

第 4 章

Linux C 语言编程基础

Linux 环境下 C 语言编程是本书所要介绍的主要内容。C 语言是 Linux 平台下最重要的开发语言之一，Linux 操作系统本身主要就是采用 C 语言开发出来的。本章将首先介绍 Linux 环境下 C 语言编程所需的编译环境，包括命令行形式的 gcc 编译器和具有图形界面的 Eclipse CDT 集成开发环境。然后本章将重点介绍 C 语言的基础知识，为后续章节的学习打下基础。如果读者已经具备了 C 语言编程的基础知识可以跳过该节，直接进入后续章节的学习。本章主要内容包括：

- gcc 编译器
- Eclipse CDT 集成开发环境
- C 语言的特点
- 数据类型
- 运算符和表达式
- 语句
- 控制结构
- 函数
- 内存管理
- 编译预处理

4.1 gcc 编译器

4.1.1 概述

在 Linux 开发环境下，gcc 是进行 C 语言程序开发不可或缺的编译工具。gcc 的名字取自 GNU C Compile 的首字母，是 Linux 系统下的标准 C 编译器。

和 Shell 脚本程序可以 Shell 环境下直接解释执行不同，C 语言程序必须被编译成二进制代码方能执行。使用 C 语言编程要经过编辑、预处理、编译、链接、调试运行等阶段。该过程如图 4-1 所示。

- (1) 编辑，通过文本编辑软件，如 gedit、vi、Emacs 和 Eclipse IDE 等，输入 C 语言程序。
- (2) 预处理，对源程序进行头文件加载和宏展开等操作。预处理过程由预处理器“`cpp`”完成。

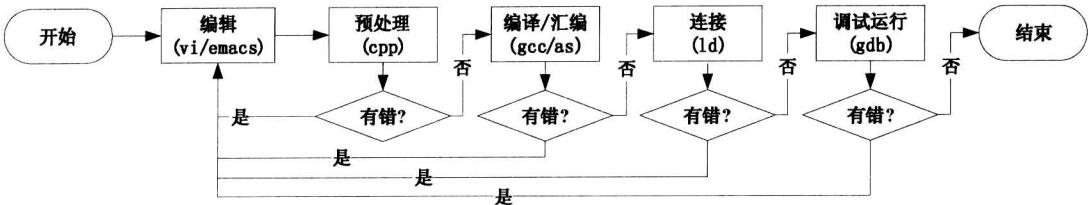


图 4-1 C 语言编程流程

(3) 编译, 将经过预处理的 C 语言程序转化为对应的计算机机器码, 生成二进制的目标文件, 也称.o 文件。编译过程实际分两步完成, 一是产生汇编代码, 二是调用汇编器处理汇编代码从而产生目标文件。gcc 使用的汇编器是 “as”。

(4) 链接, 将编译生成的多.o 文件和使用到的库文件链接成为可被操作系统执行的可执行程序 (Linux 环境下为 “ELF” 格式)。链接过程使用 GNU 的 “ld” 工具。链接过程可能会使用到两类库文件。

- 静态库: 又称为文档文件 (Archive File)。它是多个.o 文件的集合。Linux 中静态库文件的后缀为 “.a”。静态库中的各个成员 (.o 文件) 没有特殊的存在格式, 仅仅是一个.o 文件的集合。使用 “ar” 工具维护和管理静态库。

- 共享库: 也是多个.o 文件的集合, 但是这些.o 文件由编译器按照一种特殊的方式生成 (Linux 中为 “ELF” 格式)。多个可执行程序可共享库文件的代码段 (不共享数据)。

(5) 运行, 将由链接生成的可执行程序加载到内存, 由 CPU 调度运行。

(6) 调试, 对程序中的运行时错误和逻辑错误进行排查与纠正的过程。调试过程使用 GNU 的 gdb 工具。

4.1.2 第一个 C 程序

同上一章一样, 读者可先在自己家目录下建立一个 “c” 目录专门放置 C 语言程序文件, 为此输入以下命令:

```
jianglinmei@ubuntu:~/sh$ mkdir c
jianglinmei@ubuntu:~/sh$ cd c
```

下面开始第一个 C 语言程序, 使用任一文本编辑器, 输入 first.c 的程序代码如程序清单 4-1 所示。

程序清单 4-1 first.c

```
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
}
```

编译 first.c 生成可运行程序 first, 命令如下。

```
jianglinmei@ubuntu:~/c$ gcc -o first first.c
jianglinmei@ubuntu:~/c$ ./first
Hello world!
```

gcc 的“-o”选项用于指定输出的可执行文件的名称，如果不指定的话，将生成默认的名为 a.out 文件。运行程序时应使用“./”指定可执行文件的路径。如果要直接运行生成的可执行文件，可以将当前目录“.”加入到环境变量“PATH”中，如：

```
jianglinmei@ubuntu:~/c$ PATH=.:$PATH
jianglinmei@ubuntu:~/c$ first
Hello world!
```

4.1.3 编译选项

除了上一节小结介绍的“-o”选项外，gcc 还支持很多别的选项，用以进行更完全的编译控制。gcc 的一般格式是：

```
gcc [选项...] 文件...
```

gcc 的选项众多，常用的选项及其含义如表 4-1 所示。

表 4-1 gcc 常用选项

选 项	含 义
--help	显示命令帮助说明
--version	显示编译器版本信息
-o <文件>	指定输出文件名，缺省设置为“a.out”
-D MACRO	定义宏 MACRO
-E	仅进行预处理，不进行其他操作
-S	编译到汇编语言，不进行其他操作
-c	编译、汇编到目标代码，不进行链接
-g	在可执行文件中包含标准调试信息
-Wall	尽可能多地显示警告信息
-Werror	将所有的警告当作错误处理
-w	禁止所有警告
-ansi	采用标准的 ANSI C 进行编译
-l library	设定编译所需的库名称，如果一个库的文件名为“libxxx.so”那么它的库名称为“xxx”
-I path	设置头文件的路径，可以设置多个，默认路径“/usr/include”
-L path	设置库文件的路径，可以设置多个，默认路径“/usr/lib”
-static	使用静态链接，编译后可执行程序不依赖于库文件
-O N	优化编译，主要提高可执行程序的运行速度，N 可取值为 1、2、3
-Q	显示各个阶段的执行时间

下面介绍几个最常用的选项的用法。

1. -E 选项

该选项指示编译器仅对输入的文件进行预处理，且预处理的结果默认输出到标准输出设备，当指定了“-o”选项时才输出到“-o”选项指定的文件中。注意，预处理的结果仍是一个 C 语言源文件。例如：

```

jianglinmei@ubuntu:~/c$ gcc -o p.c -E first.c
jianglinmei@ubuntu:~/c$ cat p.c
# 1 "first.c"
# 1 "<built-in>"
# 1 "<命令行>"
# 1 "first.c"
# 1 "/usr/include/stdio.h" 1 3 4
.....此处省略若干源码
# 940 "/usr/include/stdio.h" 3 4

# 2 "first.c" 2

int main(void)
{
    printf("Hello world!\n");
    return 0;
}

```

可以发现，此处预处理仅对包含的头文件<stdio.h>进行了文件加载和内容替换。在预处理的结果的最后才是 first.c 文件的主要内容 main()函数的定义。

使用“-E”选项进行编译有助于解决因预处理指令书写不当，尤其是宏定义不当，所引起的错误。

2. -S 选项

该选项指示 gcc 在产生了汇编语言文件后即停止编译，产生的汇编语言的默认文件扩展名为“.s”。例如：

```

jianglinmei@ubuntu:~/c$ gcc -S first.c
jianglinmei@ubuntu:~/c$ ls -l first.s
-rw-rw-r-- 1 jianglinmei jianglinmei 491 10月 1 23:30 first.s
jianglinmei@ubuntu:~/c$ cat first.s
    .file    "first.c"
    .section      .rodata
.LC0:
    .string  "Hello world!"
    .text
    .globl  main
    .type   main, @function
main:
.....此处省略若干汇编指令
.ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
.section      .note.GNU-stack,"",@progbits

```

gcc 使用的是 AT&T 语法格式的汇编语言，使用 gcc 的“-E”选项有助于对特殊的代码进行手工优化，以提高程序执行的效率，也有益于对 AT&T 汇编语言的学习。

3. -c 选项

该选项指示 gcc 只把源代码(.c文件)编译成目标代码(.o文件)，不继续链接操作。使用“-c”选项有助于加快编译的速度，也方便使用 Make 文件（用于对中大型软件项目的多源文件进行组织管理的文件，其作用是规定编译的详细过程）对编译过程进行组织和管理。例如：

```
jianglinmei@ubuntu:~/c$ gcc -c first.c
```

```
jianglinmei@ubuntu:~/c$ ls -l first.o
-rw-rw-r-- 1 jianglinmei jianglinmei 1028 10月 1 23:47 first.o
```

4. -W 选项

指定“-Wall”选项，gcc 将显示所有的警告信息，例如：

```
jianglinmei@ubuntu:~/c$ gcc -Wall -o first first.c
first.c: 在函数 ‘main’ 中:
first.c:6:1: 警告：在有返回值的函数中，控制流程到达函数尾 [-Wreturn-type]
```

该警告指明了 main 函数声明了有返回值，但实际上却没有返回语句。

指定“-Werror”选项，gcc 会将所有的警告当作错误处理，当有错误存在时，gcc 不会生成目标文件。例如：

```
jianglinmei@ubuntu:~/c$ rm first
jianglinmei@ubuntu:~/c$ ls
1.txt first.c first.o first.s p.c
jianglinmei@ubuntu:~/c$ gcc -Werror -Wall -o first first.c
first.c: 在函数 ‘main’ 中:
first.c:6:1: 错误：在有返回值的函数中，控制流程到达函数尾 [-Werror=return-type]
cc1: all warnings being treated as errors
jianglinmei@ubuntu:~/c$ ls
1.txt first.c first.o first.s p.c
```

可见，因警告被视为错误，目标文件 first 没有生成。将程序清单 4-1 的代码更正，如程序清单 4-2 所示。

程序清单 4-2 first.c

```
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

重新加上“-Wall”选项编译 first.c 可发现不会有任何警告信息出现，目标文件也正确生成了。

5. -g 选项

初写的程序中出现错误是在所难免的，为解决程序错误（尤其是逻辑错误）需要一定的调试工具和手段。GNU 为 Linux 环境下的 C 语言程序提供了一个调试工具 gdb(GNU Debugger)。gdb 的功能非常强大，但使用 gdb 调试有一个前提，那就是需要在编译结果中包含调试符号，这些调试符号使得 gdb 能够判断目标程序与程序源代码之间的关系。

默认情况下，gcc 不会把调试符号加入到编译结果中，因为那样会增大可执行文件的体积，而且会降低可执行文件的执行效率。使用 gcc 的“-g”选项编译源代码，可以在生成的可执行程序中插入使用 gdb 进行调试所需的调试符号。例如：

```
jianglinmei@ubuntu:~/c$ gcc -o first first.c
jianglinmei@ubuntu:~/c$ ll first
-rwxrwxr-x 1 jianglinmei jianglinmei 7159 10月 6 22:46 first*
```

```
jianglinmei@ubuntu:~/c$ gcc -g -o first first.c
jianglinmei@ubuntu:~/c$ ll first
-rwxrwxr-x 1 jianglinmei jianglinmei 8043 10月 6 22:46 first*
```

从上面的显示结果可以看出，使用“-g”选项编译 first.c 生成的目标程序（8043 字节）要比不使用“-g”选项编译所生成的目标程序（7159 字节）大不少，如果源程序很大则两者的差别将更大。

gdb 是一种基于命令的调试器，在调试过程中需输入各式命令进行断点设置、单步运行、查看变量信息等调试操作。启动 **gdb** 对含调试符号的可执行文件进行调试的操作示例如下。

```
jianglinmei@ubuntu:~/c$ gcc -g -o first first.c
jianglinmei@ubuntu:~/c$ gdb first
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
.....此处省略若干关于 gdb 的说明信息
Reading symbols from /home/jianglinmei/c/first...done.
(gdb) list 列出源代码
1     #include <stdio.h>
2
3     int main(void)
4     {
5         printf("Hello world!\n");
6         return 0;
7     }
(gdb) run 运行
Starting program: /home/jianglinmei/c/first
Hello world!
[Inferior 1 (process 3480) exited normally]
(gdb) quit 退出
```

因本书重点针对于 Linux 环境编程的初学者，拟采用 Eclipse CDE 作为集成开发环境，对于 **gdb** 的详细使用方法将不做详细介绍。

6. -I 选项

gcc 一般在默认的路径 “/usr/include” 下查找头文件，如果需要在额外的路径查找头文件，可以使用-I 选项来指定。例如，gtk 的头文件 “gtk.h” 位于目录 “/usr/include/gtk-2.0/gtk/” 下，如果要包含该头文件应在程序中加上源代码：“#include <gtk-2.0/gtk/gtk.h>”，当需要包含 gtk 目录下更多的文件时，这种指定方式就显得烦琐。为此，可以将 “/usr/include/gtk-2.0” 指定为 **gcc** 的头文件查找目录，在程序中则只需写：“#include <gtk/gtk.h>”。命令如下。

```
jianglinmei@ubuntu:~$ gcc -I /usr/include/gtk-2.0/ -c -o test.o test.c
```

7. -L 和 -l 选项

gcc 一般在默认的路径 “/usr/lib” 下查找库文件，如果需要在额外的路径查找库文件，可以使用-L 选项来指定。但 “-L” 选项只是指明到哪里去找库文件，并未指出要连接哪个库，因此还需要使用 “-l” 选项来指定要连接的库的名称。库名称和库文件名是相关联的，如果一个库的文件名为 “libxxx.so” 那么它的库名称为 “xxx”，即文件名去掉 “lib” 前缀和 “.so” 或 “.a” 后缀即为库名称。例如，当需要连接 X11 库时，可以使用如下命令。

```
jianglinmei@ubuntu:~$ gcc -L /usr/lib/i386-linux-gnu/ -lX11 -o test test.c
```

4.2 Eclipse CDT

4.2.1 简介、安装和启动

Eclipse CDT (C/C++ Developing Tooling) 是一个基于 Eclipse 平台的全功能的 C/C++ 集成开发环境。Eclipse CDT 包含了以下特性：项目创建和各种工具链托管的项目（Project）建造（build）；标准的制作（make）建造；源码导航；各种源码信息查看工具，如，类型分级结构、调用关系图、包含文件浏览器、宏定义浏览器、语法高亮的代码编辑器、可折叠导航及超链接导航、源码重构和代码生成等；可视化的调试工具，包括内存查看器、寄存器查看器和反汇编查看器等。

下面介绍 Eclipse CDT 在 Unbutu12.04 环境下的安装方法。

在 Unbutu12.04 下一般使用 apt 工具从互联网获取并安装软件。为此，应首先保证连上互联网，然后打开终端，在命令行上输入以下命令更新软件包列表。

```
jianglinmei@ubuntu:~$ sudo apt-get update
```

继而输入以下命令更新升级软件包。

```
jianglinmei@ubuntu:~$ sudo apt-get upgrade
```

因更新升级过程需要下载软件包并安装，所以要等待较长时间。更新升级过程结束后，就可以安装 Eclipse CDT 了，输入以下命令来完成安装。

```
jianglinmei@ubuntu:~$ sudo apt-get install eclipse-cdt
```

安装完毕后，打开 Dash 主页（参见本书第 1.3.1 小节），并在其搜索框中输入“eclipse”，如图 4-2 所示，然后单击“Eclipse”图标即可启动 Eclipse。

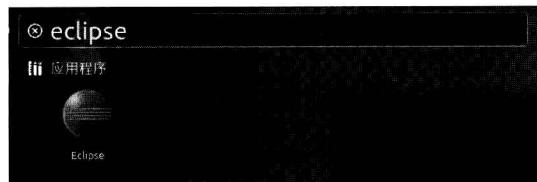


图 4-2 从 Dash 主页启动 Eclipse

Eclipse 初次启动时会要求用户设置工作空间（workspace）的位置，如图 4-3 所示。工作空间是 Eclipse 中软件项目的所有文件所存放的目录，工作空间同时也是 Eclipse 进行资源管理的一个环境，一般保持默认设置（用户家目录下的“workspace”主目录）即可。

设置工作空间位置后，单击“OK”按钮后进入 Eclipse。其初始界面是一个如图 4-4 所示的“Welcome”欢迎页面，直接单击“Welcome”标签右边的“×”关闭该页即进入 Eclipse 的主界面，如图 4-5 所示。初次启动的 Eclipse 主界面包含 4 个窗格。左上角为项目资源管理器（Project Explorer），用于列表当前工作空间下的所有项目及项目文件。右上角是一个空白的源码编辑器。左下角是大纲视图窗格，用于详细显示当前编辑项的层次信息。右下角是任务窗格，用于给出对当前项目所存在的问题的提示信息。

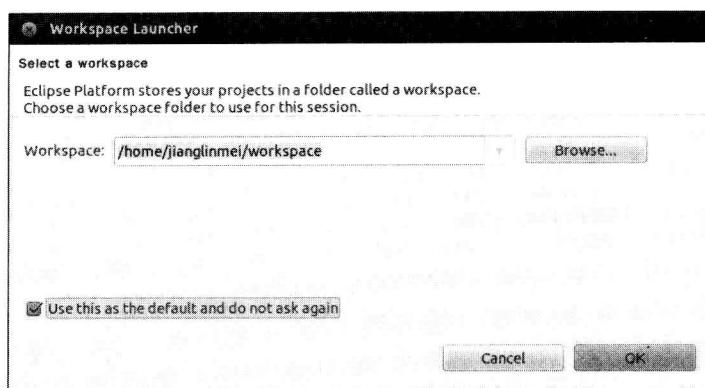


图 4-3 设置工作空间位置



图 4-4 Eclipse 欢迎页

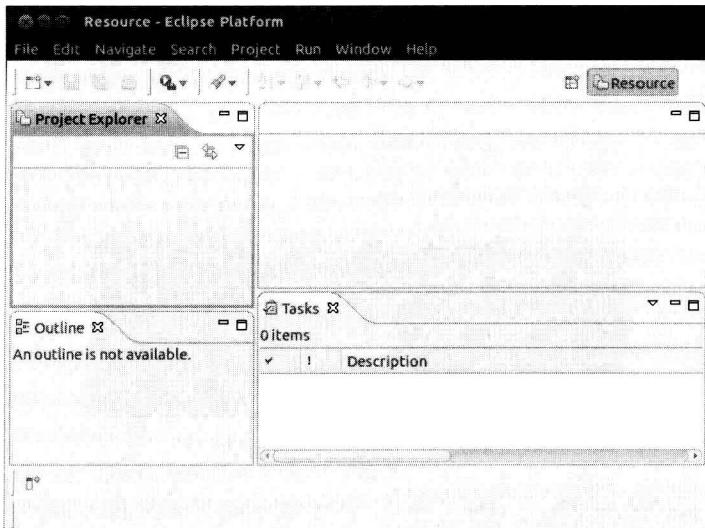


图 4-5 Eclipse 主界面

4.2.2 创建项目并运行

和大多数集成开发环境一样，Eclipse CDT 是以项目（Project）的方式来组织管理程序的。本节以一个“HelloWorld”项目为例来介绍使用 Eclipse CDT 进行 C 程序开发的整体过程。

使用 Eclipse CDT 编写程序的第一步是创建项目。为此，选择菜单“File->New->Project...”，将弹出如图 4-6 所示的新建项目对话框。展开列表中的“C/C++”项，选择其中的“C Project”，

单击“Next>”，将出现如图 4-7 所示的 C 项目设置对话框。在项目名称（Project name）文本中输入“HelloWorld”，项目类型列表中选择“Executable/Empty Project”，工具链（Toolchains）列表框中选择“Linux GCC”，单击“Finish”完成项目创建。随后将出现如图 4-8 所示的“确认打开 C/C++透视图（Perspective）”对话框。透视图是 Eclipse 界面组织的一种方式，不同的透视图的界面构成不尽相同，“C/C++透视图”透视图是最适合进行 C/C++编程开发的一种视图，因此勾选“Remember my decision”并单击“Yes”，将显示如图 4-9 所示的“C/C++透视图主界面”。

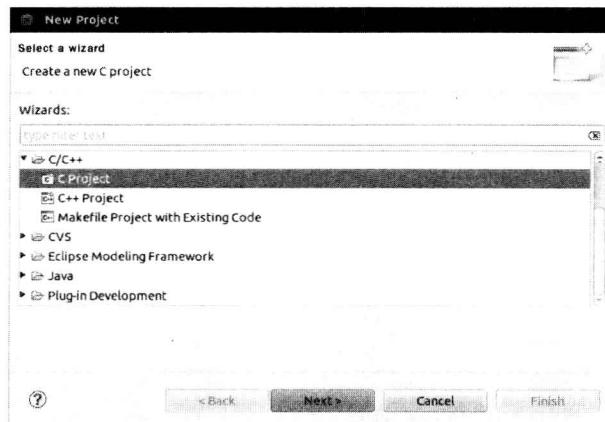


图 4-6 新建项目

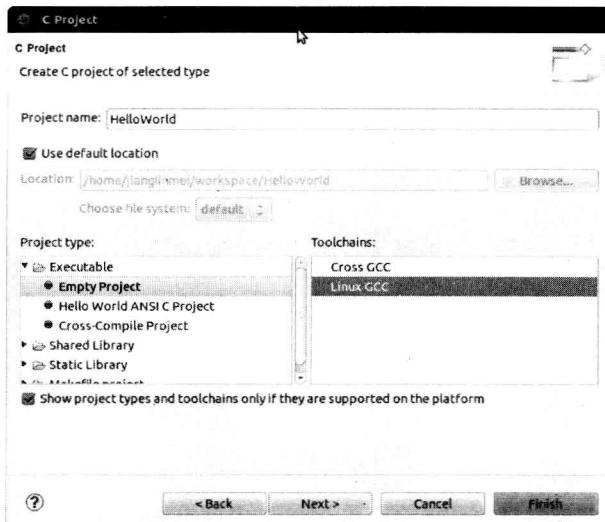


图 4-7 C 项目设置

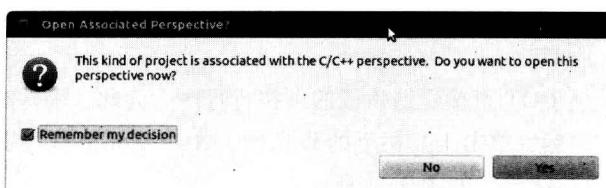


图 4-8 确认打开 C/C++透视图

以上创建了一个空白的 C 项目 (Empty Project)，接下来给项目添加源程序文件。为此，选择菜单“File->New->Source File”，将弹出如图 4-10 所示的新建源文件对话框。在“Source file”文本框中输入“hw.c”并单击“Finish”按钮。创建源文件后，主界面的“项目浏览器”中会显示源文件的名称，并可在“源代码编辑器”中编辑该源文件的代码。为此，输入本示例的代码如图 4-11 所示。

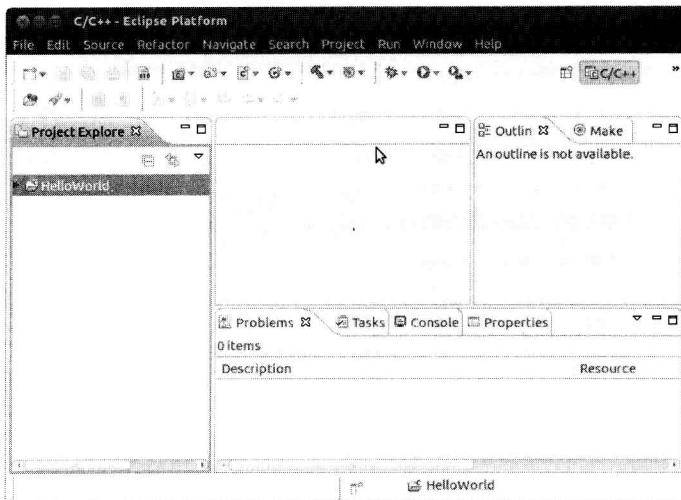


图 4-9 C/C++ 透视图主界面

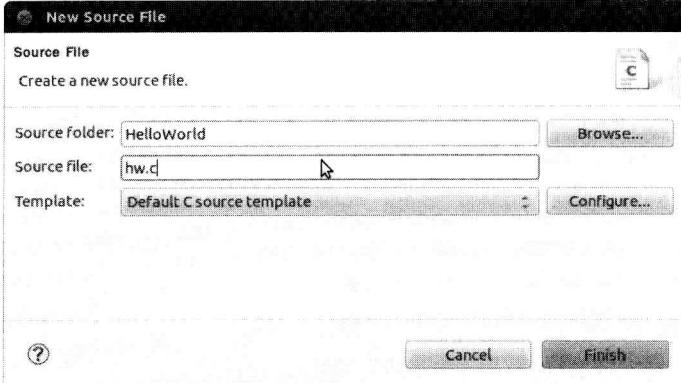


图 4-10 新建源文件

代码输入完毕后，应进行项目建造（build）以生成最终的可执行程序。为此，单击工具栏上的 按钮，注意观察位于主界面下方的控制台（Console）窗格，建造过程会在该窗格中显示一些提示信息，建造完毕后会显示“*****Build Finished****”，表示目标代码和可执行程序已生成，如图 4-12 所示。

最后，可以在 Eclipse CDT 直接建造生成的可执行程序。为此，确保在“项目浏览器”中选中项目名“HelloWorld”，然后单击工具栏上的 按钮，HelloWorld 程序即开始执行并在控制台（Console）窗格中输出执行结果，如图 4-13 所示。

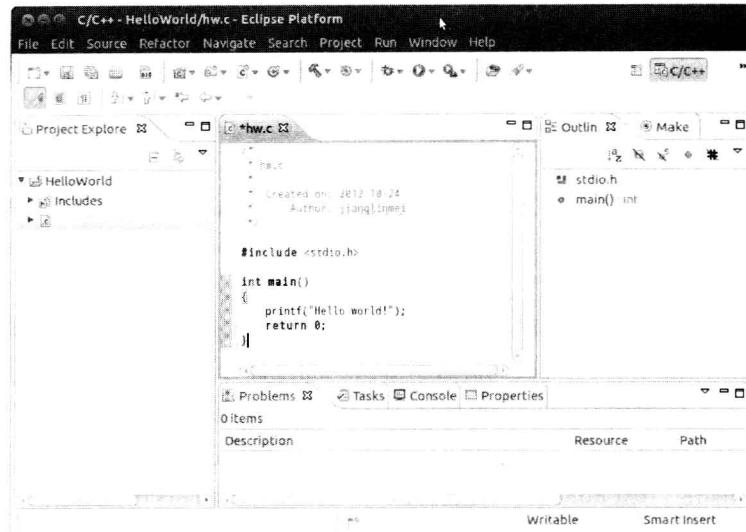


图 4-11 编辑源代码

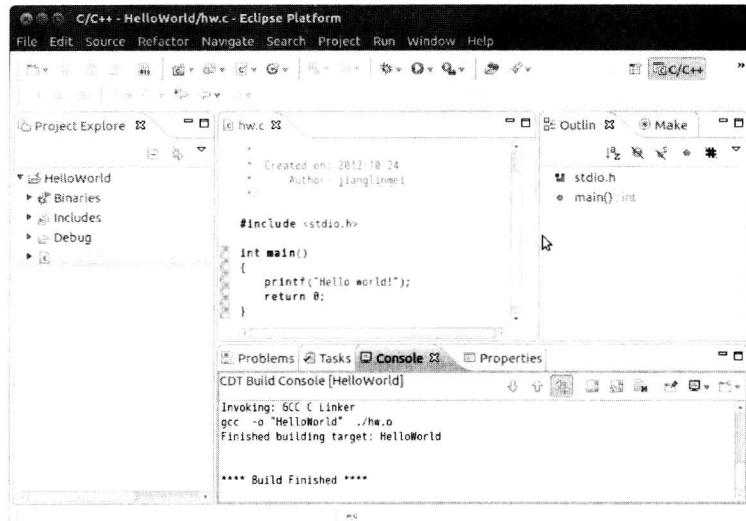


图 4-12 建造 (build) 项目

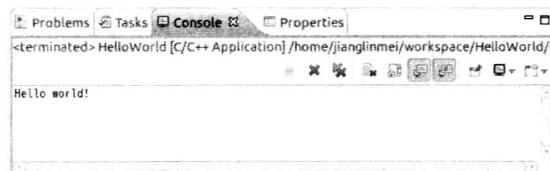


图 4-13 运行结果

除了在 Eclipse CDT 环境中直接运行建造生成的项目可执行程序外，也可以在 Linux 终端上运行它。为此，打开 Linux 终端，切换当前目录到 HelloWorld 项目所在目录下的 Debug 目录，并查看该目录下的文件，命令如下。

```
jianglinmei@ubuntu:~$ cd workspace/HelloWorld/Debug/
jianglinmei@ubuntu:~/workspace/HelloWorld/Debug$ ls
```

```
HelloWorld hw.d hw.o makefile objects.mk sources.mk subdir.mk
```

其中的“HelloWorld”文件即为项目最终的可执行程序文件，执行该文件如下。

```
jianglinmei@ubuntu:~/workspace/HelloWorld/Debug$ ./HelloWorld
Hello world!
```

到此，一个完整项目的创建和运行过程就结束了。下一小节将介绍程序调试的方法。

4.2.3 程序调试方法

程序调试的方法包括：断点设置、单步执行、变量监视、调用栈查看、内存查看、寄存器查看和反汇编等。本小节介绍程序调试的最基本方法：断点设置、单步执行和变量监视。

为方便介绍，需要更改 Eclipse 的首选项以显示源代码的行号。选择菜单“Window -> Preferences”，将弹出 Eclipse 的“首选项设置”对话框，展开左侧的列表框到“General -> Editors -> Text Editors”并单击“Text Editors”，勾选右侧的“Show line numbers”复选框，如图 4-14 所示。

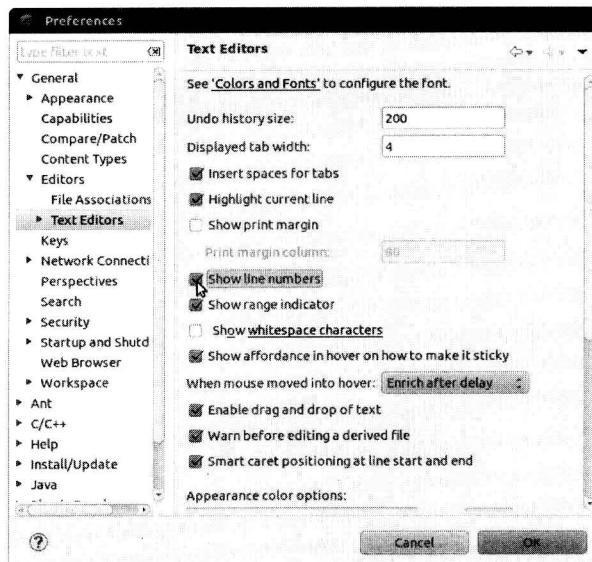


图 4-14 首选项设置

更改上一小节的 hw.c 文件的代码如程序清单 4-1 所示。更改后的 hw.c 由三个函数构成，pr() 函数输出“Hello world！”，sum() 函数计算两个整数之和，main() 函数为程序的入口，其中调用了 pr() 和 sum() 并输出调用 sum() 函数得到的结果。

程序清单 4-3

```
#include <stdio.h>

void pr()
{
    printf("Hello world!\n");
}

int sum(int a, int b)
{
    int c = a + b;
}
```

```

    return c;
}

int main()
{
    int s;

    pr();
    s = sum(2, 5);
    printf("The sum of 2 and 5 is: %d", s);

    return 0;
}

```

代码输入完毕后，建造（build）并运行，结果如图 4-15 所示。

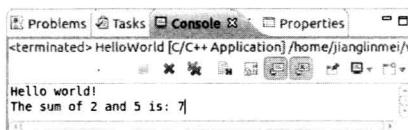


图 4-15 运行结果

下面介绍设置断点并以调试的方式运行程序。

双击源码第 19 行处的左边栏位置，该位置会显示一个蓝色的圆点，表示该行已设置了一个断点，如图 4-16 所示。



图 4-16 设置断点

设置好断点之后，确保“Project Explorer”窗格选中“HelloWorld”，然后单击工具栏上的 按钮，将弹出“确认切换透视图（Confirm Perspective Switch）”窗口，直接单击“Yes”打开“调试（Debug）”透视图，如图 4-17 所示。

调试透视图的左上角是“调试（Debug）”窗格，其中显示了函数调用栈，右上角是“变量监视（Variable）”窗格，中间主体部分是代码查看窗格，右侧则是大纲（Outline）窗格，下方是“控制台（Console）”窗格。进入调试视图后，当前运行焦点停在程序的入口处（代码第 18 行）。

接下来，按“F8”键恢复程序的运行，因第 19 行代码处设置了断点，运行焦点停在了该行。Console 窗格显示了已运行的代码行的输出结果。Variable 窗格中显示了当前可见变量“s”的值，因 s 尚未赋初值，所以其中显示的是一个随机值。再按下“F6”键，单步运行到下一条代码，可

见在 Variable 窗格中 s 的值变成了 7。继续按下“F8”键恢复程序的运行，可见在 Console 窗格中输出了“The sum of 2 and 5 is: 7”，程序运行完毕。以上步骤中，以按“F5”代替按“F6”，则运行焦点可跟进到子函数 sum()的内部，并可在 Variable 窗格中查看 sum()函数中的局部变量值。

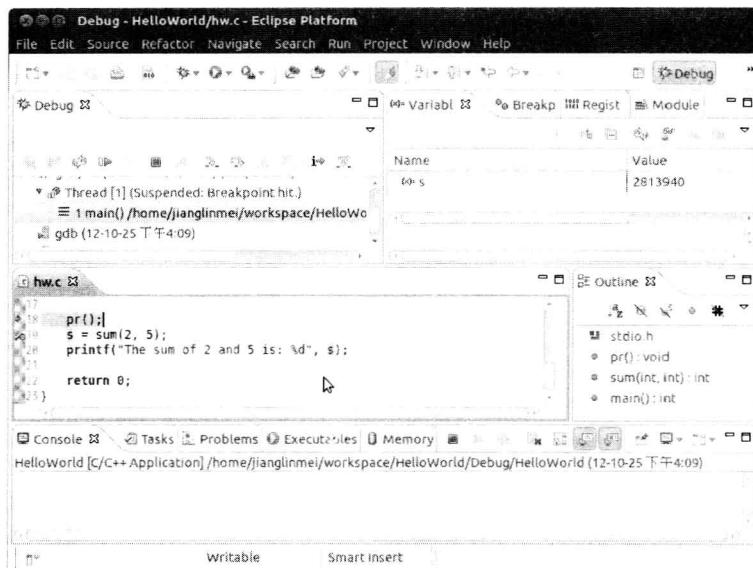


图 4-17 调试 (Debug) 透视图

4.3 C 语言基础

4.3.1 C 语言概述

C 语言用途广泛，既可以编写系统软件，也可以编写应用软件。自 20 世纪 70 年代诞生之日起，C 语言就一直是最为流行的编程语言之一。许多时下流行的编程语言，如 C++、C# 和 Java，也都是从 C 语言发展而来的。

C 语言是一种与 UNIX 操作系统密切相关的程序设计语言，它是由 Dennis M. Ritchie 为开发 UNIX 操作系统的方便而设计出来的。1978 年 Brian W. Kernighan 和 Dennis M. Ritchie 合著出版的《C 程序设计语言 (The C Programming Language)》是当时事实上的 C 语言标准，也是 C 语言编程方面最权威的教材之一。20 世纪 70 年代以来，大多数操作系统（如 UNIX、Windows 和 Linux 等）的内核的大部分内容都是用 C 语言编写的。

C 语言得以长期存在和发展与其所具有的以下优点是密不可分的。

- 语言简洁、紧凑、灵活（32 个关键字，9 种控制语句）。
- 运算符丰富（34 种），表达式类型多样、灵活、简练。
- 数据结构丰富、合理，能够方便地实现链表、树、栈、队列和图等各种复杂的结构。
- 具有结构化的控制语句，符合现代编程风格。
- 兼有高级语言和低级语言的特点。
- 可移植性好。

- 目标代码质量高，程序执行效率高。
- 语法限制不太严格，程序设计自由度大。

4.3.2 数据类型

瑞士计算机科学家图灵奖获得者沃思（Niklaus Wirth）曾提出“数据结构 + 算法 = 程序”这一著名公式，可见数据结构在程序设计中的地位。C 语言既提供了丰富的数据结构，也允许用户自定义复杂的数据结构。C 语言中的数据结构是以“数据类型”的形式来表示的。C 语言所包括的数据类型如图 4-18 所示。

数据类型决定了数据可参与的运算、所表示的数的范围和所占用的内存大小。在不同的操作系统平台上，相同的数据类型所占用的内存大小可能是不一样的。本书后文所介绍的数据类型均将基于 32 位 Linux 平台。

4.3.2.1 常量与变量

C 语言中的数据有常量和变量之分。在程序运行过程中，其值不可改变的量称为常量，反之，其值可以改变的量称为变量。常量和变量均可分为不同的类型。

常量可分为直接常量和符号常量。直接常量从其字面形式即可判别，如整型常量 2、浮点型常量 1.23456 和字符型常量 “A” 等。符号常量用一个标识符来代表一个固定不变的量，其优点是使得程序的可读性更好。标识符是程序中有具体语义的项（如常量、变量、标号、宏和函数等）的名称，C 语言规定标识符只能由字母、数字、下划线组成，并且只能由字母、下划线开头。

使用如下形式来定义一个符号常量。

```
#define 常量名 常量值
```

例如：

```
#define PI 3.1415926 /* 定义符号常量 PI，用以代替 3.1415926 这个常量值 */
```

注意，这里 “/*” 和 “*/” 的作用是括起一个注释，其间的内容一般用于解释代码的含义或该部分的程序设计思想，不会被编译。



图 4-18 C 语言的数据类型

使用如下形式来定义一个变量。

```
数据类型 变量名
```

例如：

```
int a;          /* 定义整型变量 a */
float b;        /* 定义浮点型变量 b */
```

变量应“先定义，后使用”。使用未定义过的变量，编译器将给出错误提示，并导致编译无法完成。

4.3.2.2 整型

整型可细分为基本型（int）、短整型（short）、长整型（long）和无符号型。基本型占4个字节、短整型占2个字节、长整型占8个字节。基本型、短整型和长整型均有对应的无符号型：unsigned int、unsigned short 和 unsigned long，无符号型数据的最高位不用作符号位。

- 整型常量

整型常量有三种表示形式。

- (1) 十进制整数。由数字0~9和正负号表示，如：123，-456，0。
- (2) 八进制整数。由数字0开头，后跟数字0~7表示，如：0123，011。
- (3) 十六进制整数。由0x或0X开头，后跟0~9，a~f，A~F表示，如：0x123，0Xff。

整型常量默认为int型，若其值所在范围确定超出了int的表示范围，则其类型为long型。如：123是int型，0x123456789是long型。

另外，也可以在整常量后加上字母l或L，指明它是long型，如：1234L。

- 整型变量

整型变量的定义形式为：

整型数据类型 变量名

其中整型数据为类型可以为int、long或short，并且可以在这三个关键字前加上关键字signed和unsigned指明是有符号整型还是无符号整型，当使用signed或unsigned时，默认为有符号整型。例如：

```
int x, y;           /* 定义 x, y 为整型变量 */
unsigned short m, n; /* 定义 m, n 为无符号短整型变量 */
long a;             /* 定义 a 为长整型变量 */
```

4.3.2.3 浮点型

浮点型可分为单精度浮点型（float）和双精度浮点型（double）。单精度浮点型占4个字节和双精度浮点型占8个字节。

- 浮点型常量

浮点型常量有两种表示形式。

(1) 十进制形式。这种形式由整数部分、小数点和小数部分组成，如：1.24、0.345、.222、234.0、333.0、0.0等。

(2) 指数形式。形如<实数部分>e<指数部分>，由实数部分、字母E或e、整数部分组成，常用于表示比较大的数，如：1.23E2、314.15E-2、0E3等。应注意，E之后的数字必须为整数，该数表示的是10的几次方。

浮点型常量默认为double型，可以在浮点型常量后加上字母f或F，指明它是float型，如：

1.234F。

- 浮点型变量

浮点型变量的定义形式为：

`double 变量名`

或

`float 变量名`

例如：

```
double x, y;           /* 定义 x, y 为双精度浮点型变量 */
float m, n;            /* 定义 m, n 为单精度浮点型变量 */
```

4.3.2.4 字符型

字符型可分为有符号字符型（char）和无符号字符型（unsigned char），字符型占1个字节。本质上，字符型所表示数据是该字符所对应的ASCII值，是一个整数值。

- 字符型常量

字符型常量是用单引号括起来的单个普通字符或转义字符，普通字符如：“A”、“a”、“8”等，转义字符以反斜线（“\”）后面跟一个字符或一个代码值表示，常见的转义字符以及代码值表示方式如表4-2所示。

表4-2 转义字符

转义字符	含 义	转义字符	含 义
\n	换行	\t	水平制表符
\v	垂直制表符	\b	退格
\r	回车	\f	走纸换页
\a	响铃	\\"	反斜线
\'	单引号	\"	双引号
\ddd	3位8进制数ddd代表的字符	\xhh	2位16进制数hh代表的字符

- 字符串常量

字符串常量是使用双引号括起来的字符序列，如：“abcd”。一个字符串常量由多个字符常量组成，字符串常量的字符序列末尾隐含了一个空字符'\0'（ASCII值为0），'\0'是所有字符串的结束标志。因此字符串“a”实际包含了两个字符'a'和'\0'，占用两个字节。

- 字符型变量

一个字符型变量用以存放一个字符，其定义形式为：

`char 变量名`

例如：

```
char d;           /* 定义 d 为字符型变量 */
```

程序清单4-4说明了字符型常量、字符串常量和字符变量的使用方式。

程序清单 4-4 ex_char.c

```

1 #include <stdio.h>
2
3 int main()
4 {
5     char c1, c2;
6
7     c1 = 65;
8     c2 = 'a';
9
10    /* printf() 是一个标准库函数, 用以格式化输出数据 */
11    /* 其中的%d 表示以十进制数值的形式输出变量值, */
12    /* %c 表示以字符形式输出变量值 */
13    printf("0101\tc1\tc1-ASCII\n");      /* \t 为转义字符 */
14    printf("\101\t%c\t%d\n", c1, c1);   /* \101 为以八进制表示的转义字符'A' */
15    printf("0X61\tc2\tc2-ASCII\n");
16    printf("\x61\t%c\t%d\n", c2, c2);   /* \x61 为以十六进制表示的转义字符'a' */
17
18    return 0;
19 }

```

使用 gcc 编译并执行, 命令如下。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_char ex_char.c
jianglinmei@ubuntu:~/c$ ./ex_char
0101      C1      C1-ASCII
A          A      65
0X61      C2      C2-ASCII
a          a      97

```

4.3.2.5 布尔型

布尔型只有“真”和“假”两个值, C 语言使用整数 1 表示“真”, 使用整数 0 表示“假”。在进行逻辑判断时, C 语言将任何非零值都认为是“真”, 而将零值认为是“假”。例如:

```

printf("%d\n", 3 > 2);           /* 结果为 1 */
printf("%d\n", 3 > 2 > 1);       /* 结果为 0 */
printf("%d\n", 3 & 2);           /* 结果为 1 */
printf("%d\n", 3 && 0);         /* 结果为 0 */

```

4.3.2.6 枚举型

“枚举”就是把可能值一一列举出来。枚举类型变量的取值范围只限于所列举出来的值。C 语言使用 enum 关键字来定义枚举类型, 例如:

```
enum week {sun, mon, tue, wed, thu, fri, sat};
```

这定义了一个名为 week 的枚举类型, 用以表示一周的七天。花括号括起以逗号分隔的标识符称为枚举元素, 枚举元素具有固定的整数值, 因此枚举元素也称枚举常量。注意, 在右花括号后的“;”是必不可少的语法组成部分。默认情况下, 第一个枚举元素的值为 0, 其他枚举元素的值是其前一个元素的值加 1 后得到的值。例如: sun 的值为 0, wed 的值为 3。也可以在枚举元素后用“=”为其指定一个值, 例如:

```
enum week {mon=1, tue, wed, thu, fri, sat, sun=0};
```

这样定义的 week 类型和前面定义的 week 类型各枚举元素对应地具有相同的值。

在定义好枚举类型之后，即可使用该类型来定义枚举变量。例如：

```
enum week weekday;
```

这定义了一个名为 weekday 的 week 枚举类型的变量，于是可以给 weekday 赋 week 枚举类型的七个枚举值之一，例如：

```
weekday = wed;
printf("%d", weekday); /* 结果为 3 */
```

4.3.2.7 数组

数组 (array) 是由 $n (n \geq 1)$ 个具有相同数据类型的数据元素 a_0, a_1, \dots, a_{n-1} 构成的一个有序序列(集合)。数组由一个统一的数组名来标识，数组中的某个元素由数组名和一个下标 (index) 唯一标识。

标记数组元素的下标个数决定了数组的维数，即下标个数为一个，则为一维数组，下标个数为二个，则为二维数组等。数组的所有元素存储在一块地址连续的内存单元中，最低地址对应首元素，最高地址对应末元素。

- 一维数组

1. 一维数组的定义格式

数据类型 数组名 [整型常量表达式];

如：

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

这定义了一个整型数组 a，它含有 5 个元素 (即数组长度为 5)，下标从 0 ~ 4，即 $a[0] \sim a[4]$ 。

2. 一维数组的初始化

在定义数组时可使用一对花括号括起以逗号分隔的元素的形式为数组初始化。例如：

```
int a[5] = {0, 1, 2, 3, 4};
```

这样初始化后， $a[0] = 0, a[1] = 1, a[2] = 2, a[3] = 3, a[4] = 4$ 。如果花括号中值的个数少于数组的长度 5 (这种不提供所有元素值的初始化方式称为“部分初始化”)，则后面无对应值的元素的初值为一个随机数。但如果是静态数组，则后面无对应值的元素的初值为 0。静态数组的定义并初始化的方式示例如下。

```
static int a[5] = {0, 1, 2};
```

这样对静态数组 a 进行部分初始化后， $a[0] = 0, a[1] = 1, a[2] = 2, a[3] = 0, a[4] = 0$ 。

3. 一维数组的引用

C 语言中，只能逐个引用数组元素，而不能一次引用数组的所有元素。例如：

```
static int a[3] = {7, 8, 9};
printf("a[0]=%d a[1]=%d a[2]=%d\n", a[0], a[1], a[2]); /* 输出 a[0]=7 a[1]=8 a[2]=9 */
printf("a=%d \n", a); /* 错误 */
```

- 二维数组

1. 二维数组的定义格式

数据类型 数组名[整型常量表达式][整型常量表达式]；

如：

```
int a[2][3];
```

这定义了一个二维的整型数组 a，它含有 2 行 3 列共 6 个元素（即数组长度为 6），行下标从 0~1，列下标从 0~2，即 a[0][0]、a[0][1]…a[1][2]。

2. 二维数组的初始化

在定义数组时可使用两层花括号括起以逗号分隔的元素的形式为二维数组初始化，内层的每对花括号对应一行。例如：

```
int a[2][3] = {{0, 1}, {2, 3}, {4, 5}};
```

这样初始化后，a[0][0] = 0，a[0][1] = 1，a[0][2] = 2，a[1][0] = 3，a[1][1] = 4，a[1][2] = 5。

也可以只用一对花括号为二维数组赋初值，此时按元素的排列顺序（行序优先，即排完一行的所有元素再排下一行的元素）依次为各元素赋初值。和一维数组一样，二维数组也可以部分初始化。例如：

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

这样对二维静态数组 a 进行部分初始化后，a[0][0] = 0，a[0][1] = 1，a[0][2] = 2，a[1][0] = 3，a[1][1] = 0，a[1][2] = 0。

3. 二维数组的引用

和一维数组一样，只能逐个引用二维数组的各个元素，而不能一次引用数组的所有元素。例如：

```
static int a[1][2] = {6, 7};
printf("a[0][0]=%d a[0][1]=%d \n", a[0][0], a[0][1]); /* 输出 a[0][0]=6 a[0][1]=7 */
```

- 字符数组

字符数组是一类特殊的数组，每个元素存放一个字符。

1. 字符数组的定义格式

```
char 数组名[整型常量表达式];
```

如：

```
char str[5];
```

这定义了一个字符数组 str，它可以存放 5 个字符。

2. 字符数组的初始化

字符数组和一般数组一样，可以使用花括号对元素逐个初始化。例如：

```
char str[5] = {'H', 'e', 'l', 'l', 'o'};
```

也可以用字符串常量为字符数组初始化。例如：

```
char str[6] = "Hello";
```

应当注意，用字符串常量为字符数组初始化时，要保证字符数组的长度足够容纳字符串常量末尾隐含的结束标志字符'\0'。

3. 字符数组的引用

字符数组的引用有其特殊性，可以逐个引用字符数组的各个元素，也可以以字符串的形式一次引用字符数组的所有元素，此时应保证字符数组中包括值为'\0'的元素。例如：

```
char str[3] = {'H', 'i', '\0'};
printf("str=%c%c\n", str[0], str[1]); /* 输出 str=Hi */
printf("str=%s\n", str);           /* 输出 str=Hi */
```

此处，“%s”表示以字符串形式输出后面对应变量的值。注意，若 str 数组中没有值为'\0'的元素，使用“%s”的形式输出 str 会出错。例如：

```
char str[2] = {'H', 'i'};
printf("str=%s\n", str);          /* 错误 */
```

4.3.2.8 指针

指针是 C 语言的一大特色，也是其区别于其他大多数编程语言的要点。指针的使用非常灵活，要掌握好指针的使用需要下比较大的工夫。

一个变量在内存中的地址就称为该变量的“指针”，通过指针即可找到变量的存储单元，从而可以对变量进行相关操作。

- 一级指针

1. 指针变量的定义

指针变量是用来存放另一个变量的“地址”的变量，当一个指针变量 p 所存放的内容是另一个变量 a 的地址时，称变量 p “指向” 变量 a。指针变量的定义格式如下。

数据类型 *指针变量名；

这里的数据类型指的是指针变量所指向的目标变量的数据类型，例如：

```
int *p;
```

定义了一个整型的指针 p，它可以存放整型变量的地址，也就是可以指向任一整型变量。

2. 指针变量的赋值

定义好指针变量后，应初始化或赋值后才能引用。建议在定义的时候即将指针变量初始化为空值——NULL。NULL 是一个 C 标准库中预定义的符号常量，其值是 0。例如：

```
int *p = NULL;
```

对指针变量进行赋值只有三种情况是合法的。一是赋 NULL 值，二是赋同类型变量的地址，三是赋同类型指针。例如：

```
int a, *p = NULL, *q = NULL, *r = NULL;      /* 定义整型变量 a 和三个整型指针 */
p = &a;           /* 给指针 p 赋同类型变量 a 的地址。“&”是取地址操作符 */
q = p;           /* 给指针 q 赋同类型指针 p，赋值后 p 和 q 都指向 a */
r = 0X12345678; /* 错误，不属于前面提到的三种赋值方式之一 */
```

3. 指针变量的引用

指针变量的引用要使用到“*”操作符（注意，定义指针时用到的“*”不是指针引用，而是指针变量的说明符），指的是引用指针变量所指向的目标的内容。引用指针变量之前应当确保它指向了一个有效的地址单元。例如：

```
int a = 10, *p = NULL;
p = &a;
printf("a=%d, *p=%d", a, *p); /* 输出 a=10, *p=10);
```

- 二级指针

1. 二级指针变量的定义

二级指针变量是用来存放“一级指针变量的地址”的变量，定义格式如下。

数据类型 **指针变量名；

例如：

```
int **pp;
```

定义了一个整型的二级指针 pp，它可以指向任一整型的一级指针变量。定义指针变量时使用了几个“*”，即定义了几级指针变量。例如，使用如下方式则定义了一个三级指针变量。

```
int ***ppp;
```

2. 二级指针变量的赋值

对二级指针变量进行赋值也只有三种情况是合法的。一是赋 NULL 值，二是赋同类型的一级指针变量的地址，三是赋同类型指针。例如：

```
int a, *p =NULL, **pp = NULL;      /* 定义整型变量 a、一级指针 p 和二级指针 pp */
p = &a;
pp = &p;                          /* 给二级指针 pp 赋同类型一级指针 p 地址 */
```

3. 二级指针变量的引用

二级指针变量的引用要连续用两个“*”操作符。例如：

```
int a, *p =NULL, **pp = NULL;
p = &a;
pp = &p;
printf("a=%d, *p=%d, **pp=%d", a, *p, *pp); /* 输出 a=10, *p=10, **pp=10);
```

4.3.2.9 结构体

C 语言中，结构体是多数据项的一种组织方式，是由多个数据项（每个数据项可以是任何 C 语言类型）构成的紧密关联的数据单元。

1. 结构体类型的定义

C 语言使用 struct 关键字来定义结构体类型，其一般形式如下。

```
struct 结构体名
{
    数据类型 1    成员名 1;
    数据类型 2    成员名 2;
```

```
...  
数据类型 n    成员名 n;  
};
```

其中，`struct` 是关键字，“结构体名”是用户定义的类型标识，“`{…}`”中是组成该结构体的成员，在右花括号后的“`;`”是必不可少的语法组成部分。结构体内可以有 0 到任意多个成员，成员的数据类型可以是 C 语言所允许的任何数据类型，每个成员的数据类型可以相同也可以不同。结构体类型所占内存的大小是其所有成员所占内存大小之和。以下是一个结构体类型定义的例子。

```
struct student  
{  
    int number;      /* 学号 */  
    char name[30];   /* 姓名 */  
};
```

这定义了一个名为 `student` 的结构体类型，它包含两个数据成员，一个为整型的 `number` 用以表示学号，一个为字符型数组用以表示姓名。

2. 结构体类型变量的定义

在定义好一个结构体类型之后，即可定义该结构体类型的变量，其一般形式如下。

```
struct 结构体名 变量名;
```

例如：

```
struct student a;
```

定义了一个 `student` 结构体类型的变量 `a`。

3. 结构体成员的引用

和一般数组一样，C 语言中，只能逐个引用结构体的数据成员，而不能一次引用结构体的所有成员。引用结构体成员要使用分量运算符——“`.`”，例如：

```
a.number = 1;          /* 给 a 的 number 成员赋值 1，即学号为 1 */  
a.name[0] = 'T';       /* 以下 4 行给 a 的 name 成员赋字符串"Tom"，即姓名为 Tom */  
a.name[1] = 'o';  
a.name[2] = 'm';  
a.name[3] = '\0';  
/* 下面一行输出 a 的信息，输出结果为：Student a: number=1, name=Tom */  
printf("Student a: number=%d, name=%s\n", a.number, a.name);
```

4. 结构体变量的初始化

和数组一样，可以在定义结构体变量的同时使用花括号的形式进行初始化。例如：

```
struct student a = {1, {'T', 'o', 'm', '\0'}};
```

4.3.2.10 共用体

几个不同的数据成员存放在同一段内存的结构，称为“共用体”。共用体类型的定义方式、共用体变量的定义方式以及共用体成员的引用方式和结构体相类似，区别在于使用“`union`”关键字代替了“`struct`”关键字。例如：

```
union data      /* 定义共用体类型 data */
```

```

{
    int d;
    char ch[4];
};

union data a;           /* 定义共用体 data 类型的变量 a */
a.d = 0x44434241;      /* 给成员 d 赋值 */
printf("%c%c%c%c", a.ch[0], a.ch[1], a.ch[2], a.ch[3]); /* 输出: A BCD */

```

注意，上面示例中，因共用体的成员占用相同的一段内存，故给成员 d 赋值即给成员 ch 赋值，只不过不同的成员因数据类型不同而有不同的使用方式。当各成员类型所占用的内存大小不同时，共用体所占的内存大小是其最大成员所占内存的大小。

4.3.2.11 自定义类型

C 语言中，可以使用 `typedef` 关键字为已存在的类型取一个别名，即自定义一个类型名，自定义类型的用法和功能与已有类型的用法和功能完全一样。自定义类型的语法格式为：

```
typedef 已有数据类型 新类型名;
```

例如：

```
typedef int COUNT;
```

定义了一个新的类型——COUNT，随后可以用 COUNT 来定义整型变量。例如：

```
COUNT a = 10;
printf("a=%d", a);      /* 输出 10 */
```

自定义类型更常见的场合是：为书写较为复杂的类型（如结构体、共用体）取一个别名。例如：

```
typedef struct student STUDENT;
STUDENT a;
STUDENT b;
```

4.3.3 运算符与表达式

- 运算符

运算符（也称操作符）是表述最基本的运算形式的符号。C 语言提供了 13 类共 34 个运算符，如下所示。

1. 算术运算符

+ (加)、- (减)、* (乘)、/ (除)、% (取模)、++ (自增)、-- (自减)、- (负号)

2. 关系运算符

< (小于)、<= (小于等于)、> (大于)、>= (大于等于)、== (等于)、!= (不等于)

3. 逻辑运算符

! (逻辑非)、&& (逻辑与)、|| (逻辑或)

4. 位运算符

<< (位左移)、>> (位右移)、~ (位取反)、& (位与)、| (位或)、^ (位异或)

5. 赋值运算符: = 及其扩展
6. 条件运算符: ?:
7. 逗号运算符: ,
8. 指针运算符: *(取内容)、&(取址)
9. 求字节数: sizeof
10. 强制类型转换: (类型)
11. 分量运算符: .(结构成员)、->(指针型结构成员)
12. 数组下标运算符: []
13. 其他: ()

这些运算符的优先级如表 4-3 所示。

表 4-3 C 语言的运算符和优先级

运 算 符	优 先 级
0、[]、.、->	1, 最高
!、~、-(负号)、++、--、&(取址)、*(指针)、(类型)、sizeof	2, 右结合
*(乘)、/、%	3
+、-(减)	4
<<、>>	5
<、<=、>、>=	6
==、!=	7
&(位与)	8
^	9
	10
&&	11
	12
?:	13, 右结合
=、复合赋值符号 (如: +=、*=、…)	14, 右结合
,	15, 最低

● 表达式

表达式由运算符（操作符）和运算量（操作数）组成，用以描述对什么数据以什么顺序进行什么操作。运算符对运算量进行运算的结果称为“表达式的值”。C 语言中，任何有值的式子都可以称为表达式。一个表达式中，最后一个运算的运算符的类型决定了表达式的类型。以下是一些表达式的例子。

```
int a;
double x;
a = 5           /* 赋值表达式, 值为: 5      */
x = 1.0         /* 赋值表达式, 值为: 1.0    */
(++a + 1) * (x + 0.5) /* 算术表达式, 值为: 10.5   */
a > 5           /* 关系表达式, 值为: 0      */
a && 2 || 0       /* 逻辑表达式, 值为: 1      */
a+=2, x + 3.0  /* 逗号表达式, 值为: 4.0    */
```

4.3.4 C 语言的语句

C 语言的语句是 C 程序的基本组成部分。C 语句均以分号 “;” 结尾。C 语句可分为指令语句、非指令语句和复合语句，如图 4-19 所示。

- 指令语句

一条指令语句将被编译成若干条计算机的指令，以指示计算机执行相应的操作。

指令语句包括表达式语句和流程控制语句。一个表达式后面加上一个分号即构成了一条表达式语句。流程控制语句用于控制程序的执行流程，在下一小节将专门讨论控制语句。

- 非指令语句

非指令语句不会被编译成计算机的指令，它们是 C 语言语法的必要组成部分，其作用是辅助编译器翻译指令语句，包括数据定义语句和编译预处理。

- 复合语句

复合语句是由一对花括号括起来的一组语句，其作用是将若干条语句组成一个语法上的整体。

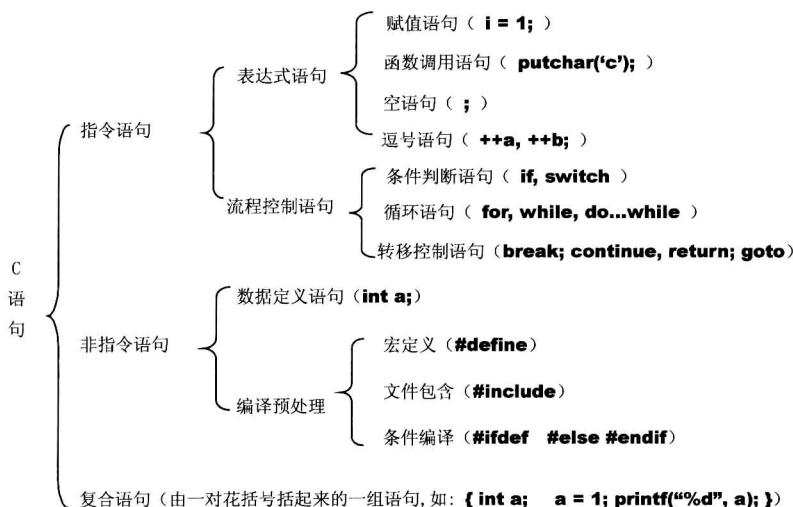


图 4-19 C 语言的语句

4.3.5 控制结构

C 语言有三类控制结构语句：条件判断语句、循环语句和转移控制语句。

- 条件判断语句

1. if 语句

if 语句判断给定的条件的真假来决定执行何种操作，有 3 种形式。

- ① 形式一

`if(条件表达式)`

语句组

如果“条件表达式”的值为“真”，则执行语句组。

- ② 形式二

```
if(条件表达式)
```

语句组 1

```
else
```

语句组 2

如果“条件表达式”的值为“真”，则执行语句组 1，否则执行语句组 2。

③ 形式三

```
if(条件表达式 1)
```

语句组 1

```
else if(条件表达式 2)
```

语句组 2

```
.....
```

```
else if(条件表达式 n)
```

语句组 n

```
else
```

语句组 n+1

如果“条件表达式 1”的值为“真”，则执行语句组 1，如果“条件表达式 2”的值为“真”，则执行语句组 2，否则按从上到下的顺序继续其他条件表达式，如果有任何一个条件表达式 x 的值为真则执行其后的语句组 x，如果所有的条件表达式的值均为假，则执行语句组 n+1。

程序清单 4-5 给出了使用 if 语句的一个示例。

程序清单 4-5 ex_if.c

```
/* 判断用户输入的年份是否是闰年 */
#include <stdio.h>

int main(void)
{
    int year, leap;

    printf("Please input a year: ");
    scanf("%d", &year); /* %d 表示把输入的值转换成整数后保存到后面的变量的地址单元 */
    if(year % 4 == 0)
    {
        if(year % 100 == 0)
        {
            if(year % 400 == 0)
                leap = 1;
            else
                leap = 0;
        }
        else
            leap = 1;
    }
    else
        leap = 0;

    if(leap)
        printf("%d is a leap year.\n", year);
    else
```

```

        printf("%d is not a leap year.\n", year);
    return 0;
}

```

输入如下命令编译运行该程序。

```

jianglinmei@ubuntu:~$ gcc -o ex_if ex_if.c
jianglinmei@ubuntu:~$ ./ex_if
Please input a year: 2000
2000 is a leap year.
jianglinmei@ubuntu:~$ ./ex_if
Please input a year: 2001
2001 is not a leap year.

```

2. switch 语句

switch 语句是多分支选择语句。它的一般形式如下所示。

```

switch(整型表达式)
{
    case 常量表达式 1:
        语句组 1;
        [break;]
    case 常量表达式 2:
        语句组 2;
        [break;]
    .....
    case 常量表达式 n:
        语句组 n;
        [break;]
    default:
        语句组 n+1;
}

```

switch 语句的执行过程是：首先计算 switch 后的整型表达式的值，然后用这个值按从上到下的顺序与各 case 后的常量表达式的值相比较，一旦比较相等就进入此 case 后面的语句组开始从上到下顺序执行，直到遇到 break 语句或执行到 switch 语句的结尾为止。各 case 后的常量表达式的值必须互不相同，方括号内的部分是可选的。

程序清单 4-6 给出了使用 switch 语句的一个示例。

程序清单 4-6 ex_switch.c

```

/* 转换百分制成绩为 5 分制等级 */
#include <stdio.h>

int main(void)
{
    int score;
    char grade;

    printf("Please input score: ");
    scanf("%d", &score);

    switch(score / 10)

```

```

{
    case 10:
    case 9:
        grade = 'A';
        break;
    case 8:
        grade = 'B';
        break;
    case 7:
        grade = 'C';
        break;
    case 6:
        grade = 'D';
        break;
    default:
        grade = 'E';
}
printf("Your grade is: %c\n", grade);
return 0;
}

```

输入如下命令编译运行该程序。

```

jianglinmei@ubuntu:~$ gcc -o ex_switch ex_switch.c
jianglinmei@ubuntu:~$ ./ex_switch
Please input score: 67
Your grade is: D
jianglinmei@ubuntu:~$ ./ex_switch
Please input score: 91
Your grade is: A

```

● 循环语句

1. while 循环语句

while 语句的一般形式是：

```

while(条件表达式)
{
    语句组;
}

```

当条件表达式的值为“真”时，反复执行语句组，当条件表达式的值为“假”时退出循环。

2. do...while 循环语句

do...while 语句的一般形式是：

```

do
{
    语句组;
} while(条件表达式);

```

反复执行语句组，直到条件表达式的值为“假”时退出循环。

3. for 循环语句

for 语句的一般形式是：

```

for(初始化表达式; 条件表达式; 值更改表达式)
{
    语句组;
}

```

首先执行初始化表达式，然后判断条件表达式，当条件表达式的值为“真”时，反复执行语句组和值更改表达式，当条件表达式的值为“假”时退出循环，如图 4-20 所示。

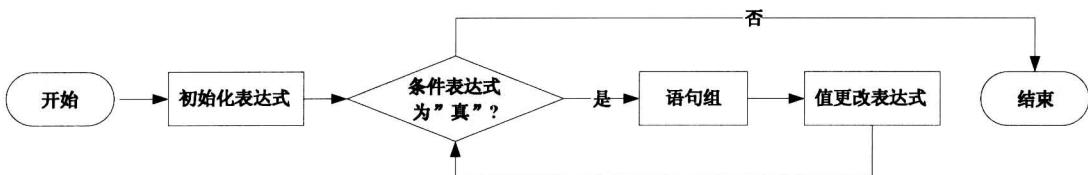


图 4-20 for 语句执行流程

程序清单 4-7 给出了使用 for 语句的一个示例。

程序清单 4-7 ex_for.c

```

/* 计算 0~100 之间所有偶数之和 */
#include <stdio.h>

int main(void)
{
    int i, sum;

    for(i = 0, sum = 0; i <= 100; i += 2)
    {
        sum += i;
    }

    printf("The sum of all even number digit between 0 and 100 is: %d\n", sum);
    return 0;
}

```

```

jianglinmei@ubuntu:~$ gcc -o ex_for ex_for.c
jianglinmei@ubuntu:~$ ./ex_for
The sum of all even number digit between 0 and 100 is: 2550

```

● 转移控制语句

1. break 语句

C 语言中，break 语句可出现在两个地方：一是 switch 语句的 case 分支中，用以退出 switch；另一处是循环语句中，用以退出循环。

2. continue 语句

continue 语句只能用在循环语句中，作用是中止一次循环，即跳过循环语句组中位于 continue 之后的语句，进入下一次循环条件表达式的判断（对于 for 循环还要先执行更改表达式）。

3. return 语句

用于函数体内，作用是退出函数，同时可以给调用者返回一个值。

4. goto 语句

goto 语句用以将程序执行流程无条件转移到一个标号处。标号是一个后面跟一个冒号 “:” 的

标识符。

程序清单 4-8 给出了使用 for 语句的一个示例。

程序清单 4-8 ex_transfer.c

```
/* 计算 n~100 之间所有偶数之和, n 为用户输入的一个整数 */
#include <stdio.h>

int main(void)
{
    int i, sum, digit;

    printf("Please input a positive digit:");
    scanf("%d", &digit);
    if(digit > 100)
        goto END;      /* 跳到标号 END 后面的行 */

    for(i = digit, sum = 0; ; i++)          /* 条件表达式省略, 永远为真 */
    {
        if(i > 100)
            break;                      /* 退出循环 */
        if(i % 2 == 1)
            continue;                  /* 终止本次循环, 流程转入 “更改表达式——i++” */

        sum += i;
    }

    printf("The sum of all even number digit between 0 and 100 is: %d\n", sum);
    return 0;
END:      /* 定义标号 */
    printf("The digit you inputted is too large\n");
    return 0;
}
```

输入如下命令编译运行该程序。

```
jianglinmei@ubuntu:~$ ./ex_transfer
Please input a positive digit:0
The sum of all even number digit between 0 and 100 is: 2550
jianglinmei@ubuntu:~$ ./ex_transfer
Please input a positive digit:120
The digit you inputted is too large
```

4.3.6 函数

一个 C 源程序由一个或多个文件构成, 每个文件就是一个编译单位。一个 C 语言源文件由一个或多个函数构成。每个程序有且只有一个主函数 (main), 其他都是子函数。主函数可以调用子函数, 子函数可以相互调用, 但子函数不能调用主函数。

1. 函数的定义

函数定义的一般格式是:

```
数据类型 函数名([形式参数说明])
{
```

```
    函数体
```

```
}
```

形式参数说明方法：

数据类型 变量名 [, 数据类型 变量名 ...]

例如：

```
int sum(int x, int y)
{
    int z;
    z = x + y;
    return z;
}
```

2. 函数原型

在程序中调用函数需满足以下条件：

- (1) 被调用函数必须存在，且必须遵循“先定义后使用”的原则。
- (2) 如果被调用函数定义在主调函数之后，可以在调用之前给出原型声明。

C 语言函数原型声明的一般格式是：

数据类型 函数名(形式参数类型 [变量名], [形式参数类型[变量名], ...]);

可见，函数原型声明的格式即函数定义格式除去花括号与函数体的部分，然后另加一个分号结束，并且形式参数的变量名是可选的。例如，上面定义的 sum() 函数的原型为：

```
int sum(int x, int y);
```

或

```
int sum(int, int);
```

3. 函数的调用

函数调用一般形式是：

函数名([实参列表])

应注意：

- (1) 如果被调用没有形式参数，则不能有实参列表，但必须有圆括号。
- (2) 实参数个数和形参数个数必须相同。
- (3) 实参和形参的数据类型一一对应，必须相同或实参的类型能安全转换为形参的类型。
- (4) 调用函数时，按位置顺序将实参的值一一赋值给对应的形参，这是按值传递参数的过程。
- (5) 被调用函数有返回值时，主调函数中可以获取该返回值。

程序清单 4-9 给出了使用函数的一个示例。

程序清单 4-9 ex_function.c

```
#include <stdio.h>

/* 声明函数原型 */
int sum(int x, int y);
```

```

int main(void)
{
    int result;

    result = sum(3, 5);           // 调用自定义函数

    printf("the sum of 3 and 5 is: %d\n", result);
    return 0;
}

/* 定义函数，计算两数之和 */
int sum(int x, int y)
{
    int z;
    z = x + y;
    return z;
}

```

输入如下命令编译运行该程序：

```

jianglinmei@ubuntu:~$ ./ex_function
the sum of 3 and 5 is: 8

```

4.3.7 内存管理

C语言的标准库中提供了3个函数用以分配和回收内存。

1. 动态分配内存函数 malloc()

函数原型：void * malloc(unsigned size);

功能：在堆中分配一块size字节的内存。调用结果为新分配的内存的首地址，是一个void类型指针。若分配失败，则返回NULL。

2. 动态分配内存函数 calloc()

函数原型：void *calloc(unsigned int n ,unsigned int size);

功能：在堆中分配一块n * size字节的内存。调用结果为新分配的内存的首地址，是一个void类型指针。若分配失败，则返回NULL。

3. 释放动态分配的内存函数 free()

函数原型：void free(void *p);

功能：释放p所指向的动态分配的内存。注意，实参必须是一个指向动态分配的内存的指针，它可以是任何类型的指针变量。

程序清单4-10给出了使用函数的一个示例。

程序清单4-10 ex_malloc.c

```

#include <stdio.h>

/* 声明函数原型 */
int sum(int x, int y);

int main(void)
{
    int result;

```

```

result = sum(3, 5);           // 调用自定义函数

printf("the sum of 3 and 5 is: %d\n", result);
return 0;
}

/* 定义函数，计算两数之和 */
int sum(int x, int y)
{
    int z;
    z = x + y;
    return z;
}

```

输入如下命令编译运行该程序。

```

jianglinmei@ubuntu:~$ ./ex_function
the sum of 3 and 5 is: 8

```

4.3.8 编译预处理

编译预处理即在编译之前对源文件进行的处理，包括头文件加载和宏展开等操作。编译预处理主要包括：宏定义、文件包含和条件编译。

- 宏定义

1. 不带参数的宏定义

不带参数的宏定义的功能是：用一个标识符（宏名）代替一段“字符序列”（宏体）。其一般形式如下。

```
#define 宏名 [宏体]
```

例如：

```
#define PI 3.1415926
```

这定义了一个名为 PI 的宏，以后在源码中如果出现 PI，则它表示 3.1415926。

使用宏定义应注意以下几点。

- (1) 宏名一般用大写字母表示，以区别于一般的变量名。
- (2) 宏定义位置任意，但一般在函数外面，其作用域为从定义位置直到文件结束。
- (3) 使用#define可终止宏名作用域，格式为：

```
#undef 宏名
```

例如：

```
#undef PI
```

(4) 预编译时，对宏进行展开操作，即用宏体来替换宏名，替换过程不作语法检查。但程序中使用双引号括起的内容即使与宏名相同也不会替换。

(5) 宏定义中应注意使用圆括号以保证宏展开后的式子依然正确。例如：

```

#define WIDTH      80
#define LENGTH     WIDTH + 40

```

```
var = LENGTH * 2;
```

宏展开过程：

```
var = WIDTH + 40 * 2
var = 80 + 40 * 2
```

这和正常的理解不一致，正确定义方法应当是：

```
#define WIDTH      80
#define LENGTH     (WIDTH + 40)
var = LENGTH * 2;
```

宏展开过程：

```
var = (WIDTH + 40) * 2
var = (80 + 40) * 2
```

2. 带参数的宏定义

带参数的宏定义的一般形式如下。

```
#define 宏名(形参表) 宏体
```

例如：

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

当参的宏在展开时，除了进行简单的字符序列替换外，还要进行形参替换。应注意以下几点。

(1) 宏体及各形参应用圆括号括起来，以免歧义。例如：

```
#define MUL(x, y) x * y
var = MUL(3 + 5, 2 + 1);
```

展开后为：

```
var = 3 + 5 * 2 + 1
```

(2) 宏名与参数之间不能有空格。例如：

```
#define S (r) PI*r*r
```

相当于定义了不带参的宏 S，代表字符序列“(r) PI*r*r”。

(3) 引用带参的宏时要求实参数个数与形参数个数相同。其形式与函数调用相同，但处理方式不同。函数传参是在运行阶段，宏参数替换是在预处理阶段。

(4) 符号“#”、“##”和“\”。

- # 把宏参数变为一个字符串，即在宏参数两边加上“#”。
- ## 把两个宏参数粘合在一起，即顺序连接两参数的值。
- \ 续行符，允许分多行书写宏体，注意在“\”后不能有任何字符。

程序清单 4-11 给出了使用这些特殊符号的宏的一个示例。

程序清单 4-11 ex_presymbol.c

```
#include <stdio.h>
#define STR(s) #s
```

```

#define VERSION(major, minor)      major##.##minor
#define DEFINE_FUNCTION(id, name) \
    static void name##id() \
    { \
        printf("function is: %s\n", STR(name##id)); \
    } \
}

DEFINE_FUNCTION(10, output);

int main()
{
    printf(STR(it is a string));           // 输出: it is a string
    printf("\n%.1f\n", VERSION(5, 3));     // 输出: 5.3

    output10();

    return 0;
}

```

本示例程序的宏展开后的代码如下所示。

```

static void output10() { printf("function is: %s\n", "output10"); };

int main()
{
    printf("it is a string");
    printf("\n%.1f\n", 5.3);

    output10();

    return 0;
}

```

输入如下命令编译运行该程序。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_presymbol ex_presymbol.c
jianglinmei@ubuntu:~/c$ ./ex_presymbol
it is a string
5.3
function is: output10

```

● 文件包含

使用文件包含指令，一个源文件将另一个源文件的内容全部包含进来，其一般形式如下。

```
#include "文件名"
```

或

```
#include <文件名>
```

这两种形式的区别在于搜索所包含文件的路径不同。“””” 形式先在当前目录搜索，如搜索不到则再搜索标准头文件路径或 gcc 的 “-I” 选项指定的路径搜索。“<>” 形式则略去了在当前目录搜索的过程。

#include一般用于包含“.h”头文件，如用于包含“.c”文件很容易引起变量重复定义的错误，应当谨慎。在预编译时，#include指令行会被所包含的文件的内容取代。例如：

```
/* a.h 文件 */
extern int max(int, int);

/* a.c 文件 */
#include "a.h"      /* 在 a.c 文件中包含 a.h 文件 */
int max(int a, int b)
{
    return a > b ? a : b;
}
```

经预处理后，a.c文件的内容变为：

```
extern int max(int, int);
int max(int a, int b)
{
    return a > b ? a : b;
}
```

● 条件编译

条件编译即在某个条件成立的情况下才进行编译。条件编译有三种形式。

1. 形式一

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

如果“标识符”代表的宏已经定义过，则对程序段1进行编译，否则（如果有#else部分）对程序段2进行编译。

2. 形式二

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

如果“标识符”代表的宏没有定义过，则对程序段1进行编译，否则（如果有#else部分）对程序段2进行编译。

3. 形式三

```
#if 常量表达式 1
    程序段 1
[ #elif 常量表达式 2
    程序段 2
#else
    程序段 n ]
```

```
#endif
```

如果指定的“常量表达式 1”值为真（非零），则对“程序段 1”进行编译，否则如果指定的“常量表达式 2”值为真（非零），则对“程序段 2”进行编译，否则对“程序段 n”进行编译。其中 #elif 指令可以有任意多个。

程序清单 4-12 给出了条件编译的一个示例。

程序清单 4-12 ex_preif.c

```
#include <stdio.h>

int main()
{
    #if CIRCLE
        printf("draw circle\n");
    #elif RECTANGLE
        printf("draw rectangle\n");
    #else
        printf("draw line\n");
    #endif

    return 0;
}
```

输入如下命令编译该程序。

```
jianglinmei@ubuntu:~/c$ gcc -E -o ex_preif.e ex_preif.c
```

然后查看 ex_preif.e 的内容可见，main()函数的内容如下。

```
int main()
{
    printf("draw line\n");
    return 0;
}
```

再次，输入如下命令，指定预定义宏值编译该程序。

```
jianglinmei@ubuntu:~/c$ gcc -E -D RECTANGLE=1 -o ex_preif.e ex_preif.c
```

然后查看 ex_preif.e 的内容可见，main()函数的内容如下。

```
int main()
{
    printf("draw rectangle\n");
    return 0;
}
```

从以上预处理结果可以看出，对条件编译指令预处理后的源代码中仅含有满足条件的部分。

4.4 小结

本章首先介绍了 Linux 环境下两类编译环境的配置和使用。命令行形式的 gcc 编译器轻巧、

高效，有图形界面的 Eclipse CDT 集成开发环境则更贴近 Linux 初学者尤其是从 Windows 平台转入 Linux 平台的用户的使用习惯。本章重点从 C 语言的数据类型、运算符与表达式、表达式语句、控制语句、函数、内存管理和编译预处理等方面介绍了 C 语言的基础知识。这是学习后续章节章节的基础。

4.5 习题

- (1) 请简述编写一个 C 语言程序的基本流程。
- (2) 请举例说明 5 个以上 gcc 常用编译选项的作用。
- (3) 请简述在 Eclipse CDT 中创建 C 项目并进行构造和运行的过程。
- (4) 在 Eclipse CDT 中如何设置断点、单步运行及查看变量值？有哪些快捷键？
- (5) C 语言有哪些特点？
- (6) 什么是常量？什么是变量？
- (7) 如何表达长整型常量？如何表示单精度浮点常量？
- (8) 整型可细分为哪些类型？它们在 32 位 Linux 下占用的字节数和表示范围分别是什么？
- (9) 字符型和整型有何共性？
- (10) 如何确定枚举常量的值？
- (11) 二维数组部分初始化的规则是怎样的？
- (12) 字符数组和一般一维数组的区别在哪里？
- (13) 什么是指针？引用指针所指的变量时应注意什么？指针赋值有哪些情况是合法的？
- (14) 什么是结构体，如何引用结构体的数据成员？
- (15) 共用体和结构体的区别在哪？
- (16) 自定义类型的语法形式是怎样的？使用自定义类型有何好处？
- (17) C 语言有哪些运算符，其优先级是怎样的？
- (18) C 语言有哪些控制语句？请简要描述 for 语句的执行流程。
- (19) 如何定义函数？如何定义函数原型？函数调用的流程是怎样的？
- (20) C 语言如何动态分配内存，如何释放？
- (21) C 语言有哪些预处理指令，其作用分别是什么？
- (22) 使用宏定义有何应注意的方面？带参的宏与函数在形式上相似，它们有哪些区别？

第 5 章

文 件

在 Linux 中，几乎任何事物都可以用一个文件来表示。Linux 中的文件类型多样，既包含普通的磁盘文件，也包含特殊的硬件设备文件、管道（PIPE）文件、套接字（SOCKET）文件和目录文件等。在 C 语言编程环境中，与文件有关的操作主要是 I/O（输入/输出）操作。Linux 环境下的 I/O 操作可以分为两类，它们是基于文件描述符的底层系统调用 I/O 和基于流的 C 标准库函数调用 I/O。这两类 I/O 操作各有优缺点，需视情况而使用。基于流的 I/O 将在下一章介绍。本章主要介绍基于文件描述符的 I/O，内容包括：

- Linux 文件 I/O 概述
- 底层文件访问
- 链接文件的操作
- 目录文件的操作
- 设备文件

5.1 Linux 文件 I/O 概述

5.1.1 简介

Linux 环境中的文件具有特别重要的意义。在 Linux 中，几乎任何事物都可以用一个文件来表示，或者通过特殊的文件提供支持。通过将设备表示成设备文件，程序可以像使用磁盘文件一样使用串行口、打印机和其他设备。Linux 中，文件系统被组织成一树的形状，树枝是目录，树叶是文件。其中的目录也是一类特殊的文件。另外，Linux 中用于进程间通信的管道和用于网络通信的 SOCKET，也都以文件接口的方式提供服务。因此，文件操作编程是其他应用编程的基础之一。

文件为操作系统服务和设备提供了一个简单而一致的接口，大多数 Linux 文件 I/O 只需用到 5 个函数：open、read、write、lseek 和 close。除此之外，使用 stat、access 等其他 I/O 函数可以获取或设置文件的状态和权限等信息。对于目录文件的操作，Linux 则提供了一些简单而特殊的编程接口。

5.1.2 文件和目录

文件，除了本身包含的内容外，还会有一个名字和一些属性，即“管理信息”。文件的属性被

保存在文件的索引节点 (inode) 中。索引节点是文件系统中的一个特殊数据块，用以保存文件自身的属性，包含如下信息。

1. 文件使用的设备号
2. 索引节点号
3. 文件访问权限和文件类型
4. 文件的硬连接数
5. 所有者用户 ID
6. 组 ID
7. 设备文件的设备号
8. 文件大小 (单位为字节)
9. 包含该文件的磁盘块的大小
10. 该文件所占的磁盘块
11. 文件的最后访问时间
12. 文件的最后修改时间
13. 文件的状态最后改变时间

正如本书第1章所述，Linux文件系统将文件索引节点号和文件名同时保存在目录中。目录是用于保存其他文件的节点号和名字的文件，是将文件的名称和它的索引节点号结合在一起的一张表，目录中每一对文件名称和索引节点号称为一个“连接”。目录文件中的每个数据项都是指向某个文件节点的连接。删除一个文件时，实质上是删除目录中与该文件对应的数据项，同时将文件的连接数减1。

通常文件中包含一定的数据，磁盘中的普通文件和目录文件都有相应的磁盘区域存储数据。这些数据是存储在由索引节点指定的位置上的。而其他一些特殊文件，如设备文件等，则不具有这样的磁盘存储区域。

5.1.3 文件和设备

硬件设备在UNIX/Linux中通常也被表示(映射)为文件。这些设备文件被放在Linux的/dev目录下。硬件设备可分为字符设备和块设备，两者的区别在于访问设备时是否需要一次读写一整块。比如，键盘是一种字符设备，一次仅能读写一个字节。而典型地，硬盘是一种块设备，每次至少读写一个扇区(一整块数据)。

Linux环境下一类比较重要的设备是终端设备。终端是一种字符设备，它有多种类型，通常使用tty来简称各种类型的终端设备。tty是Teletype的缩写。

这里介绍几个Linux中比较重要的设备文件，分别如下。

1. 控制终端 (/dev/tty)

代表进程的控制终端(键盘和显示屏或键盘和窗口)。如果当前进程有控制终端(Controlling Terminal)的话，那么/dev/tty就是当前进程的控制终端的设备特殊文件。可以使用命令“ps -ax”来查看进程与哪个控制终端相连。

对于交互命令行方式登录的Shell，/dev/tty就是登录时所使用的终端，使用命令“tty”可以查看它具体对应哪个实际终端设备的设备文件。

没有控制终端的进程不能打开/dev/tty。

2. 控制台终端 (/dev/ttyN, /dev/console)

代表系统控制台。错误信息和诊断信息通常会被发送到这个设备（打印终端、虚拟控制台、控制台窗口）。

在 UNIX/Linux 系统中，计算机显示器通常被称为控制台终端（Console）。它仿真了类型为 Linux 的一种终端（TERM=Linux），并且有一些设备特殊文件与之相关联：tty0、tty1、tty2 等。

当用户在控制台上登录时，使用的是 tty1。tty1 ~ tty6 等称为虚拟终端，而 tty0 则是当前所使用虚拟终端的一个别名，系统所产生的信息会发送到该终端上。/dev/console 就是 tty0。因此不管目前正在使用哪个虚拟终端，系统信息都会发送到控制台终端上。

/dev/pts 是用户通过 telnet 或 ssh 远程登录系统后，Linux 所创建的控制台设备文件所在的目录。由于不同时刻登录系统的用户可能不同，登录用户数量也可能不同，所以不像其他设备文件是构建系统时就已经产生的硬盘节点，/dev/pts 下的设备文件其实是动态生成的。第一个用户登录，console 的设备文件为 /dev/pts/0，第二个为 /dev/pts/1，以此类推。这里的 0、1、2、3 不是具体的标准输入或输出，而是整个控制台。

3. /dev/null

代表空（null）设备。所有写向这个设备的输出都将被丢弃，而读该设备会立即返回文件尾标志。空设备常用于输出重定向，以忽略某些错误输出。

5.1.4 系统调用和标准函数库

系统调用是 UNIX/Linux 内核直接提供的编程接口，在内核空间运行。C 语言标准函数库是由一些函数构成的集合，完全运行在用户空间，其中可能使用系统调用来完成诸如访问硬件设备的底层功能。

直接使用底层系统调用进行 I/O 操作的效率非常低。原因如下。

- (1) 执行系统调用时，Linux 必须从用户空间切换到内核空间，然后再切换回来。
- (2) 硬件会限制系统调用一次需读写的数据块大小，如块设备。

为此，应让每次系统调用完成尽可能多的工作。这正是 C 语言标准函数库所做的。C 语言标准函数库带缓冲机制，允许在缓冲区满或必须的情况下才使用底层系统调用，这样就减少了系统调用的次数，提高了效率。另外，有的库函数完全不使用系统调用，只在用户空间完成特定的功能。

有关系统调用和标准函数库的文档一般分别放在手册页的第 2 节和第 3 节。

5.2 底层文件访问

底层文件访问即使用底层系统调用进行的 I/O 操作。底层文件访问的大多数操作都是通过一个与文件相关联的“文件描述符”来进行的。

5.2.1 文件描述符

文件描述符是一个非负整数。对于内核而言，所有打开的文件都由文件描述符引用。

当打开一个现存文件或创建一个新文件时，内核向进程返回一个文件描述符。当读、写一个文件时，用 open 或 creat 返回的文件描述符标识该文件，将其作为参数传送给 read 或 write。从内

核源码的角度看，文件描述符其实是当前进程所打开的文件结构数组的下标。

按照惯例，UNIX/Linux Shell 使文件描述符 0 与进程的标准输入相结合，文件描述符 1 与标准输出相结合，文件描述符 2 与标准出错输出相结合。在头文件<unistd.h>中定义了常量 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO，其值分别为 0、1、2。

一个进程能打开的文件数由<limits.h>文件中的 OPEN_MAX 限定，这个限定值每个系统可能都不太一样，LINUX 一般默认为 1024。

5.2.2 文件的创建、打开和关闭

在 Linux 底层，可使用 open 函数创建或打开文件，用 close 函数关闭已打开的文件。

5.2.2.1 open 函数

调用 open 函数可以打开或创建一个文件，其原型如下。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

打开一个文件后，即在文件描述符和文件之间建立了一个关联。open 函数既可打开一个已存在的文件，也可以创建一个新的文件。具体执行打开操作还是创建操作由 flag 参数指定。open 函数各参数和返回值的含义如下。

- | | |
|-------------|---|
| 1. pathname | 要打开或创建的文件的名字 |
| 2. flags | 由下列一个或多个常数进行或运算构成 |
| • O_RDONLY | 只读打开 |
| • O_WRONLY | 只写打开 |
| • O_RDWR | 读、写打开 |
| • O_APPEND | 每次写时追加到文件的尾端 |
| • O_CREAT | 若此文件不存在则创建它，应结合第三个参数 mode 使用 |
| • O_EXCL | 结合 O_CREATE，当文件不存在时，才创建文件 |
| • O_TRUNC | 如果此文件存在，而且为只读或只写则将其长度截短为 0 |
| 3. mode | 存取许可权位，一个 32 位无符号整数，仅当创建新文件时才使用，由下列一个或多个常数进行或运算构成。应注意，最终文件权限受系统变量 umask 限制，是所设权限和 umask 的二进制“非”进行二进制“与”所得的结果（即，mode & ~umask） |

- S_IRUSR 文件所有者-读
- S_IWUSR 文件所有者-写
- S_IXUSR 文件所有者-执行
- S_IRGRP 组用户-读
- S_IWGRP 组用户-写
- S_IXGRP 组用户-执行
- S_IROTH 其他用户-读
- S_IWOTH 其他用户-写
- S_IXOTH 其他用户-执行

4. 返回值

- 成功时返回一个文件描述符（非负整数）
- 失败时返回-1，并设置全局变量 `errno` 指明失败原因

如果使用 `open` 创建新文件，新文件的所有者是当前进程的有效用户，文件的所属组是当前进程的有效组或父目录的所属组（与文件系统的类型和挂载方式有关）。

由 `open` 返回的文件描述符一定是最小的未用描述符数字。因此，如果先关闭标准输入对应的文件描述符，再打开任一文件，该文件对应的文件描述符将为 0。用此策略，可将自打开的文件作为标准输入使用。

5.2.2.2 `close` 函数

使用 `open` 函数打开的文件在操作结束后，应当使用 `close` 函数关闭。`close` 函数的原型如下。

```
#include<unistd.h>
int close(int filedes);
```

调用 `close` 函数后，终止了文件描述符与文件之间的关联，被关闭的文件描述符重新变为可用。关闭一个文件的同时也释放该进程加在该文件上的所有记录锁。当一个进程终止时，它所打开的所有文件都将由内核自动关闭。

`close` 函数的参数和返回值的含义如下：

1. `filedes` 待关闭的文件描述符。

2. 返回值

成功时返回 0，失败时返回-1。

5.2.3 文件的读、写

在 Linux 底层，可使用 `read` 函数读取已打开文件中的数据，用 `write` 函数写新的数据到已打开的文件中。

5.2.3.1 `write` 函数

调用 `write` 函数可向已打开的文件中写入数据，其原型如下。

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

`write` 函数各参数和返回值的含义如下。

1. `fd` 文件描述符

2. `buf` 待写入文件的数据的缓冲区，是一个常量指针

3. `count` 待写的字节数，该字节数应当小于或等于缓冲区的大小

4. 返回值

- 若成功为已写的字节数；若出错为-1，错误值记录在 `errno`

- 返回值类型 `ssize_t` 为一个“有符号字”类型“`_SWORD_TYPE`”，该类型定义在 `/usr/include/i386-linux-gnu/bits/types.h` 文件中。在 32 位机中是 `int`，在 64 位机中是 `long int`

程序清单 5-1 给出了一个使用 `write` 函数的简单示例。

程序清单 5-1 `ex_write.c`

```
1 #include <unistd.h>
2 #include <stdlib.h>
```

```

3
4 int main()
5 {
6     if ((write(1, "Here is some data\n", 18)) != 18)
7         write(2, "A write error has occurred on file descriptor 1\n", 46);
8
9     return 0;
10 }

```

程序第1、2行包含了必要的头文件。第6行调用write函数向文件描述符1（也就是标准输出——屏幕）写入18字节的数据“Here is some data\n”，如果返回值即实际写入的字节数不等于18则表示写入出错。如果写入出错，则执行第7行，向文件描述符2（也就是标准错误输出）写入46字节的数据“A write error has occurred on file descriptor 1\n”。

在命令行编译运行ex_write.c，如下所示。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_write ex_write.c
jianglinmei@ubuntu:~/c$ ./ex_write
Here is some data

```

5.2.3.2 read函数

调用read函数可从已打开的文件中读取数据，其原型如下。

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

read函数各参数和返回值的含义如下。

1. fd 文件描述符
2. buf 用于放置读取到的数据的缓冲区
3. count 要读取的字节数
4. 返回值

- 若成功为已读取的字节数（0表示已到达文件尾）；若出错为-1，错误值记录在errno
- 程序清单5-2给出了一个使用read函数的简单示例。

程序清单5-2 ex_read.c

```

1 #include <unistd.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char buffer[128];
7     int nread;
8
9     nread = read(0, buffer, 128);
10    if (nread == -1)
11        write(2, "A read error has occurred\n", 26);
12    else if ((write(1,buffer,nread)) != nread)
13        write(2, "A write error has occurred\n", 27);
14
15    return 0;
16 }

```

程序第6行定义了一个用于存储读取到的数据的缓冲区buffer。第9行调用read函数从文件

描述符 1（也就是标准输入——键盘）读取最多 128 字节数据，`read` 的返回值（实际读取到的字节数）记录在 `nread` 变量中。第 10 行判断 `read` 的返回值是否为 -1，如为 -1 表示读取时发生了错误，在第 11 行向标准错误输出设备输出提示信息。否则，在第 12 行调用 `write` 函数，将读取到 `buffer` 中的数据重新输出到标准输出设备。如果输出的字节数与读取到的字节数不一致，则在第 13 行向标准错误输出设备输出错误提示。

在命令行编译运行 `ex_read.c`，如下所示。

```
jianglinmei@ubuntu:~/c$ gcc -o ex_read ex_read.c
jianglinmei@ubuntu:~/c$ ./ex_read
This is an "read" example.
This is an "read" example.
```

5.2.4 文件的定位

对于可随机访问的文件，如磁盘文件，人们往往希望能够按需定位到文件的某个位置进行读、写操作。这可以通过调用 `lseek` 函数来完成。

实际上，每个已打开的文件都有一个与其相关联的“当前文件位移量”。通常，读、写操作都从当前文件位移量处开始，并在读、写完成后使位移量增加所读或写的字节数。

当打开一个文件时，如果指定 `O_APPEND` 选项，该位移量被设置为文件的长度，否则该位移量被设置为 0。如果要随机访问的文件内容，可调用 `lseek` 函数显式地定位一个已打开文件的“当前文件位移量”。

`lseek` 函数的原型如下。

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

`lseek` 仅将当前的文件位移量记录在内核变量内，并不引起任何 I/O 操作。`lseek` 函数各参数和返回值的含义如下。

1. `fd` 文件描述符
 2. `offset` 位移量。`off_t` 类型一般为“`long int`”的 `typedef`
 3. `whence` 指定位移量相对于何处开始，可取以下三个常量值
 - `SEEK_SET(=0)` 文件开始位置
 - `SEEK_CUR(=1)` 文件读写指针当前位置
 - `SEEK_END(=2)` 文件结束位置
 4. 返回值
 - 若成功为当前读写位置相对于文件头的位移量；若出错为 -1，错误值记录在 `errno`
- 程序清单 5-3 给出了一个综合使用 `open`、`close`、`write` 和 `lseek` 函数的示例。

程序清单 5-3 `ex_lseek.c`

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5
6 char buf1[] = "abcdefghijkl";
7 char buf2[] = "ABCDEFGHIJ";
```

```

8
9 int main(void)
10 {
11     int fd;
12
13     if((fd = open("file.hole", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)) < 0)
14     {
15         write(2, "create error\n", 13);
16         return -1;
17     }
18     if(write(fd, buf1, 10) != 10)
19     {
20         write(2, "buf1 write error\n", 17);
21         return -1;
22     }
23
24     /* offset now = 10 */
25     if(lseek(fd, 40, SEEK_SET) == -1)
26     {
27         write(2, "lseek error\n", 12);
28         return -1;
29     }
30
31     /* offset now = 40 */
32     if(write(fd, buf2, 10) != 10)
33     {
34         write(2, "buf2 write error\n", 17);
35         return -1;
36     }
37     /* offset now = 50 */
38
39     return 0;
40 }

```

程序第 6、7 行定义了两个全局数组，其中保存了两个待写入文件的字符串。第 11 行定义了一个文件描述符变量 fd。第 13 行调用 open 函数以只写 (O_WRONLY) 和创建 (O_CREAT) 方式在当前目录创建一个所有者具有读 (S_IRUSR) 和写 (S_IWUSR) 权限的普通文件 file.hole，打开的文件描述符记录于 fd 变量。如果 open 的返回值小于 0，说明打开文件失败，输出错误提示并退出程序。

程序第 18 行，向文件写入 “abcdefghijkl”，如写入成功则写入后“当前文件位移量”将为 10。第 25 行调用 lseek 函数将“当前文件位移量”设置为 40。第 32 行，从位移量 40 处开始写入“ABCDEFGHIJ”，如写入成功则写入后“当前文件位移量”将为 50。

在命令行编译运行 ex_lseek.c，如下所示。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_lseek ex_lseek.c
jianglinmei@ubuntu:~/c$ ./ex_lseek
jianglinmei@ubuntu:~/c$ od -c file.hole
00000000 a b c d e f g h i j \0 \0 \0 \0 \0 \0 \0 \0
00000020 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
00000040 \0 \0 \0 \0 \0 \0 \0 \0 A B C D E F G H
00000060 I J

```

其中 od -c 命令表示以字符形式显示二进制文件的内容。

5.2.5 文件属性的读取

5.2.5.1 stat 系列函数

在操作 Linux 文件时，考察文件的所有者、文件大小、文件的连接数、文件类型、文件的权限等信息，是一种常见的需求。为此，Linux 提供 stat 系列函数以读取文件的属性信息。这些函数的原型如下。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

这些函数各参数和返回值的含义如下。

1. file_name 文件名
2. filedes 文件描述符。使用 fstat 需先打开文件
3. buf 文件信息结构缓冲区，该缓冲区为一个结构体，定义如下。

```
struct stat {
    dev_t      st_dev;        /* 保存本文件的设备的 ID */
    ino_t      st_ino;        /* 与文件关联的索引节点号 (inode) */
    mode_t     st_mode;       /* 文件权限和文件类型信息 */
    nlink_t    st_nlink;      /* 该文件上硬链接的个数 */
    uid_t      st_uid;        /* 文件所有者的 UID 号 */
    gid_t      st_gid;        /* 文件所有者的 GID 号 */
    dev_t      st_rdev;       /* 特殊文件的设备 ID */
    off_t      st_size;       /* 文件大小 (字节数) */
    blksize_t  st_blksize;    /* 文件系统 I/O 的块大小 */
    blkcnt_t   st_blocks;     /* 块数 (512 字节的块) */
    time_t     st_atime;      /* 最后访问时间 */
    time_t     st_mtime;      /* 最后修改时间 */
    time_t     st_ctime;      /* 最后状态改变时间 */
};
```

4. 返回值

- 成功为 0；若出错为 -1，错误值记录在 errno

这三个函数中，stat 和 lstat 的参数完全相同，两者区别在于：当文件是一个符号链接时，lstat 返回的是该符号链接本身的信息，而 stat 返回的是该链接指向的文件的信息。

调用 stat 系列函数后，文件的属性信息均保存于 struct stat 结构体类型的 buf 中。如果要进一步获知文件的权限和类型详情，应当对 struct stat 结构体的 st_mode 成员进行分析。st_mode 成员的每一个位代表了一种权限或文件类型，可以将该成员与表 5-1 所示的标志位进行二进制“与”运算以测试权限或类型。

表 5-1

struct stat 的 st_mode 成员相关的标志位

标志位	常量值	含义
S_IFMT	0170000	文件类型掩码
S_IFSOCK	0140000	套接字 (socket)
S_IFLNK	0120000	符号链接
S_IFREG	0100000	普通文件
S_IFBLK	0060000	块设备
S_IFDIR	0040000	目录
S_IFCHR	0020000	字符设备
S_IFIFO	0010000	FIFO (命名管道)
S_ISUID	0004000	设置了 SUID
S_ISGID	0002000	设置了 SGID
S_ISVTX	0001000	设置了黏滞位
S_IRWXU	00700	文件所有者权限掩码
S_IRUSR	00400	文件所有者可读
S_IWUSR	00200	文件所有者可写
S_IXUSR	00100	文件所有者可执行
S_IRWXG	00070	文件所属组权限掩码
S_IRGRP	00040	文件所属组可读
S_IWGRP	00020	文件所属组可写
S_IXGRP	00010	文件所属组可执行
S_IRWXO	00007	其他用户权限掩码
S_IROTH	00004	其他用户可读
S_IWOTH	00002	其他用户可写
S_IXOTH	00001	其他用户可执行

除直接使用二进制与的方式进行文件类型测试外，Linux 还提供了以下宏用于文件类型的判断（其中的 m 参数即 st_mode 成员）。

- S_ISREG(m) 是否为普通文件
- S_ISDIR(m) 是否为目录
- S_ISCHR(m) 是否为字符设备
- S_ISBLK(m) 是否为块设备
- S_ISFIFO(m) 是否为 FIFO (命名管道)
- S_ISLNK(m) 是否为符号链接
- S_ISSOCK(m) 是否为套接字 (socket)

程序清单 5-4 给出了一个使用 stat 函数的示例，该示例程序的功能是，显示用户以命令行参数的形式提供的文件的属性信息。

程序清单 5-4 ex_stat.c

```
1 #include <sys/types.h>
```

```

2 #include <sys/stat.h>
3 #include <time.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main(int argc, char *argv[])
8 {
9     struct stat sb;
10
11    if (argc != 2) {
12        printf("Usage: %s <pathname>\n", argv[0]);
13        exit(EXIT_FAILURE);
14    }
15
16    if (stat(argv[1], &sb) == -1) {
17        perror("stat");
18        exit(EXIT_FAILURE);
19    }
20
21    printf("File type:           ");
22
23    switch (sb.st_mode & S_IFMT) {
24        case S_IFBLK: printf("block device\n");      break;
25        case S_IFCHR: printf("character device\n");   break;
26        case S_IFDIR: printf("directory\n");          break;
27        case S_IFIFO: printf("FIFO/pipe\n");         break;
28        case S_IFLNK: printf("symlink\n");            break;
29        case S_IFREG: printf("regular file\n");       break;
30        case S_IFSOCK: printf("socket\n");             break;
31        default:    printf("unknown?\n");            break;
32    }
33
34    printf("I-node number:      %ld\n", (long) sb.st_ino);
35
36    printf("Mode:              %lo (octal)\n",
37          (unsigned long) sb.st_mode);
38
39    printf("Link count:        %ld\n", (long) sb.st_nlink);
40    printf("Ownership:         UID=%ld  GID=%ld\n",
41          (long) sb.st_uid, (long) sb.st_gid);
42
43    printf("Preferred I/O block size: %ld bytes\n", (long) sb.st_blksize);
44    printf("File size:          %lld bytes\n", (long long) sb.st_size);
45    printf("Blocks allocated:  %lld\n", (long long) sb.st_blocks);
46
47    printf("Last status change:  %s", ctime(&sb.st_ctime));
48    printf("Last file access:    %s", ctime(&sb.st_atime));
49    printf("Last file modification: %s", ctime(&sb.st_mtime));
50
51    exit(EXIT_SUCCESS);
52 }

```

程序第 11~14 行，判断命令行的合法性，若参数（含程序名本身）个数不为 2，则输出正确的命令格式提示并退出。

程序第 16 行，调用 stat 函数读取命令行参数所指定的文件的属性信息，读取到的属性信息存

储在 sb 变量中。若 stat 函数出错，则调用 perror 函数输出错误提示并退出。perror 函数的作用是以字符串可读的形式输出全局变量 errno 所代表的错误。

程序第 23~32 行，输出文件类型信息。

程序第 34 行，输出文件的索引节点号；第 36~37 行，以八进制格式输出 st_mode 成员的值；第 39 行输出硬链接数；第 40 行输出文件所有者用户 ID 和组 ID。

程序第 43~45 行，输出文件大小相关信息。

程序第 47~49 行，输出文件改变时间相关信息。其中的 ctime 函数用于将 time_t 结构体表示的时间转换为字符串可读形式的时间。

在命令行编译运行 ex_stat.c，如下所示。

```
jianglinmei@ubuntu:~/c$ gcc -o ex_stat ex_stat.c
jianglinmei@ubuntu:~/c$ ./ex_stat ex_stat
File type: regular file
I-node number: 271850
Mode: 100755 (octal)
Link count: 1
Ownership: UID=1001 GID=1002
Preferred I/O block size: 4096 bytes
File size: 7510 bytes
Blocks allocated: 16
Last status change: Fri Nov 9 11:57:26 2012
Last file access: Fri Nov 9 11:57:33 2012
Last file modification: Fri Nov 9 11:57:26 2012
```

5.2.5.2 access 函数

在对文件编程时，判断文件是否存在以及是否可访问是一类常见操作。使用前面所述的 stat 系列函数获取文件属性信息再进一步判断是一种可选的方法。但 stat 系列函数的使用稍显复杂，使用 access 函数进行文件的存取许可权测试会是一个更好的选择。access 函数的原型如下。

```
#include <unistd.h>
int access(const char *pathname, int mode);
```

access 函数按实际用户 ID 和实际组 ID 进行存取许可权测试，其各参数和返回值的含义如下。

1. pathname 文件名
2. mode 测试项，其值可以是以下值之一或多个值按位或的结果
 - R_OK 测试读许可权
 - W_OK 测试写许可权
 - X_OK 测试执行许可权
 - F_OK 测试文件是否存在
3. 返回值

- 成功（具有所有测试项权限）为 0；若出错为 -1，错误值记录在 errno

程序清单 5-5 给出了一个使用 access 函数的示例。该示例程序的功能是，判断用户以命令行参数的形式提供的文件是否存在并且可读，如果不存在或不可读则输出相应的提示信息，如果可读则读取前 20 个字节的数据并输出。

程序清单 5-5 ex_access.c

```
1 #include <sys/types.h>
```

```

2 #include <unistd.h>
3 #include <sys/stat.h>
4 #include <fcntl.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 int main(int argc, char *argv[])
9 {
10     int i, num;
11     int fd;
12     char buf[20];
13
14     if (argc != 2) {
15         printf("Usage: %s <pathname>\n", argv[0]);
16         exit(EXIT_FAILURE);
17     }
18
19     if(access(argv[1], F_OK) != 0) {
20         printf("The file '%s' is not existed!\n", argv[1]);
21         exit(EXIT_FAILURE);
22     }
23     else if(access(argv[1], R_OK) != 0) {
24         printf("The file '%s' can not be read!\n", argv[1]);
25         exit(EXIT_FAILURE);
26     }
27
28     if((fd = open(argv[1], O_RDONLY)) < 0) {
29         printf("Fail to open file '%s'!\n", argv[1]);
30         exit(EXIT_FAILURE);
31     }
32
33     if((num = read(fd, buf, 20)) < 0) {
34         close(fd);
35         printf("Fail to read file '%s'!\n", argv[1]);
36         exit(EXIT_FAILURE);
37     }
38
39     printf("The starting %d bytes of '%s' is:\n", num, argv[1]);
40     for(i = 0; i < num; i++) {
41         printf("%c(%x) ", buf[i], buf[i]);
42     }
43     printf("\n");
44
45     close(fd);
46
47     exit(EXIT_SUCCESS);
48 }

```

程序第 14~17 行，判断命令行的合法性，若参数（含程序名本身）个数不为 2，则输出正确的命令格式提示并退出。

程序第 19~22 行，判断参数所指定的文件是否存在，如果不存在则输出错误提示并退出。

程序第 23~26 行，判断参数所指定的文件是否可读，如果不可读则输出错误提示并退出。

程序第 28~31 行，打开参数所指定的文件，如果打开失败则输出错误提示并退出。

程序第 33~37 行，读取文件前 20 个字节，实际读取到的字节数保存于变量 num，如果读取

失败则关闭文件描述符，然后输出错误提示并退出。

程序第 39~43 行，以字符 (%d) 和十六进制 (%x) 的格式输出前面读取到的文件数据的每个字节。

在命令行编译运行 ex_access.c，如下所示。

```
jianglinmei@ubuntu:~/c$ gcc -o ex_access ex_access.c
jianglinmei@ubuntu:~/c$ ./ex_access nofile
The file 'nofile' is not existed!
jianglinmei@ubuntu:~/c$ ./ex_access ex_access.c
The starting 20 bytes of 'ex_access.c' is:
#(23) i(69) n(6e) c(63) l(6c) u(75) d(64) e(65) (20) <(3c) s(73) y(79) s(73) /(2f)
t(74) y(79) p(70) e(65) s(73) .(2e)
```

5.2.6 文件属性的修改

为了更好地使用文件，有时需要使用系统调用来修改文件的属性信息，如文件的所有者、文件名、文件的长度等。

5.2.6.1 改变文件的所有者

文件所有者是 Linux 系统中的文件所具有的一种属性，使用 chown 系列函数可以改变该属性，同时还可以改变文件的所属组。这些函数的原型如下。

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

这些函数各参数和返回值的含义如下。

1. path 文件名
2. owner 新文件所有者的用户 ID（一个 32 位无符号整数）。当指定为-1 时，表示保持所有者不变
3. group 新文件所属组的组 ID（一个 32 位无符号整数）。当指定为-1 时，表示保持所属组不变
4. fd 文件描述符。使用 fchown 函数需先打开文件
5. 返回值
 - 成功为 0；若出错为-1，错误值记录在 errno

这三个函数中，chown 和 lchown 的参数完全相同，两者区别在于：当文件是一个符号链接时，lchown 改变的是该符号链接本身的所有者，而 chown 改变的是该链接指向的文件的所有者。

因为涉及权限管理问题，只有 root 用户才可以使用这些函数来改变任意文件的所有者和所属组，而普通用户只能改变属于自己的文件的所属组，并且指定的所属组只能是用户自身所在组之一。

程序清单 5-6 给出了一个使用 chown 函数的示例。该示例程序的功能是改变用户指定的文件的所有者。用户在命令行应提供两个额外的参数，第一个参数为所有者用户 ID，第二个为要改变所有者的文件的文件名。

程序清单 5-6 ex_chown.c

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
```

```

3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(int argc, char *argv[])
7 {
8     if (argc != 3) {
9         printf("Usage: %s <uid> <pathname>\n", argv[0]);
10        exit(EXIT_FAILURE);
11    }
12
13    if (chown(argv[2], atoi(argv[1]), -1) != 0) {
14        printf("Fail to change owner of %s\n", argv[2]);
15        exit(EXIT_FAILURE);
16    }
17
18    exit(EXIT_SUCCESS);
19 }

```

程序第 8~11 行，判断命令行的合法性，若参数（含程序名本身）个数不为 3，则输出正确的命令格式提示并退出。

程序第 13~16 行，更改文件的所有者，如果更改失败则输出错误提示并退出。其中第 13 行的 atoi 函数的作用是将一个字符串转换成一个对应的十进制整数，在此即将命令行的第一个参数转换成十进制数表示的 UID。

在命令行编译运行 ex_chown.c，如下所示。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_chown ex_chown.c
# 注：创建一个用于测试的用户 alice
jianglinmei@ubuntu:~/c$ sudo useradd -m -s /bin/bash alice
[sudo] password for jianglinmei:
# 注：id 命令可显示用户的 UID 和 GID
jianglinmei@ubuntu:~/c$ id alice
uid=1002(alice) gid=1003(alice) 组=1003(alice)
#注：创建一个用于测试文件所有者的临时文件
jianglinmei@ubuntu:~/c$ touch testowner
jianglinmei@ubuntu:~/c$ ll testowner
-rw-rw-r-- 1 jianglinmei jianglinmei 0 2012-11-11 21:32 testowner
#注：下条命令调用 ex_chown 将 testowner 的所有者更改为 alice
jianglinmei@ubuntu:~/c$ sudo ./ex_chown 1002 testowner
jianglinmei@ubuntu:~/c$ ll testowner
-rw-rw-r-- 1 alice jianglinmei 0 2012-11-11 21:32 testowner

```

5.2.6.2 改变文件的访问权限

在 Linux 环境下，文件的读、写和执行等访问权限是文件的重要属性之一。要改变文件的访问权限，需要使用 chmod 系列函数，这些函数的原型如下。

```

#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);

```

这两个函数各参数和返回值的含义如下。

1. path 文件名

2. mode 文件的权限(一个32位无符号整数)。一般由一组八进制数进行二进制“或”运算构成，其中的各个位表示一种权限，称权限位，这些权限位如表5-2所示。

表5-2

文件权限位及其含义

权限位	八进制常量值	含义
S_ISUID	04000	设置SUID
S_ISGID	02000	设置SGID
S_ISVTX	01000	设置黏滞位
S_IRUSR	00400	文件所有者可读
S_IWUSR	00200	文件所有者可写
S_IXUSR	00100	文件所有者可执行
S_IRGRP	00040	文件所属组可读
S_IWGRP	00020	文件所属组可写
S_IXGRP	00010	文件所属组可执行
S_IROTH	00004	其他用户可读
S_IWOTH	00002	其他用户可写
S_IXOTH	00001	其他用户可执行

3. fd 文件描述符。使用fchmod需先打开文件

4. 返回值

成功为0；若出错为-1，错误值记录在errno

程序清单5-7给出了一个使用chmod函数的示例。该示例程序的功能是将用户指定的文件的权限更改为所有者可写、组用户和其他用户可读。

程序清单5-7 ex_chmod.c

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(int argc, char *argv[])
7 {
8     if (argc != 2) {
9         printf("Usage: %s <pathname>\n", argv[0]);
10        exit(EXIT_FAILURE);
11    }
12
13    if (chmod(argv[1], S_IWUSR | S_IRGRP | S_IROTH) != 0) {
14        printf("Fail to change privilege of %s\n", argv[1]);
15        exit(EXIT_FAILURE);
16    }
17
18    exit(EXIT_SUCCESS);
19 }
```

本程序的代码结构与程序清单5-6基本一样，因此不再多做解释。在命令行编译运行

ex_chmod.c，如下所示。

```
jianglinmei@ubuntu:~/c$ gcc -o ex_chmod ex_chmod.c
jianglinmei@ubuntu:~/c$ touch testmod
jianglinmei@ubuntu:~/c$ ll testmod
-rw-rw-r-- 1 jianglinmei jianglinmei 0 2012-11-11 22:07 testmod
jianglinmei@ubuntu:~/c$ ./ex_chmod testmod
jianglinmei@ubuntu:~/c$ ll testmod
--w-r--r-- 1 jianglinmei jianglinmei 0 2012-11-11 22:07 testmod
```

5.2.6.3 重命名文件

普通文件和目录文件均可使用 rename 函数重命名，该函数的原型如下。

```
#include <stdio.h>
int rename(const char *oldpath, const char *newpath);
```

函数各参数和返回值的含义如下。

1. oldpath 旧文件名
2. newpath 新文件名
3. 返回值

成功为 0；若出错为-1，错误值记录在 errno

rename 函数的旧文件名、新文件名与返回结果有密切关系，如表 5-3 所示。

表 5-3 rename 函数的参数与返回结果的关系

oldpath	newpath 文件不存在	newpath 为普通文件	newpath 为目录文件
普通文件	文件被重命名，返回成功	原 newpath 文件被删除，oldpath 文件被重命名为 newpath，返回成功	返回错误
目录文件	文件被重命名，返回成功	返回错误	newpath 目录文件为空则该目录被删除，oldpath 目录被重命名为 newpath，返回成功。否则返回错误

另外，如果新文件名和旧文件名完全一样，rename 函数不做任何操作而返回成功。对于目录文件还应注意，newpath 不能以 oldpath 为前缀，即不能将一个目录文件重命名为该目录的子目录或子孙目录文件。

程序清单 5-8 给出了一个使用 rename 函数的示例。该示例程序的功能是将命令行第一个参数所指的文件改名为第二个参数所指的字符串。

程序清单 5-8 ex_rename.c

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(int argc, char *argv[])
7 {
8     if (argc != 3) {
9         printf("Usage: %s <oldpathname> <newpathname>\n", argv[0]);
10        exit(EXIT_FAILURE);
11    }
```

```

12
13     if (rename(argv[1], argv[2]) != 0) {
14         printf("Fail to change name of %s\n", argv[1]);
15         exit(EXIT_FAILURE);
16     }
17
18     exit(EXIT_SUCCESS);
19 }
```

本程序的代码结构与程序清单 5-6 基本一样，因此不再多做解释。在命令行编译运行 ex_rename.c，如下所示。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_rename ex_rename.c
jianglinmei@ubuntu:~/c$ ./ex_rename
Usage: ./ex_rename <oldpathname> <newpathname>
jianglinmei@ubuntu:~/c$ ./ex_rename testowner testrename
jianglinmei@ubuntu:~/c$ ll testrename
-rw-rw-r-- 1 alice jianglinmei 0 2012-11-11 21:32 testrename
```

5.2.6.4 复制文件描述符

复制文件描述符需要用到 dup 或 dup2 函数。这两个函数常用于重定向一个已打开的文件描述符。其原型如下。

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

函数各参数和返回值的含义如下。

1. oldfd 旧文件描述符
2. newfd 新文件描述符
3. 返回值

若成功为复制后的文件描述符；若出错为-1，错误值记录在 errno

这两个函数都将返回复制后的文件描述符。也就是说，复制得到的文件描述符和旧文件描述符将指向同一个打开的文件。不同的是，dup 函数返回的是最小未用的文件描述符，而 dup2 函数返回的是预先指定的文件描述符 newfd。如果 newfd 正在使用，则会先关闭 newfd。但如果 newfd 与 oldfd 一样，则关闭该文件正常返回。

5.2.6.5 上锁和解锁文件

Linux 下可以调用 flock 函数来上锁或解锁一个文件。该函数的原型如下。

```
#include <sys/file.h>
int flock(int fd, int operation);
```

函数各参数和返回值的含义如下。

1. fd 文件描述符
2. operation 上锁或解锁方式，可取以下值之一
 - LOCK_SH 共享锁
 - LOCK_EX 独占锁
 - LOCK_UN 解锁
3. 返回值

若成功为 0；若出错为 -1，错误值记录在 errno

一个进程对一个文件只能有一个独占锁，但可以有多个共享锁。上锁的作用只有在别的进程要对该文件上锁时才能表现出来。如果一个程序不试图去上锁一个已经被上锁的文件，就不应该对其进行访问。

应当注意：对文件的操纵本身与锁其实没有什么关系。无论文件是否被上锁，用户都可以随便对文件进行正常情况下的任何操作。上锁文件的目的是为了同步多个进程之间的操作，参与同步的各进程必须遵守约定的规则（上锁后再操作），上锁才有意义。

默认情况下，flock 是阻塞式的。也就是说，如果另一个进程已持有该文件的锁且该锁与本进程请求的锁不兼容，flock 将会阻塞，直到拥有该文件的锁的进程对其解锁为止。如果要进行非阻塞式调用，应将 operation 参数与常量“LOCK_NB”进行二进制“或”操作后再传递给 flock 函数。非阻塞式调用 flock 后，flock 会立即返回 -1，并且 errno 的值将为 EWOULDBLOCK。

5.3 链接文件的操作

链接文件是 Linux 系统的一种特殊文件，它实际上是指向一个现实存在的文件的链接。链接文件又分为硬链接文件和符号链接文件。下面分别介绍这两类链接文件的相关系统调用。

5.3.1 创建硬链接

当需要对一个已经存在的文件建立新的链接时，需要使用系统调用 link 函数。该函数的原型如下。

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
```

函数各参数和返回值的含义如下。

1. oldpath 已存在的文件名
2. newpath 要新建的链接文件名
3. 返回值

若成功为 0；若出错为 -1，错误值记录在 errno

使用 link 函数时应注意，新的链接文件 newpath 和已存在的文件 oldpath 应在同一个文件系统中。如果新的链接文件 newpath 已经存在，已存在的文件不会被覆盖。

使用 link 创建的新的链接文件和原文件是一模一样的，地位也是对等的，它们都指向相同的文件（引用相同的文件索引节点）。创建之后，就没必要也无法区分哪个是原始文件了。

5.3.2 创建和读取符号链接

与硬链接直接指向文件的索引节点不同，符号链接是指向某一文件的指针。符号链接可以跨文件系统创建。

5.3.2.1 创建符号链接

创建符号链接文件的系统调用函数为 symlink，其原型如下。

```
#include <unistd.h>
```

```
int symlink(const char *oldpath, const char *newpath);
```

函数各参数和返回值的含义如下。

1. oldpath 已存在的文件名
2. newpath 要新建的符号链接文件名
3. 返回值

若成功为 0；若出错为 -1，错误值记录在 errno

和创建硬链接一样，如果新的链接文件 newpath 已经存在，已存在的文件不会被覆盖。符号链接文件可以指向一个并不存在的文件，这种符号链接被称为“悬浮链接（dangling link）”。

符号链接文件的权限和原文件的权限是无关的。

5.3.2.2 读取符号链接

读取符号链接所指向的目标文件需要使用系统调用 readlink 函数，该函数的原型如下。

```
#include <unistd.h>
ssize_t readlink(const char *path, char *buf, size_t bufsiz);
```

函数各参数和返回值的含义如下。

1. path 符号链接文件名
2. buf 用于存储获取到的信息（所指向的目标文件名）的缓冲区
3. bufsiz 缓冲区大小
4. 返回值

若成功为实际写入缓冲区的字节数；若出错为 -1，错误值记录在 errno

5.3.3 删除链接

要删除链接，包括硬链接和符号链接，可通过系统调用 unlink 函数实现。该函数的型如下。

```
#include <unistd.h>
int unlink(const char *pathname);
```

函数各参数和返回值的含义如下。

1. path 要删除的文件的文件名
2. 返回值
 - 若成功为 0；若出错为 -1，错误值记录在 errno

unlink 函数的作用是从文件系统中删除一个文件名。

对于硬链接，如果被删除的文件名是引用一个文件的最后一个文件名，并且没有任何进程打开该文件，则该文件将被删除，其占用的存储空间也被释放。如果被删除的文件名是引用一个文件的最后一个文件名，但是还有进程打开该文件，该文件将在最后一个打开的文件描述符被关闭时才会被删除。

对于符号链接，仅该符号链接被删除，而不会影响其指向的目标文件。

对于套接字（socket）、管道（fifo）或设备文件，文件名会被删除，但是已打开该文件的进程仍可继续使用它。

程序清单 5-9 给出了一个综合使用各链接文件操作函数的示例。该示例程序的功能类似 linux 命令 ln，可为指定文件创建硬链接（使用 “-h” 选项）或符号链接（使用 “-s” 选项），还可删除

链接（使用“-d”选项）和读取符号连接（使用“-r”选项）。

程序清单 5-9 ex_link.c

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main(int argc, char *argv[])
8 {
9     char buf[512];
10
11    if (argc != 4 && argc != 3) goto ERR_FORMAT;
12
13    if (strcmp(argv[1], "-h") == 0) {
14        if (argc == 4) {
15            if (link(argv[2], argv[3]) != 0) goto ERROR;
16        }
17        else goto ERR_FORMAT;
18    }
19    else if (strcmp(argv[1], "-s") == 0) {
20        if (argc == 4) {
21            if (symlink(argv[2], argv[3]) != 0) goto ERROR;
22        }
23        else goto ERR_FORMAT;
24    }
25    else if (strcmp(argv[1], "-r") == 0) {
26        if (readlink(argv[2], buf, 512) < 0) goto ERROR;
27
28        printf("%s is linked to %s\n", argv[2], buf);
29    }
30    else if (strcmp(argv[1], "-d") == 0) {
31        if (unlink(argv[2]) != 0) goto ERROR;
32    }
33    else goto ERR_FORMAT;
34
35    exit(EXIT_SUCCESS);
36
37 ERR_FORMAT:
38    printf("Usage: \n");
39    printf("\t%s <-h | -s> <pathname> <linkpathname>\n", argv[0]);
40    printf("\t%s <-r | -d> <linkpathname>\n", argv[0]);
41    exit(EXIT_FAILURE);
42 ERROR:
43    printf("Fail to create link!\n");
44    exit(EXIT_FAILURE);
45 }
```

程序第 37 行定义了一个表示命令格式错误的标签 ERR_FORMAT，第 38~41 行是输出正确的命令格式并退出。

程序第 42 行定义了一个表示命令执行出错的标签，第 43~44 行是输出相应的错误提示并退出。

程序第 11 行判断参数个数是否正确,如果不正确则程序流程跳转到标签 ERR_FORMAT 之后的语句。

程序第 13 ~ 18 行创建硬链接。其中, strcmp 函数用于比较两个字符串,当它的两个参数所指的字符串相等时, strcmp 返回 0。在此, strcmp 用于判断命令行第二个参数是否为 “-h”。如果是 “-h”, 则进一步判断参数个数是否为 4, 为 4 则创建硬链接, 否则程序流程跳转到标签 ERR_FORMAT 之后的语句, 执行错误处理。如果创建硬链接出错, 则程序流程跳转到标签 ERROR 之后的语句, 执行错误处理。

程序第 19 ~ 24 行创建符号链接。

程序第 25 ~ 29 行读取符号链接。

程序第 30 ~ 32 行删除链接。

在命令行编译运行 ex_link.c, 如下所示。

```
jianglinmei@ubuntu:~/c$ gcc -g -o ex_link ex_link.c
jianglinmei@ubuntu:~/c$ ./ex_link
Usage:
      ./ex_link <-h | -s> <pathname> <linkpathname>
      ./ex_link <-r | -d> <linkpathname>
jianglinmei@ubuntu:~/c$ ./ex_link -h ex_link ex_hard_link
jianglinmei@ubuntu:~/c$ ./ex_link -s ex_link ex_symbol_link
jianglinmei@ubuntu:~/c$ ll ex_*_link
-rwxrwxr-x 2 jianglinmei jianglinmei 8576 2012-11-11 22:55 ex_hard_link*
lrwxrwxrwx 1 jianglinmei jianglinmei    7 2012-11-11 22:56 ex_symbol_link -> ex_link*
jianglinmei@ubuntu:~/c$ ./ex_link -r ex_symbol_link
ex_symbol_link is linked to ex_link
jianglinmei@ubuntu:~/c$ ./ex_link -d ex_hard_link
jianglinmei@ubuntu:~/c$ ll ex_*_link
lrwxrwxrwx 1 jianglinmei jianglinmei 7 2012-11-11 22:56 ex_symbol_link -> ex_link*
```

5.4 目录文件的操作

目录文件是一类比较特殊的文件, 它在构造 Linux 的树型文件系统中起着重要的作用。Linux 系统为目录文件的操作提供了一些专用的系统调用。

5.4.1 目录文件的创建与删除

5.4.1.1 创建目录文件

创建目录文件可使用 mkdir 函数, 其原型如下。

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
```

函数各参数和返回值的含义如下。

1. pathname 新创建的目录文件名
2. mode 存取许可权位, 由下列一个或多个常数进行或运算构成。最终权限受系统变量 umask 限制 (mode & ~umask & 0777)

- S_IRUSR 文件所有者-读
- S_IWUSR 文件所有者-写
- S_IXUSR 文件所有者-执行
- S_IRGRP 组用户-读
- S_IWGRP 组用户-写
- S_IXGRP 组用户-执行
- S_IROTH 其他用户-读
- S_IWOTH 其他用户-写
- S_IXOTH 其他用户-执行

3. 返回值

若成功为 0；若出错为 -1，错误值记录在 `errno`

创建后，新目录文件的所有者是当前进程的有效用户。文件的所属组是当前进程的有效组或父目录的所属组（与文件系统的类型和挂载方式有关）。如果父目录具有 SGID 属性，则新建目录也具有该属性。

5.4.1.2 删除目录文件

可以使用系统调用 `rmdir` 函数删除目录文件，其原型如下。

```
#include <unistd.h>
int rmdir(const char *pathname);
```

函数各参数和返回值的含义如下。

1. `pathname` 要删除的目录文件名
2. 返回值

若成功为 0；若出错为 -1，错误值记录在 `errno`

使用 `rmdir` 函数应注意，它只能删除空目录。如果 `pathname` 所指目录中含有任何文件，函数调用将失败。

5.4.2 目录文件的打开与关闭

同普通文件的访问一样，使用一个目录文件之前需将其打开，使用完毕之后应将其关闭。

5.4.2.1 打开目录文件

打开目录文件可以使用 `opendir` 函数（非系统调用），其原型如下。

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

函数各参数和返回值的含义如下。

1. `name` 要打开的目录文件名
2. 返回值

若成功为指向目录文件的结构指针；若出错为 NULL，错误值记录在 `errno`

如果 `opendir` 函数执行成功会返回一个目录流（directory stream），该流定位在目录块的第一项。`DIR` 类型是一个内部结构，其定义对用户是透明的。

5.4.2.2 关闭目录文件

关闭目录文件可以使用 `closedir` 函数（非系统调用），其原型如下。

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

函数各参数和返回值的含义如下。

1. dirp 已打开的目录流
2. 返回值
- 若成功为 0；若出错为 -1，错误值记录在 errno

5.4.3 目录文件的读取

读取目录文件可以使用 readdir 函数（非系统调用），其原型如下。

```
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

函数各参数和返回值的含义如下。

1. dirp 已打开的目录流
2. 返回值
- 若成功为下一个目录项的指针；若已读取到目录流末尾则返回 NULL 且 errno 的值不变；若出错为 NULL，错误值记录在 errno

函数返回的结构体类型 struct dirent 定义如下。

```
struct dirent {
    ino_t          d_ino;           /* 索引节点 */
    off_t          d_off;           /* 下一目录项的位移量 */
    unsigned short d_reclen;        /* 本记录的长度 */
    unsigned char   d_type;          /* 文件类型 */
    char            d_name[256];      /* 文件名 */
};
```

应当注意：该结构中，仅 d_ino 和 d_name 两个数据成员是 POSIX.1 标准中强制规定的；其他成员尚未标准化，亦未得到支持，故不应该使用它们。

程序清单 5-10 给出了一个打开、读取并关闭目录的示例。该示例程序的功能是：列出用户指定目录下的所有文件。

程序清单 5-10 ex_dir.c

```
1 #include <sys/types.h>
2 #include <dirent.h>
3 #include <unistd.h>
4 #include <stdio.h>
5
6 int main(int argc, char *argv[])
7 {
8     DIR * dir;
9     struct dirent * ptr;
10    int i = 0;
11
12    if (argc != 2) {
13        printf("Usage: %s <dir>\n", argv[0]);
```

```

14     return -1;
15 }
16
17 if ((dir = opendir(argv[1])) == NULL) {
18     printf("Fail to open directory %s\n", argv[1]);
19     return -1;
20 }
21
22 while((ptr = readdir(dir)) != NULL) {
23     printf("%-16s\t", ptr->d_name);
24     if(++i % 4 == 0)
25         printf("\n");
26 }
27 printf("\n");
28
29 closedir(dir);
30
31 return 0;
32 }

```

程序第 8~9 行定义了目录操作所需的两个结构体变量。第 12~15 行判断命令行参数的合法性。第 17~20 行打开目录，如打开失败则输出提示并退出。第 22~27 行以每行 4 个文件的形式输出所打开的目录下的所有（包括隐藏文件）文件的名称。

在命令行编译运行 ex_dir.c，如下所示。

```

jianglinmei@ubuntu:~/c$ gcc -o ex_dir ex_dir.c
jianglinmei@ubuntu:~/c$ ./ex_dir /home
.
..          alice          jianglinmei
hadoop      molin

```

5.5 设备文件

设备文件是 Linux 中的一类非常特别的文件。Linux 为外部设备提供了一种标准接口，使得用户可以非常方便地以访问文件的方式访问外部设备。

由于在 Linux 系统中，所有的外部设备都被看作是/dev 目录下的一个文件，所以本章前文所述的各种关于文件的系统调用均可使用于外部设备文件，因此可以方便地使用基于文件描述符的 I/O 操作实现对外部设备的操作。但因设备各异，有些设备具有其特殊性。例如，磁带是一种非随机访问的存储介质，只能顺序存取。因此系统调用 lseek 对它来说是无效的。

5.6 小结

本章首先介绍了 Linux 环境下文件操作的统一接口、文件和目录的组织管理方式和特殊的设备文件的作用，然后重点介绍了底层文件访问的方法以及使用文件描述符访问文件所需用到的系统调用函数，同时介绍了读取和修改文件属性的方法。链接文件是 Linux 文件管理的一大特性，本章随后介绍了链接文件的创建、删除和读取的方法。目录文件的操作具有其特殊性，并且使用