绝这一个连接。
- ENOBUFS：系统的缓冲内存不足。
- ENOMEM：核心内存不足。

在使用这个函数前，需要在程序中包含下面的头文件。

```
#include<sys/types.h>
#include<sys/socket.h>
```

13.3.4 accept 函数使用实例

本节将讲述一个 accept 函数使用实例。在程序中，先建立一个 socket，然后用 bind 函数在这个 socket 函数上面绑定一个端口，然后使用 listen 函数使这个端口处于监听状态。当有连接请求时，accpet 函数会产生一个新的 socket，然后输出提示信息。程序的代码如下所示。

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>                                  /*包含头文件。*/

#define PORT 5678                                   /*定义一个端口常量。*/

main()
{
   int sockfd,newsockfd,fd;                         /*定义相关的变量。*/
   struct sockaddr_in addr;
   int addr_len = sizeof(struct sockaddr_in);
   fd_set myreadfds;
   char msgbuffer[256];                             /*定义一个字符串。*/

   if ((sockfd = socket(AF_INET,SOCK_STREAM,0))<0) /*建立一个 socket。*/
   {
      perror("socket");
      exit(1);
   }
   else
   {
      printf("socket created .\n");                      /*socket 建立成功。*/
      printf("socked id: %d \n",sockfd);
```

```
    }

    bzero(&addr,sizeof(addr));                              /*设置 addr 的信息。*/
    addr.sin_family =AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);

    if(bind(sockfd,&addr,sizeof(addr))<0)                   /*绑定端口号。*/
    {
       perror("connect");
       exit(1);
    }
    else
    {
        printf("connected.\n");
        printf("local port:%d\n",PORT)   ;
    }

    if(listen(sockfd,3)<0)                                  /*监听一个端口号。*/
    {
       perror("listen");
       exit(1);
    }
    else
    {
       printf("listenning......\n");
    }

    if((newsockfd = accept (sockfd,&addr,&addr_len))<0) /*接受一个连接。*/
    {
       perror("accept");
    }
    else
    {
       printf("accepted a new connecttion.\n");
    }
}
```

输入下面的命令，编译这个程序。

```
gcc 17.21.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
socket created .
```

```
socked id: 3
connected.
local port:5678
listenning......
```

结果表明，本地计算机的 5678 号端口处于监听状态。打开浏览器，在浏览器的地址栏中输入下面的网址，然后按“Enter”键。这样浏览器会请求连接本地计算机的 5678 号端口。

```
http://127.0.0.1:5678/
```

浏览器会显示无法打开这个网页。在终端中显示的结果如下所示。表明这个程序已经接受了这个连接，然后退出了程序。

```
accepted a new connecttion.
```

13.3.5 请求连接函数 connet

所谓请求连接，指的是客户机向服务器发送信息时，需要发送一个连接请求。connect 函数可以完成这项功能，这个函数的使用方法如下所示。

```
int connect (int sockfd,struct sockaddr * serv_addr,int addrlen);
```

在参数列表中，sockfd 表示已经建立的 socket，serv_addr 是一个结构体指针，指向一个 sockaddr 结构体，这个结构体存储着远程服务器的 IP 与端口信息，Addrlen 表示 sockaddr 结构体的内存长度，可以用 sizeof 函数来获取。sockaddr 结构体的定义见前面节中的 bind 函数所述。

函数会将本地的 socket 连接到 serv_addr 所指定的服务器 IP 与端口。如果连接成功，返回值为 0，连接失败则返回-1。函数可能发生下面的错误，可以用 error 来捕获发生的错误。

- EBADF：参数 sockfd 不是一个合法的 socket。
- EFAULT：参数 serv_addr 指针指向了一个无法读取的内存空间。
- ENOTSOCK：参数 sockfd 是文件描述词，而不是一个正常的 socket。
- EISCONN：参数 sockfd 的 socket 已经处于连接状态。
- ECONNREFUSED：连接要求被服务器拒绝。
- ETIMEDOUT：需要的连接操作超过限定时间仍未得到响应。
- ENETUNREACH：无法传送数据包至指定的主机。
- EAFNOSUPPORT：sockaddr 结构的 sa_family 不正确。
- EALREADY：socket 不能阻断，但是以前的连接操作还未完成。

如果需要使用这个函数，需要在程序的最前面包含下面的头文件。

```
#include<sys/types.h>
#include<sys/socket.h>
```

13.3.6　connet 函数使用实例

本节将讲解一个 connet 函数的实例，在程序中连接到远程服务器。在实例中连接的远程服务器是央视国际网站，域名是 www.cctv.com。可以在终端中输入下面的命令来取得这个域名的 IP 地址。

```
ping www.cctv.com
```

显示的结果如下所示。

```
PING www.cctv.com (202.108.249.216) 56(84) bytes of data.
```

所以央视国际的服务器 IP 地址是 202.108.249.216。网站服务的端口是 80。程序的代码如下所示。

```
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>

#define PORT 80                                    /*定义一个端口号。*/
#define REMOTE_IP "202.108.249.216"                /*定义一个 IP 地址。*/

int   main(int argc,char *argv[])
{
   int s ;                                         /*定义相关的变量。*/
   struct sockaddr_in addr ;
   char mybuffer[256];

   if( (s=socket(AF_INET,SOCK_STREAM,0))<0 )       /*建立一个 socket。*/
   {
      perror("socket");
      exit(1);
   }
   else
   {
    printf("socket created .\n");
    printf("socked id: %d \n",s);
   }

   bzero(&addr,sizeof(addr));                      /*设置 addr 的信息。*/
   addr.sin_family =AF_INET;
   addr.sin_port=htons(PORT);
   addr.sin_addr.s_addr=inet_addr(REMOTE_IP);

   if(connect(s,&addr,sizeof(addr))<0)             /*连接到远程服务器。*/
   {
```

```
        perror("connect");
        exit(1);
    }
    else                                              /*输出信息。*/
    {
        printf("connected ok!\n");
        printf("remote ip:%s\n",REMOTE_IP);
        printf("remote port:%d\n",PORT);
     }
}
```

输入下面的命令，编译这个程序。

```
gcc 17.22.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。结果表明程序已经正常连接到了远程服务器。

```
socket created .
socked id: 3
connected ok!
remote ip:202.108.249.216
remote port:80
```

13.4 数据的发送与接收

建立套接字并完成网络连接以后，可以把信息传送到远程主机上，这个过程就是信息的发送。对于远程主机发送来的信息，本地主机需要进行接收处理。本节将讲述这种面向连接的套接字信息发送与接收操作。

13.4.1 数据接收函数 recv

函数 recv 可以接收远程主机发送来的数据，并把这些数据保存到一个数组中。该函数的使用方法如下所示。

```
int recv(int s,void *buf,int len,unsigned int flags);
```

在参数列表中，s 表示已经建立的 socket，buf 是一个指针，指向一个数组，接收到的数据会保存到这个数组上，len 表示数组的长度，可以用 sizeof 函数来取得，flags 一般设置为 0，其他可能的赋值与含义如下所示。

- MSG_OOB：接收以 out-of-band 送出的数据。
- MSG_PEEK：返回来的数据并不会在系统内删除，如果再调用 recv 时会返回相同的数据内容。

- MSG_WAITALL：强迫接收到 len 大小的数据后才能返回，除非有错误或信号产生。
- MSG_NOSIGNAL：此操作被 SIGPIPE 信号中断。

recv 函数如果接收到数据，会把这些数据保存在 buf 指针指向的内存中，然后返回接收到字符的个数。如果发生错误则会返回-1。函数可能发生下面这些错误，可以用 errno 来捕获错误。

- EBADF：参数 s 不是一个合法的 socket。
- EFAULT：参数中的指针指向了无法读取的内存空间。
- ENOTSOCK：参数 s 是文件描述词，而不是一个 socket。
- EINTR：被信号中断。
- EAGAIN：此动作会阻断进程，但参数 s 的 socket 不可阻断。
- ENOBUFS：系统的缓冲内存不足。
- ENOMEM：核心内存不足
- EINVAL：参数不正确。

在使用这个函数前，需要在程序的最前面包含下面的头文件。

```
#include <sys/types.h>
#include <sys/socket.h>
```

13.4.2　recv 函数使用实例

本节将讲解一个 recv 函数使用实例。在程序中，连接到北京大学的 FTP 服务器，然后用 recv 函数取得 ftp 服务器返回的信息。北京大学的 FTP 服务器域名如下所示。

```
ftp.pkp.edu.cn
```

在终端中输入下面的命令，取得这个域名的 IP 地址。

```
ping ftp.pku.edu.cn
```

终端中显示的结果如下所示。

```
PING vineyard.pku.edu.cn (202.38.97.197) 56(84) bytes of data.
```

所以北京大学 FTP 服务器的 IP 地址是 202.38.97.197。FTP 服务的端口号是 21。程序的代码如下所示。

```
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>                                  /*包含头文件。*/

#define PORT 21                                          /*定义一个端口号。*/
#define REMOTE_IP "202.38.97.197"                        /*定义一个 IP 地址。*/
```

```
int   main(int argc,char *argv[])
{
   int s ;                                              /*定义相关的变量。*/
   struct sockaddr_in addr ;
   char mybuffer[256];

   if( (s=socket(AF_INET,SOCK_STREAM,0))<0 )            /*建立一个 socket。*/
   {
      perror("socket");
      exit(1);
   }
   else
   {
      printf("socket created .\n");                     /*socket 建立成功。*/
      printf("socked id: %d \n",s);
   }

   bzero(&addr,sizeof(addr));                           /*清空 addr 所占的内存。*/
   addr.sin_family =AF_INET;                            /*设置 addr 的成员。*/
   addr.sin_port=htons(PORT);
   addr.sin_addr.s_addr=inet_addr(REMOTE_IP);

   if(connect(s,&addr,sizeof(addr))<0)                  /*连接远程服务器。*/
   {
      perror("connect");
      exit(1);
   }
   else
   {
       printf("connected ok!\n");                       /*连接成功。*/
       printf("remote ip:%s\n",REMOTE_IP);
       printf("remote port:%d\n",PORT);
   }

   recv(s ,mybuffer,sizeof(mybuffer),0);                /*接收信息。*/
   printf("%s\n",mybuffer);                             /*输出接收到的信息。*/
}
```

输入下面的命令，编译这个程序。

```
gcc 17.23.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。结果表明程序已经正确连接到了北京大学的 FTP 服务器。服务器返回了一段欢迎信息。

```
socket created .
socked id: 3
connected ok!
remote ip:202.38.97.197
remote port:21
220 Welcome to VINEYARD FTP service.
```

13.4.3　信息发送函数 send

用 connect 函数连接到远程计算机以后，可以用 send 函数将信息发送到对方的计算机。这个函数的使用方法如下所示。

```
int send(int s,const void * msg,int len,unsigned int flags);
```

在参数列表中，s 表示已经建立的 socket，msg 是需要发送数据的指针，len 表示需要发送数据的长度。这个长度可以用 sizeof 函数来取得，参数 flags 一般设置为 0，可能的赋值与含义如下所示。

- MSG_OOB：传送的数据以 out-of-band 的方式送出。
- MSG_DONTROUTE：取消路由表查询。
- MSG_DONTWAIT：设置为不可阻断传输。
- MSG_NOSIGNAL：此传输不可被 SIGPIPE 信号中断。

如果发送数据成功，函数会返回已经传送的字符个数，否则会返回-1。函数可能发生下面这些错误，可以用 errno 来捕获函数的错误。

- EBADF：参数 s 不是一个正确的 socket。
- EFAULT：参数中的指针指向了不可读取的内存空间。
- ENOTSOCK：参数 s 是一个文件，而不是一个 socket。
- EINTR：被信号中断。
- EAGAIN：此操作会中断进程，但 socket 不允许中断。
- ENOBUFS：系统的缓冲内存不足。
- ENOMEM：核心内存不足。
- EINVAL：传给系统调用的参数不正确。

在使用这个函数前，需要在程序的最前面包含下面的头文件。

13.4.4　数据传输函数 write 与 read

在前面的章节讲述过，write 函数可以向文件中写入数据，read 函数可以从文件中读取数据。socket 建立连接以后，向这个 socket 中写入数据表示向远程主机传送数据，从 socket 中读取数据相当于接受远程主机传送过来的数据。这两个函数的使用方法如下所示。

```
ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
```

在参数列表中，fd 表示已经建立的 socket，buf 是指向一段内存的指针，count 表示 buf 指向内存的长度。read 函数读取字节时，会把读取的内容保存到 buf 指向的内存中，然后返

回读取到字节的个数。使用 write 函数传输数据时，会把 buf 指针指向的内存中的数据发送到 socket 连接的远程主机，然后返回实际发送字节的个数。

在使用这个函数前，需要在程序的最前面包含下面的头文件。

```
#include <unistd.h>
```

13.4.5 read 函数接收数据实例

本节将讲解一个 read 函数读取数据的实例。在程序中，监听一个端口，如果有客户端连接这个端口则接受这个连接，然后用 read 函数读取远程主机发送的数据，输出这些数据以后结束这个程序。程序的代码如下所示。

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>                                    /*包含头文件。*/

#define PORT 6677                                     /*定义一个端口号。*/

main()
{
   int sockfd,newsockfd,fd;                           /*定义相关的变量。*/
   struct sockaddr_in addr;
   int addr_len = sizeof(struct sockaddr_in);
   fd_set myreadfds;
   char msgbuffer[256];
   char msg[] ="This is the message from server.Connected.\n";

   if ((sockfd = socket(AF_INET,SOCK_STREAM,0))<0) /*建立一个 socket。*/
   {
      perror("socket");
      exit(1);
   }
   else
   {
      printf("socket created .\n");                  /*socket 建立成功。*/
      printf("socked id: %d \n",sockfd);
   }

   bzero(&addr,sizeof(addr));                         /*清空 zero 所在的内存。*/
   addr.sin_family =AF_INET;
   addr.sin_port = htons(PORT);
   addr.sin_addr.s_addr = htonl(INADDR_ANY);

   if(bind(sockfd,&addr,sizeof(addr))<0)                /*绑定 IP 端口。*/
   {
      perror("connect");
      exit(1);
   }
```

```
    else
    {
        printf("connected.\n");
        printf("local port:%d\n",PORT)    ;
    }

    if(listen(sockfd,3)<0)                                    /*监听一个端口号。*/
    {
       perror("listen");
       exit(1);
    }
    else
    {
       printf("listenning......\n");
    }

    if((newsockfd = accept (sockfd,&addr,&addr_len))<0) /*接受一个连接。*/
    {
       perror("accept");
    }
    else                                                      /*输出结果。*/
    {
        printf("cnnect from %s\n",inet_ntoa(addr.sin_addr));
        if(read(newsockfd,msgbuffer,sizeof(msgbuffer))<=0)
                                                 /*接收信息。*/
        {
          perror("accept");
        }
        else
        {
           printf("message:\n%s \n",msgbuffer);           /*输出接收到的信息。*/
          }
     }
}
```

输入下面的命令，编译这个程序。

```
gcc 17.24.c
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x a.out
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。结果表明这个程序正在监听本地计算机的 6677 号端口。

```
socket created .
socked id: 3
connected.
local port:6677
```

```
    listenning......
```

打开浏览器，在浏览器中输入下面的网址，然后按“Enter”键，使浏览器访问本地计算机的 6677 号端口。

```
    http://127.0.0.1:6677/
```

浏览器显示无法打开网页。在终端中显示了下面的代码，这些代码是浏览器向本机的 6677 号端口请求打开网页的数据报。

```
    cnnect from 127.0.0.1
    message:
    GET / HTTP/1.1
    Host: 127.0.0.1:6677
    User-Agent: Mozilla/5.0 (X11; U; Linux i686; zh-CN; rv:1.8.1.8) Gecko/20071030
Fedora/2.0.0.8-2.fc8 Firefox/2.0.0.8
    Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,image/png,*/*
```

13.5 面向连接套接字通信实例

本节将讲述一个面向连接的套接字通信实例。面向连接的网络通信，包括客户端和服务器两个部分的程序。服务器实现监听功能，如果有客户端连接，则接受这个连接，并将用户发送来的信息发回。客户端实现请求连接的功能，可以连接到服务器，向服务器发送信息，并接收服务器发回的信息。

13.5.1 服务器程序

所谓服务器程序，指的是在网络通信时这个程序始终处于等待状态，可以接受用户的连接请求，并且对用户发送的信息进行处理，本节的实例是面向连接的套接字通信服务器程序。程序的重点是端口的监听和接收发送的信息。

```
    #include<sys/types.h>
    #include<sys/socket.h>
    #include<netinet/in.h>
    #include<arpa/inet.h>
    #include<unistd.h>                                  /*包含头文件。*/

    #define PORT 5678                                   /*定义端口号。*/
    #define MAX 10                                      /*最多的连接数。*/

    main()
    {
       int sockfd,newsockfd,is_connected[MAX],fd;       /*定义相关的变量。*/
       struct sockaddr_in addr;
       int addr_len = sizeof(struct sockaddr_in);
       fd_set myreadfds;
       char msgbuffer[256];
       char msg[] ="This is the message from server.Connected.\n";
```

```
if ((sockfd = socket(AF_INET,SOCK_STREAM,0))<0) /*建立一个 socket。*/
{
   perror("socket");
   exit(1);
}
else
{
   printf("socket created .\n");              /*socket 建立成功，输出提示。*/
   printf("socked id: %d \n",sockfd);
}

bzero(&addr,sizeof(addr));                    /*清空 addr 所在的内存。*/
addr.sin_family =AF_INET;
addr.sin_port = htons(PORT);
addr.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(sockfd,&addr,sizeof(addr))<0)         /*绑定端口。*/
{
   perror("connect");
   exit(1);
}
else
{
    printf("connected.\n");
    printf("local port:%d\n",PORT)    ;
}

if(listen(sockfd,3)<0)                        /*开始监听。*/
{
   perror("listen");
   exit(1);
}
else
{
   printf("listenning......\n");
}

for(fd=0;fd<MAX;fd++)
{
   is_connected[fd]=0;                        /*设置所有的标记为 0。*/
}

while(1)                                      /*进入一个循环，处理所有的连接。*/
{
   FD_ZERO(&myreadfds);                       /*清空一个标志。*/
   FD_SET(sockfd,&myreadfds);                 /*设置标志。*/

   for(fd=0;fd<MAX;fd++)
   {
      if(is_connected[fd])                    /*判断有没有连接。*/
```

```
            {
               FD_SET(fd,&myreadfds);              /*设置标志。*/
             }
         }

         if(!select(MAX,&myreadfds,NULL,NULL,NULL))     /*如果到达了最大的连接数则进
                                                        入下次循环。*/
         {
             continue;
         }

         for(fd=0;fd<MAX;fd++)                      /*进入一个循环。*/
         {
            if(FD_ISSET(fd,&myreadfds))             /*判断标志。*/
            {
               if(sockfd==fd)                       /*如果新建的 socket 与 fd 相同。*/
               {
                  if((newsockfd = accept (sockfd,&addr,&addr_len))<0)
                                                    /*接受一个连接，新建一个 socket。*/
                  {
                     perror("accept");
                   }
                  write(newsockfd,msg,sizeof(msg));     /*给客户端发送一段信息。*/
                  is_connected[newsockfd] =1;           /*设置标志。*/
                  printf("cnnect from %s\n",inet_ntoa(addr.sin_addr));
                                                        /*输出客户端的 IP。*/
               }
               else
               {
                  bzero(msgbuffer,sizeof(msgbuffer));   /*清空字符串。*/
                  if(read(fd,msgbuffer,sizeof(msgbuffer))<=0)
                                                        /*读取结果。*/
                  {
                     printf("connect closed.\n");       /*输出连接关闭。*/
                     is_connected[fd]=0;                /*设置标志。*/
                     close(fd);                         /*关闭一个 socket。*/
                   }
                   else
                   {
                      write(fd,msgbuffer,sizeof(msgbuffer));
                                                    /*发送接收到的信息。*/
                      printf("message:%s \n",msgbuffer);
                                                    /*输出接收到的信息。*/
                    }
                }
            }
         }
      }
   }
```

输入下面的命令，编译这个程序。需要加-o 参数指定一个输出文件名。

```
gcc 17.25.c -o tcpser
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x udpcli
```

13.5.2　客户端程序

客户端指的是在网络通信时主动向服务器发送连接请求，主动发送信息的程序。本节将讲述面向连接的套接字通信的客户端程序。这个程序的主要内容是向服务器申请连接，并且发送用户输入的内容。

```
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>                                  /*包含头文件。*/

#define PORT 5678                                        /*定义远程端口。*/
#define REMOTE_IP "127.0.0.1"                            /*定义远程 IP。*/

int   main(int argc,char *argv[])
{
   int s ;                                               /*定义相关变量。*/
   struct sockaddr_in addr ;
   char mybuffer[256];

   if( (s=socket(AF_INET,SOCK_STREAM,0))<0 )             /*新建一个 socket。*/
   {
      perror("socket");
      exit(1);
   }
   else
   {
      printf("socket created .\n");
      printf("socked id: %d \n",s);
   }

   bzero(&addr,sizeof(addr));                            /*设置 addr。*/
   addr.sin_family =AF_INET;
   addr.sin_port=htons(PORT);
   addr.sin_addr.s_addr=inet_addr(REMOTE_IP);

   if(connect(s,&addr,sizeof(addr))<0)                   /*连接远程服务器。*/
   {
      perror("connect");
```

```
        exit(1);
    }
    else
    {
        printf("connected ok!\n");
        printf("remote ip:%s\n",REMOTE_IP);
        printf("remote port:%d\n",PORT);
     }

    recv(s ,mybuffer,sizeof(mybuffer),0);              /*接收服务器发送的信息。*/
    printf("%s\n",mybuffer);                           /*输出信息。*/

    while(1)                                           /*进入一个循环。*/
    {
      bzero(mybuffer,sizeof(mybuffer));                /*清空 mybuffer 字符串。*/
      read(STDIN_FILENO,mybuffer,sizeof(mybuffer));    /*读取输入。*/

      if(send(s,mybuffer,sizeof(mybuffer),0)<0)        /*发送信息。*/
      {
        perror("send");                                /*错误处理。*/
        exit(1);
      }
      else
      {
        bzero(mybuffer,sizeof(mybuffer));              /*清空 mybuffer 内存。*/
        recv(s ,mybuffer,sizeof(mybuffer),0);          /*接收信息。*/
        printf("received:%s\n",mybuffer);              /*输出信息。*/
      }
    }
}
```

输入下面的命令，编译这个程序。需要加-o 参数指定一个输出文件名。

```
gcc 17.26.c -o tcpcli
```

输入下面的命令，对这个程序添加可执行权限。

```
chmod +x udpcli
```

13.5.3 实例程序测试

本节将对面向连接套接字通信的两个实例程序进行测试。在运行这两个程序时，需要先打开服务程序，使相应的端口处于监听状态，然后打开客户端程序，请求连接服务器的端口。

（1）输入下面的命令，运行服务器程序。

```
./tcpser
```

（2）程序的运行结果如下所示。

```
socket created .
socked id: 3
connected.
local port:5678
```

```
listenning......
```

（3）结果显示，已经建立 socket，本地端口是 5678。程序正在这个端口上进行监听。

（4）这时打开另一个终端，在终端中输入下面的命令，打开客户端程序。

```
./tcpcli
```

（5）程序显示的结果如下所示。结果显示程序已经建立了 socket 连接到了 127.0.0.1 这个 IP 上的 5678 号端口。服务器发送回一条信息。

```
socket created .
socked id: 3
connected ok!
remote ip:127.0.0.1
remote port:5678
This is the message from server.Connected.
```

（6）在服务器程序的终端中查看，显示的结果如下所示。

```
cnnect from 127.0.0.1
```

（7）结果表示服务器已经接受了客户端的连接请求。

（8）这时进入客户程序的终端，输入“hello”，然后按“Enter”键。显示的结果如下所示。

```
received:hello
```

（9）结果表明，程序把输入的字符串发送到了服务器，服务器又返回了这个字符串。

（10）进入到服务器的终端，可以查看到客户端发送的信息如下所示。

```
message:hello
```

（11）这两个程序进入了 while 循环以后，会始终执行程序无法退出。可以按“Ctrl”+“C”组合键结束程序。

13.6　小结

本节讲述了面向连接的套接字网络通信，这种信息传输方式是网络中最常用最重要的信息传输方式。本节的知识重点是端口监听、远程连接、接受连接、接收和发送信息四个操作。最后一节的程序实例演示了面向连接的信息传输过程，通过这个实例的学习，可以学习本节中所讲述的套接字连接的综合操作。这个实例是网络传输的最基本的原理和形式，用这种方法可以实现不同计算机之间的文本、文件等类型数据的传输。