附录 E　实验室：指南

这是一份非常简短的文档，可以帮助你熟悉 UNIX 系统上 C 编程环境的基础知识。它不是面面俱到或特别详细，只是给你足够的知识让你继续学习。

关于编程的几点一般建议：如果想成为一名专业程序员，需要掌握的不仅仅是语言的语法。具体来说，应该了解你的工具，了解你的库，并了解你的文档。与 C 编译相关的工具是 gcc、gdb 和 ld。还有大量的库函数也可供你使用，但幸运的是 libc 包含了许多功能，默认情况下它与所有 C 程序相关联——需要做的就是包含正确的头文件。最后，了解如何找到所需的库函数（例如，学习查找和阅读手册页）是一项值得掌握的技能。我们稍后将更详细地讨论这些内容。

就像生活中（几乎）所有值得做的事情，成为这些领域的专家需要时间——事先花时间了解有关工具和环境的更多信息，绝对值得付出努力。

E.1　一个简单的 C 程序

我们从一个简单的 C 程序开始，它保存在文件“hw.c”中。与 Java 不同，文件名和文件内容之间不一定有关系。因此，请以适当的方式，利用你的常识来命名文件。

第一行指定要包含的文件，在本例中为 stdio.h，它包含许多常用输入/输出函数的“原型”。我们感兴趣的是 printf()。当你使用#include 指令时，就告诉 C 预处理器（cpp）查找特定文件（例如，stdio.h），并将其直接插入到#include 的代码中。默认情况下，cpp 将查看目录/usr/include/，尝试查找该文件。

下面一部分指定 main()函数的签名，即它返回一个整数（int），并用两个参数来调用，一个整数 argc，它是命令行上参数数量的计数。一个指向字符（argv）的指针数组，每个指针都包含命令行中的一个单词，最后一个单词为 null。下面的指针和数组会更多。

```
/* header files go up here */
/* note that C comments are enclosed within a slash and a star, and
   may wrap over lines */
// if you use gcc, two slashes will work too (and may be preferred)
#include <stdio.h>

/* main returns an integer */
int main(int argc, char *argv[]) {
    /* printf is our output function;
       by default, writes to standard out */
    /* printf returns an integer, but we ignore that */
    printf("hello, world\n");
```

```
    /* return 0 to indicate all went well */
    return(0);
}
```

程序然后简单打印字符串“hello，world”，并将输出流换到下一行，这是由 printf()调用结束时的“\n”实现的。然后，程序返回一个值并结束，该值被传递回执行程序的 shell。终端上的脚本或用户可以检查此值（在 csh 和 tcsh shell 中，它存储在状态变量中），以查看程序是干净地退出还是出错。

E.2 编译和执行

我们现在将学习如何编译程序。请注意，我们将使用 gcc 作为示例，但在某些平台上，可以使用不同的（本机）编译器 cc。

在 shell 提示符下，只需键入：

```
prompt> gcc hw.c
```

gcc 不是真正的编译器，而是所谓的“编译器驱动程序”，因此它协调了编译的许多步骤。通常有 4～5 个步骤。首先，gcc 将执行 cpp（C 预处理器）来处理某些指令（例如#define 和#include。程序 cpp 只是一个源到源的转换器，所以它的最终产品仍然只是源代码（ 即一个 C 文件）。然后真正的编译将开始，通常是一个名为 cc1 的命令。这会将源代码级别的 C 代码转换为特定主机的低级汇编代码。然后执行汇编程序 as，生成目标代码（机器可以真正理解的数据位和代码位），最后链接编辑器（或链接器）ld 将它们组合成最终的可执行程序。幸运的是（!），在大多数情况下，你可以不明白 gcc 如何工作，只需愉快地使用正确的标志。

上面编译的结果是一个可执行文件，命名为（默认情况下）a.out。然后运行该程序，只需键入：

```
prompt> ./a.out
```

运行该程序时，操作系统将正确设置 argc 和 argv，以便程序可以根据需要处理命令行参数。具体来说，argc 将等于 1，argv [0]将是字符串“./a.out”，而 argv[1]将是 null，表示数组的结束。

E.3 有用的标志

在继续使用 C 语言之前，我们首先指出一些有用的 gcc 编译标志。

```
prompt> gcc -o hw hw.c  # -o: to specify the executable name
prompt> gcc -Wall hw.c  # -Wall: gives much better warnings
prompt> gcc -g hw.c     # -g: to enable debugging with gdb
prompt> gcc -O hw.c     # -O: to turn on optimization
```

当然，你可以根据需要组合这些标志（例如 gcc -o hw -g -Wall hw.c）。在这些标志中，你应该总是使用-Wall，这会提供很多关于可能出错的额外警告。不要忽视警告！相反，要修复它们，让它们幸福地消失。

E.4 与库链接

有时，你可能想在程序中使用库函数。因为 C 库中有很多函数（可以自动链接到每个程序），所以通常要做的就是找到正确的#include 文件。最好的方法是通过手册页（manual page），通常称为 man page。

例如，假设你要使用 fork()系统调用[①]。在 shell 提示符下输入 man fork，你将获得 fork()如何工作的文本描述。最顶部的是一个简短的代码片段，它告诉你在程序中需要#include 哪些文件才能让它通过编译。对于 fork()，需要#include sys/types.h 和 unistd.h，按如下方式完成：

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

但是，某些库函数不在 C 库中，因此你必须做更多的工作。例如，数学库有许多有用的函数（正弦、余弦、正切等）。如果要在代码中包含函数 tan()，应该先查手册页。在 tan 的 Linux 手册页的顶部，你会看到以下两行代码：

```
#include <math.h>
...
Link with -lm.
```

你已经应该理解了第一行——你需要#include 数学库，它位于文件系统的标准位置（即/usr/include/math.h）。但是，下一行告诉你如何将程序与数学库“链接”。有许多有用的库可以链接。其中许多都放在/usr/lib 中，数学库也确实在这里。

有两种类型的库：静态链接库（以.a 结尾）和动态链接库（以.so 结尾）。静态链接库直接组合到可执行文件中。也就是说，链接器将库的低级代码插入到可执行文件中，从而产生更大的二进制对象。动态链接通过在程序可执行文件中包含对库的引用来改进这一点。程序运行时，操作系统加载程序动态链接库。这种方法优于静态方法，因为它节省了磁盘空间（没有不必要的大型可执行文件），并允许应用程序在内存中共享库代码和静态数据。对于数学库，静态和动态版本都可用，静态版本是/usr/lib/libm.a，动态版本是/usr/lib/libm.so。

在任何情况下，要与数学库链接，都需要向链接编辑器指定库。这可以通过使用正确的标志调用 gcc 来实现。

```
prompt> gcc -o hw hw.c -Wall -lm
```

-l×××标志告诉链接器查找 lib×××.so 或 lib×××.a，可能按此顺序。如果出于某种原因，你坚持使用动态库而不是静态库，那么可以使用另一个标志——看看你是否能找到它是什么。人们有时更喜欢库的静态版本，因为使用动态库有一点点性能开销。

① 请注意，fork()是一个系统调用，而不仅仅是一个库函数。 但是，C 库为所有系统调用提供了 C 包装函数，每个系统调用函数都会陷入操作系统。

最后要注意：如果你希望编译器在不同于常用位置的路径中搜索头文件，或者希望它与你指定的库链接，可以使用编译器标志-I /foo/bar，来查找目录/foo/ bar 中的头文件，使用 -L /foo/bar 标志来查找/foo/bar 目录中的库。以这种方式指定的一个常用目录是“.”（称为“点”），它是 UNIX 中当前目录的简写。

请注意，-I 标志应该针对编译，而-L 标志针对链接。

E.5 分别编译

一旦程序开始变得足够大，你可能希望将其拆分为单独的文件，分别编译每个文件，然后将它们链接在一起。例如，假设你有两个文件，hw.c 和 helper.c，希望单独编译它们，然后将它们链接在一起。

```
# we are using -Wall for warnings, -O for optimization
prompt> gcc -Wall -O -c hw.c
prompt> gcc -Wall -O -c helper.c
prompt> gcc -o hw hw.o helper.o -lm
```

-c 标志告诉编译器只是生成一个目标文件——在本例中是名为 hw.o 和 helper.o 的文件。这些文件不是可执行文件，而只是每个源文件中代码的机器代码表示。要将目标文件组合成可执行文件，必须将它们“链接”在一起。这是通过第三行 gcc -o hw hw.o helper.o）完成的。在这种情况下，gcc 看到指定的输入文件不是源文件（.c），而是目标文件（.o），因此直接跳到最后一步，调用链接编辑器 ld 将它们链接到一起，得到单个可执行文件。由于它的功能，这行通常被称为“链接行”，并且可以指定特定的链接命令，例如-lm。类似地，仅在编译阶段需要的标志，诸如-Wall 和-O，就不需要包含在链接行上，只是包含在编译行上。

当然，你可以在一行中为 gcc 指定所有 C 源文件（gcc -Wall -O -o hw hw.c helper.c），但这需要系统重新编译每个源代码文件，这个过程可能很耗时。通过单独编译每个源文件，你只需重新编译编辑修改过的文件，从而节省时间，提高工作效率。这个过程最好由另一个程序 make 来管理，我们接下来介绍它。

E.6 Makefile 文件

程序 make 让你自动化大部分构建过程，因此对于任何认真的程序（和程序员）来说，都是一个至关重要的工具。来看一个简单的例子，它保存在名为 Makefile 的文件。

要构建程序，只需输入：

```
prompt> make
```

这会（默认）查找 Makefile 或 makefile，将其作为输入（你可以用标志指定不同的 makefile，阅读手册页，找出是哪个标志）。gmake 是 make 的 gnu 版本，比传统的 make 功能更多，所以我们将在下面的部分中重点介绍它（尽管我们互换使用这两个词）。这些讨论大多数都基于 gmake 的 info 页面，要了解如何查找这些页面，请参阅“E.8 文档”部分。另

外请注意：在 Linux 系统上，gmake 和 make 是一样的。

```
hw: hw.o helper.o
    gcc -o hw hw.o helper.o -lm

hw.o: hw.c
    gcc -O -Wall -c hw.c

helper.o: helper.c
    gcc -O -Wall -c helper.c

clean:
    rm -f hw.o helper.o hw
```

Makefile 基于规则，这些规则决定需要发生的事情。规则的一般形式是：

```
target: prerequisite1 prerequisite2 ...
    command1
    command2
    ...
```

target（目标）通常是程序生成的文件的名称。目标的例子是可执行文件或目标文件。目标也可以是要执行的操作的名称，例如在我们的示例中为“clean”。

prerequisite（先决条件）是用于生成目标的输入文件。目标通常依赖于几个文件。例如，要构建可执行文件 hw，需要首先构建两个目标文件：hw.o 和 helper.o。

最后，command（命令）是一个执行的动作。一条规则可能有多个命令，每个命令都在自己的行上。重要提示：必须在每个命令行的开头放一个制表符！如果你只放空格，make 会打印出一些含糊的错误信息并退出。

通常，命令在具有先决条件的规则中，如果任何先决条件发生更改，就要重新创建目标文件。但是，为目标指定命令的规则不需要先决条件。例如，包含 delete 命令、与目标“clean”相关的规则中，没有先决条件。

回到我们的例子，在执行 make 时，大致工作如下：首先，看到目标 hw，并且意识到要构建它，它必须具备两个先决条件，hw.o 和 helper.o。因此，hw 依赖于这两个目标文件。然后，Make 将检查每个目标。在检查 hw.o 时，看到它取决于 hw.c。这是关键：如果 hw.c 最近被修改，但 hw.o 没有被创建，make 会知道 hw.o 已经过时，应该重新生成。在这种情况下，会执行命令行 gcc -O -Wall -c hw.c，生成 hw.o。因此，如果你正在编译大型程序，make 会知道哪些目标文件需要根据其依赖项重新生成，并且只会执行必要的工作来重新创建可执行文件。另外请注意，如果 hw.o 根本不存在，也会被创建。

继续，基于上面定义的相同标准，helper.o 也会重新生成或创建。当两个目标文件都已创建时，make 现在可以执行命令来创建最终的可执行文件，然后返回并执行以下操作：gcc -o hw hw.o helper.o -lm。

到目前为止，我们一直没提 makefile 中的 clean 目标。

要使用它，必须明确提出要求，键入以下代码：

```
prompt> make clean
```

这会在命令行上执行该命令。因为 clean 目标没有先决条件，所以输入 make clean 将导致命令被执行。在这种情况下，clean 目标用于删除目标文件和可执行文件，如果你希望从头开始重建整个程序，就非常方便。

现在你可能会想，“好吧，这似乎没问题，但这些 makefile 确实很麻烦！”你说得对——如果它们总是这样写的话。幸运的是，有很多快捷方式，让使用更容易。例如，这个 makefile 具有相同的功能，但用起来更好：

```
# specify all source files here
SRCS = hw.c helper.c
# specify target here (name of executable)
TARG = hw
# specify compiler, compile flags, and needed libs
CC = gcc
OPTS = -Wall -O
LIBS = -lm

# this translates .c files in src list to .o's
OBJS = $(SRCS:.c=.o)

# all is not really needed, but is used to generate the target
all: $(TARG)

# this generates the target executable
$(TARG): $(OBJS)
    $(CC) -o $(TARG) $(OBJS) $(LIBS)

# this is a generic rule for .o files
%.o: %.c
    $(CC) $(OPTS) -c $< -o $@

# and finally, a clean line
clean:
    rm -f $(OBJS) $(TARG)
```

虽然我们不会详细介绍 make 语法，但如你所见，这个 makefile 可以让生活更轻松一些。例如，它允许你轻松地将新的源文件添加到构建中，只需将它们加入 makefile 顶部的 SRCS 变量即可。你还可以通过更改 TARG 行轻松更改可执行文件的名称，并且可以轻松修改指定编译器，标志和库。

关于 make 的最后一句话：找出目标的先决条件并非总是很容易，特别是在大型复杂程序中。毫不奇怪，有另一种工具可以帮助解决这个问题，称为 makedepend。自己阅读它，看看是否可以将它合并到一个 makefile 中。

E.7 调试

最后，在创建了良好的构建环境和正确编译的程序之后，你可能会发现程序有问题。

解决问题的一种方法是认真思考——这种方法有时会成功，但往往不会。问题是缺乏信息。你只是不知道程序中到底发生了什么，因此无法弄清楚为什么它没有按预期运行。幸运的是，有某种帮助工具：gdb，GNU 调试器。

将以下错误代码保存在 buggy.c 文件中，然后编译成可执行文件。

```
#include <stdio.h>

struct Data {
    int x;
};

int
main(int argc, char *argv[])
{
    struct Data *p = NULL;
    printf("%d\n", p->x);
}
```

在这个例子中，主程序在变量 p 为 NULL 时引用它，这将导致分段错误。当然，这个问题应该很容易通过检查来解决，但在更复杂的程序中，找到这样的问题并非总是那么容易。

要为调试会话做好准备，请重新编译程序，并确保将-g 标志加入每个编译行。这让可执行文件包含额外的调试信息，这些信息在调试会话期间非常有用。另外，不要打开优化（-O）。尽管这可能也行，但在调试过程中也可能导致困扰。

使用-g 重新编译后，你就可以使用调试器了。在命令提示符处启动 gdb，如下所示：

```
prompt> gdb buggy
```

这让你进入与调试器的交互式会话。请注意，你还可以使用调试器来检查在错误运行期间生成的“核心”文件，或者连上已在运行的程序。阅读文档以了解更多相关信息。

进入调试器后，你可能会看到以下内容：

```
prompt> gdb buggy
GNU gdb ...
Copyright 2008 Free Software Foundation, Inc.
(gdb)
```

你可能想要做的第一件事就是继续运行程序。这只需在 gdb 命令提示符下输入 run。在这个例子中，你可能会看到：

```
(gdb) run
Starting program: buggy

Program received signal SIGSEGV, Segmentation fault.
0x8048433 in main (argc=1, argv=0xbffff844) at buggy.cc:19
19          printf("%d\n", p->x);
```

从示例中可以看出，在这种情况下，gdb 会立即指出问题发生的位置。在我们尝试引用 p 的行中产生了“分段错误”。这就意味着我们访问了一些我们不应该访问的内存。这时，

精明的程序员可以检查代码，然后说“啊哈！肯定是 p 没有指向任何有效的地址，因此不应该引用!”，然后继续修复该问题。

但是，如果你不知道发生了什么，可能想要检查一些变量。gdb 允许你在调试会话期间以交互方式执行此操作。

```
(gdb) print p
1 = (Data *) 0x0
```

通过使用 print 原语，我们可以检查 p，并看到它是指向 Data 类型结构的指针，并且它当前设置为 NULL（即零，即十六进制零，此处显示为“0x0”）。

最后，你还可以在程序中设置断点，让调试器在某个函数中停止程序。执行此操作后，单步执行（一次一行），看看发生了什么，这通常很有用。

```
(gdb) break main
Breakpoint 1 at 0x8048426: file buggy.cc, line 17.
(gdb) run
Starting program: /homes/hacker/buggy

Breakpoint 1, main (argc=1, argv=0xbffff844) at buggy.cc:17
17           struct Data *p = NULL;
(gdb) next
19           printf("%d\n", p->x);
(gdb)

Program received signal SIGSEGV, Segmentation fault.
0x8048433 in main (argc=1, argv=0xbffff844) at buggy.cc:19
19          printf("%d\n", p->x);
```

在上面的例子中，在 main()函数中设置了断点。因此，当我们运行程序时，调试器几乎立即停止在 main 执行。在示例中的该点处，发出“next”命令，它将执行下一行源代码级指令。“next”和“step”都是继续执行程序的有用方法——在文档中阅读，以获取更多详细信息[①]。

这里的讨论真的对 gdb 不公平，它是丰富而灵活的调试工具，有许多功能，而不只是这里有限篇幅中描述的功能。在闲暇之余阅读更多相关信息，你将成为一名专家。

E.8 文档

要了解有关所有这些事情的更多信息，你必须做两件事：第一是使用这些工具；第二是自己阅读更多相关信息。了解更多关于 gcc、gmake 和 gdb 的一种方法是阅读它们的手册页。在命令提示符下输入 man gcc、man gmake 或 man gdb。你还可以使用 man -k 在手册页中搜索关键字，但这并非总如人意。谷歌搜索可能是更好的方法。

关于手册页有一个棘手的事情：如果有多个名为×××的东西，输入 man ×××可能

① 特别是，你可以在使用 gdb 进行调试时使用交互式的“help”命令。

不会得到你想要的东西。例如，如果你正在寻找 kill()系统调用手册页，如果只是在提示符下键入 man kill，会得到错误的手册页，因为有一个名为 kill 的命令行程序。手册页分为几个部分（section），默认情况下，man 将返回找到的最低层部分的手册页，在本例中为第 1 部分。请注意，你可以通过查看页面的顶部来确定你看到的手册页：如果看到 kill（2），就知道你在第 2 节的正确手册页中，这里放的是系统调用。有关手册页的每个不同部分中存储的内容，请键入 man man 以了解更多信息。另外请注意，man -a kill 可用于遍历名为“kill”的所有手册页。

手册页对于查找许多内容非常有用。特别是，你经常需要查找要传递给库调用的参数，或者需要包含哪些头文件才能使用库调用。所有这些都在手册页中提供。例如，如果查找 open()系统调用，你会看到：

```
SYNOPSIS
     #include <sys/types.h>
     #include <sys/stat.h>
     #include <fcntl.h>

     int open(const char *path, int oflag, /* mode_t mode */...);
```

这告诉你包含头文件 sys/types.h、sys/stat.h 和 fcntl.h，以便使用 open 调用。它还告诉你要传递给 open 的参数，即名为 path 的字符串和整数标志 oflag，以及指定文件模式的可选参数。如果你需要链接某个库以使用该调用，这里也会告诉你。

需要一些努力才能有效使用手册页。它们通常分为许多标准部分。主体将描述如何传递不同的参数，以使函数具有不同的行为。

一个特别有用的部分是手册页的 RETURN VALUES 部分，它告诉你成功或失败时函数将返回什么。再次引用 open()的手册页：

```
RETURN VALUES
     Upon successful completion, the open() function opens the
     file and return a non-negative integer representing the
     lowest numbered unused file descriptor. Otherwise, -1 is
     returned, errno is set to indicate the error, and no files
     are created or modified.
```

因此，通过检查 open 的返回值，你可以看到是否成功打开。如果没有，open（以及许多标准库函数）会将一个名为 errno 的全局变量设置为一个值，来告诉你错误。有关更多详细信息，请参见手册页的 ERRORS 部分。

你可能还想做一件事，即查找未在手册页本身中指定的结构的定义。例如，gettimeofday()的手册页有以下概要：

```
SYNOPSIS
     #include <sys/time.h>
     int gettimeofday(struct timeval *restrict tp,
                      void *restrict tzp);
```

在这个页面中，你可以看到时间被放入 timeval 类型的结构中，但是手册页可能不会告诉你这个结构有哪些字段！（在这个例子中，它包含在内，但你可能并非总是如此幸运）因

此，你可能不得不寻找它。所有包含文件都位于/usr/include 目录下，因此你可以用 grep 这样的工具来查找它。例如，你可以键入：

```
prompt> grep 'struct timeval' /usr/include/sys/*.h
```

这让你在/usr/include/sys 中以.h 结尾的所有文件中查找该结构的定义。遗憾的是，这可能不一定有效，因为包含文件可能包括在别处的其他文件。

更好的方法是使用你可以使用的工具，即编译器。编写一个包含头文件 time.h 的程序，假设名为 main.c。然后，使用编译器调用预处理器，而不是编译它。预处理器处理文件中的所有指令，例如#define 指令和#include 指令。为此，请键入 gcc -E main.c。结果是一个 C 文件，其中包含所有需要的结构和原型，包括 timeval 结构的定义。

可能还有找到这些东西的更好方法：google。你应该总是 google 那些你不了解的东西——只要通过查找就可以学到很多东西，这令人惊奇！

info 页面

info 页面在寻找文档方面也非常有用，它为许多 GNU 工具提供了更详细的文档。你可以通过运行 info 程序或通过 emacs（黑客的首选编辑器）执行 Meta-x info 来访问 info 页面。像 gcc 这样的程序有数百个标志，其中一些标志非常有用。gmake 还有许多功能可以改善你的构建环境。最后，gdb 是一个非常复杂的调试器。阅读 man 和 info 页面，尝试以前没有尝试过的功能，成为编程工具的强大用户。

E.9 推荐阅读

除了 man 和 info 页面之外，还有许多有用的书籍。请注意，许多此类信息可在线免费获取，然而，有时书本形式的东西似乎更容易学习。另外，总是在 O'Reilly 书籍中寻找你感兴趣的主题，它们几乎总是高品质的。

Brian Kernighan 和 Dennis Ritchie 编写的《The C Programming Language》，是最权威的 C 语言图书。

Andrew Oram 和 Steve Talbott 编写的《Managing Projects with make》。关于 make 的价格公道的小书。

Richard M. Stallman 和 Roland H. Pesch 编写的《Debugging with GDB: The GNU Source-Level Debugger》。关于使用 GDB 的一本小书。

W. Richard Stevens 和 Steve Rago 编写的《Advanced Programming in the UNIX Environment》。Stevens 写了一些优秀的图书，这是 UNIX 黑客必读的书。他还有一套关于 TCP/IP 和套接字编程的好书。

Peter Van der Linden 编写的《Expert C Programming》。上面关于编译器等的许多有用提示，都直接来自这里。读这本书！ 虽然有点过时，但这本书很精彩，令人大开眼界。