笨办法学 C 语言
一本清晰直观的现代 *C* 语言编程介绍

Zed A. Shaw

July 2011

Contents

译者前言

本书作者尚未完成，翻译也在进行中。原书的错漏不少，翻译的错漏也许更多，但是我们将努力保证成品的质量。请通过 翻译代码库关注我们的进度。

欢迎提出宝贵意见和建议！

作者前言

这本书目前还只是一个粗糙的半成品。里边的句法可能不规整，有的章节还没写完，不过你可以看到并且了解到我写书的过程。

你可以加入本书的邮件组 lcthw@librelist.com，我会在那里发布本书的新内容，你有问题或者遇到困难的话也可以在那里提问。

该邮件组是一个讨论组，而不是公告组。你可以在里边讨论本书并提出问题。

最后别忘了我还写了一本 Learn Python The Hard Way, 2nd Edition，如果你没有编程经验的话，那你应该先读一下这本书。LCTHW 的阅读对象不是初学者，而是至少读过 LPTHW 或者学过一门其他语言的人。

引言：笛卡尔的 C 语言之梦

一切迄今我以为最接近于“真实”的东西都来自感觉和对感觉的传达。但是，我发现，这些东西常常欺骗我们。因此，唯一明智的是：再也不完全信赖那些哪怕仅仅欺骗过我们一次的东西。

(笛卡尔《第一哲学沉思集》)

如果一定要找一段名人名言来描述 C 语言编程的话，那就是这一段了。对于很多程序员来说，这让 C 语言显得恐怖和邪恶。C 语言就是恶魔、撒旦、捣乱的洛基[1]，他用指针迷惑你，给你直接访问硬件的权力，从而毁掉你的工作效率。然后，一旦这位路西法大人吸引你上了钩，他就会用邪恶的“segfault（段错误）”毁掉你的世界，并在邪恶的笑声中让你知道你最终栽在了跟魔鬼的交易里。

但是，造成这一切并不是 C 语言的过错。不，我的朋友，真正捣蛋的恶魔其实是你的计算机和操作系统。它们对你隐瞒了真实的内部工作原理，所以你从不曾真正了解到究竟发生了什么。而 C 语言唯一失败的地方，就是带你去了那些你看不到的地方，告诉你冷酷无情的真相。C 语言给了你一颗红药丸[2]，C 语言展示给你幕后的巫师。*C 语言就是真相*。

既然 C 语言如此危险，我们依然要用 C 呢？因为 C 能让你战胜抽象带来的虚假现实，并从愚蠢中将你解放出来。

你将学到的东西

本书的目的是让你掌握足够的 C 语言，从而可以写自己的软件，或者修改别人的代码。本书的结尾部分我们将从一本更著名的叫做 *"K&R C"* 的书中节选一些代码出来，并让你用自己学到的东西进行代码审查 (code review)。要达到这一程度，你需要学习一些东西：

1. C 语言的基本语法和术语。
2. 编译、make 文件、连接器。
3. 找出 Bug 并防止 Bug 产生。
4. 防御性编程。
5. 将 C 代码改坏。
6. 编写基础的 Unix 系统程序。

完成本书的最后一章后，你应该已经拥有了足够的弹药来应对基本的系统软件、库文件、以及小规模项目的撰写了。

[1]译注：洛基是北欧神话里的火神，以惹是生非闻名。撒旦和路西法都是恶魔的名称

[2]译注：《黑客帝国》典故，选择红药丸让主角层层深入探求真相，蓝药丸让主角继续醉生梦死。

如何阅读本书

本书是写给有至少学过一门编程语言的程序员的，如果你还没学过编程，我推荐你去看 Learn Python The Hard Way 或者 Learn Ruby The Hard Way。这两本书是为初学者写的，而且效果不错。学完其中一本后，你就可以回来对这本书开始下手了。

对于已经学过编程的人来说，本书乍看上去似乎有些怪。别的书一般都是让你一段接着一段地读下去，然后断断续续地写一点代码，这本书完全不同。取而代之的是我要求你立即输入代码，然后我将告诉你刚才做了什么。解释你体验过的事情更容易一些，因此这种方式相对更好一些。

正因为本书结构这样特殊，你在看书时必须遵守一些规矩：

1. 键入所有的代码，禁止复制粘贴！
2. 一字不差地键入代码，即使是注释也一样。
3. 让程序运行起来，保证输出都和本书的相同。
4. 如果有 bug 就将其改正。
5. 做加分习题，有不会的也没关系，跳过就好了。
6. 在求助之前一定要确认你已经尽力了。

如果你遵从这些规则，完成了书里的所有作业，还是不会用 C 语言编程，那也没关系，因为你至少尝试过了。C 语言并不适合每个人，但尝试的过程也会让你变成一个更好的程序员。

关键技能

我猜你是从一门菜鸟语言来到这里的。[3]要么你来自像 Python 或者 Ruby 这样“还算能用”的语言，这些语言让思维不清半吊子瞎鼓捣的你也能写出能运行的程序来。要么你用过 Lisp 这样的语言，这些语言假装计算机是某个纯函数的仙境，四周还装了五彩的婴儿墙。也许你学过 Prolog，因而认为整个世界应该只是一个供你上下求索的数据库。更糟糕的还在后面呢，我打赌你还用过某个 IDE，所以你的脑子充满里记忆空洞，如果你不是每打三个字母就按一次 CTRL-SPACE 的话，我怕你连一个完整的函数名称都敲不出来。

不管背景如何，你都可能有四样技能有待提高：

读写能力 如果你平时使用 IDE 的话，这一点就尤为正确。不过大体来说我发现程序员略读的时候太多了，从而导致理解性阅读能力有些问题。他们将代码扫视一遍就觉得自己读懂了，其实不然。其他的语言还提供了各种工具，从而避免让程序员直接撰写代码，所以一旦面对 C 语言这样的东西时，他们就立马崩溃了。最简单的办法就是要理解每个人都有这样的问题，解决方案就是强迫自己慢下来，倍加细致地去读写代码。一开始你也许会觉着很痛苦很烦躁，那就增加自己休息的频率，最后你会觉得这其实也很容易做到。

关注细节 这方面没有人能做得好，这也是劣质软件的最大成因。其他的语言会让不够专注的你蒙混过关，但 C 语言却要求你完全聚精会神，因为 C 语言直接和计算机打交道，而计算机硬件又是极其挑剔的。在 C 的语境中没有“有点像”或是“差不多”这样的说法，所以你需要专注。反复检查你的工作。在证明正确之前，要先假设一切都可能是错的。

发现差异 从别的语言来的程序员有一个问题，就是他们的大脑已经被训练成可以发现那种语言中的差异，而不是 C 语言

[3]我只是在挑逗你而已，如果你能看得出来的话

中的差异。当你在对比你的代码和标准答案时，你的视线会直接跳过那写你认为不重要或不熟悉的部分。我给你的解决办法是：强迫自己观察自己的错误，如果你的代码跟标准答案不是*一字不差*，那它就是错的。

规划和除错 我喜欢其他更简单的语言，因为我可以胡搞乱来。我把想法打出来，然后就能直接在编译器里看到结果。这些语言可以让你很方便地尝试新的主意，但你有没有发现呢：如果你一直用“乱改直到能用”的方法写代码，到头来就是什么都不能用了。C 语言对你要求比较高，因为它要求你先计划好要创建的东西。当然你也可以偶尔瞎搞搞，但和其他语言相比，你需要在更早的阶段就开始认真做计划。在你写代码之前，我会教你如何规划程序的关键部分，希望这能同时使你成为一个更优秀的程序员。即使是微小的计划也能让你的后续工作更为顺利。

在学习 C 语言的过程中，你将被迫更早地、更多地应对这些问题，所以学习 C 语言更能让你成为一个好程序员。你不能思维不清半吊子地瞎鼓捣，否则什么都不会做出来。C 语言的优势是作为一门简单的语言，你可以自己把它弄明白，所以如果你要学习机器的工作原理，并增强这些关键的编程技能的话，C 语言是上佳的选择。

C 语言比一些其他的语言难学一些，但这只是因为 C 语言做到了不对你隐藏机器的工作原理，其他语言试图对工作原理进行模糊处理，这也是其他处理失败的地方。

协议

本书可以免费阅读，但在我完成修改之前，你不能进行分发或做任何修改。我得保证别把一本没完成的书传出去，以免不小心误人子弟。

Part I

基础技能

Chapter 1

习题 0: 准备工作

本章你将学会架设 C 语言编程的系统环境。如果你使用的是 Linux 或者 Mac OSX，那么有个好消息可以告诉你，你的系统本来就是为 C 语言编程设计的。C 语言的发明人也曾是 Unix 操作系统的创作者，而 Linux 和 OSX 都是基于 Unix 的操作系统。所以，整个架设过程会非常简单。

对于 Windows 的用户，我就只有坏消息了：在 Windows 下学习 C 语言是一件痛苦的事情。在 Windows 下写 C 语言代码不是问题，问题是 Windows 下所有的库、函数、以及工具和 Unix 下的 C 语言环境比起来就是有那么一点“跑偏”的感觉。恐怕这也是你不得不接受的事实了。

你也别被这条突如其来的坏消息吓到。我不是说要完全避开 Windows，我的意思是说，如果你想最不费力地学习 C 语言，那么你最好还是从 Unix 下手。了解一点 Unix 还有另外一个好处，那就是你可以学到一些 C 语言的惯用技巧，从而扩展你的编程技术。

这也意味着你将会用到命令行。没错，我是说命令行。你需要在命令行输入命令。不过别害怕，我会告诉你该输入什么命令，以及执行命令会有什么样的结果，这样你同时也会学到不少让你大开眼界的技能。

1.1 Linux

对于大部分 Linux 系统来说，你只需要安装若干软件包就可以了。在基于 Debian 的系统（例如 Ubuntu）上面，你只要使用下面的命令安装即可：

在 Ubuntu 上面安装需求软件包

```
$ sudo aptitude install build-essential
```

以上是一个命令行的例子，所以你要先找到系统里的“命令行终端（terminal）”并且把它运行起来，才能执行上述的命令。你会看到一个类似上面提到的’$’ 的提示界面，然后键入上述命令。 *’$’ 这个符号是无需键入的，只要把后面的内容键入即可。*

以下是基于 RPM 的 Linux 发行版要做的准备工作，以 Fedora 为例：

在 Fedora 上面安装需求软件包

```
$ su -c yum groupinstall development-tools
```

执行完上述指令后，你应该可以顺利完成第一个习题了。如果不行，请向作者反馈。

1.2 Mac OSX

Mac OSX 上面的安装就更简单了。首先你需要从 Apple 下载最新版的 *XCode*，或者从你的安装 DVD 中找出来安装也可以。文件很大，可能你一辈子都下不下来，所以我还是建议你从 DVD 安装好了。另外，你可以上网搜索一下“安装 xcode”，找点说明来看看。

装完 XCode 可能需要重启电脑。一切完成以后，你可以找到命令行终端 (Terminal) 程序并把它放到 dock 中。本书里会大量用到命令行终端，所以还是把它放到方便的地方比较好。

1.3 Windows

对于 Windows 用户来说，我就教教你们怎样在虚拟机里安装并运行 Ubuntu Linux 吧。这样你可以有一个做本书习题的环境，但也省得面对各种痛苦的 Linux 安装问题了。

【本节内容尚未完成】... have to figure this one out.

1.4 文本编辑器

编辑器的选择对于程序员来说总是个难题。对于初学者来说我跟他们讲用 Gedit 就可以了，Gedit 功能简单，对代码支持也不错，不过 Gedit 在某些非英语环境下会有问题，而且如果你已经写过一阵子程序的话，你没准已经有自己喜欢的编辑器了。

在此前提下，我要求你试着用用几种支持你的系统平台的代码编辑器，然后选一个你最喜欢的坚持用下去。如果你喜欢 GEdit 就继续用 GEdit，如果你想要换个不一样的，那就简单地试试别的编辑器，然后从中选一个就行了。

最重要的一点是*不要纠结于挑选一个完美的编辑器*。文本编辑器总有各种奇怪的不好用之处。选一个然后用下去就行了。如果你看到别的喜欢的编辑器，就拿来试试。别没玩没了花时间去配置打造那所谓的完美编辑器。

以下是你要去试试的编辑器：

1. Linux 和 OSX 下的 Gedit。
2. OSX 下的 TextWrangler。
3. Nano，这是一个命令行终端下的编辑器，基本任何地方都支持。
4. Emacs 以及 Emacs for OSX，不过要做好学习的准备。

5. Vim 和 MacVim。

编辑器的种类实在太多了，给大街上的人们每人分一个大概都够，不过上面列出的只是我确定可以用的。试试其它我没提到的编辑器，收费版的也可以试试，直到找到你喜欢的为止。

1.4.1 警告：不要使用 IDE

IDE，也就是“集成开发环境（Integrated Development Environment）”，它会让你变傻。如果你想要成为一个好程序员，那么 IDE 就是最坏的工具。因为 IDE 会为你隐藏真正进行中的事情，而你的任务正是知道真正发生了什么。如果你想要完成某件任务，而该任务的系统平台又是围绕这个 IDE 设计的，这种情况下 IDE 还是有用的。不过对于学习 C 以及很多其他语言来说，IDE 是毫无意义的。

Note 1 *IDE 和吉他谱*

玩过吉他的人都知道吉他谱是什么东西，不过还是让我给其他没玩过吉他的人解释一下吧。有一种约定俗成的音乐记谱方式叫做“线谱”，这是一种普遍的，古老的，通用的记录如何演奏乐器的方法。线谱很大程度上是为钢琴和作曲家而生，所以如果你弹钢琴的话，线谱是很容易使用的。

然而吉他这种乐器有些古怪，它并不适合这种记谱方式，所以演奏吉他的人使用了一种另类的记谱方式，称作“吉他谱（tablature）”。吉他谱告诉你的不是要演奏的音调，而是你在某一时刻要弹的指位和琴弦。你可以在不了解任何曲调的情况下学会弹奏一首曲子，很多人也是这么去学的。然而如果你想从中读出你弹奏的曲调，吉他谱就没什么用处了。

传统的记谱方式也许比吉他谱难学，但它可以告诉你如何演奏音乐，而不仅仅是如何弹吉他。拿着一份线谱，我可以走到一架钢琴前面弹出同样的一首歌曲，我可以用贝司把它弹出来，我还可以把它输入到计算机中重新设计整份乐谱。然而拿着吉他谱，我就只能用它弹弹吉他。

IDE 和吉他谱类似。毫无疑问你可以使用 IDE 快速地写出代码，但你只能在一个固定的平台上使用一种特定的语言。这也是公司企业喜欢兜售这些东西给你的原因。他们知道你是个懒人，而 IDE 只在他们的平台上面工作，就这样，由于你的懒惰，他们就把你禁锢在他们的平台上了。

打破这个循环的方法也不是没有，你需要卧薪尝胆，最终学会如何不使用 IDE 进行编程。简单的文本编辑器，或者像 Vim 和 Emacs 这样的程序员编辑器，会让代码真正成为你的工作对象。比起使用 IDE 来这样会更难一些，不过最终的结果就是你可以应对任何代码，不管它在什么样的计算机平台上，不管它使用的是什么语言，而且你懂它的深层原理。

Chapter 2

习题 1：重拾编译器

这是你用 C 语言写的第一个简单程序：

ex1.c

```c
int main()
{
    puts("Hello world.");

    return 0;
}
```

把它保存为 `ex1.c` 然后输入：

Build ex1

```
$ make ex1
cc     ex1.c   -o ex1
```

你的系统使用的命令有可能会不大一样，但最终应该都会编译出一个可运行文件叫 `ex1`

2.1 你应该看到的结果

你运行这个程序应该会看到以下输出。

运行 *ex1*

```
$ ./ex1
Hello world.
```

如果输出不同，你需要回到前一步，找出问题所在并修正它。

2.2 让程序出错

在本书中，我为每段程序都准备了这样一个小章节，在这个小章节里，将让你对程序做一些非常规的事情，用一些怪异的方法运行程序或者改变代码，以此而造成程序出错或编译器报错。

对于这个程序，打开编译器的所有警告项，重新编译它：

Build ex1，使用 *-Wall* 参数

```
$ rm ex1
$ CFLAGS="-Wall" make ex1
cc -Wall    ex1.c   -o ex1
ex1.c: In function 'main':
ex1.c:3: warning: implicit declaration of function 'puts'
$ ./ex1
Hello world.
$
```

现在你得到了一个警告说函数“puts”是隐式申明（implictly declared）的。C 编译器足够聪明，它可以够猜出你想要什么，但是你应该尽力消除所有的编译器警告。对于这个警告，你可以把以下几行加入到 `ex1.c` 的顶部，并重新编译：

```
#include <stdio.h>
```

现在，按前面的步骤再做一次，你会发现警告消失了。

2.3 加分习题

1. 用你的文本编辑器打开 `ex1` 改变或者删除任意部分。试试运行它，看会发生什么。
2. 再多打印 5 行的文字或者比“hello world”复杂一些的句子。
3. 运行 `man 3 puts` 阅读一下关于这个函数的内容以及别的内容。

Chapter 3

习题 2: ”Make” 是你的 Python 了

在 Python 中你只用键入python 和你希望运行的代码文件就能城启动脚本。Python 解释器 (interpreter) 会直接运行他们，并实时载入任何其他你可能需要的库 (libraries)。C 相对来说是只完全不同的怪兽，在 C 中你需要编译 (*compile*) 源文件并将它们拼凑成一个可以自己运行的二进制程序。这个过程若要手动完成的话那就是个悲剧。在上个习题中，你是通过直接运行 `make` 来实现的。

在本章习题是一节 GUN make 的快速入门课，当然你也会在往后的 C 语言学习过程中继续学习如何使用它。在这本书剩下的章节中，Make 将担任你的”Python 解释器”。它将帮助你 build 代码、测试代码、完成各种准备工作，以及为你做所有 Python 程序通常能做的事。

当然 make 和 python 还是有区别的，后面我还会向你展示更加智能化的 Makefile 魔法——生成程序时你将不需要指定每一条愚蠢的细节。这些内容不会出现在本章中，不过等你用了一段时间婴儿级别的 make 之后，大师级别的 make 就会闪亮登场了。

3.1 使用 Make

有些程序 make 已经知道如何 build 了，使用 make 的第一阶段就是去 build 这样的程序。Make 已经拥有了数十年积累的丰富相关知识，用以将一些文件 build 出各种其他文件。在上一个练习里你已经使用了像这样的命令：

Build ex1，使用 -Wall 参数

```
$ make ex1
# or this one too
$ CFLAGS="-Wall" make ex1
```

在第一个命令里你在告诉 make，“我希望创建一个叫做 ex1 的文件”，然后 make 就执行了以下任务：

1. `ex1` 这个文件是不是已经存在了？
2. 否。好吧，那么是否有另外一个文件以 `ex1` 开头呢?

3. 是，它叫 `ex1.c`。那么，我知道如何 build `.c` 的文件吗?
4. 是，我将运行 `cc ex1.c -o ex1` 来 build 它们。
5. 我将通过使用 `cc` 从 `ex1.c` 中 build `ex1`.

第二个命令是将修饰参数传递给 make 的一种方式。如果你不熟悉 Unix shell 是如何工作的，简单的解释就是你可以创建这些“环境变量 (environment variable)”，而你的程序会在运行中获取这些变量。取决于你使用的 shell，有时你可以通过运行 `export CFLAGS="-Wall"` 这样的命令来创建环境变量。你也可以直接将它们放在你想要运行的命令之前，这样的话，变量将只在当前命令中生效。

在这个例子中我使用了 `CFLAGS="-Wall" make ex1` ，结果就是 shell 将命令行选项 `-Wall` 加到 *make* 会运行的 `cc` 这个命令中。该命令行选项将让编译器 (compiler) *cc* 报告所有警告信息。（由于命运的纠结，其实它不能报告出所有可能的警告信息。）

事实上单单使用 *make* 你就已经可以做很多事情了，但还是让我们来看一看怎样写一个 **Makefile** ，这会帮你更好的理解它。请创建一个文件并输入以下内容：

一个简单的 *Makefile*

```
CFLAGS=-Wall -g

clean:
        rm -f ex1
```

把它命名为 **Makefile** 并保存到当前目录中。make 这个命令会自动假设当前目录存在一个叫做 **Makefile** 的文件，并且会直接运行它。同时，*警告: 确保你只输入了制表符 (TAB)，而非制表符和空格的混合。*

这个 **Makefile** 将给你展现一些 make 的新玩意儿。首先我们在里边设定好 *CFLAGS*，这样我们就无需在别处再次设定了，同样，我们还添加了 -g 这个实现调试功能的参数。然后我们使用了 *clean* 这一小节，它的作用是为你的小项目清理文件。

先确保它在与文件 `ex1.c` 在相同路径下, 然后运行以下命令：

运行简单的 *Makefile*

```
$ make clean
$ make ex1
```

3.2 你应该看到的结果

如果一切正常你将可以看到：

使用 *Makefile* 做完整的 *build*

```
$ make clean
rm -f ex1
$ make ex1
cc -Wall -g    ex1.c   -o ex1
ex1.c: In function 'main':
ex1.c:3: warning: implicit declaration of function 'puts'
$
```

这里你可以看到我通过运行 `make clean` 来实现我们的目标 *clean*。现在再来看看 Makefile，你会发现我在文件中写入了想要 *make* 去帮我运行的 shell 命令。在这里你可以想放多少命令就放多少命令，所以它还是个不错的自动化工具。

Note 2 你修正 *ex1.c* 了吗？

如果你为 `ex1.c` 添加了 `#include <stdio.h>`，那么你的输出应该不会有关于 puts 的警告信息（虽然它显示为警告信息 warning，其实说它是个错误信息 error 更合适）。我得到了警告是因为我并没有修复它。

同时还需要注意的是，尽管我们没有在 `Makefile` 中提到 `ex1` ，*make* 依然知道如何去 build 它并使用了我们的特殊设定。

3.3 让程序出错

以上这些应该已经够你起步了，但我们还是将这个 make 文件用某种方式打乱一下，这样你可以进一步了解到底发生了什么。请选择 `rm -f ex1` 这一行并取消缩进 (将整行左移)，你将得到：

运行 *make* 出错

```
$ make clean
Makefile:4: *** missing separator.  Stop.
```

一定要记得缩进，如果你看到了类似的奇怪错误，那就要检查一下你在是不是统一使用了制表符。要知道有些类型的 make 的是非常挑剔的。

3.4 加分习题

1. 创建一个 `all: ex1` 的目标，然后仅通过命令 `make` 来 build 它。
2. 阅读 `man make` 来获取更多的使用说明。
3. 阅读 `man cc`，找出更多关于 `-Wall` 和 `-g` 这两个 flag 功能的说明.

4. 上网研究一下 Makefile，看看还能不能改进你写的 makefile。
5. 在其他 C 语言项目中找一个 `Makefile` 并尝试了解一下它的功能。

Chapter 4

Exercise 3: 格式化打印输出

请保留 **Makefile** 文件，因为它可以帮你找到错误，而且将来我们要把更多任务自动化时，我们还会往里面添加内容。

许多编程语言使用 C 风格的格式化输出, 让我们来试一下吧:

ex3.c

```c
#include <stdio.h>

int main()
{
    int age = 10;
    int height = 72;

    printf("I am %d years old.\n", age);
    printf("I am %d inches tall.\n", height);

    return 0;
}
```

完成之后运行 *make ex3* 去 build 并运行它. 确保你解决了所有警告信息。

这些练习里只有很少的代码，下面就来详解一下：

1. 首先引入另一个叫做 **stdio.h** 的 “头文件（header file）”。这样做就是告诉编译器你要使用” 标准输入输出函数 (standard Input/Output functions)”。*printf* 就是这些函数之一。
2. 接着你用一个叫做 *age* 的变量并且给它赋值为 10。
3. 下一步你给变量 *height* 赋值为 72。
4. 然后你用 *printf* 函数打印了这个星球上个子最高的 10 岁小孩的年龄和身高。
5. 在 *printf* 函数中你会注意到你传进了一个字符串，这个字符串和别的语言中的格式化字符串很像。

6. 在这个格式化字符串的后面，你放置了一些变量，这些变量是用以“取代” *printf* 中格式化字符串的。

这么做的结果就是你把几个变量交给 *printf*，然后它构造出一个新的字符串并且在终端中打印这个新的字符串。

4.1 你应该看到的结果

当你做完了整个 build，你会看到类似如下的结果：

Build 并运行 ex3.c

```
$ make ex3
cc -Wall -g    ex3.c   -o ex3
$ ./ex3
I am 10 years old.
I am 72 inches tall.
$
```

很快我就不会再告诉你让你运行 `make` 、查看 build 信息这些事情了，所以请确保你现在能把这些东西弄对，确保没有差错。

4.2 使用外部资源做研究

在每个练习的*加分习题*部分，我也许会让你亲自去查找一些信息并弄明白它们。要成为一名自食其力的程序员，这一点是很重要的。如果你老是自己不尝试解决就跑去别人那问问题，你就永远也学不会如何独立的解决问题。这就导致你永远树立不了技术方面的信心，而且你总需要有人在身旁才能做自己的工作。

打破这种习惯的方法就是*强制*你自己先回答自己的问题，然后确认你是对的。寻找答案的方法可以是试验着让东西出错，用你认为可能的答案去做实验，以及自己学习专研。

这个练习中我希望你能自己上网，找出*所有 printf* 的转义符和修饰符。转义符是 `\n` 和 `\t` 这类的，它们可以（分别）让你打印出一个空行和制表符. 修饰符是 `%s` 和 `%d` 这类的，它们可以让你打印出一个字符串和一个整形数。找出所有可能的符号，学习它们的用法，以及你可以控制它们实现什么样的“精度”和显示宽度。

从现在起，这样的任务就会出现在你的加分习题中，这些练习是你应该做的。

4.3 让程序出错

试试以下这些方法来让你的程序出错，也许这些方法会让程序崩溃，不过也不一定：

1. 在第一次调用 *printf* 时，删掉 *age* 变量然后重新编译。应该会收到几条警告信息。
2. 运行这个程序的话，你的程序可能会报错，或者输出一个相当不合理的年龄值。
3. 将 *printf* 改回原样，但不要给 *age* 变量赋值，取而代之的是将这行改为 *int age;* 然后重新 build 并运行.

让 *ex3.c* 出错

```
# edit ex3.c to break printf
$ make ex3
cc -Wall -g    ex3.c   -o ex3
ex3.c: In function 'main':
ex3.c:8: warning: too few arguments for format
ex3.c:5: warning: unused variable 'age'
$ ./ex3
I am -919092456 years old.
I am 72 inches tall.
# edit ex3.c again to fix printf, but don't init age
$ make ex3
cc -Wall -g    ex3.c   -o ex3
ex3.c: In function 'main':
ex3.c:8: warning: 'age' is used uninitialized in this function
$ ./ex3
I am 0 years old.
I am 72 inches tall.
$
```

4.4 加分习题

1. 尽你所能的用别的方法去修改 `ex3.c` 让它出错。
2. 运行 `man 3 printf` 然后阅读其他带'%' 的修饰符，如果你在别的语言中使用过字符串修饰符，这会看起来很熟悉 (其他语言中的字符串修饰符就是从 *printf* 来的).
3. 把 `ex3` 加入到你的 `Makefile` 的 *all* 列表，使用这个 Makefile 运行 `make clean all` 来重新 build 你做过的所有习题。
4. 把 `ex3` 这个文件也加到 `Makefile` 的 *clean* 列表中。现在使用 `make clean` 将会把它也删除掉。

Chapter 5

习题 4: 介绍 Valgrind

是时候学习另外一个工具 *Valgrind* 了，它将伴随你学习 C 语言的整个过程。我现在介绍 *Valgrind* 给你，因为从现在起，在“让程序出错”这个小节里，每个练习你都将用到它。*Valgrind* 运行你的程序，然后报告你犯下的所有致命错误。它是一个很棒的自由软件，我经常在我写 C 代码的时候用到它。

还记得在上一个习题里，我让你修改你的代码移除了 *printf* 函数的参数吗？它打印出了一些不寻常的结果，但我没有告诉你为什么它会打印出这些结果。在这个习题里，我们将用 *Valgrind* 来探个究竟。

Note 3 为什么要介绍这些工具?

只经过几章，我们就学习了本书所需要的所有工具，而只学写了一点点代码。这是因为这本书的大部分读者不熟悉编译语言，当然也不知道自动化处理和有用的工具。让你马上接触 *make* 和 *Valgrind* ，我就能用他们更快的教会你 C 语言并帮助你找到你在程序中犯的错误。

在这章练习后，一段时间内，我们将不会介绍其他任何工具，大部分将是代码和语法。但我们还将会学习一些工具，用来查看程序如何运行和帮我们理解一些常见错误和问题。

5.1 安装 Valgrind

你能通过操作系统的包管理器来安装 *Valgrind*，但我要你学习如何从源代码安装程序。它包括以下几个步骤：

1. 下载源代码包。
2. 解压文件到你的电脑。
3. 运行 *./configure* 设置配置。
4. 运行 *make* 创建程序, 就像你以前做过的那样。
5. 运行 *sudo make install* 把程序安装到你的电脑。

下面是我流程的一个脚本，我要你试着用同样的方法做一遍。

ex4.sh

```
# 1) Download it (use wget if you don't have curl)
curl -O http://valgrind.org/downloads/valgrind-3.6.1.tar.bz2

# use md5sum to make sure it matches the one on the site
md5sum valgrind-3.6.1.tar.bz2

# 2) Unpack it.
tar -xjvf valgrind-3.6.1.tar.bz2

# cd into the newly created directory
cd valgrind-3.6.1

# 3) configure it
./configure

# 4) make it
make

# 5) install it (need root)
sudo make install
```

照着上面的做，不过如果你用了更新的 Valgrind 版本，就记得更新脚本里的版本号。如果它 build 失败，那么查出为什么出错。

5.2 使用 Valgrind

使用 *Valgrind* 很简单, 你只要运行 `valgrind theprogram` 它就会运行你的程序，然后打印出你程序运行时的错误。在这个习题里，我们将让程序出错，然后我们修正这个程序。

首先，我们把 `ex3.c` 的代码拿来用，换个名字叫 `ex4.c` ，当然，代码会故意弄错，为了练习，你需要重新输入一遍。

ex4.c

```c
#include <stdio.h>

/* Warning: This program is wrong on purpose. */

int main()
{
    int age = 10;
    int height;

    printf("I am %d years old.\n");
    printf("I am %d inches tall.\n", height);

    return 0;
}
```

你可以看到，除了我在上面犯了两个经典错误之外，其余都是一样的。

1. 我没有初始化 *height* 变量.
2. 我忘记给头一个 *printf* 函数加上 *age* 变量。

5.3 你应该看到的结果

现在我们像平常一样创建它，然后用 *Valgrind* 运行它，而不是像以前那样直接运行程序（看 Source: “Build 并用 Valgrind 运行 ex4.c”）：

Build 并用 Valgrind 运行 ex4.c

```
$ make ex4
cc -Wall -g    ex4.c   -o ex4
ex4.c: In function 'main':
ex4.c:10: warning: too few arguments for format
ex4.c:7: warning: unused variable 'age'
ex4.c:11: warning: 'height' is used uninitialized in this function
$ valgrind ./ex4
==3082== Memcheck, a memory error detector
==3082== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==3082== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
==3082== Command: ./ex4
==3082==
I am -16775432 years old.
==3082== Use of uninitialised value of size 8
==3082==    at 0x4E730EB: _itoa_word (_itoa.c:195)
==3082==    by 0x4E743D8: vfprintf (vfprintf.c:1613)
==3082==    by 0x4E7E6F9: printf (printf.c:35)
==3082==    by 0x40052B: main (ex4.c:11)
==3082==
==3082== Conditional jump or move depends on uninitialised value(s)
==3082==    at 0x4E730F5: _itoa_word (_itoa.c:195)
==3082==    by 0x4E743D8: vfprintf (vfprintf.c:1613)
==3082==    by 0x4E7E6F9: printf (printf.c:35)
==3082==    by 0x40052B: main (ex4.c:11)
```

```
25 ==3082==
26 ==3082== Conditional jump or move depends on uninitialised value(s)
27 ==3082==    at 0x4E7633B: vfprintf (vfprintf.c:1613)
28 ==3082==    by 0x4E7E6F9: printf (printf.c:35)
29 ==3082==    by 0x40052B: main (ex4.c:11)
30 ==3082==
31 ==3082== Conditional jump or move depends on uninitialised value(s)
32 ==3082==    at 0x4E744C6: vfprintf (vfprintf.c:1613)
33 ==3082==    by 0x4E7E6F9: printf (printf.c:35)
34 ==3082==    by 0x40052B: main (ex4.c:11)
35 ==3082==
36 I am 0 inches tall.
37 ==3082==
38 ==3082== HEAP SUMMARY:
39 ==3082==     in use at exit: 0 bytes in 0 blocks
40 ==3082==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
41 ==3082==
42 ==3082== All heap blocks were freed -- no leaks are possible
43 ==3082==
44 ==3082== For counts of detected and suppressed errors, rerun with: -v
45 ==3082== Use --track-origins=yes to see where uninitialised values come from
46 ==3082== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 4 from 4)
47 $
```

输出很长，因为 *Valgrind* 会精确告知你程序每一个错误的所在。你要逐行从头到尾的读一遍（最左边的是行号，这样你可以对照查看）：

1 你像平常一样用 `make ex4` build 程序。请确认你的 *cc* 指令编译时用了同样的参数，没有 -g 选项，*Valgrind* 的输出将不带行号。

2-6 可以注意到编译器也对你发出了警告，它提醒你"too few arguments for format". 那是你忘了加 *age* 变量。

7 用 `valgrind ./ex4` 运行你的程序。

8 然后 *Valgrind* 就抓狂了，丢了一堆错误出来:

> **14-18** 在 `main (ex4.c:11)` 这一行 (读作"文件 ex4.c 的第 11 行，main 函数里面) 你看到了"Use of uninitialised value of size 8"，你从错误中找到这条，然后在它下面就能看到所谓的"栈追踪 (stack trace)"。你看到的 (ex4.c:11) 这行是栈的最底下一层，如果你看不出哪里有问题那就继续往栈的上一层看，这回你要检查代码的 printf.c:35 这个位置。通常最底下的一行是最重要的（在这里是第 18 行）。
>
> **20-24** 这个接下来的错误依然是 main 函数 ex4.c:11 这行的错误 *Valgrind* 恨死这行了。这个错误是说某个 if 语句或者 while 循环是基于这个未被初始化的变量运行的，这次是 height 这个变量。
>
> **25-35** 剩下的错误基本是一样的，因为这个变量被重复用到了。

37-46 最后，程序退出然后 *Valgrind* 做了个概要，向你展示你的程序有多糟糕。

要学的东西真不少，不过你要用下面的方法来应对：

1. 每次运行你的 C 程序，都在 *Valgrind* 下重跑一遍检查一下。
2. 针对你看到的每一个错误，都到 source:line 所指的位置检查并修正它们。你可能需要上网搜索错误信息的意义。
3. 如果你的程序被 Valgrind “认证通过”了，那这个程序应该就不错了，而且你也许还从中学到了关于怎样写代码的一些知识。

在这个习题里，我没指望你能立马就熟练运用 *Valgrind* ，只是让你装好它并学习如何快速上手，这样我们就能在以后的习题里用上它。

5.4 加分习题

1. 根据 *Valgrind* 和编译器的提示，修正错误。
2. 在网上研读 *Valgrind* 。
3. 下载其他软件的源码，自己 build 一下。尝试一些你已经用过但还从没自己动手 build 过的软件。
4. 看看 *Valgrind* 的源码的路径组织架构，读一下它的 Makefile ，别担心，我也觉得这东西一团糟。

Chapter 6

Exercise 5: C 程序的结构

你现在已经知道了怎么使用 *printf*，也有了一些可供你支配使用的基本工具，所以我们现在来逐行分析一个简单 C 语言程序的结构。在这个程序里你将会敲一些你还不熟悉的东西，我将会慢慢地拆解他们。在接下来的一些练习中我们将继续与这些概念打交道。

ex5.c

```c
#include <stdio.h>

/* This is a comment. */
int main(int argc, char *argv[])
{
    int distance = 100;

    // this is also a comment
    printf("You are %d miles away.\n", distance);

    return 0;
}
```

把这些代码敲出来，然后使它运行并且保证 *Valgrind* 没有报错。你也许没法确保，但是你要养成检查的好习惯。

6.1 你应该看到的结果

这是个很无聊的输出，但是这个练习的关键是分析代码:

ex5 的输出

```
$ make ex5
```

```
cc -Wall -g    ex5.c   -o ex5
$ ./ex5
You are 100 miles away.
$
```

6.2 代码详解

在敲这些代码的时候，也许你已经察觉到这些代码中包含的 C 语言特征。让我们来快速地逐行解读一下这些代码，然后通过练习来更好地理解每一个部分。

ex5.c:1 *include* 指令的作用是把被包含的文件内容导出到当前源文件中。C 语言的惯例是使用 `.h` 后缀来表示头文件，头文件包含了你在程序中需要用到的函数的列表。

ex5.c:3 这是一个多行的注释 *(comment)*，你可以随意在`/*` 和`*/` 之间包含任意多的字符。

ex5.c:4 到现在为止你一直在" 盲目地" 使用一个复杂形式的 主函数 *(main function)*。C 程序是怎么运行的呢？首先由操作系统加载你的程序，然后运行一个名为 *main* 的函数。这个函数完全结束的时候，它会返回一个 整数型 的返回值，另外，该函数需要两个参数，一个是参数个数，它的类型为 *int*，还有一个参数是数组，它用来保存所有输入的参数，它是一个字符串 (*char **) 数组。是不是很难理解？别担心，我们很快就会讲到它们。

ex5.c:5 开始任何一个函数体的时候，你需要使用 `{` 来指出这是一个" 块 (block)" 的开始。在 Python 中你只需要写一个 : 并且缩进即可。在其他的一些编程语言中也许你需要用 *begin* 或者 *do* 来表示一个块的开始。

ex5.c:6 一个变量的声明和赋值可以在同一时间完成。你可以使用 `type name = value;` 这个语法来创建一个变量。C 语言的表达式 (除了逻辑条件) 以 `';'`(semicolon) 字符来结尾。

ex5.c:8 另一种形式的注释，它和 Python 和 Ruby 的注释一样，从 `//` 开始，到一行的结尾结束。

ex5.c:9 又一次调用你的老朋友，*printf* 函数。像很多其他语言一样，函数调用的语法是 `name(arg1, arg2);` ，也可以没有参数，或者有任意数量的参数。稍后我们会看到，*printf* 函数是一个比较怪异的函数，因为它的参数数量是可变的。

ex5.c:11 函数的返回语句，它将会给操作系统返回一个程序退出的值，可能你还不熟悉 UNIX 软件怎样使用返回值，同样，我们一会就会看到。

ex5.c:12 最终，我们用一个右花括号 (`}`) 结束 main 函数。这也是程序的结尾。

在这段分析里有很多的信息，所以请逐行学习并且确保你至少对接下来要做的事情有一点把握。你可能不会知道所有的东西，但是你可以在继续前进之前大概猜测到。

6.3 加分习题

1. 针对每一行，写下那些你不理解的符号，看看你是否能猜到他们是什么意思。在纸上画一个表格把你的猜测记下来，等会可以回来检查一下你是否猜对了。
2. 回到之前练习题中的代码并做类似的解析，看看你是否理解了这些代码。写下你自己不知道和不能解释的东西。

Chapter 7

习题 6：变量类型

现在你应该对 C 语言的结构有了大概的了解，那么接下来我们再做些十分简单的事情，即输出不同类型的变量：

ex6.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int distance = 100;
    float power = 2.345f;
    double super_power = 56789.4532;
    char initial = 'A';
    char first_name[] = "Zed";
    char last_name[] = "Shaw";

    printf("You are %d miles away.\n", distance);
    printf("You have %f levels of power.\n", power);
    printf("You have %f awesome super powers.\n", super_power);
    printf("I have an initial %c.\n", initial);
    printf("I have a first name %s.\n", first_name);
    printf("I have a last name %s.\n", last_name);
    printf("My whole name is %s %c. %s.\n",
            first_name, initial, last_name);

    return 0;
}
```

在这个程序中我们将：1. 声明不同类型的变量，并且 2. 使用不同的字符串格式 (string format) *printf* 来打印输出它们。

7.1 你应该看到的结果

你的输出应该和我的一样，你也许也会发现这些 C 的字符串格式与 Python 和其他语言的字符串格式是多么的相似。其实它们已经存在很长时间了。

ex6 的输出

```
$ make ex6
cc -Wall -g    ex6.c   -o ex6
$ ./ex6
You are 100 miles away.
You have 2.345000 levels of power.
You have 56789.453200 awesome super powers.
I have an initial A.
I have a first name Zed.
I have a last name Shaw.
My whole name is Zed A. Shaw.
$
```

你将看到一系列的“类型（types）”, 它们将告诉 C 编译器哪些变量应该被提出来，并格式化字符串以匹配不同类型。以下是它们匹配过程的细节：

整数型 (Integer) 使用 *int* 声明整数，用 %d 打印输出。

浮点型 (float) 根据大小决定使用单精度浮点型 (*float*) 或双精度浮点型 (*double*) 声明浮点数 (双精度型更大些)，用 %f 打印输出。

字符型 (char) 使用 *char* 声明字符型，写法是用单引号 ' 包围字符，用 %c 打印输出。

字符串 (string) 使用 char name[] 声明字符串型，写法是用双引号 " 包围字符，用 %s 打印输出。

这里需要注意的是 C 语言非常注意区分单双引号。单引号是为字符 *char* 所准备的，而双引号则是为一堆字符，既字符串型，*char[]* 所准备的。

Note 4 *C 数据类型的英文缩写*

当我们谈论到 C 数据类型时，在英语中我将直接使用 char[] 来表示，而不会使用完整字符名 SOMENAME[].
但事实上作为 C 代码这是不对的，缩写仅仅是书中一种表达数据类型的简便方式。

7.2 让程序出错

你可以通过给 printf 赋错误的值来搅坏这个程序。用那条打印我的名字的指令来打个比方，如果在参数中，把变量 *initial* 放在变量 *first_name* 的前面，bug 就来了。真这样运行的话，编译器会对你发飙，你没准会看到个“段故障（Segmentation fault）”，就像下面这样：

ex6 的爆炸现场

```
$ make ex6
cc -Wall -g    ex6.c   -o ex6
ex6.c: In function 'main':
ex6.c:19: warning: format '%s' expects type 'char *', but argument 2 has type 'int'
ex6.c:19: warning: format '%c' expects type 'int', but argument 3 has type 'char *'
$ ./ex6
You are 100 miles away.
You have 2.345000 levels of power.
You have 56789.453125 awesome super powers.
I have an initial A.
I have a first name Zed.
I have a last name Shaw.
Segmentation fault
$
```

在 Valgrind 下再次运行，看看关于“Invalid read of size 1”的内容会告诉你些什么。

7.3 加分习题

1. 除了改变 *printf* 以外，再想想有没有别的办法可以弄坏这个代码，并找出修复它的方法。
2. 上网搜索”printf formats”，找出并尝试一些稀有格式。
3. 探索一下有多少种方式可以写出数字。尝试以下几个，并找到更多：octal, hexadecimal。
4. 尝试打印输出一个空字符串 `""`.

Chapter 8

习题 7：再讲点变量，加上点算术

让我们通过声明更多的变量类型来熟悉变量的用法，这些类型包括 *int*(整数型)、*float*(单精度浮点型)、*char*(字符型)、以及 *double*(双精度浮点型)。我们会在稍后的各种数学表达式中用到它们，同时你还会学到 C 语言的基础数学。

ex7.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int bugs = 100;
    double bug_rate = 1.2;

    printf("You have %d bugs at the imaginary rate of %f.\n",
            bugs, bug_rate);

    long universe_of_defects = 1L * 1024L * 1024L * 1024L;
    printf("The entire universe has %ld bugs.\n",
            universe_of_defects);

    double expected_bugs = bugs * bug_rate;
    printf("You are expected to have %f bugs.\n",
            expected_bugs);

    double part_of_universe = expected_bugs / universe_of_defects;
    printf("That is only a %e portion of the universe.\n",
            part_of_universe);

    // this makes no sense, just a demo of something weird
    char nul_byte = '\0';
    int care_percentage = bugs * nul_byte;
    printf("Which means you should care %d%%.\n",
            care_percentage);

    return 0;
}
```

以下就是这段毫无意义的代码所做的事情：

ex7.c:1-4 常见的 C 程序开头。

ex7.c:5-6 为一些虚假的 bug 数据声明一个 *int* 和一个 *double* 类型的变量。

ex7.c:8-9 没什么新的知识点，就是把这俩变量打印出来。

ex7.c:11 用一种新的变量类型 *long* 声明一个大数字。

ex7.c:12-13 使用 `%ld` 输出那个数，`%ld` 是在 `%d` 前加了个修饰符。增加的字母 "l" 的意思是" 按长整型（long decimal）打印出这个数"。

ex7.c:15-17 更多的数学计算和打印输出而已。

ex7.c:19-21 打造出一个变量，用来描述你的 bug 和宇宙中 bug 总数的比值，当然这个计算结果是完全不准确的。我们可以用修饰符 `%e` 用科学计数法输出结果.

ex7.c:24 用 `'\0'` 这样的特殊语法创建一个字符变量，这样就建立了一个 "空字节 (nul byte)" 字符. 它其实就是数字 0。

ex7.c:25 用这个字符乘以 bugs 变量，你期待的问题就有了 0 这个结果。这里向你演示了一个丑陋的代码技巧，有时你会在一些地方看到这种用法。

ex7.c:26-27 输出那个结果, 注意到我使用 `%%` (两个百分号)，这样我就可以打印出'%' 这个百分号字符来。

ex7.c:28-30 *main* 函数的结尾。

这些代码只是一个练习，用来为你演示算术的原理。在代码的结尾处你也看见了只有 C 中有，而别的语言中没有的东西。对于 C 语言来说，字符只是整数而已。其实它是一种非常小的整数，仅此而已。这就意味着你可以用它们做数学运算，很多软件就是这么做的，不管目的是好是坏。

最后的这点内容是让你初窥 C 语言是直接访问机器的过程。我们会在后面的联系中进一步探索这个课题。

8.1 你应该看到的结果

和平常一样，这些是你应该在输出中看到的结果：

ex7 输出

```
$ make ex7
cc -Wall -g    ex7.c   -o ex7
$ ./ex7
```

```
You have 100 bugs at the imaginary rate of 1.200000.
The entire universe has 1073741824 bugs.
You are expected to have 120.000000 bugs.
That is only a 1.117587e-07 portion of the universe.
Which means you should care 0%.
$
```

8.2 让程序出错

同样的，试着往 *printf* 中传递错的参数来让程序出错。对比使用修饰符%s 和 %c 输出 *nul_byte* 有什么不同结果。当你让程序出错后，在 *Valgrind* 运行你的程序，看看它对你的破坏行为如何报错。

8.3 加分习题

1. 给 *universe_of_defects* 赋不同大小的值直到从编译器获得警告信息。
2. 这些庞大数字实际上打印出的是什么？
3. 把 *long* 换成 *unsigned long*，试着找到比刚才那个数更大的值。
4. 上网查查 *unsigned* 的功能是什么。
5. 在下一个练习之前，试着向自己解释为什么 *char* 可以和 *int* 相乘.

Chapter 9

习题 8：数组和大小

上一个习题中你做了些数学运算，不过你在运算中使用了 `'\0'` (null) 字符。如果你学过其他的语言，这对你来说就会有些奇怪，因为其他的语言会将字符串 (string) 和字节数组 (byte array) 当作不同的怪兽来对待，然而 C 语言则将字符串直接当字节数组对待，而只有那些实现打印功能的 C 函数才知道它们是不同的。

明白 C 语言将字符串和字节数组等同对待是很重要的，而在解释这一点之前，有几个概念一定要讲清楚：*sizeof* 和数组。以下是我们即将探讨的代码：

ex8.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int areas[] = {10, 12, 13, 14, 20};
    char name[] = "Zed";
    char full_name[] = {
        'Z', 'e', 'd',
         ' ', 'A', '.', ' ',
         'S', 'h', 'a', 'w', '\0'
    };

    // WARNING: On some systems you may have to change the
    // %ld in this code to a %u since it will use unsigned ints
    printf("The size of an int: %ld\n", sizeof(int));
    printf("The size of areas (int[]): %ld\n",
            sizeof(areas));
    printf("The number of ints in areas: %ld\n",
            sizeof(areas) / sizeof(int));
    printf("The first area is %d, the 2nd %d.\n",
            areas[0], areas[1]);

    printf("The size of a char: %ld\n", sizeof(char));
    printf("The size of name (char[]): %ld\n",
            sizeof(name));
    printf("The number of chars: %ld\n",
            sizeof(name) / sizeof(char));

    printf("The size of full_name (char[]): %ld\n",
            sizeof(full_name));
    printf("The number of chars: %ld\n",
            sizeof(full_name) / sizeof(char));

    printf("name=\"%s\" and full_name=\"%s\"\n",
            name, full_name);

    return 0;
}
```

在这段代码中我们创建了一些包含不同数据类型的数组。由于数组是 C 语言工作原理的核心，我们可以用很多种不同的方法来创建数组。这里将就用 `type name[] = {initializer};` 这种格式了，往后我们还会碰到更多的方式。这个语法的意思就是：“我需要一个初始化为.. 类型的数组”，当 C 接收到这个指令时，将会：

1. 检查数据类型，这里我们看到的是 *int*。
2. 检查 `[]`，发现长度值没有给出。
3. 检查初始值：发现 `{10, 12, 13, 14, 20}`，于是知道了你要将这 5 个整数放入数组。
4. 在本机创建出一篇内存区域，用以顺次保存这五个整数。
5. 把上面创建的内存位置赋给 *areas* 这个名字。

也就是说，*areas* 这一行的作用就是创建了容纳这 5 个整数的数组。而到了 `char name[] = "Zed";` 这一行，它实现的事情也是一样的，只不过它创建了另一个有三个字符的数组并赋值到 *name*。我们创建出来的最终数组是 *full_name*，但逐字符的拼写方式却很恼人。对于 C 语言来说，*name* 和 *full_name* 都是创建字符数组的方式。

接下来我们将使用一个叫做 *sizeof* 的关键字来询问东西的大小 (以 *bytes* 为单位)。C 语言就是一门关于内存大小，内存位置，以及如何处理内存区块的语言。更简单一点说，C 语言提供了 *sizeof*，就是为了让你在用到某样东西之前，可以问到这个东西的大小。

这里可能会让人有些头晕，所以我们先运行下面这段代码再来解释。

9.1 你应该看到的结果

ex8 的输出

```
```

```
1  $ make ex8
2  cc -Wall -g    ex8.c   -o ex8
3  $ ./ex8
4  The size of an int: 4
5  The size of areas (int[]): 20
6  The number of ints in areas: 5
7  The first area is 10, the 2nd 12.
8  The size of a char: 1
9  The size of name (char[]): 4
10 The number of chars: 4
11 The size of full_name (char[]): 12
12 The number of chars: 12
13 name="Zed" and full_name="Zed A. Shaw"
14 $
```

现在你看到了不同的 *printf* 指令的输出，也大致看到了 C 是如何工作的了。由于你的电脑的整型大小可能不一样，你的输出有可能跟我的会完全不同，我就拿输出来讲讲吧：

5 我的电脑认为 *int* 的大小是 4 个字节. 电脑系统有 32-bit 和 64-bit 之分，所以你的电脑可能会使用一个不同的整形大小。。

6 数组 *areas* 里有 5 个整数，我的电脑就合情合理地用了 20 个字节来存储它。

7 如果用 *areas* 的大小除以 *int* 的大小，我们将得到元素数 5。再看看代码，这正是初始值中的设定。

8 我们访问了数组，得到了 `areas[0]` 和 `areas[1]`，这意味着 C 和 Python/Ruby 一样，数组的索引也是从 0 开始的。

9-11 一样的方法去访问 *name* 这个数组，有没有发现数组的大小有些奇怪？”Zed” 是三个字符，可它的长度怎么是 *4* 个字节呢？这第 4 个字符是哪来的呢？

12-13 一样的方法去读取 *full_name*，我们看到这次的值是正确的。

13 最后我们打印输出 *name* 和 *full_name* 以证明对于 printf 来说，他们都是字符串。

请确保弄懂这些代码，并且明白这些输出结果跟输入代码的对应关系。之后我们将在这个基础上更多地探索数组和存储。

9.2 让程序出错

让这个程序出错很简单，试一试这些：

1. 删除 *full_name* 尾部的 `'\0'` 并重新运行程序。再在 Valgrind 下运行一次。现在将 *full_name* 的定义放到 *main* 的顶部，*areas* 的前面。在 Valgrind 下运行几次，看看会不会得到新的错误信息。有时候你可能会幸运到一个错误都看不见。
2. 尝试打印输出 `areas[10]` 而非 `areas[0]` ，并看看 Valgrind 是怎样认为的。
3. 多做几个不同的尝试，在 *name* 和 *full_name* 上面也试一下。

9.3 加分习题

1. 使用 `areas[0] = 100;` 这样的语句，对数组 *areas* 中的元素进行赋值。
2. 尝试对 *name* 和 *full_name* 的元素进行赋值。
3. 尝试把 *areas* 中的一个元素设置成 *name* 中的一个字符.
4. 上网搜索一下，了解一下不同 CPU 整形大小差异。

Chapter 10

习题 9：数组和字符串

在上一个习题中我们学习了如何创建基本数组以及数组和字符串的对应关系。这个习题中我们将更全面的了解数组与字符串之间的相似之处，并更多地了解关于内存分布的知识。

这节习题展示给你的是 C 语言将字符串作为字节数组存放的，该数组以 `'\0'` (nul) 作为终止。在上一习题中我们是手动这样去做的，所以你可能已经得出这个结论了。这里我们将用另一种方式更清晰地去展示这一点，将字符串与一个数字数组相比较：

ex9.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int numbers[4] = {0};
    char name[4] = {'a'};

    // first, print them out raw
    printf("numbers: %d %d %d %d\n",
            numbers[0], numbers[1],
            numbers[2], numbers[3]);

    printf("name each: %c %c %c %c\n",
            name[0], name[1],
            name[2], name[3]);

    printf("name: %s\n", name);

    // setup the numbers
    numbers[0] = 1;
    numbers[1] = 2;
    numbers[2] = 3;
    numbers[3] = 4;

    // setup the name
    name[0] = 'Z';
    name[1] = 'e';
    name[2] = 'd';
    name[3] = '\0';

    // then print them out initialized
    printf("numbers: %d %d %d %d\n",
            numbers[0], numbers[1],
            numbers[2], numbers[3]);

    printf("name each: %c %c %c %c\n",
            name[0], name[1],
            name[2], name[3]);

    // print the name like a string
    printf("name: %s\n", name);

    // another way to use name
    char *another = "Zed";

    printf("another: %s\n", another);

    printf("another each: %c %c %c %c\n",
            another[0], another[1],
            another[2], another[3]);

    return 0;
}
```

在这段代码中我们建立了一些数组，用的是对每一个元素一一赋值这种无聊乏味的方法。在 *numbers* 我们是在创建一些数值，但在 *name* 中我们实际上是在手动创建字符串。

10.1 你应该看到的结果

运行这段代码，你应该看到数组的内容被打印出来。首先打印出来的是初始为 0 的内容，接着打印出来的是初始化后的内容。

ex9 的输出

```
```

```
$ make ex9
cc -Wall -g    ex9.c   -o ex9
$ ./ex9
numbers: 0 0 0 0
name each: a
name: a
numbers: 1 2 3 4
name each: Z e d
name: Zed
another: Zed
another: Z e d
$
```

关于这个程序，你将注意到以下几点有趣的事：

1. 初始化数组的时候，我并不需要将 4 个元素一一赋予数组。这是 C 语言的一个快捷功能：当你只设置了一个元素的值，它就会把剩下的都初始化为 0。
2. 打印 *numbers* 中的每一个元素时，打印出来的值都是 0。
3. 打印 *name* 中的每一个元素时，只有第一个元素'a' 被打印出来了，因为 `'\0'` 这个特殊字符是不会被显示出来的。
4. 第一次打印 *name* 这个数组（字符串）的时候，它只输出了"a"。由于这个数组在第一个元素"a"之后的其它元素都是被 0 填充的，所以这个字符串正确地使用了 `'\0'` 作为终结字符。
5. 接下来我们再次创建数组，用的还是最单调乏味的手动一一赋值的方法，然后再次将它们打印出来。看看结果有什么变化。现在数字设置是生效了，不过你有没有看到我的名字也被正确地打印出来了？
6. 处理字符串有两种语法：第 6 行的 `char name[4] = {'a'}` 和第 44 行的 `char *another = name` 。第一种不太常用，第二种才是你在处理这样的字符串常值 (string literal) 时应该使用的语法。

注意，我使用了完全一样的语法和代码风格来操作整数数组和字符数组进行，但是 *printf* 认为 *name* 只是一个字符串。这里再次解释一下，这是因为对于 C 语言来说，字符串和字符数组时是没有区别的。

最后一点，当你创建字符串时，你一般应该使用 `char *another = "Literal"` 这样的语法。这其实和前面的方法是一回事，只不过这样写更符合习惯，而且写起来更容易而已。

10.2 让程序出错

几乎所有的 C 程序的 bug 都来自两个地方，一是没有分配足够的空间，二是字符串的结尾没有放 `'\0'` 。事实上这些 bug 及其常见而且很难弄对，以至于大部分高质量的 C 代码都会避免使用 C 风格的字符串。后面的习题中我们将学到如何完全避免使用 C 语言的字符串。

对本节的程序而言，让它出错的关键方法就是忘记在字符串结尾放 `'\0'` 。试一下下面的方法：

1. 取消 *name* 的初始设定。
2. 不小心设了 `name[3] = 'A';` ，这样一来就没有终止符了.

3. 将初始值设为 `{'a','a','a','a'}`，这样就会字符'a' 过多而终止符 `'\0'` 没地方放了.

想想还有没有其他弄坏它的办法，并且就像之前一样，将它们通通放在 Valgrind 下运行，这样你就会知道到底发生了什么以及各种错误的名字。有时候你的错误连 Valgrind 都找不出来，但是你可以试变更一下申明变量的位置然后看会不会遇到错误。这也是 C 语言歪门邪道的地方，有时候你只不过动了一下变量的位置，bug 就奇怪地不见了。

10.3 加分习题

1. 将字符赋值给 *numbers*，并使用 *printf* 将字符逐一的打印输出。看看你会得到什么样的编译器警告？
2. 反过来处理一下 *name*, 试着像操作 *int* 数组一样，并逐一打印 (*int*) 值。Valgrind 是怎样认为的？
3. 你还可以用多少种其他方式来打印输出它们？
4. 如果一个字符数组的长度是 4 个字节，一个整数的长度也是 4 个字节，你可以把整个 *name* 数组当做一个整数来处理吗？你要通过怎样的手段才能做到这一点？
5. 在纸上把数组画出来，用一排盒子的格式表示出这些数组。然后将你上面的操作在纸上模拟出来，看看你是否弄对了。
6. 将 *name* 用 *another* 的申明风格申明一次，看看代码是否还能正常运行。

Chapter 11

习题 10：字符串数组，循环

你可以用多种数据类型组成一个数组，要记得“字符串”和“字节数组”就是一个东西。下一步就是将字符串放到数组里。这里还会为你介绍第一个循环结构——*for-loop* （for 循环）——它会帮助你打印出这个新的数据结构。

有个有趣的地方是，其实已经有个字符串数组在你的程序中躲藏了一段时间了，那就是在 *main* 函数参数里的 *char *argv[]* 。下面一段代码会打印出你通过命令行传递的所有参数：

ex10.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;

    // go through each string in argv
    // why am I skipping argv[0]?
    for(i = 1; i < argc; i++) {
        printf("arg %d: %s\n", i, argv[i]);
    }

    // let's make our own array of strings
    char *states[] = {
        "California", "Oregon",
        "Washington", "Texas"
    };
    int num_states = 4;

    for(i = 0; i < num_states; i++) {
        printf("state %d: %s\n", i, states[i]);
    }

    return 0;
}
```

for-loop 的格式是这样的：

```
for(INITIALIZER; TEST; INCREMENTER) {
    CODE;
}
```

接下来讲讲 *for-loop* 的工作原理：

1. *INITIALIZER* 是完成循环前的准备工作的代码，在本例中就是 `i = 0` 这句。
2. 接下来运行的是 *TEST* 这个布尔表达式的检查动作，如果它的值是 false (0)，那么接下来就什么都不做了。
3. 运行代码 *CODE*，完成它要做的事情。
4. *CODE* 运行之后，*INCREMENTER* 部分就会被运行，通常都是将某个变量递增，比如本例中的 `i++` 。
5. 然后代码会从第 2 步开始重新运行，直到 *TEST* 的值成为 false (0) 为止。

这个 *for-loop* 将使用 *argc* 和 *argv* 这两个变量来对命令行参数做如下处理：

1. 操作系统将每一条命令行参数当做字符串传递到数组 *argv* 中，程序的名称 (./ex10) 的位置是 0，其它的参数向后依次排列。
2. 操作系统还将 *argc* 设为 *argv* 中参数的个数，这样你处理它的时候就不必担心超过最后一个参数了。
3. *for-loop* 做了预备工作，设定 `i = 1` 。
4. 之后它通过 `i < argc` 这句判断 *i* 是否小于 *argc*。一开始 $0 < 1$，所以判断通过。
5. 然后它接着运行代码，打印出 *i* ，然后使用 *i* 作为 *argv* 数组内部的索引。
6. 接着递增器将通过语法 `i++` 运行。这算是 `i = i + 1` 的简写方式。
7. 然后它就会重复上述代码，直到 `i < argc` 的值成为 false （0），然后循环就结束了，余下的程序将继续运行。

11.1 你应该看到的结果

跟这个程序打交道你需要用两种方式运行它。第一种是传递一些命令行参数从而赋值给 *argc* 和 *argv* 。第二种是运行时不传递任何参数，这样的话由于 `i < argc` 一开始就是 false，*for-loop* 将不会运行。

ex10 的输出

```
$ make ex10
cc -Wall -g    ex10.c   -o ex10
$ ./ex10 i am a bunch of arguments
arg 1: i
arg 2: am
arg 3: a
arg 4: bunch
arg 5: of
arg 6: arguments
state 0: California
state 1: Oregon
state 2: Washington
state 3: Texas
$
$ ./ex10
state 0: California
state 1: Oregon
state 2: Washington
state 3: Texas
$
```

11.1.1 理解字符串数组

从习题中你应该已经知道了 C 语言中“字符串数组”是怎样构造出来的了——通过结合 `char *str = "blah"` 和 `char str[] = {'b','l','a','h'}` 来构建一个二维数组。第 14 行的 `char *states[] = {...}` 就是这个二维组合的动作，其中一维是以字符串为元素，另一维度则是以构成字符串的字符作为元素。

头晕了吧？很多人都没有想过多重维度的概念，所以你应该试着在纸上构建出这个字符串数组：

1. 在这张纸上做一个矩形网格，最左列填入每一个 *string(字符串)* 的索引值。
2. 最上面一行填入每一个 *character(字符)* 的索引值。
3. 然后将字符串填入表格中间的空位，每个字符占一格。
4. 做好这个表格后，用它来手动追踪一遍代码。

另一个弄明白的方法是通过使用你更熟悉的语言 (比如 Python 或者 Ruby) 建造一个相同的结构。

11.2 让程序出错

1. 选一个你喜欢的别种语言，用它来运行这个程序，但是要给它尽可能多的命令行参数。看看你能不能通过给出太多的形参把这程序给搞崩了。
2. 将 *i* 初始化为 0 看看会发生什么。你需要同时调整 *argc* 吗？还是说它可以直接运行？为什么这里可以用基于 0 的索引方式呢？
3. 把 *num_states* 设置得更高导致错误，看看会发生什么。

11.3 加分习题

1. 弄明白什么样的代码你可以放到 *for-loop* 中。
2. 查找一下应该如何在 *for-loop* 中使用字符 `','` (comma) 分隔多条命令。
3. 阅读以下什么是 *NULL*，并尝试将数组 *states* 的元素之一设为 *NULL*，看看它会打印输出什么。
4. 试试你可不可以在打印输出之前将数组 *states* 的元素赋值到数组 *argv* 中，然后再反过来试一下。

Chapter 12

Exercise 11: While 循环和布尔表达式

你已经体验会过了 C 语言的循环，但你可能对布尔表达式 `i < argc` 的理解还不清晰。在我们看到 *while* 循环 是如何工作之前，让我先解释一些东西。

在 C 语言中根本就没有“布尔（boolean）”这种类型，取而代之的是数字零为“false(假)”，其他非 0 数都为“true（真）”。在最后一个练习中表达式 `i < argc` 实际上只返回 1 或 0，而不是像 Python 中那样返回 *True* 或 *False*。这是又一个 C 语言更能展示计算机工作原理的例子，因为对于计算机来说表示真假的值都只是整数而已。

现在你需要实现上一个习题一样的程序，不过你要使用 *while* 循环来解决。接下来会让你比较两者，看看它们之间的关系是什么。

ex11.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    // go through each string in argv

    int i = 0;
    while(i < argc) {
        printf("arg %d: %s\n", i, argv[i]);
        i++;
    }

    // let's make our own array of strings
    char *states[] = {
        "California", "Oregon",
        "Washington", "Texas"
    };

    int num_states = 4;
    i = 0;  // watch for this
    while(i < num_states) {
        printf("state %d: %s\n", i, states[i]);
        i++;
    }

    return 0;
}
```

从下面你可以看出 *while* 循环 更简单一些:

```
while(TEST) {
    CODE;
}
```

它只是简单地在 *TEST* 为 true(1) 时运行 *CODE* 而已。这就是说，如果我们要事项 *for* 循环 那样的功能，那就得自己初始化并递增 *i*。

12.1 你应该看到的结果

输出和上一习题基本一样，所以我只是变动了一下运行方式，这样你就看到了不同的运行结果.

ex11 的输出

```
$ make ex11
cc -Wall -g    ex11.c   -o ex11
$ ./ex11
arg 0: ./ex11
state 0: California
state 1: Oregon
state 2: Washington
state 3: Texas
$
$ ./ex11 test it
arg 0: ./ex11
arg 1: test
arg 2: it
state 0: California
state 1: Oregon
state 2: Washington
state 3: Texas
$
```

12.2 让程序出错

在你自己写代码时，你应该多用 *for* 循环结构，少用 *while* 循环。因为 *for* 循环更难出错。这里有几种常见的错误:

1. 忘了初始化一开始的 `int i;`，造成循环错误.
2. 忘了初始化第二次循环里的 *i* 于是循环里的值和第一次循环时一样。现在你的第二次循环可能运行也可能运行不了。
3. 忘了在循环的结尾执行 `i++` 自增，造成了“死循环”，这在编程史的前一二十年里可谓是最恐怖的问题之一。

12.3 加分习题

1. 使用 `i--` 让循环从 `argc` 循环计数递减到 0。你可能需要做一些计算让数组的索引正常工作。
2. 使用 while 循环把 *argv* 中的值拷贝到 *states* 中。
3. 让这个拷贝循环永远不会失败，例如 *argv* 中元素太多时，就不把它们都拷贝到 *states* 中。
4. 深入研究一下你是否真的拷贝了这些字符串。不过答案可能会使你感到惊讶与迷惑。

Chapter 13

Exercise 12: If, Else-If, Else

在每一种语言中都共有的就是 *if* 语句，C 语言中同样也有。下面的代码使用的就是 *if* 语句以确认你只输入一个或两个参数:

ex12.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;

    if(argc == 1) {
        printf("You only have one argument. You suck.\n");
    } else if(argc > 1 && argc < 4) {
        printf("Here's your arguments:\n");

        for(i = 0; i < argc; i++) {
            printf("%s ", argv[i]);
        }
        printf("\n");
    } else {
        printf("You have too many arguments. You suck.\n");
    }

    return 0;
}
```

if 语句的格式是这样的：

```
if(TEST) {
    CODE;
} else if(TEST) {
    CODE;
} else {
    CODE;
}
```

这和其他大多数语言里的类似，除了几处 C 语言特有的不同点:

1. 正如前面所提到的，如果 *TEST* 的部分值为 0，结果就为假，其他情况则为真。
2. 你必须把 *TEST* 元素放到圆括号中，但其他的一些语言不需要你这样做。
3. 你不需要用 `{}` 花括号把代码包括起来，但这是一种非常不好的格式。花括号可以让你更清楚地看出某个代码分支开始和结束的位置。如果你不使用花括号，各种令人讨厌的错误就会出现。

除了这些，它和其他语言中的功能一样。*else if* 和 *else* 部分也都不是必须的。

13.1 你应该看到的结果

这个程序运行起来很简单:

ex12 的输出

```
$ make ex12
cc -Wall -g    ex12.c   -o ex12
$ ./ex12
You only have one argument. You suck.
$ ./ex12 one
Here's your arguments:
./ex12 one
$ ./ex12 one two
Here's your arguments:
./ex12 one two
$ ./ex12 one two three
You have too many arguments. You suck.
$
```

13.2 让程序出错

这个例子不容易出错，因为它太简单了，不过可以试着把 *if* 语句 里的测试条件改乱掉试一下。

1. 移除结尾的 *else*，这样程序就永远不会找到边缘条件（edge case）了。
2. 把 `&&` 换成 `||` 你就用“或（or）”代替了“与（and）”测试，看看结果是怎样的。

13.3 加分习题

1. 你已经简单学到了 `&&` ，它的功能是执行“与”的比较，再上网研究一下各种不同的“布尔操作符 (boolean operator)”。
2. 多写写条件判断语句，看看你还能得出些什么来。
3. 回到习题 10 和 11，使用 *if* 语句将循环提前结束。完成操作你需要 *break* 语句。自己查查怎么用。
4. 第一个条件判断是否真的是对的呢？对于你来说” 第一个参数” 和用户输入的第一个参数是不一样的。把这里修改正确。

Chapter 14

Exercise 13: Switch 语句

在其他诸如 Ruby 这样的语言里，你可以在 *switch* 语句 中用任何形式的语句，在另外一些像 Python 这样的语言则不提供 *switch* 语句，因为 *if* 语句就能起到相同的作用。对这些语言来说，*switch* 只是将 *if* 语句换了一种写法而已，而两者的内部工作原理也是一样的。

而在 C 语言里，*switch* 语句是完全不一样的东西，它其实是一个“跳转表”。你只能把结果是整数的表达式放到 *switch* 语句中，而不是随便的布尔表达式都可以，这些整数被用作和 *switch* 语句头部的值比较，如果两者匹配，代码就会跳到子句的位置继续运行。以下是一些我们将要来分析和理解” 跳转表” 概念的代码。

ex13.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    if(argc != 2) {
        printf("ERROR: You need one argument.\n");
        // this is how you abort a program
        return 1;
    }

    int i = 0;
    for(i = 0; argv[1][i] != '\0'; i++) {
        char letter = argv[1][i];

        switch(letter) {
            case 'a':
            case 'A':
                printf("%d: 'A'\n", i);
                break;

            case 'e':
```

```
            case 'E':
                printf("%d: 'E'\n", i);
                break;

            case 'i':
            case 'I':
                printf("%d: 'I'\n", i);
                break;

            case 'o':
            case 'O':
                printf("%d: 'O'\n", i);
                break;

            case 'u':
            case 'U':
                printf("%d: 'U'\n", i);
                break;

            case 'y':
            case 'Y':
                if(i > 2) {
                    // it's only sometimes Y
                    printf("%d: 'Y'\n", i);
                }
                break;

            default:
                printf("%d: %c is not a vowel\n", i, letter);
        }
    }

    return 0;
}
```

在这个程序里我们从命令行参数获得一个参数，然后输出参数里所有的元音字母，这是一个乏味的演示如何使用 *switch* 语句的例子。让我们来看看它是怎么工作的：

1. 编译器会在程序中 *switch* 语句开始的地方做一个标记，我们记这个位置为 Y。
2. 然后它会评估 `switch(letter)` 表达式并从中得到一个值。在我们的例子中，这个值是命令行参数 `argv[1]` 中的某个字母对应的原始 ASCII 码。
3. 编译器对于像 `case 'A':` 这样的 *case* 块也会产生一个相对于位置 Y 的偏移值，这个偏移值就是'A'。所以对于在 `case 'A'` 下的代码来说，它们在程序中的位置是 Y + 'A'。

4. 然后它将会做数学运算来得出 Y+letter 对应于 *switch* 语句中的位置，如果得到的位置太远 (即没有找到匹配的值), 它将会把这个位置值置为 Y+default。
5. 一旦程序知道了这个位置，就会跳转到代码中的那个点，然后继续运行。这就是为什么有的代码块有 *break* 语句而有的没有。[1]
6. 如果输入了'a'，那么程序将会跳转到 `case 'a'`，这里没有 break 语句，所以它将会继续往下执行，进入 `case 'A'` 代码块，那里有一些代码和 break 语句。
7. 最终，它会运行该代码块，碰到 break 语句，然后退出整个 *switch* 语句。

这是对 *switch* 语句如何工作的一个比较深入的研究，不过在实际应用时你只要记住这些简单的规则：

1. 每次都包含一个 *defaule:* 分支，这样就能捕捉到任何没有预料到的输入。
2. 除非你真的需要，否则不要允许代码中出现" 运行所有分支 (fall through)" 这样的现象。如果你使用了，那就添加一个注释 `//fallthrough`，这样其他人就知道这是你特意而为，而不是因为疏忽。
3. 在你写具体的分支代码之前，最好始终同时写好 *case* 语句和 *break* 语句 ([2])。
4. 如果可以的话，就用 *if* 语句来取代使用 *switch* 语句。

14.1 你会看到什么

下面是我如何使用这个程序的例子，同时来演示几种不同的给程序传递参数的方式。

ex13 的输出

```
$ make ex13
cc -Wall -g    ex13.c   -o ex13
$ ./ex13
ERROR: You need one argument.
$
$ ./ex13 Zed
0: Z is not a vowel
1: 'E'
2: d is not a vowel
$
$ ./ex13 Zed Shaw
ERROR: You need one argument.
$
$ ./ex13 "Zed Shaw"
0: Z is not a vowel
1: 'E'
2: d is not a vowel
```

[1]译者注：作者的意思应该是没有 *break* 语句的话程序将会继续执行下面的 *case* 语句，反之则跳出 *switch* 代码块。
[2]译者注：通过这种方式能很大程序上减少遗忘 break 语句而带来的麻烦

```
3:   is not a vowel
4: S is not a vowel
5: h is not a vowel
6: 'A'
7: w is not a vowel
$
```

注意在代码的最开始部分，有一个 `return 1;` 语句，它可以保证在你没有提供足够的参数时退出程序。使用一个不为 0 的返回值可以通知操作系统该程序发生了错误。任何大于 0 的返回值可以被脚本和其他程序用来检查到底发生了什么。

14.2 让程序出错

要破坏一个 *switch* 语句的正常运行是非常简单的，以下就是一些你可以把事情搞糟的方法:

1. 忘记 *break* 语句，程序将会运行两个或者更多你本不期望运行的代码块。
2. 忘记 default 语句，它就会默默地忽略那些你没考虑到的值。
3. 不小心放一个值为未预期的变量在 *switch* 语句中，比如一个值很怪异的 *int* 值。
4. 在 *switch* 语句中使用未初始化的变量。

你也可以用其他方法来使程序出错，看看你能否破坏它。

14.3 加分习题

1. 写一个新程序，对字母做一些数学操作，使他们成为小写字母，然后去除 switch 中的那些多余的大写字母的分支。
2. 使用 `','` (逗号) 来初始化 *for* 循环中的 *letter*。
3. 使它能处理你使用一个 *for* 循环传递的所有参数。
4. 把这个 *switch* 表达式用 *if* 语句改写，你更喜欢哪一个呢？
5. 在’Y’ 这个分支中，我把 break 放在 *if* 语句外面。如果把它放在 *if* 语句里面会有什么影响？证明一下你的猜想是正确的。

Chapter 15

习题 14：创建和使用函数

目前为止你已经使用了部分由头文件 **stdio.h** 包含的函数。在这个练习中你将会写一些函数并使用一些别的函数。

ex14.c

```c
#include <stdio.h>
#include <ctype.h>

// forward declarations
int can_print_it(char ch);
void print_letters(char arg[]);

void print_arguments(int argc, char *argv[])
{
    int i = 0;

    for(i = 0; i < argc; i++) {
        print_letters(argv[i]);
    }
}

void print_letters(char arg[])
{
    int i = 0;

    for(i = 0; arg[i] != '\0'; i++) {
        char ch = arg[i];

        if(can_print_it(ch)) {
            printf("'%c' == %d ", ch, ch);
        }
27      }
28
29      printf("\n");
30  }
31
32  int can_print_it(char ch)
33  {
34      return isalpha(ch) || isblank(ch);
35  }
36
37
38  int main(int argc, char *argv[])
39  {
40      print_arguments(argc, argv);
41      return 0;
42  }
```

在这个例子中你创建了一个函数，用来打印属于“alpha(字母)”或者“blanks(空白符)”类型的字符和它们对应的 ASCII 码值。以下是代码详解：

ex14.c:2 为了能使用函数 *isalpha* 和 *isblank*, 我们需要包含一个新的头文件。

ex14.c:5-6 你可以在没有实际定义函数之前就告诉 C 你稍后会在程序中使用的函数。这就是“前置声明 (forward declaration)”，它解决了在未定义函数之前就使用函数这个鸡生蛋还是蛋生鸡的问题。

ex14.c:8-15 定义函数 *print_arguments*, 用来打印由 *main* 函数接收到的参数数组的值。

ex14.c:17-30 定义下一个函数 *print_letters*, 它由 *print_arguments* 函数调用, 用来输出每个字母的字符值和它们的 ASCII 码值。

ex14.c:32-35 定义函数 *can_print_it* 用来给调用它的函数 *print_letters* 返回一个真值（0 或者 1），由 `isalpha(ch) || isblan` 决定该值。

ex14.c:38-42 最后，*main* 函数调用了函数 *print_arguments*，整个函数调用链就从这里开始了。

其实我没必要描述各个函数中的实际内容，因为这些都是你在之前的练习中遇到过的。你应该看到的是我用和定义 *main* 函数一样的方式定义了一些函数。唯一的不同是你需要提前告诉 C 那些还没有在文件中遇到但之后要使用的函数，也就是那些代码头部的“前置声明 (forward declaration)”所做的事情。

15.1 你应该看到的结果

要体验这个程序，你只需给它提供不同的参数即可，这些参数将被传递到你的函数里边。以下是我的演示：

ex14 的输出

```
$ make ex14
cc -Wall -g    ex14.c   -o ex14

$ ./ex14
'e' == 101 'x' == 120

$ ./ex14 hi this is cool
'e' == 101 'x' == 120
'h' == 104 'i' == 105
't' == 116 'h' == 104 'i' == 105 's' == 115
'i' == 105 's' == 115
'c' == 99 'o' == 111 'o' == 111 'l' == 108

$ ./ex14 "I go 3 spaces"
'e' == 101 'x' == 120
'I' == 73 ' ' == 32 'g' == 103 'o' == 111 ' ' == 32 ' ' == 32 's' == 115 'p' == 112
    ↪'a' == 97 'c' == 99 'e' == 101 's' == 115
$
```

函数 *isalpha* 和 *isblank* 所做的工作就是来判断一个输入的字符是否是字母 (letter) 或者空白符 (blank)。我最后一个示例中它打印了除了'3' 以外的所有字符，因为'3' 是一个数字[1]。

15.2 让程序出错

有两种不同的方式来破坏这个程序:

1. 把前置声明去除以此来迷惑编译器，它将会抱怨找不到函数 *can_print_it* 和 *print_letters*。
2. 当你在 *main* 函数中调用 *print_arguments* 时对 *argc* 加 1，这样就越过了 *argv* 数组的最后一个参数[2]。

15.3 加分习题

1. 重构这些代码似的你能使用更少的函数完成相同的工作。比如，你真的需要 *can_print_it* 这个函数吗？
2. 让 *print_arguments* 函数调用 *strlen* 函数来获知每个字符串参数的长度，然后把长度作为参数传递给 *print_letters*。然后重写 *print_letters* 函数使它只处理这个固定长度的字符串，这样就不需要依赖于 '\0' 来决定字符串长度了。
3. 使用 *man* 来查询 *isalpha* 和 *isblank* 函数的信息。用其他类似的函数来实现只打印数字或其他字符的功能。[3]。
4. 去看看不同的人们格式化函数的不同风格。不要使用“K&R 语法”，这是一种过时也是让人迷惑的方式。不过你要

[1]译者注：这里的 3 是指数字 3 而不是 ASCII 表中的字符'3'

[2]译者注：这时候函数 *print_arguments* 函数最后将会访问越界

[3]译者注：man 程序是 Unix 和类 Unix 系统上一个软件文档描述程序，使用它可以用来查询库函数等信息。通常在控制台执行。

了解这种风格的成因，这样等你碰到喜欢这种风格的人，你就知道他们为什么喜欢了。

Chapter 16

Exercise 15: 指针啊指针，恐怖的指针

指针是 C 语言中闻名遐迩而又神秘莫测的东西。我将教给你们一些处理指针所用的词汇，从而揭开指针的神秘面纱。事实上指针并不复杂，只是常常被以各种方式滥用，导致它显得十分难用。如果你尽量避免那些愚蠢的使用方法，指针实际上是相当容易的。

为了便于讨论，我写了一个简单的小程序，能够用三种不同的方式打印一组人员年龄: [1]

ex15.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    // create two arrays we care about
    int ages[] = {23, 43, 12, 89, 2};
    char *names[] = {
        "Alan", "Frank",
        "Mary", "John", "Lisa"
    };
    // safely get the size of ages
    int count = sizeof(ages) / sizeof(int);
    int i = 0;

    // first way using indexing
    for(i = 0; i < count; i++) {
        printf("%s has %d years alive.\n",
                names[i], ages[i]);
    }

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

[1]记住，学习本书时你要输入这些程序，也许一时不能理解，但请尝试在我解释以前自己弄明白。

```c
22
23      // setup the pointers to the start of the arrays
24      int *cur_age = ages;
25      char **cur_name = names;
26
27      // second way using pointers
28      for(i = 0; i < count; i++) {
29          printf("%s is %d years old.\n",
30                  *(cur_name+i), *(cur_age+i));
31      }
32
33      printf("---\n");
34
35      // third way, pointers are just arrays
36      for(i = 0; i < count; i++) {
37          printf("%s is %d years old again.\n",
38                  cur_name[i], cur_age[i]);
39      }
40
41      printf("---\n");
42
43      // fourth way with pointers in a stupid complex way
44      for(cur_name = names, cur_age = ages;
45              (cur_age - ages) < count;
46              cur_name++, cur_age++)
47      {
48          printf("%s lived %d years so far.\n",
49                  *cur_name, *cur_age);
50      }
51
52      return 0;
53  }
```

在解释指针的工作原理之前，我们先来逐行分析这个程序，了解它的工作过程。在你阅读这些详细描述的同时，试着在纸上写下答案，留待和我之后的解释相比较，看看是否相同。

ex15.c:6-10 创建两个数组, *ages* 用以存储一些 *int* 数据，而数组 *names* 则存储一些字符串。

ex15.c:12-13 稍候 *for* 循环要用到的一些变量。

ex15.c:16-19 你懂的，循环访问两个数组并打印出每人的年龄。此处使用 *i* 作为数组的索引。

ex15.c:24 创建指针，指向 *ages*。注意 `int *` 的作用是创建 “指向整数类的指针”。这点和 “指向字符类的指针” `char *` 相似，而字符串就是一个字符的数组。发现相似之处了嘛？

ex15.c:25 创建一个指针，指向 *names*。`char *` 已经是一个 “指向字符类的指针”，也就是一个字符串。而 *names* 是二维的，所以你需要的两级指针，也就是 “（字符指针）的指针” `char **`。请推敲研究此处，尝试解释给自己听。

ex15.c:28-31 依次访问 *ages* 和 *names*，但这次使用指针加上 *i* 的偏移量. 写为 `*(cur_name+i)` 和`name[i]` 是等价的，你可以将它读成“(指针 *cur_name* 加 i) 的值”。

ex15.c:35-39 这显示了 C 语言如何奇妙的将指针和数组当作同一种事物来处理，仍然用数组的语法来访问数组，只要换上指针的名称就可以了。C 语言自然能分辨。

ex15.c:44-50 像前两个一样，又一个让人抓狂的循环，只是它用了不同的指针算法：

ex15.c:44 通过将 *cur_name* 和 *cur_age* 设置到 *names* 和 *ages* 数组的起点来初始化 *for* 循环。

ex15.c:45 然后 *for-loop* 的测试部分比较了指针 *cur_age* 到起点 *ages* 的 距离。为何这样可行呢？

ex15.c:46 接下来 *for-loop* 的增进部分同时增进了 *cur_name* 和 *cur_age*，以便二者指向数组 *name* 和 *age* 的下一个元素。

ex15.c:48-49 指针 *cur_name* 和 *cur_age* 现在数组中正在操作的元素，我们只用 `*cur_name` 和 `*cur_age` 就可以将他们打印出来，也就是“任何 *cur_name* 指向位置的值”。

这段程序看似简单，却包含大量信息，它的目的是使你在看我解释之前，自己尝试理解指针。在你写下自己想法之前，请不要继续阅读

16.1 你应该看到的结果

运行程序之后，请你尝试根据每行打印结果追溯源代码。必要时，更改 *printf* 以确保你得到正确的行号。

ex15 输出

```
$ make ex15
cc -Wall -g    ex15.c   -o ex15
$ ./ex15
Alan has 23 years alive.
Frank has 43 years alive.
Mary has 12 years alive.
John has 89 years alive.
Lisa has 2 years alive.
---
Alan is 23 years old.
Frank is 43 years old.
Mary is 12 years old.
John is 89 years old.
Lisa is 2 years old.
---
Alan is 23 years old again.
Frank is 43 years old again.
Mary is 12 years old again.
John is 89 years old again.
```

```
Lisa is 2 years old again.
---
Alan lived 23 years so far.
Frank lived 43 years so far.
Mary lived 12 years so far.
John lived 89 years so far.
Lisa lived 2 years so far.
$
```

16.2 解释指针

当你输入诸如 `ages[i]` 的东西时，其实是在数组 *ages* 中“检索”, 借助 *i* 中存储的数字实现所需功能。当 *i* 被设为 0 时，相当于输入 `ages[0]`。因为数字 *i* 表示我们在 `age[0]` 需要访问的位置，所以一直被称作“索引”。它也可以称作“地址”，比如说“我想要知道数组 *ages* 中地址 *i* 的整数是多少？”

如果 *i* 是一个索引，那么 *ages* 又是什么呢？对于 C 语言来说 *ages* 是电脑内存中，所有整数的起始位置。它也是一个地址，C 编译器将会用它替换任何带有 ages 中第一个整数地址的 *ages*。另一种思考 *ages* 的方法是，它是“ages 中第一个整数的地址”。诀窍在于 *ages* 是整个电脑里的内存地址。不像 *i* 只是 *ages* 的内部地址。*ages* 的数组名就是电脑里的实际地址。

至此我们得到一种认识：C 语言将整个电脑视为一个巨大的字节数组。显然这不是很管用，但是接下来 C 在这个巨大的字节数组之上加上了类型和这些类型大小这类概念。从前面的练习中你已经见过这是如何运行的，现在你可以开始了解 C 是如何对数组进行以下操作的：

1. 在你的电脑上创建一块内存区域。
2. 将该内存块起始初“指向”*ages* 名称。
3. 通过基础地址 *ages* “检索”该内存块并获取 *i* 个字节以外的元素。
4. 将 *ages+i* 处的地址转化为大小合适的有效整型，以便索引正常工作并返回你所需求的：索引 *i* 处的整型值。

如果你能取得基础地址，比如 *ages*，再“加”上另一个地址比如 *i* 以产生新的地址，那么你是否能得到一个始终指向此地址的某种东西呢？正式如此，你所得到的这种东西，就叫做“指针”。这就是指针 *cur_age* 和 *cur_name* 所做的事情。它们是指向你电脑内存里 *ages* 和 *names* 地址的变量。范例程序将它们移来移去，通过相关计算获得内存存储数值。在其中一例，它们只是将 *i* 加到 *cur_age*，这就相当于 `array[i]`。在最后一个 *for* 循环 里，两个指针没有 *i* 的帮忙也能自行移动。这个循环中，指针被看作是数组和整数偏移量的整合。

指针仅仅是指向电脑内存的一个地址，明确类型之后你就可以得到大小正确的数据。这有点像 *ages* 与 *i* 合二为一的数据类型。C 语言知道指针指向何处，知道所指数据类型，该类型数据大小以及如何为你获取它们。正如 *i* 一样，你可以让它们自增、自减，或者做加、减运算。也可以像 *ages* 一样，你通过它们得到数值，输入新值，进行所有数组操作。

指针的目的，在于当数组不能完全胜任的时候，手动检索一块内存区域。只要是不使用数组的情况，你大可放心使用指针。然后又些时候你不得不操作原始内存区域，这才是指针的任务。指针为你操作内存提供了原始、直接的连接方式。

本节最后一件要知道的事是，你既可以使用数组操作语法，也可以使用指针操作语法来编写程序。你可以用指针指向某个东西，然后用数组语法接入它，也可以利用指针算数来操作数组。在以上的范例程序中，我演示了这一点，它也

是 C 语言的基本特征：指针和数组（基本上）是同一回事。

16.3 指针使用练习

在 C 语言中有四种主要的指针处理方法：

1. 向系统请求一块内存作为指针工作区域。这包括字符串和你还没见过的 *structs*。
2. 利用指针向函数传递一大块内存（比如字符串和数组），这样你就不必传递整个数据。
3. 掌握函数地址以便于动态回调。
4. 内存区域的复杂扫描，比如从网络套接字中将字节转换成数据结构或者解析文件。

对于几乎其他所有你能看到的指针使用方法，应该都是作为数组。在 C 语言编程的初期，由于编译器对数组的优化还很糟糕，人们便使用指针来加速程序的运行。而现今，不同的数组与指针语法被翻译成相同的机器码，并被同样的优化，所以指针并不是必须的。相反，在能使用数组的情况下应该尽量使用，而只在必须使用指针进行性能优化的时候才使用指针。

16.4 指针词典

现在我要教给你一个小小的指针词典，用来读写指针。任何遇到复杂指针声明的时候就来参考它，然后一点点的解决问题。(或者如果代码本身不太好就直接放弃使用):

type *ptr “一个叫做 ptr 的 type 型指针”

***ptr** “ptr 所指向地址对应的值”

***(ptr + i)** “ptr 所指地址加 i 的位置的值”

&thing “thing 的地址”

type *ptr = &thing “将名为 ptr 的 type 型指针设置到 thing 的地址”

ptr++ “增进 ptr 的指向位置”

我们将用这本简单的词典解决往后书中遇到的所有指针问题。

16.5 如何让程序出错

你只需将指针指向错误的地方就能让这个程序出错。

1. 尝试将 *cur_age* 指向 *names*。你需要用 C 语言中的指针映射 (cast) 来强制执行，研究一下具体需要怎样做。
2. 尝试让最后一个 *for* 循环中，用各种方法让算术出错。
3. 尝试重写这些循环，从结尾向开头访问数组。这比看上去要困难。

16.6 加分习题

1. 将程序中所有数组重写为指针。
2. 将程序中所有指针重写为数组。
3. 回顾以往使用数组的程序，尝试使用指针替代。
4. 只用指针处理命令行参数，方法与本例中处理 *names* 相似。
5. 尝试指针方法获取各种东西的地址与值。
6. 在结尾加入另一个 *for* 循环，打印指针被调用时地址。使用 *printf* 打印时将会用到 %p 格式。
7. 重写程序，每种打印方式使用一个函数。尝试将指针传递给函数以便它们获得数据。记住你可以声明一个函数接受指针，但只能按数组的方式使用它。
8. 将 *for* 循环改为 *while* 循环，看看对于不同种类的指针使用，哪种循环效果更好。

Chapter 17

Exercise 16: Structs And Pointers To Them

In this exercise you'll learn how to make a `struct`, point a pointer at them, and use them to make sense of internal memory structures. I'll also apply the knowledge of pointers from the last exercise and get you constructing these structures from raw memory using `malloc`.

As usual, here's the program we'll talk about, so type it in and make it work:

ex16.c

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

struct Person {
    char *name;
    int age;
    int height;
    int weight;
};

struct Person *Person_create(char *name, int age, int height, int weight)
{
    struct Person *who = malloc(sizeof(struct Person));
    assert(who != NULL);

    who->name = strdup(name);
    who->age = age;
    who->height = height;
    who->weight = weight;

    return who;
}

void Person_destroy(struct Person *who)
{
    assert(who != NULL);

    free(who->name);
    free(who);
}

void Person_print(struct Person *who)
{
    printf("Name: %s\n", who->name);
    printf("\tAge: %d\n", who->age);
    printf("\tHeight: %d\n", who->height);
    printf("\tWeight: %d\n", who->weight);
}

int main(int argc, char *argv[])
{
    // make two people structures
    struct Person *joe = Person_create(
            "Joe Alex", 32, 64, 140);

    struct Person *frank = Person_create(
            "Frank Blank", 20, 72, 180);

    // print them out and where they are in memory
    printf("Joe is at memory location %p:\n", joe);
    Person_print(joe);

    printf("Frank is at memory location %p:\n", frank);
    Person_print(frank);

    // make everyone age 20 years and print them again
    joe->age += 20;
    joe->height -= 2;
    joe->weight += 40;
    Person_print(joe);

    frank->age += 20;
```

```c
    frank->weight += 20;
    Person_print(frank);

    // destroy them both so we clean up
    Person_destroy(joe);
    Person_destroy(frank);

    return 0;
}
```

To describe this program, I'm going to use a different approach than before. I'm not going to give you a line-by-line breakdown of the program, but I'm going to make *you* write it. I'm going to give you a guide through the program based on the parts it contains, and your job is to write out what each line does.

includes I include some new header files here to gain access to some new functions. What does each give you?

struct Person This is where I'm creating a structure that has 4 elements to describe a person. The final result is a new compound type that lets me reference these elements all as one, or each piece by name. It's similar to a row of a database table or a class in an OOP language.

function Person_create I need a way to create these structures so I've made a function to do that. Here's the important things this function is doing:

1. I use *malloc* for "memory allocate" to ask the OS to give me a piece of raw memory.
2. I pass to *malloc* the `sizeof(struct Person)` which calculates the total size of the struct, given all the fields inside it.
3. I use *assert* to make sure that I have a valid piece of memory back from malloc. There's a special constant called *NULL* that you use to mean "unset or invalid pointer". This *assert* is basically checking that malloc didn't return a NULL invalid pointer.
4. I initialize each field of *struct Person* using the `x->y` syntax, to say what part of the struct I want to set.
5. I use the *strdup* function to duplicate the string for the name, just to make sure that this structure actually owns it. The *strdup* actually is like *malloc* and it also copies the original string into the memory it creates.

function Person_destroy If I have a create, then I always need a destroy function, and this is what destroys *Person* structs. I again use *assert* to make sure I'm not getting bad input. Then I use the function *free* to return the memory I got with *malloc* and *strdup*. If you don't do this you get a "memory leak".

function Person_print I then need a way to print out people, which is all this function does. It uses the same `x->y` syntax to get the field from the struct to print it.

function main In the main function I use all the previous functions and the *struct Person* to do the following:

1. Create two people, *joe* and *frank*.
2. Print them out, but notice I'm using the `%p` format so you can see *where* the program has actually put your struct in memory.
3. Age both of them by 20 years, with changes to their body too.

4. Print each one after aging them.
5. Finally destroy the structures so we can clean up correctly.

Go through this description carefully, and do the following:

1. Look up every function and header file you don't know about. Remember that you can usually do `man 2 function` or `man 3 function` and it'll tell you about it. You can also search online for the information.
2. Write a *comment* above each and every single line saying what the line does in English.
3. Trace through each function call and variable so you know where it comes from in the program.
4. Look up any symbols you don't know as well.

17.1 What You Should See

After you augment the program with your description comments, make sure it really runs and produces this output:

ex16 output

```
$ make ex16
cc -Wall -g    ex16.c   -o ex16

$ ./ex16
Joe is at memory location 0xeba010:
Name: Joe Alex
        Age: 32
        Height: 64
        Weight: 140
Frank is at memory location 0xeba050:
Name: Frank Blank
        Age: 20
        Height: 72
        Weight: 180
Name: Joe Alex
        Age: 52
        Height: 62
        Weight: 180
Name: Frank Blank
        Age: 40
        Height: 72
        Weight: 200
```

17.2 Explaining Structures

If you've done the work I asked you then structures should be making sense, but let me explain them explicitly just to make sure you've understood it.

A structure in C is a collection of other data types (variables) that are stored in one block of memory but let you access each variable independently by name. They are similar to a record in a database table, or a very simplistic class in an object oriented language. We can break one down this way:

1. In the above code, you make a `struct` that has the fields you'd expect for a person: name, age, weight, height.
2. Each of those fields has a type, like `int`.
3. C then packs those together so they can all be contained in one single `struct`.
4. The `struct Person` is now a *compound data type*, which means you can now refer to `struct Person` in the same kinds of expressions you would other data types.
5. This lets you pass the whole cohesive grouping to other functions, as you did with `Person_print`.
6. You can then access the individual parts of a `struct` by their names using `x->y` if you're dealing with a pointer.
7. There's also a way to make a struct that doesn't need a pointer, and you use the `x.y` (period) syntax to work with it. You'll do this in the Extra Credit.

If you didn't have `struct` you'd need to figure out the size, packing, and location of pieces of memory with contents like this. In fact, in most early assembler code (and even some now) this is what you do. With C you can let C handle the memory structuring of these compound data types and then focus on what you do with them.

17.3 How To Break It

With this program the ways to break it involve how you use the pointers and the `malloc` system:

1. Try passing `NULL` to `Person_destroy` to see what it does. If it doesn't abort then you must not have the `-g` option in your Makefile's `CFLAGS`.
2. Forget to call `Person_destroy` at the end, then run it under `Valgrind` to see it report that you forgot to free the memory. Figure out the options you need to pass to `Valgrind` to get it to print how you leaked this memory.
3. Forget to free `who->name` in `Person_destroy` and compare the output. Again, use the right options to see how `Valgrind` tells you exactly where you messed up.
4. This time, pass `NULL` to `Person_print` and see what `Valgrind` thinks of that.
5. You should be figuring out that `NULL` is a quick way to crash your program.

17.4 Extra Credit

In this exercise I want you to attempt something difficult for the extra credit: Convert this program to *not* use pointers and `malloc`. This will be hard, so you'll want to research the following:

1. How to create a *`struct`* on the *stack*, which means just like you've been making any other variable.
2. How to initialize it using the `x.y` (period) character instead of the `x->y` syntax.
3. How to pass a structure to other functions without using a pointer.

Chapter 18

Exercise 17: Heap And Stack Memory Allocation

In this exercise you're going to make a big leap in difficulty and create an entire small program to manage a database. This database isn't very efficient and doesn't store very much, but it does demonstrate most of what you've learned so far. It also introduces memory allocation more formally and gets you started working with files. We use some file I/O functions, but I won't be explaining them too well so you can try to figure them out first.

As usual, type this whole program in and get it working, then we'll discuss:

ex17.c

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

#define MAX_DATA 512
#define MAX_ROWS 100

struct Address {
    int id;
    int set;
    char name[MAX_DATA];
    char email[MAX_DATA];
};

struct Database {
    struct Address rows[MAX_ROWS];
};
```

```c

struct Connection {
    FILE *file;
    struct Database *db;
};

void die(const char *message)
{
    if(errno) {
        perror(message);
    } else {
        printf("ERROR: %s\n", message);
    }

    exit(1);
}

void Address_print(struct Address *addr)
{
    printf("%d %s %s\n",
            addr->id, addr->name, addr->email);
}

void Database_load(struct Connection *conn)
{
    int rc = fread(conn->db, sizeof(struct Database), 1, conn->file);
    if(rc != 1) die("Failed to load database.");
}

struct Connection* Database_open(const char *filename, char mode)
{
    struct Connection *conn = malloc(sizeof(struct Connection));
    if(!conn) die("Memory error");

    conn->db = malloc(sizeof(struct Database));
    if(!conn->db) die("Memory error");

    if(mode == 'c') {
        conn->file = fopen(filename, "w");
    } else {
        conn->file = fopen(filename, "r+");

        if(conn->file) {
            Database_load(conn);
        }
    }

    if(!conn->file) die("Failed to open the file");

    return conn;
}

void Database_close(struct Connection *conn)
{
    if(conn) {
        if(conn->file) fclose(conn->file);
        if(conn->db) free(conn->db);
        free(conn);
    }
}

void Database_write(struct Connection *conn)
{
    rewind(conn->file);

    int rc = fwrite(conn->db, sizeof(struct Database), 1, conn->file);
    if(rc != 1) die("Failed to write database.");

    rc = fflush(conn->file);
    if(rc == -1) die("Cannot flush database.");
}

void Database_create(struct Connection *conn)
{
    int i = 0;

    for(i = 0; i < MAX_ROWS; i++) {
        // make a prototype to initialize it
        struct Address addr = {.id = i, .set = 0};
        // then just assign it
        conn->db->rows[i] = addr;
    }
}

void Database_set(struct Connection *conn, int id, const char *name, const char *email)
{
    struct Address *addr = &conn->db->rows[id];
    if(addr->set) die("Already set, delete it first");

    addr->set = 1;
    // WARNING: bug, read the "How To Break It" and fix this
    char *res = strncpy(addr->name, name, MAX_DATA);
    // demonstrate the strncpy bug
    if(!res) die("Name copy failed");

    res = strncpy(addr->email, email, MAX_DATA);
    if(!res) die("Email copy failed");
}

void Database_get(struct Connection *conn, int id)
{
    struct Address *addr = &conn->db->rows[id];

    if(addr->set) {
        Address_print(addr);
    } else {
        die("ID is not set");
    }
}

void Database_delete(struct Connection *conn, int id)
{
    struct Address addr = {.id = id, .set = 0};
    conn->db->rows[id] = addr;
}

void Database_list(struct Connection *conn)
{
    int i = 0;
    struct Database *db = conn->db;

    for(i = 0; i < MAX_ROWS; i++) {
        struct Address *cur = &db->rows[i];

        if(cur->set) {
            Address_print(cur);
        }
    }
}

int main(int argc, char *argv[])
{
    if(argc < 3) die("USAGE: ex17 <dbfile> <action> [action params]");

    char *filename = argv[1];
    char action = argv[2][0];
    struct Connection *conn = Database_open(filename, action);
    int id = 0;

    if(argc > 3) id = atoi(argv[3]);
    if(id >= MAX_ROWS) die("There's not that many records.");

    switch(action) {
        case 'c':
            Database_create(conn);
            Database_write(conn);
            break;

        case 'g':
            if(argc != 4) die("Need an id to get");

            Database_get(conn, id);
            break;

        case 's':
            if(argc != 6) die("Need id, name, email to set");

            Database_set(conn, id, argv[4], argv[5]);
            Database_write(conn);
            break;

        case 'd':
            if(argc != 4) die("Need id to delete");

            Database_delete(conn, id);
            Database_write(conn);
            break;

        case 'l':
            Database_list(conn);
            break;
        default:
            die("Invalid action, only: c=create, g=get, s=set, d=del, l=list");
    }

    Database_close(conn);
```

```c

    return 0;
}
```

In this program I am using a set of structures to create a simple database for an address book. In it I'm using some things you've never seen, so you should go through it line-by-line, explain what each line does, and look up any functions you do not recognize. There are few key things I'm doing that you should pay attention to as well:

#define for constants I use another part of the "C Pre-Processor" to create constant settings of *MAX_DATA* and *MAX_ROWS*. I'll cover more of what the CPP does, but this is a way to create a constant that will work reliably. There's other ways but they don't apply in certain situations.

Fixed Sized Structs The *Address* struct then uses these constants to create a piece of data that is fixed in size making it less efficient, but easier to store and read. The *Database* struct is then also fixed size because it is a fixed length array of *Address* structs. That lets you write the whole thing to disk in one move later on.

die function to abort with an error In a small program like this you can make a single function that kills the program with an error if there's anything wrong. I call this *die*, and it's used after any failed function calls or bad inputs to exit with an error using *exit*.

errno and perror() for error reporting When you have an error return from a function, it will usually set an "external" variable called *errno* to say exactly what error happened. These are just numbers, so you can use *perror* to "print the error message".

FILE functions I'm using all new functions like *fopen*, *fread*, *fclose*, and *rewind* to work with files. Each of these functions works on a *FILE* struct that's just like your structs, but it's defined by the C standard library.

nested struct pointers There's use of nested structures and getting the address of array elements that you should study. Specifically code like `&conn->db->rows[i]` which reads "get the *i* element of *rows*, which is in *db*, which is in *conn*, then get the address of (*&*) it".

copying struct prototypes best shown in *Database_delete*, you can see I'm using a temporary local *Address*, initializing its *id* and *set* fields, and then simply copying it into the *rows* array by assigning it to the element I want. This trick makes sure that all fields but *set* and *id* are initialized to 0s and is actually easier to write. Incidentally, you shouldn't be using *memcpy* to do these kinds of struct copying operations. Modern C allows you to simply assign one struct to another and it'll handle the copying for you.

processing complex arguments I'm doing some more complex argument parsing, but this isn't really the best way to do it. We'll get into better option parsing later in the book.

converting strings to ints I use the *atoi* function to take the string for the id on the command line and convert it to the *int id* variable. Read up on this function and similar ones.

allocating large data on the "heap" The whole point of this program is that I'm using *malloc* to ask the OS for a large amount of memory to work with when I create the *Database*. I cover this in more detail below.

NULL is 0 so boolean works In many of the checks I'm testing that a pointer is not NULL by simply doing `if(!ptr) die("fail!` this is valid because NULL is the same as 0 so it will evaluate to false. You could be explicit and say `if(ptr == NULL) die("fail` as well.

18.1 What You Should See

You should spend as much time as you can testing that it works, and running it with *Valgrind* to confirm you've got all the memory usage right. Here's a session of me testing it normally and then using *Valgrind* to check the operations:

ex17 output

```
$ make ex17
cc -Wall -g    ex17.c   -o ex17
$ ./ex17 db.dat c
$ ./ex17 db.dat s 1 zed zed@zedshaw.com
$ ./ex17 db.dat s 2 frank frank@zedshaw.com
$ ./ex17 db.dat s 3 joe joe@zedshaw.com
$
$ ./ex17 db.dat l
1 zed zed@zedshaw.com
2 frank frank@zedshaw.com
3 joe joe@zedshaw.com
$ ./ex17 db.dat d 3
$ ./ex17 db.dat l
1 zed zed@zedshaw.com
2 frank frank@zedshaw.com
$ ./ex17 db.dat g 2
2 frank frank@zedshaw.com
$
$ valgrind --leak-check=yes ./ex17 db.dat g 2
# cut valgrind output...
$
```

The actual output of *Valgrind* is taken out since you should be able to detect it.

Note 5 *OSX Valgrind "Leaks"*

Valgrind will report that you're leaking small blocks of memory, but sometimes it's just over-reporting from OSX's internal APIs. If you see it showing leaks that aren't inside your code then just ignore them.

18.2 Heap vs. Stack Allocation

You kids these days have it great. You play with your Ruby or Python and just make objects and variables without any care for where they live. You don't care if it's on the "stack", and the heap? Fuggedaboutit. You don't even know, and you know what, chances are your language of choice doesn't even put the variables on stack at all. It's all heap, and you don't even *know* if it is.

C is different because it's using the real CPU's actual machinery to do its work, and that involves a chunk of ram called the stack and another called the heap. What's the difference? It all depends on where you get the storage.

The heap is easier to explain as it's just all the remaining memory in your computer, and you access it with the function `malloc` to get more. Each time you call `malloc`, the OS uses internal functions to register that piece of memory to you, and then returns a pointer to it. When you're done with it, you use `free` to return it to the OS so that it can be used by other programs. Failing to do this will cause your program to "leak" memory, but `Valgrind` will help you track these leaks down.

The stack is a special region of memory that stores temporary variables each function creates as locals to that function. How it works is each argument to a function is "pushed" onto the stack, and then used inside the function. It is really a stack data structure, so the last thing in is the first thing out. This also happens with all local variables like `char action` and `int id` in `main`. The advantage of using a stack for this is simply that, when the function exits, the C compiler "pops" these variables off the stack to clean up. This is simple and prevents memory leaks if the variable is on the stack.

The easiest way to keep this straight is with this mantra: If you didn't get it from `malloc` or a function that got it from `malloc`, then it's on the stack.

There's three primary problems with stacks and heaps to watch for:

1. If you get a block of memory from `malloc`, and have that pointer on the stack, then when the function exits, the pointer will get popped off and lost.
2. If you put too much data on the stack (like large structs and arrays) then you can cause a "stack overflow" and the program will abort. In this case, use the heap with `malloc`.
3. If you take a pointer to something on the stack, and then pass that or return it from your function, then the function receiving it will "segmentation fault" (segfault) because the actual data will get popped off and disappear. You'll be pointing at dead space.

This is why in the program I've created a `Database_open` that allocates memory or dies, and then a `Database_close` that frees everything. If you create a "create" function, that makes the whole thing or nothing, and then a "destroy" function that cleans up everything safely, then it's easier to keep it all straight.

Finally, when a program exits the OS will clean up all the resources for you, but sometimes not immediately. A common idiom (and one I use in this exercise) is to just abort and let the OS clean up on error.

18.3 How To Break It

This program has a lot of places you can break it, so try some of these but also come up with your own:

1. The classic way is to remove some of the safety checks such that you can pass in arbitrary data. For example, if you remove the check on line 159 that prevents you from passing in any record number.
2. You can also try corrupting the data file. Open it in any editor and change random bytes then close it.
3. You could also find ways to pass bad arguments to the program when it's run, such as getting the file and action backwards will make it create a file named after the action, then do an action based on the first character.
4. There is a bug in this program because of `strncpy` being poorly designed. Go read about `strncpy` then try to find out what happens when the `name` or `address` you give is *greater* than 512 bytes. Fix this by simply forcing the last

character to `'\0'` so that it's always set no matter what (which is what strncpy should do).

5. In the extra credit I have you augment the program to create arbitrary size databases. Try to see what the biggest database is before you cause the program to die for lack of memory from `malloc`.

18.4 Extra Credit

1. The `die` function needs to be augmented to let you pass the `conn` variable so it can close it and clean up.
2. Change the code to accept parameters for `MAX_DATA` and `MAX_ROWS`, store them in the `Database` struct, and write that to the file, thus creating a database that can be arbitrarily sized.
3. Add more operations you can do on the database, like `find`.
4. Read about how C does it's struct packing, and then try to see why your file is the size it is. See if you can calculate a new size after adding more fields.
5. Add some more fields to the `Address` and make them searchable.
6. Write a shell script that will do your testing automatically for you by running commands in the right order. Hint: Use `set -e` at the top of a `bash` to make it abort the whole script if any command has an error.
7. Try reworking the program to use a single global for the database connection. How does this new version of the program compare to the other one?
8. Go research ”stack data structure” and write one in your favorite language, then try to do it in C.

Chapter 19

习题 18: 指向函数的指针

C 中的函数（译注：函数名即为指针名）实际上只是指向程序中的代码的指针。就像你创建的指向结构体（structs），字符串（strings），和数组（arrays）的指针一样，你也能用指针指向函数。这么做的主要用途就是向其他函数传递" 回调函数"（callbacks），或者传递给模拟的类与对象。这个练习中，我们会用到一些回调（callback）函数，在下一个中我们会做一个简单的对象系统。

函数指针的格式看起来像这样:

`int (*POINTER_NAME)(int a, int b)`

这些是帮助你想起如何写出函数指针的方法:

1. 写一个普通的函数声明: `int callme(int a, int b)`
2. 用指针的语法格式包围函数名：`int (*callme)(int a, int b)`
3. 把名字改为指针名：`int (*compare_cb)(int a, int b)`

记住这个的关键就是，当你用这种方法做完后，指针的变量名叫做 *compare_cb*然后你就像使用函数一样使用它。这和指向数组的指针可以像被指向的数组那样使用相似。指向函数的指针可以向被指向的函数那样使用，只是名字不用而已。

使用行函数指针（*Raw Function Pointer*）

```
int (*tester)(int a, int b) = sorted_order;
printf("TEST: %d is same as %d\n", tester(2, 3), sorted_order(2, 3));
```

即使函数指针返回一个指向别的什么东西的指针这也会正常工作:

1. 写下了: `char *make_coolness(int awesome_levels)`
2. 包裹它: `char *(*make_coolness)(int awesome_levels)`
3. 重命名: `char *(*coolness_cb)(int awesome_levels)`

接下来需要解决的在使用函数指针时的问题是很难把它们作为形参传给函数，就像你想把一个回调函数传给另一

个函数的时候那样。解决方法就是使用 *typedef* — C 语言中为其它复杂类型重新命名的关键字。唯一需要做的就是把 *typedef* 放到声明函数指针语法的前面，之后你就可以像使用一种类型一样的使用函数指针名。我用下面的练习代码演示一下:

ex18.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <errno.h>
#include <string.h>

/** Our old friend die from ex17. */
void die(const char *message)
{
    if(errno) {
        perror(message);
    } else {
        printf("ERROR: %s\n", message);
    }

    exit(1);
}

// a typedef creates a fake type, in this
// case for a function pointer
typedef int (*compare_cb)(int a, int b);

/**
 * A classic bubble sort function that uses the
 * compare_cb to do the sorting.
 */
int *bubble_sort(int *numbers, int count, compare_cb cmp)
{
    int temp = 0;
    int i = 0;
    int j = 0;
    int *target = malloc(count * sizeof(int));

    if(!target) die("Memory error.");

    memcpy(target, numbers, count * sizeof(int));

    for(i = 0; i < count; i++) {
        for(j = 0; j < count - 1; j++) {
            if(cmp(target[j], target[j+1]) > 0) {
                temp = target[j+1];
                target[j+1] = target[j];
                target[j] = temp;
            }
        }
    }

    return target;
}

int sorted_order(int a, int b)
{
    return a - b;
}

int reverse_order(int a, int b)
{
    return b - a;
}

int strange_order(int a, int b)
{
    if(a == 0 || b == 0) {
        return 0;
    } else {
        return a % b;
    }
}

/**
 * Used to test that we are sorting things correctly
 * by doing the sort and printing it out.
 */
void test_sorting(int *numbers, int count, compare_cb cmp)
{
    int i = 0;
    int *sorted = bubble_sort(numbers, count, cmp);

    if(!sorted) die("Failed to sort as requested.");

    for(i = 0; i < count; i++) {
        printf("%d ", sorted[i]);
```

```
83         }
84         printf("\n");
85 
86         free(sorted);
87     }
88 
89 
90     int main(int argc, char *argv[])
91     {
92         if(argc < 2) die("USAGE: ex18 4 3 1 5 6");
93 
94         int count = argc - 1;
95         int i = 0;
96         char **inputs = argv + 1;
97 
98         int *numbers = malloc(count * sizeof(int));
99         if(!numbers) die("Memory error.");
100 
101        for(i = 0; i < count; i++) {
102            numbers[i] = atoi(inputs[i]);
103        }
104 
105        test_sorting(numbers, count, sorted_order);
106        test_sorting(numbers, count, reverse_order);
107        test_sorting(numbers, count, strange_order);
108 
109        free(numbers);
110 
111        return 0;
112    }
```

在这个程序中，你创建了一个可以通过比较回调函数（comparison callback）排序整型数组的动态排序算法。这是修改后的程序，你可以清晰的理解它:

ex18.c:1-6 所有我们调用的函数都需要 include 包含。

ex18.c:7-17 这是我们前一个练习中的 *die* 函数，我们会用它来做错误检查。

ex18.c:21 在这里使用了 *typedef*，稍后我们会像在 *bubble_sort* 和 *test_sorting* 中使用 *int* 或 *char* 一样把 *compare_cb* 当做类型使用。

ex18.c:27-49 一种冒泡排序（bubble sort）的实现，这是一种排序整型数效率很差的方法。这个函数包含:

> **ex18.c:27** 这个我为最后一个参数 *cmp* 使用 *typedef* 类型 *compare_cb*。这是一个为了排序而返回的两个整型的比较结果的函数。
>
> **ex18.c:29-34** 先是在栈上创建变量的一般方法，接着是在堆上使用 *malloc* 创建整型数组的方法。确保你知道

`count * sizeof(int)` 做的什么事。

ex18.c:38 冒泡排序的外层循环。

ex18.c:39 冒泡排序的内层循环。

ex18.c:40 现在我像调用一般函数一样的调用 *cmp* 这个回调函数，但是用我们定义好的一些名字代替，只是指向它的指针。这就让调用器（caller）传递近任何它们想要的东西，只要满足 *compare_cb typedef* 的“特征”（signature）。

ex18.c:41-43 冒泡排序的实际需要做的交换操作。

ex18.c:48 最后返回新创建并排序的结果数组 *target*。

ex18.c:51-68 *compare_cb* 的三种不同的函数类型，我们需要和我们创建的 *typedef* 定义一样的定义。如果你得到了这个错误，C 编译其就会向你抱怨说这些类型不匹配。

ex18.c:74-87 这是 *bubble_sort* 函数的测试用例。现在你也可以看看我演示如何把传入的 *compare_cb* 参数传递给 *bubble_sort*。

ex18.c:90-103 一个简单的 main 函数，通过你从命令行传入的整型数建立数组，然后调用 *test_sorting* 函数。

ex18.c:105-107 最后，你看到了 *typedef* 定义的函数指针 *compare_cb* 是如何使用的。我只是简单的调用了 *test_sorting* 但是把 *sorted_order*，*reverse_order*，和 *strange_order* 作为函数名来使用。C 编译器接着找到这些函数的地址，然后把它作为一个指针供 *test_sorting* 去使用。如果你看看 *test_sorting* 你会看到它把这些依次传递给 *bubble_sort* 但事实上它根本就不知道它们是做什么的，只有满足 *compare_cb* 原型（prototype）的才能工作。

ex18.c:109 最后我们要做的就是释放我们构造的数值数组。

19.1 你应该看到的结果

运行这个程序很简单，那就尝试不同数值的组合，甚至是非数值的组合并产看结果。

ex18 output

```
$ make ex18
cc -Wall -g    ex18.c   -o ex18
$ ./ex18 4 1 7 3 2 0 8
0 1 2 3 4 7 8
8 7 4 3 2 1 0
3 4 2 7 1 0 8
$
```

19.2 让程序出错

我将让你做一些奇怪的事情去破坏它。这些函数指针和别的指针一样，所以它们指向的是内存块（blocks of memory）。C 可以把一种指针类型转换成另一种所以你可以用不同的方法处理数据。这通常不是必要的，但还是让你看看如何修改（hack）你的电脑，我希望你在 *test_sorting* 函数结尾加上这些:

Function Pointer Evil

```
    unsigned char *data = (unsigned char *)cmp;

    for(i = 0; i < 25; i++) {
        printf("%0x:", data[i]);
    }
    printf("\n");
```

这个循环像先把你的函数转换成字符串然后在排序，接着输出它们的内容。这个循环不会中断（break）你的程序，除非你正在使用的 CPU 或操作系统因为你的操作而产生问题。当它输出这个排序后的数组，你会看到一个十六进制数的字符串:

```
55:48:89:e5:89:7d:fc:89:75:f8:8b:55:fc:8b:45:f8:29:d0:c9:c3:55:48:89:e5:89:
```

那应该是这个函数自身的汇编字节码，你会看到它们开头都相同，但结尾不同。还可能是这个循环没取得完整的函数或者是取的太多了，取了一部分别的函数的代码。不多分析一下你永远都不会知道。

19.3 加分习题

1. 找一个十六进制编辑器打开 *ex18*，然后找到包含这个十六进制位串的函数，看看你是否可以在这个程序中找到这个函数。
2. 在你的十六进制编辑器中随便找点别的东西，修改它们。回到你的程序看看发生了什么。修改你找到的字符串是相当容易的。
3. 给 *compare_cb* 传入错误的参数，然后看看 C 编译器提示什么信息。
4. 传入 NULL 看看你的程序出了什么问题。然后运行 *Valgrind* 看看报告了什么。
5. 写另外一个排序算法，修改 *test_sorting* 调用这两个函数并比较排序结果。

Chapter 20

习题 19：一个简单的对象系统

I learned C before I learned Object Oriented Programming, so it helped me to build an OOP system in C to understand the basics of what OOP meant. You are probably the kind of person who learned an OOP language before you learned C, so this kind of bridge might help you as well. In this exercise, you will build a simple object system, but also learn more about the C Pre-Processor or CPP.

This exercise will build a simple game where you kill a Minotaur in a small little castle. Nothing fancy, just four rooms and a bad guy. This project will also be a multi-file project, and look more like a real C software project than your previous ones. This is why I'm introducing the CPP here because you need it to start using multiple files in your own software.

20.1 How The CPP Works

The C Pre-Processor is a template processing system. It's a highly targeted one that helps make C easier to work with, but it does this by having a syntax aware templating mechanism. Traditionally people just used the CPP to store constants and make ”macros” to simplify repetitive coding. In modern C you'll actually use the CPP as a code generator to create templated pieces of code.

How the CPP works is you give it one file, usually a .c file, and it processes various bits of text starting with the `#` (octothorpe[1]) character. When it encounters one of these it performs a specific replacement on the text of the input file. It's main advantage though is it can *include* other files, and then augment its list of macros based on that file's contents.

A quick way to see what the CPP does is take the last exercise and run this:

```
cpp ex18.c | less
```

It will be a huge amount of output, but scroll through it and you'll see the contents of the other files you included with `#include`. Scroll down to the original code and you can see how the ***cpp*** is altering the source based on various `#define` macros in the header files.

The C compiler is so tightly integrated with ***cpp*** that it just runs this for you and understands how it works intimately. In modern C, the ***cpp*** system is so integral to C's function that you might as well just consider it to be part of the language.

[1]A.K.A. pound, hash, mesh, number symbol, pick whatever makes you happy

In the remaining sections, we'll be using more of the CPP syntax and explaining it as we go.

20.2 The Prototype Object System

The OOP system we'll create is a simple ”prototype” style object system more like JavaScript. Instead of classes, you start with prototypes that have fields set, and then use those as the basis of creating other object instances. This ”classless” design is much easier to implement and work with than a traditional class based one.

20.2.1 The Object Header File

I want to put the data types and function declarations into a separate header file named **`object.h`**. This is standard C practice and it lets you ship binary libraries but still let the programmer compile against it. In this file I have several advanced CPP techniques I'm going to quickly describe and then have you see in action later:

object.h

```
#ifndef _object_h
#define _object_h

typedef enum {
    NORTH, SOUTH, EAST, WEST
} Direction;

typedef struct {
    char *description;
    int (*init)(void *self);
    void (*describe)(void *self);
    void (*destroy)(void *self);
    void *(*move)(void *self, Direction direction);
    int (*attack)(void *self, int damage);
} Object;

int Object_init(void *self);
void Object_destroy(void *self);
void Object_describe(void *self);
void *Object_move(void *self, Direction direction);
int Object_attack(void *self, int damage);
void *Object_new(size_t size, Object proto, char *description);

#define NEW(T, N) Object_new(sizeof(T), T##Proto, N)
#define _(N) proto.N

#endif
```

Taking a look at this file, you can see we have a few new pieces of syntax you haven't encountered before:

#ifndef You've seen a `#define` for making simple constants, but the CPP can also do logic and remove sections of code. This `#ifndef` is "if not defined" and checks if there's already a `#define _object_h` and if there is it skips all of this code. I do this so that we can include this file any time we want and not worry about it defining things multiple times.

#define With the above `#ifndef` shielding this file from we then add the *_object_h* define so that any attempts to include it later cause the above to skip.

#define NEW(T,N) This makes a macro, and it works like a template function that spits out the code on the right, whenever you write use the macro on the left. This one is simply making a short version of the normal way we'll call *Object_new* and avoids potential errors with calling it wrong. The way the macro works is the *T* and *N* parameters to *NEW* are "injected" into the line of code on the right. The syntax `T##Proto` says to "concat Proto at the end of T", so if you had `NEW(Room, "Hello.")` then it'd make *RoomProto* there.

#define _(N) This macro is a bit of "syntactic sugar" for the object system and basically helps you write `obj->proto.blah` as simply `obj->_(blah)`. It's not necessary, but it's a fun little trick that I'll use later.

20.2.2 The Object Source File

The `object.h` file is declaring functions and data types that are defined (created) in the `object.c`, so that's next:

object.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "object.h"
#include <assert.h>

void Object_destroy(void *self)
{
    Object *obj = self;

    if(obj) {
        if(obj->description) free(obj->description);
        free(obj);
    }
}

void Object_describe(void *self)
{
    Object *obj = self;
    printf("%s.\n", obj->description);
}

int Object_init(void *self)
{
    // do nothing really
    return 1;
}

void *Object_move(void *self, Direction direction)
{
    printf("You can't go that direction.\n");
    return NULL;
}

int Object_attack(void *self, int damage)
{
    printf("You can't attack that.\n");
    return 0;
}

void *Object_new(size_t size, Object proto, char *description)
{
    // setup the default functions in case they aren't set
    if(!proto.init) proto.init = Object_init;
    if(!proto.describe) proto.describe = Object_describe;
    if(!proto.destroy) proto.destroy = Object_destroy;
    if(!proto.attack) proto.attack = Object_attack;
    if(!proto.move) proto.move = Object_move;

    // this seems weird, but we can make a struct of one size,
    // then point a different pointer at it to "cast" it
    Object *el = calloc(1, size);
    *el = proto;

    // copy the description over
    el->description = strdup(description);

    // initialize it with whatever init we were given
    if(!el->init(el)) {
        // looks like it didn't initialize properly
        el->destroy(el);
        return NULL;
    } else {
        // all done, we made an object of any type
        return el;
    }
}
```

There's really nothing new in this file, except one *tiny* little trick. The function *Object_new* uses an aspect of how *struct*s work by putting the base prototype at the beginning of the struct. When you look at the **ex19.h** header later, you'll see how I make the first field in the struct an *Object*. Since C puts the fields in a struct in order, and since a pointer just points at a chunk of memory, I can "cast" a pointer to anything I want. In this case, even though I'm taking a potentially larger block of memory from *calloc*, I'm using a *Object* pointer to work with it.

I explain this a bit better when we write the **ex19.h** file since it's easier to understand when you see it being used.

That creates your base object system, but you'll need a way to compile it and link it into your **ex19.c** file to create a complete program. The **object.c** file on its own doesn't have a *main* so it isn't enough to make a full program. Here's a **Makefile** that will do this based on the one you've been using:

The Makefile

```
CFLAGS=-Wall -g

all: ex19

ex19: object.o

clean:
	rm -f ex19
```

This **Makefile** is doing nothing more than saying that *ex19* depends on *object.o*. Remember how *make* knows how to build different kinds of files by their extensions? Doing this tells make the following:

1. When I say run *make* the default *all* should just build *ex19*.
2. When you build *ex19*, you need to also build **object.o** and include it in the build.
3. *make* can't see anything in the file for **object.o**, but it does see an **object.c** file, and it knows how to turn a **.c** into a **.o**, so it does that.
4. Once it has **object.o** built it then runs the correct compile command to build *ex19* from **ex19.c** and **object.o**.

20.3 The Game Implementation

Once you have those files you just need to implement the actual game using the object system, and first step is putting all the data types and function declarations in a `ex19.h` file:

ex19.h

```c
#ifndef _ex19_h
#define _ex19_h

#include "object.h"

struct Monster {
    Object proto;
    int hit_points;
};

typedef struct Monster Monster;

int Monster_attack(void *self, int damage);
int Monster_init(void *self);

struct Room {
    Object proto;

    Monster *bad_guy;

    struct Room *north;
    struct Room *south;
    struct Room *east;
    struct Room *west;
};

typedef struct Room Room;

void *Room_move(void *self, Direction direction);
int Room_attack(void *self, int damage);
int Room_init(void *self);


struct Map {
    Object proto;
    Room *start;
    Room *location;
};

typedef struct Map Map;

void *Map_move(void *self, Direction direction);
int Map_attack(void *self, int damage);
int Map_init(void *self);

#endif
```

That sets up three new Objects you'll be using: `Monster`, `Room`, and `Map`.

Taking a look at `object.c:52` you can see where I use a pointer `Object *el = calloc(1, size)`. Go back and look at the `NEW` macro in `object.h` and you can see that it is getting the `sizeof` another struct, say `Room`, and I allocate that much. However, because I've pointed a `Object` pointer at this block of memory, and because I put an `Object proto` field at the from of `Room`, I'm able to treat a `Room` like it's an `Object`.

The way to break this down is like so:

1. I call `NEW(Room, "Hello.")` which the CPP expands as a macro into `Object_new(sizeof(Room), RoomProto, "Hello.")`.
2. This runs, and inside `Object_new` I allocate a piece of memory that's `Room` in size, *but* point a `Object *el` pointer at it.
3. Since C puts the `Room.proto` field first, that means the `el` pointer is really only pointing at enough of the block of memory to see a full `Object` struct. It has no idea that it's even called `proto`.
4. It then uses this `Object *el` pointer to set the contents of the piece of memory correctly with `*el = proto;`. Remember that you can copy structs, and that `*el` means "the value of whatever el points at", so this means "assign the proto struct to whatever el points at".
5. Now that this mystery struct is filled in with the right data from `proto`, the function can then call `init` or `destroy` on the `Object`, but the cool part is whoever called this function can *change* these out for whatever ones they want.

And with that, we have a way to get this one function to construct new types, and give them new functions to change their behavior. This may seem like "hackery" but it's stock C and totally valid. In fact there's quite a few standard system functions that work this same way, and we'll be using some of them for converting addresses in network code.

With the function definitions and data structures written out I can now actually implement the game with four rooms and a minotaur to beat up:

ex19.c

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

```
#include "ex19.h"


int Monster_attack(void *self, int damage)
{
    Monster *monster = self;

    printf("You attack %s!\n", monster->_(description));

    monster->hit_points -= damage;

    if(monster->hit_points > 0) {
        printf("It is still alive.\n");
        return 0;
    } else {
        printf("It is dead!\n");
        return 1;
    }
}

int Monster_init(void *self)
{
    Monster *monster = self;
    monster->hit_points = 10;
    return 1;
}

Object MonsterProto = {
    .init = Monster_init,
    .attack = Monster_attack
};


void *Room_move(void *self, Direction direction)
{
    Room *room = self;
    Room *next = NULL;

    if(direction == NORTH && room->north) {
        printf("You go north, into:\n");
        next = room->north;
    } else if(direction == SOUTH && room->south) {
        printf("You go south, into:\n");
        next = room->south;
    } else if(direction == EAST && room->east) {
        printf("You go east, into:\n");
        next = room->east;
    } else if(direction == WEST && room->west) {
        printf("You go west, into:\n");
        next = room->west;
    } else {
        printf("You can't go that direction.");
        next = NULL;
    }

    if(next) {
        next->_(describe)(next);
    }

    return next;
}


int Room_attack(void *self, int damage)
{
    Room *room = self;
    Monster *monster = room->bad_guy;

    if(monster) {
        monster->_(attack)(monster, damage);
        return 1;
    } else {
        printf("You flail in the air at nothing. Idiot.\n");
        return 0;
    }
}


Object RoomProto = {
    .move = Room_move,
    .attack = Room_attack
};


void *Map_move(void *self, Direction direction)
{
    Map *map = self;
    Room *location = map->location;
```

```c
    Room *next = NULL;

    next = location->_(move)(location, direction);

    if(next) {
        map->location = next;
    }

    return next;
}

int Map_attack(void *self, int damage)
{
    Map* map = self;
    Room *location = map->location;

    return location->_(attack)(location, damage);
}


int Map_init(void *self)
{
    Map *map = self;

    // make some rooms for a small map
    Room *hall = NEW(Room, "The great Hall");
    Room *throne = NEW(Room, "The throne room");
    Room *arena = NEW(Room, "The arena, with the minotaur");
    Room *kitchen = NEW(Room, "Kitchen, you have the knife now");

    // put the bad guy in the arena
    arena->bad_guy = NEW(Monster, "The evil minotaur");

    // setup the map rooms
    hall->north = throne;

    throne->west = arena;
    throne->east = kitchen;
    throne->south = hall;

    arena->east = throne;
    kitchen->west = throne;

    // start the map and the character off in the hall
    map->start = hall;
    map->location = hall;

    return 1;
}

Object MapProto = {
    .init = Map_init,
    .move = Map_move,
    .attack = Map_attack
};

int process_input(Map *game)
{
    printf("\n> ");

    char ch = getchar();
    getchar(); // eat ENTER

    int damage = rand() % 4;

    switch(ch) {
        case -1:
            printf("Giving up? You suck.\n");
            return 0;
            break;

        case 'n':
            game->_(move)(game, NORTH);
            break;

        case 's':
            game->_(move)(game, SOUTH);
            break;

        case 'e':
            game->_(move)(game, EAST);
            break;

        case 'w':
            game->_(move)(game, WEST);
            break;

        case 'a':
```

```c

            game->_(attack)(game, damage);
            break;
        case 'l':
            printf("You can go:\n");
            if(game->location->north) printf("NORTH\n");
            if(game->location->south) printf("SOUTH\n");
            if(game->location->east) printf("EAST\n");
            if(game->location->west) printf("WEST\n");
            break;

        default:
            printf("What?: %d\n", ch);
    }

    return 1;
}

int main(int argc, char *argv[])
{
    // simple way to setup the randomness
    srand(time(NULL));

    // make our map to work with
    Map *game = NEW(Map, "The Hall of the Minotaur.");

    printf("You enter the ");
    game->location->_(describe)(game->location);

    while(process_input(game)) {
    }

    return 0;
}
```

Honestly there isn't much in this that you haven't seen, and only you might need to understand how I'm using the macros I made from the headers files. Here's the important key things to study and understand:

1. Implementing a prototype involves creating its version of the functions, and then creating a single struct ending in "Proto". Look at *MonsterProto*, *RoomProto* and *MapProto*.

2. Because of how *Object_new* is implemented, if you don't set a function in your prototype, then it will get the default implementation created in **object.c**.

3. In *Map_init* I create the little world, but more importantly I use the *NEW* macro from **object.h** to build all of the objects. To get this concept in your head, try replacing the *NEW* usage with direct *Object_new* calls to see how it's

being translated.

4. Working with these objects involves calling functions on them, and the *_(N)* macro does this for me. If you look at the code `monster->_(attack)(monster, damage)` you see that I'm using the macro, which gets replaced with `monster->proto.attack(monster, damage)`. Study this transformation again by rewriting these calls back to their original. Also, if you get stuck then run *cpp* manually to see what it's going to do.

5. I'm using two new functions *srand* and *rand*, which setup a simple random number generator good enough for the game. I also use *time* to initialize the random number generator. Research those.

6. I use a new function *getchar* that gets a single character from the stdin. Research it.

20.4 What You Should See

Here's me playing my own game:

ex19 output

```
$ make ex19
cc -Wall -g   -c -o object.o object.c
cc -Wall -g    ex19.c object.o   -o ex19
$ ./ex19
You enter the The great Hall.

> l
You can go:
NORTH

> n
You go north, into:
The throne room.

> l
You can go:
SOUTH
EAST
WEST

> e
You go east, into:
Kitchen, you have the knife now.

> w
You go west, into:
```

```
The throne room.

> s
You go south, into:
The great Hall.

> n
You go north, into:
The throne room.

> w
You go west, into:
The arena, with the minotaur.

> a
You attack The evil minotaur!
It is still alive.

> a
You attack The evil minotaur!
It is dead!

> ^D
Giving up? You suck.
$
```

20.5 Auditing The Game

As an exercise for you I have left out all of the *assert* checks I normally put into a piece of software. You've seen me use *assert* to make sure a program is running correctly, but now I want you to go back and do the following:

1. Look at each function you've defined, one file at a time.
2. At the top of each function, add *asserts* that make sure the input parameters are correct. For example, in *Object_new* you want a `assert(description != NULL)`.
3. Go through each line of the function, and find any functions being called. Read the documentation (man page) for that function, and confirm what it returns for an error. Add another assert to check that the error didn't happen. For example, in *Object_new* you need one after the call to *calloc* that does `assert(el != NULL)`.
4. If a function is expected to return a value, either make sure it returns an error value (like NULL), or have an assert to make sure that the returned variable isn't invalid. For example, in *Object_new*, you need to have `assert(el != NULL)` again before the last return since that part can never be NULL.

5. For every *`if-statement`* you write, make sure there's an else clause unless that if is an error check that causes an exit.

6. For every *`switch-statement`* you write, make sure that there's a *`default`* case that handles anything you didn't anticipate.

Take your time going through every line of the function and find any errors you make. Remember that the point of this exercise is to stop being a "coder" and switch your brain into being a "hacker". Try to see how you could break it, then write code to prevent it or abort early if you can.

20.6 Extra Credit

1. Update the `Makefile` so that when you do *`make clean`* it will also remove the `object.o` file.

2. Write a test script that works the game in different ways and augment the `Makefile` so you can run *`make test`* and it'll thrash the game with your script.

3. Add more rooms and monsters to the game.

4. Put the game mechanics into a third file, compile it to .o, and then use that to write another little game. If you're doing it right you should only have a new *`Map`* and a *`main`* function in the new game.

Chapter 21

习题 20: Zed 的强悍的 debug 宏命令

There is a constant problem in C that you have been dancing around but which I am going to solve in this exercise using a set of macros I developed. You can thank me later when you realize how insanely awesome these macros are. Right now you won't realize how awesome they are, so you'll just have to use them and then you can walk up to me one day and say, "Zed, those Debug Macros were the bomb. I owe you my first born child because you saved me a decade of heartache and prevented me from killing myself more than once. Thank you good sir, here's a million dollars and the original Snakehead Telecaster prototype signed by Leo Fender."

Yes, they are that awesome.

21.1 The C Error Handling Problem

In almost every programming language handling errors is a difficult activity. There's entire programming languages that try as hard as they can to avoid even the concept of an error. Other languages invent complex control structures like exceptions to pass error conditions around. The problem exists mostly because programmers assume errors don't happen and this optimism infects the type of languages they use and create.

C tackles the problem by returning error codes and setting a global *errno* value that you check. This makes for complex code that simply exists to check if something you did had an error. As you write more and more C code you'll write code with the pattern:

1. Call a function.
2. If the return value is an error (must look that up each time too).
3. Then cleanup all the resource created so far.
4. and print out an error message that hopefully helps.

This means for every function call (and yes, *every* function) you are potentially writing 3-4 more lines just to make sure it worked. That doesn't include the problem of cleaning up all of the junk you've built to that point. If you have 10 different structures, 3 files, and a database connection, when you get an error then you would have 14 more lines.

In the past this wasn't a problem because C programs did what you've been doing when there's an error: die. No point in bothering with cleanup when the OS will do it for you. Today though many C programs need to run for weeks, months, or years and handle errors from many different sources gracefully. You can't just have your webserver die at the slightest touch, and you definitely can't have a library you've written nuke a the program its used in. That's just rude.

Other languages solve this problem with exceptions, but those have problems in C (and in other languages too). In C you only have one return value, but exceptions are an entire stack based return system with arbitrary values. Trying to marshal exceptions up the stack in C is difficult, and no other libraries will understand it.

21.2 The Debug Macros

The solution I've been using for years is a small set of "debug macros" that implement a basic debugging and error handling system for C. This system is easy to understand, works with every library, and makes C code more solid and clearer.

It does this by adopting the convention that whenever there's an error, your function will jump to an "error:" part of the function that knows how to cleanup everything and return an error code. You use a macro called `check` to check return codes, print an error message, and then jump to the cleanup section. You combine that with a set of logging functions for printing out useful debug messages.

I'll now show you the entire contents of the most awesome set of brilliance you've ever seen:

dbg.h

```c
#ifndef __dbg_h__
#define __dbg_h__

#include <stdio.h>
#include <errno.h>
#include <string.h>

#ifdef NDEBUG
#define debug(M, ...)
#else
#define debug(M, ...) fprintf(stderr, "DEBUG %s:%d: " M "\n", __FILE__, __LINE__, ##__VA_ARGS__)
#endif

#define clean_errno() (errno == 0 ? "None" : strerror(errno))

#define log_err(M, ...) fprintf(stderr, "[ERROR] (%s:%d: errno: %s) " M "\n", __FILE__, __LINE__, clean_err

#define log_warn(M, ...) fprintf(stderr, "[WARN] (%s:%d: errno: %s) " M "\n", __FILE__, __LINE__, clean_err

#define log_info(M, ...) fprintf(stderr, "[INFO] (%s:%d) " M "\n", __FILE__, __LINE__, ##__VA_ARGS__)

#define check(A, M, ...) if(!(A)) { log_err(M, ##__VA_ARGS__); errno=0; goto error; }

```

```
24 #define sentinel(M, ...)  { log_err(M, ##__VA_ARGS__); errno=0; goto error; }
25
26 #define check_mem(A) check((A), "Out of memory.")
27
28 #define check_debug(A, M, ...) if(!(A)) { debug(M, ##__VA_ARGS__); errno=0; goto error; }
29
30 #endif
```

Yes, that's it, and here's what every line does:

dbg.h:1-2 The usual defense against accidentally including the file twice, which you saw in the last exercise.

dbg.h:4-6 Includes for the functions that these macros need.

dbg.h:8 The start of a `#ifdef` which lets you recompile your program so that all the debug log messages are removed.

dbg.h:9 If you compile with `NDEBUG` defined, then "no debug" messages will remain. You can see in this case the `#define debug()` is just replaced with nothing (the right side is empty).

dbg.h:10 The matching `#else` for the above `#ifdef`.

dbg.h:11 The alternative `#define debug` that translates any use of `debug("format", arg1, arg2)` into an `fprintf` call to `stderr`. Many C programmers don't know, but you can create macros that actually work like `printf` and take variable arguments. Some C compilers (actually cpp) don't support this, but the ones that matter do. The magic here is the use of `##__VA_ARGS__` which says "put whatever they had for extra arguments (...) here". Also notice the use of `__FILE__` and `__LINE__` to get the current file:line for the debug message. *Very* helpful.

dbg.h:12 The end of the `#ifdef`.

dbg.h:14 The `clean_errno` macro that's used in the others to get a safe readable version of `errno`. That strange syntax in the middle is a "ternary operator" and you'll learn what it does later.

dbg.h:16-20 The `log_err`, `log_warn`, and `log_info`, macros for logging messages meant for the end user. Works like `debug` but can't be compiled out.

dbg.h:22 The best macro ever, `check` will make sure the condition `A` is true, and if not logs the error `M` (with variable arguments for `log_err`), then jumps to the function's `error:` for cleanup.

dbg.h:24 The 2nd best macro ever, `sentinel` is placed in any part of a function that shouldn't run, and if it does prints an error message then jumps to the `error:` label. You put this in `if-statements` and `switch-statements` to catch conditions that shouldn't happen, like the `default:`.

dbg.h:26 A short-hand macro `check_mem` that makes sure a pointer is valid, and if it isn't reports it as an error with "Out of memory."

dbg.h:28 An alternative macro `check_debug` that still checks and handles an error, but if the error is common then you don't want to bother reporting it. In this one it will use `debug` instead of `log_err` to report the message, so when you define `NDEBUG` the check still happens, the error jump goes off, but the message isn't printed.

21.3 Using dbg.h

Here's an example of using all of **dbg.h** in a small program. This doesn't actually do anything but demonstrate how to use each macro, but we'll be using these macros in all of the programs we write from now on, so be sure to understand how to use them.

ex20.c

```c
#include "dbg.h"
#include <stdlib.h>
#include <stdio.h>


void test_debug()
{
    // notice you don't need the \n
    debug("I have Brown Hair.");

    // passing in arguments like printf
    debug("I am %d years old.", 37);
}

void test_log_err()
{
    log_err("I believe everything is broken.");
    log_err("There are %d problems in %s.", 0, "space");
}

void test_log_warn()
{
    log_warn("You can safely ignore this.");
    log_warn("Maybe consider looking at: %s.", "/etc/passwd");
}

void test_log_info()
{
    log_info("Well I did something mundane.");
    log_info("It happened %f times today.", 1.3f);
}

int test_check(char *file_name)
{
    FILE *input = NULL;
    char *block = NULL;

    block = malloc(100);
    check_mem(block); // should work

    input = fopen(file_name,"r");
    check(input, "Failed to open %s.", file_name);

    free(block);
    fclose(input);
    return 0;

error:
    if(block) free(block);
    if(input) fclose(input);
    return -1;
}

int test_sentinel(int code)
{
    char *temp = malloc(100);
    check_mem(temp);

    switch(code) {
        case 1:
            log_info("It worked.");
            break;
        default:
            sentinel("I shouldn't run.");
    }

    free(temp);
    return 0;

error:
    if(temp) free(temp);
    return -1;
}

int test_check_mem()
{
    char *test = NULL;
    check_mem(test);

    free(test);
```

```c
    return 1;

error:
    return -1;
}

int test_check_debug()
{
    int i = 0;
    check_debug(i != 0, "Oops, I was 0.");

    return 0;
error:
    return -1;
}

int main(int argc, char *argv[])
{
    check(argc == 2, "Need an argument.");

    test_debug();
    test_log_err();
    test_log_warn();
    test_log_info();

    check(test_check("ex20.c") == 0, "failed with ex20.c");
    check(test_check(argv[1]) == -1, "failed with argv");
    check(test_sentinel(1) == 0, "test_sentinel failed.");
    check(test_sentinel(100) == -1, "test_sentinel failed.");
    check(test_check_mem() == -1, "test_check_mem failed.");
    check(test_check_debug() == -1, "test_check_debug failed.");

    return 0;

error:
    return 1;
}
```

Pay attention to how `check` is used, and how when it is `false` it will jump to the `error:` label to do a cleanup. The way to read those lines is, ”check that A is true and if not say M and jump out.”

21.4 What You Should See

When you run this, give it some bogus first parameter and you should see this:

ex20 output

```
$ make ex20
cc -Wall -g -DNDEBUG    ex20.c   -o ex20
$ ./ex20 test
[ERROR] (ex20.c:16: errno: None) I believe everything is broken.
[ERROR] (ex20.c:17: errno: None) There are 0 problems in space.
[WARN] (ex20.c:22: errno: None) You can safely ignore this.
[WARN] (ex20.c:23: errno: None) Maybe consider looking at: /etc/passwd.
[INFO] (ex20.c:28) Well I did something mundane.
[INFO] (ex20.c:29) It happened 1.300000 times today.
[ERROR] (ex20.c:38: errno: No such file or directory) Failed to open test.
[INFO] (ex20.c:57) It worked.
[ERROR] (ex20.c:60: errno: None) I shouldn't run.
[ERROR] (ex20.c:74: errno: None) Out of memory.
```

See how it reports the exact line number where the *check* failed? That's going to save you hours of debugging later. See also how it prints the error message for you when *errno* is set? Again, that will save you hours of debugging.

21.5 How The CPP Expands Macros

It's now time for you to get a small introduction to the CPP so that you know how these macros actually work. To do this, I'm going to break down the most complex macro from **dbg.h** and have you run *cpp* so you can see what it's actually doing.

Imagine I have a function called *dosomething()* that return the typical 0 for success and -1 for an error. Every time I call *dosomething* I have to check for this error code, so I'd write code like this:

```
int rc = dosomething();
if(rc != 0) {
    fprintf(stderr, "There was an error: %s\n", strerror());
    goto error;
}
```

What I want to use the CPP for is to encapsulate this *if-statement* I have to use all the time into a more readable and memorable line of code. I want what you've been doing in **dbg.h** with the *check* macro:

```
int rc = dosomething();
check(rc == 0, "There was an error.");
```

This is *much* clearer and explains exactly what's going on: check that the function worked, and if not report an error. To do this, we need some special CPP "tricks" that make the CPP useful as a code generation tool. Take a look at the *check* and *log_err* macros again:

```
#define log_err(M, ...) fprintf(stderr, "[ERROR] (%s:%d: errno: %s) " M "\n", __FILE__,
    ↪__LINE__, clean_errno(), ##__VA_ARGS__)
#define check(A, M, ...) if(!(A)) { log_err(M, ##__VA_ARGS__); errno=0; goto error; }
```

The first macro, *log_err* is simpler and simply replace itself with a call to *fprintf* to *stderr*. The only tricky part of this macro is the use of `...` in the definition `log_err(M, ...)`. What this does is let you pass variable arguments to the macro, so you can pass in the arguments that should go to *fprintf*. How do they get injected into the *fprintf* call? Look at the end to the `##__VA_ARGS__` and that's telling the CPP to take the args entered where the `...` is, and inject them at that part of the *fprintf* call. You can then do things like this:

```
log_err("Age: %d, name: %s", age, name);
```

The arguments `age, name` are the `...` part of the definition, and those get injected into the fprintf output to become:

```
fprintf(stderr, "[ERROR] (%s:%d: errno: %s) Age %d: name %d\n",
     __FILE__, __LINE__, clean_errno(), age, name);
```

See the `age, name` at the end? That's how `...` and `##__VA_ARGS__` work together, and it will work in macros that call other variable argument macros. Look at the *check* macro now and see it calls *log_err*, but *check* is *also* using the `...` and `##__VA_ARGS__` to do the call. That's how you can pass full *printf* style format strings to *check*, which go to *log_err*, and then make both work like *printf*.

Next thing to study is how *check* crafts the *if-statement* for the error checking. If we strip out the *log_err* usage we see this:

```
if(!(A)) { errno=0; goto error; }
```

Which means, if *A* is false, then clear errno and goto the error label. That has *check* macro being replaced with the *if-statment* so if we manually expanded out the macro `check(rc == 0, "There was an error.")` we'd get:

```
if(!(rc == 0) {
     log_err("There was an error.");
     errno=0;
     goto error;
}
```

What you should be getting from this trip through these two macros is that the CPP replaces macros with the expanded version of their definition, but that it will do this *recursively*, expanding all the macros in macros. The CPP then is just a recursive templating system, as I mentioned before. Its power comes from its ability to generate whole blocks of parameterized code thus becoming a handy code generation tool.

That leaves one question: Why not just use a function like *die*? The reason is you want file:line numbers and the *goto* operation for an error handling exit. If you did this inside a function, you wouldn't get a line number for where the error actually happened, and the goto would be much more complicated.

Another reason is you still have to write the raw *if-statement*, which looks like all the other *if-statements* in your code, so it's not as clear that this one is an error check. By wrapping the *if-statement* in a macro called *check* you make it clear that this is just error checking, and not part of the main flow.

Finally, CPP has the ability to *conditionally compile* portions of code, so you can have code that's only present when you build a developer or debug version of the program. You can see this already in the **`dbg.h`** file where the ***`debug`*** macro has a body only if it's asked for by the compiler. Without this ability, you'd need a wasted ***`if-statement`*** that checks for "debug mode", and then still wastes CPU doing that check for no value.

21.6 Extra Credit

1. Put `#define NDEBUG` at the top of the file and check that all the debug messages go away.
2. Undo that line, and add `-DNDEBUG` to ***`CFLAGS`*** at the top of the **`Makefile`** then recompile to see the same thing.
3. Modify the logging so that it include the function name as well as the file:line.

Chapter 22

习题 21：高级数据类型与流程控制

这个习题将会是 C 语言中你可使用的数据类型以及流程控制结构的一个完备的缩影。它可以作为你补充与巩固知识的参考，而且不需要你写任何代码。我会通过制作一些学习卡来让你记忆这个信息，这让你可以夯实你脑海中的重要概念。

为了让习题有所帮助，你应该至少花一个星期去仔细推敲其内容并且完成填空。你应该写出每一个题在表达什么意思，然后写一个程序去验证你的研究

22.1 可用数据类型

整型 (int) 通常存放一个整数, 一般大小默认是 32 位。

双精度浮点型 (double) 存放一个较大的浮点数。

单精度浮点型 (float) 存放一个较小的浮点数。

字符型 (char) 存放单个字符

空类型 (void) 表示“没有类型”，用于表示一个函数不返回信息，或者一个没有类型的指针，例如 `void *thing` 。

枚举类型 (enum) 枚举类型，像整型一样，值是整型，但是会返回集合中对应的符号名称。在 分支选择语句 *(switch-statements)* 中，当你没有涵盖一个枚举类型里的所有元素的时候，一些编译器会发出警告。

22.1.1 类型修饰符

unsigned 使得某一类型不用表示负数，这样虽然不会有负数，但是会有一个较大的上界。

signed 有正数与负数，会将无符号的上界一分为二，转换成相同大小的负下界[1]。

long 会为这个类型使用一个较大的存储来存放更大的数值，通常是原来大小的两倍。

short 为这个类型使用较小的存储，所以它存放的数值小一些，但是空间节省了一半。

[1]译者注：unsigned 与 signed 可以用来修饰 char 和任意整型

22.1.2 类型限定符

const 表明这个变量在初始化之后就不会再发生改变。

volatile 表明这个变量的变化无法预测，编译器不会去假设这个变量的值，也不会做任何优化。只有当你要对这个变量做一些怪事情的时候才需要使用这个限定符。

register 强制要求编译器将这个变量放在寄存器里，编译器可以就这么忽略掉你。如今编译器更善于决定一个变量应该放在哪里，所以最好只有当你确定这样可以提高速度的时候才这么做。

22.1.3 类型转换

C 语言使用了一种“阶梯式的类型提升”的机制，当面对一个表达式两边的操作数时，会在执行这个运算之前，提升较小的一边去适应较大的一边。如果表达式的一边在这个列表中，那么另一边会在运算完成前按照如下顺序被转换:

1. long double
2. double
3. float
4. int (仅指 *char* 和 *short int*);
5. long

如果你想弄清除表达式中的类型转换到底是怎么回事，那么就不要把这个工作交给编译器来做。使用明确的转换操作将其转换成你想要的类型。例如，如果给你：

```
long + char - int * double
```

与其揣测结果是否会被正确的转换成 double ，不如使用如下转换:

```
(double)long - (double)char - (double)int * double
```

将你想要转换的类型放在变量的前面的括号中，就可以进行强制转换。重要的一点是 *数据类型转换只能提升不能降低*. 不要试图将 *long* 转换成 *char* 除非你明确的知道你这么做的后果。

22.1.4 类型大小

头文件 `stdint.h` 为整数类型定义了一组 *typdefs* ，同时也定义了一组所有数据类型大小的宏。这个在兼容性上优于老的 `limits.h` ，使用起来更加方便，类型定义如下：

int8_t 8 比特符号整型

uint8_t 8 比特无符号整型

int16_t 16 比特符号整型

uint16_t 16 比特无符号整型

int32_t 32 比特符号整型

uint32_t 32 比特无符号整型

int64_t 64 比特符号整型

uint64_t 64 比特无符号整型

这个匹配的模式是 (u)int(BITS)_t ，其中 *u* 放在前面表明是”unsigned”，*BITS* 是比特数。这个模式也用在了返回这些数据类型最大值的宏定义上：

INT*N*_MAX 比特数为 *N* 的符号整型的最大正数值。

INT*N*_MIN 比特数为 *N* 的符号整型的最小负数值。

UINT*N*_MAX 比特数为 *N* 的无符号整型的最大正数值。因为是无符号的，所以最小值是 0，没有负数。

头文件 `stdint.h` 中也有表示 size_t 类型大小的宏，这个类型是大小可以用来装指针的整型还有其他关于大小定义的方便的宏。编译器必须得有这些定义，这样才能允许有其他更大的类型。

这是 `stdint.h` 中的一个列表：

int_least*N*_t 能存储至少 *N* 比特的整型[2]。

uint_least*N*_t 能存储至少 *N* 比特的无符号整型。

INT_LEAST*N*_MAX 至少能存储 *N* 比特的整型对应的最大值。

INT_LEAST*N*_MIN 至少能存储 *N* 比特的整型对应的最小值。

UINT_LEAST*N*_MAX 至少能存储 *N* 比特的无符号整型对应的最小值。。

int_fast*N*_t 同 *int_leastN_t* 接近，但是要求的是“最快速”的至少 *N* 比特的整型[3]。

uint_fast*N*_t 无符号字长最小快速整型.

INT_FAST*N*_MAX 字长最小快速整型对应的最大值。

INT_FAST*N*_MIN 字长最小快速整型对应的最小。

UINT_FAST*N*_MAX 字长最小快速无符号整型。

intptr_t 大小容纳一个指针的 *signed* 整型。

uintptr_t 大小容纳一个指针的 *unsigned* 整型。

INTPTR_MAX *intptr_t* 的最大值。

INTPTR_MIN *intptr_t* 的最小值。

UINTPTR_MAX 无符号 *uintptr_t* 的最大值。

intmax_t 系统允许的最大的符号整型。

uintmax_t 系统允许的最大的无符号整型。

INTMAX_MAX 最大的符号整型的最大值。

INTMAX_MIN 最大的符号整型的最小值。

UINTMAX_MAX 最大的无符号整型的最大值。

[2]译者注：字长最小整型，即规定的最小长度为 *N* 比特的整型。
[3]译者注：字长最小快速整型，理论上来说，CPU 处理和其字长（word）长度相同的整型会比较快。

PTRDIFF_MIN *ptrdiff_t* 的最小值。

PTRDIFF_MAX identptrdiff_t 的最大值。

SIZE_MAX *size_t* 的最大值。

22.2 可用运算符

这是 C 语言中可用的运算符的完整列表。在这个列表中，我做了如下说明：

双目运算符 (binary) 有左右两个运算对象: `X + Y`.

单目运算符 (unary) 对自身做运算: `-X`.

前置运算符 (prefix) 运算符在变量的前面: `++X`.

后置运算符 (postfix) 通常是与 *(prefix)* 相同，但是放置在后面会有不同的含义: `X++`.

三目运算符 (ternary) 只有一个，虽然被叫做三目运算符，但是实际上是“三个操作数”: `X ? Y : Z`.

22.2.1 数学运算符

这里有一些基本的数学运算符, 而且我把 `()` 也放了进来，因为它是调用一个函数，很像一个“数学”运算符。

() 调用函数

*** (binary)** 乘

/ 除

+ (binary) 加

+ (unary) 正数

++ (postfix) 先读值，再自增

++ (prefix) 先自增，再读值

−− (postfix) 先读值，再自减

−− (prefix) 先自减，再读值

- (binary) 减

- (unary) 负数

22.2.2 数据运算符

这些运算符使用不同的方法和形式来存取数据。

-> 访问指针指向的结构体的成员

. 访问结构体成员

[] 数组下标.

sizeof 类型或变量的大小

& (unary) 取地址

*** (unary)** 取值

22.2.3 逻辑运算符

这些运算符用来检验变量相等或者不相等。

!= 不相等

< 小于

<= 小于等于或者不大于

== 相等 (不是赋值的等于)

> 大于

>= 大于等于或者不小于

22.2.4 位运算符

这些运算符要更高级一些，用来对正数的原始二进制数进行移位和修改的。

& (binary) 按位与.

<< 左移位

>> 右移位

^ 按位异或（互斥或）

| 按位或

~ 按位取反（翻转所有的位）

22.2.5 布尔运算符

用在真值检验中。好好学习三目运算符，非常的好用。

! 非

&& 与

|| 或

?: 三目运算符，条件运算符，将 `X ? Y : Z` 读作“如果 X 则 Y 否则 Z”.

22.2.6 赋值运算符

复合赋值运算符是在赋值的时候，加上其他的运算符。上面的大部分运算符，都可以用来组成复合运算符。

= 等于

%= 取余等于

&= 按位与等于

*= 乘等于

+= 加等于

-= 减等于

/= 除等于

<<= 左移位等于

>>= 右移位等于

^= 按位异或等于

|= 按位或等于

22.3 可用的控制结构

还有一些你还没有接触到的控制结构：

do-while `do { ... } while(X);` 先执行循环体中的代码，再验证 *X* 表达式，直到结束。

break 把这个放入到循环体中，就会提前结束循环。

continue 跳出循环体，直接验证表达式，进行下一次循环。

goto 跳到代码中，你放置了 `label:` 的地方，你已经在 **`dbg.h`** 的宏中用过这个跳到 `error:` 标签处。

22.3.1 加分习题

1. 浏览 **`stdint.h`** 文件或者它的介绍然后写出所有可能的可用大小标识符。
2. 浏览这的每一项，然后写出其在代码中是做什么的。可以通过网上查找来确定你的研究是否正确。
3. 可以利用记忆卡片每天花费 15 分钟来加强记忆，夯实基础。
4. 写一个程序打印出各种类型的例子来确认你的研究结果是正确的。

Chapter 23

Exercise 22: The Stack, Scope, And Globals

The concept of "scope" seems to confuse quite a few people when they first start programming. Originally it came from the use of the system stack (which we lightly covered earlier) and how it was used to store temporary variables. In this exercise, we'll learn about scope by learning about how a stack data structure works, and then feeding that concept back in to how modern C does scoping.

The real purpose of this exercise though is to learn where the hell things live in C. When someone doesn't grasp the concept of scope, it's almost always a failure in understanding where variables are created, exist, and die. Once you know where things are, the concept of scope becomes easier.

This exercise will require three files:

ex22.h A header file that sets up some external variables and some functions.

ex22.c Not your main like normal, but instead a source file that will become a object file `ex22.o` which will have some functions and variables in it defined from `ex22.h`.

ex22_main.c The actual *`main`* that will include the other two and demonstrate what they contain as well as other scope concepts.

23.0.2 ex22.h and ex22.c

Your first step is to create your own header file named `ex22.h` which defines the functions and "extern" variables you need:

ex22.h

```
#ifndef _ex22_h
#define _ex22_h

// makes THE_SIZE in ex22.c available to other .c files
```

```c
extern int THE_SIZE;

// gets and sets an internal static variable in ex22.c
int get_age();
void set_age(int age);

// updates a static variable that's inside update_ratio
double update_ratio(double ratio);

void print_size();

#endif
```

The important thing to see is the use of `extern int THE_SIZE`, which I'll explain after you also create the matching `ex22.c`:

ex22.c

```c
#include <stdio.h>
#include "ex22.h"
#include "dbg.h"

int THE_SIZE = 1000;

static int THE_AGE = 37;

int get_age()
{
    return THE_AGE;
}

void set_age(int age)
{
    THE_AGE = age;
}


double update_ratio(double new_ratio)
{
    static double ratio = 1.0;

    double old_ratio = ratio;
    ratio = new_ratio;

    return old_ratio;
}

void print_size()
{
    log_info("I think size is: %d", THE_SIZE);
}
```

These two files introduce some new kinds of storage for variables:

extern This keyword is a way to tell the compiler "the variable exists, but it's in another 'external' location". Typically this means that one .c file is going to use a variable that's been defined in another .c file. In this case, we're saying `ex22.c` has a variable *THE_SIZE* that will be accessed from `ex22_main.c`.

static (file) This keyword is kind of the inverse of *extern* and says that the variable is only used in this .c file, and should not be available to other parts of the program. Keep in mind that *static* at the file level (as with *THE_AGE* here) is different than in other places.

static (function) If you declare a variable in a function *static*, then that variable acts like a *static* defined in the file, but it's only accessible from that function. It's a way of creating constant state for a function, but in reality it's *rarely* used in modern C programming because they are hard to use with threads.

In these two files then, you have the following variables and functions that you should understand:

THE_SIZE This is the variable you declared *extern* that you'll play with from `ex22_main.c`.

get_age and set_age These are taking the static variable *THE_AGE*, but exposing it to other parts of the program through functions. You couldn't access *THE_AGE* directly, but these functions can.

update_ratio This takes a new *ratio* value, and returns the old one. It uses a function level static variable *ratio* to keep track of what the ratio currently is.

print_size Prints out what `ex22.c` thinks *THE_SIZE* is currently.

23.0.3 ex22_main.c

Once you have that file written, you can then make the main function which uses all of these and demonstrates some more scope conventions:

ex22_main.c

```c
#include "ex22.h"
#include "dbg.h"

const char *MY_NAME = "Zed A. Shaw";

```

```c
void scope_demo(int count)
{
    log_info("count is: %d", count);

    if(count > 10) {
        int count = 100;  // BAD! BUGS!

        log_info("count in this scope is %d", count);
    }

    log_info("count is at exit: %d", count);

    count = 3000;

    log_info("count after assign: %d", count);
}

int main(int argc, char *argv[])
{
    // test out THE_AGE accessors
    log_info("My name: %s, age: %d", MY_NAME, get_age());

    set_age(100);

    log_info("My age is now: %d", get_age());

    // test out THE_SIZE extern
    log_info("THE_SIZE is: %d", THE_SIZE);
    print_size();

    THE_SIZE = 9;

    log_info("THE SIZE is now: %d", THE_SIZE);
    print_size();

    // test the ratio function static
    log_info("Ratio at first: %f", update_ratio(2.0));
    log_info("Ratio again: %f", update_ratio(10.0));
    log_info("Ratio once more: %f", update_ratio(300.0));

    // test the scope demo
    int count = 4;
    scope_demo(count);
    scope_demo(count * 20);
50
51      log_info("count after calling scope_demo: %d", count);
52
53      return 0;
54  }
```

I'll break this file down line-by-line, and as I do you should find each variable I mention and where it lives.

ex22_main.c:4 Making a *const* which stands for constant and is an alternative to using a *define* to create a constant variable.

ex22_main.c:6 A simple function that demonstrates more scope issues in a function.

ex22_main.c:8 Prints out the value of *count* as it is at the top of the function.

ex22_main.c:10 An *if-statement* that starts a new *scope block*, and then has another *count* variable in it. This version of *count* is actually a whole new variable. It's kind of like the *if-statement* started a new "mini function".

ex22_main.c:11 The *count* that is local to this block is actually different from the one in the function's parameter list. What what happens as we continue.

ex22_main.c:13 Prints it out so you can see it's actually 100 here, not what was passed to *scope_demo*.

ex22_main.c:16 Now for the freaky part. You have *count* in two places: the parameters to this function, and in the *if-statement*. The *if-statement* created a new block, so the *count* on line 11 *does not impact the parameter with the same name*. This line prints it out and you'll see that it prints the value of the parameter, not 100.

ex22_main.c:18-20 Then I set the parameter *count* to 3000 and print that out, which will demonstrate that you can change function parameters and they don't impact the caller's version of the variable.

Make sure you trace through this function, but don't think that you understand scope quite yet. Just start to realize that if you make a variable inside a block (as in *if-statements* or *while-loops*), then those variables are *new* variables that exist only in that block. This is crucial to understand, and is also a *source of many bugs*. We'll address why you shouldn't do this shortly.

The rest of the **ex22_main.c** then demonstrates all of these by manipulating and printing them out:

ex22_main.c:26 Prints out the current values of *MY_NAME* and gets *THE_AGE* from **ex22.c** using the accessor function *get_age*.

ex22_main.c:27-30 Uses *set_age* in **ex22.c** to change *THE_AGE* and then print it out.

ex22_main.c:33-39 Then I do the same thing to *THE_SIZE* from **ex22.c**, but this time I'm accessing it directly, and also demonstrating that it's actually changing in that file by printing it here and with *print_size*.

ex22_main.c:42-44 Show how the static variable *ratio* inside *update_ratio* is maintained between function calls.

ex22_main.c:46-51 Finally running *scope_demo* a few times so you can see the scope in action. Big thing to notice is that the local *count* variable remains unchanged. You *must* get that passing in a variable like this will not let you change it in the function. To do that you need our old friend the pointer. If you were to pass a pointer to this *count*, then the called function has the address of it and can change it.

That explains what's going on in all of these files, but you should trace through them and make sure you know where everything is as you study it.

23.1 What You Should See

This time, instead of using your **Makefile** I want you to build these two files manually so you can see how they are actually put together by the compiler. Here's what you should do and what you should see for output.

ex22 output

```
$ cc -Wall -g -DNDEBUG   -c -o ex22.o ex22.c
$ cc -Wall -g -DNDEBUG    ex22_main.c ex22.o   -o ex22_main
$ ./ex22_main
[INFO] (ex22_main.c:26) My name: Zed A. Shaw, age: 37
[INFO] (ex22_main.c:30) My age is now: 100
[INFO] (ex22_main.c:33) THE_SIZE is: 1000
[INFO] (ex22.c:32) I think size is: 1000
[INFO] (ex22_main.c:38) THE SIZE is now: 9
[INFO] (ex22.c:32) I think size is: 9
[INFO] (ex22_main.c:42) Ratio at first: 1.000000
[INFO] (ex22_main.c:43) Ratio again: 2.000000
[INFO] (ex22_main.c:44) Ratio once more: 10.000000
[INFO] (ex22_main.c:8) count is: 4
[INFO] (ex22_main.c:16) count is at exit: 4
[INFO] (ex22_main.c:20) count after assign: 3000
[INFO] (ex22_main.c:8) count is: 80
[INFO] (ex22_main.c:13) count in this scope is 100
[INFO] (ex22_main.c:16) count is at exit: 80
[INFO] (ex22_main.c:20) count after assign: 3000
[INFO] (ex22_main.c:51) count after calling scope_demo: 4
```

Make sure you trace how each variable is changing and match it to the line that gets output. I'm using *log_info* from the **dbg.h** macros so you can get the exact line number where each variable is printed and find it in the files for tracing.

23.2 Scope, Stack, And Bugs

If you've done this right you should now see many of the different ways you can place variables in your C code. You can use *extern* or access functions like *get_age* to create globals. You can make new variables inside any blocks, and they'll retain their own values until that block exits, leaving the outer variables alone. You also can pass a value to a function, and change the parameter but not change the caller's version of it.

The most important thing to realize though is that all of this causes bugs. C's ability to place things in many places in your machine and then let you access it in those places means you get confused easily about where something lives. If you don't where it lives then there's a chance you'll not manage it properly.

With that in mind, here's some rules to follow when writing C code so you avoid bugs related to the stack:

1. Do not "shadow" a variable like I've done here with *`count`* in *`scope_demo`*. It leaves you open to subtle and hidden bugs where you *think* you're changing a value and you actually aren't.
2. Avoid too many globals, especially if across multiple files. If you have to then use accessor functions like I've done with *`get_age`*. This doesn't apply to constants, since those are read-only. I'm talking about variables like *`THE_SIZE`*. If you want people to modify or set this, then make accessor functions.
3. When in doubt, put it on the heap. Don't rely on the semantics of the stack or specialized locations and instead just create things with *`malloc`*.
4. Don't use function static variables like I did in *`update_ratio`*. They're rarely useful and end up being a huge pain when you need to make your code concurrent in threads. They are also hard as hell to find compared to a well done global variable.
5. Avoid reusing function parameters as it's confusing whether you're just reusing it or if you think you're changing the *caller's* version of it.

As with all things, these rules can be broken when it's practical. In fact, I guarantee you'll run into code that breaks all of these rules and is perfectly fine. The constraints of different platforms makes it necessary sometimes.

23.3 How To Break It

For this exercise, breaking the program involves trying to access or change things you can't:

1. Try to directly access variables in **`ex22.c`** from **`ex22_main.c`** that you think you can't. For example, you can't get at *`ratio`* inside *`update_ratio`*? What if you had a pointer to it?
2. Ditch the *`extern`* declaration in **`ex22.h`** to see what you get for errors or warnings.
3. Add *`static`* or *`const`* specifiers to different variables and then try to change them.

23.4 Extra Credit

1. Research the concept of "pass by value" vs. "pass by reference". Write an example of both.
2. Use pointers to gain access to things you shouldn't have access to.
3. Use valgrind to see what this kind of access looks like when you do it wrong.
4. Write a recursive function that causes a stack overflow. Don't know what a recursive function is? Try calling *`scope_demo`* at the bottom of *`scope_demo`* itself so that it loops.
5. Rewrite the **`Makefile`** so that it can build this.

Chapter 24

习题 23：接触 Duff 的设备

本节习题可以看做是一个脑力游戏，你将接触到的是一个叫“Duff 的设备”的东西，它的名称来自于其发明者 Tom Duff，而这也是最著名的 C 语言实现范例之一。这一小段杰作（恶魔的杰作？）几乎包含了你学过的所有的内容。研究它的工作原理也是一件乐事。

Note 6 仅供练习

C 语言的乐趣之一就是你可以用它产生和本习题一样的各种令人匪夷所思的实现方案，不过这也是 C 语言让人恼火的一个原因。学习这些技巧能让你更深的了解 C 语言和你的计算机。然而，你绝对不能把这些技巧用到实际编程中去。简单易读的代码是你永远的奋斗目标。

Duff 的设备实际上是作者 Tom Duff 跟 C 编译器玩的一个骗招，其实按理说这种用法不应该生效才对。这个一道供你思考解决的迷题，所以我不会告诉你它实现的功能。你需要运行这段代码，从中理解它的功能，以及为什么可以通过这种方法实现。

ex23.c

```c
#include <stdio.h>
#include <string.h>
#include "dbg.h"


int normal_copy(char *from, char *to, int count)
{
    int i = 0;

    for(i = 0; i < count; i++) {
        to[i] = from[i];
    }

    return i;
}

```

```c
int duffs_device(char *from, char *to, int count)
{
    {
        int n = (count + 7) / 8;

        switch(count % 8) {
            case 0: do { *to++ = *from++;
                        case 7: *to++ = *from++;
                        case 6: *to++ = *from++;
                        case 5: *to++ = *from++;
                        case 4: *to++ = *from++;
                        case 3: *to++ = *from++;
                        case 2: *to++ = *from++;
                        case 1: *to++ = *from++;
                    } while(--n > 0);
        }
    }

    return count;
}

int zeds_device(char *from, char *to, int count)
{
    {
        int n = (count + 7) / 8;

        switch(count % 8) {
            case 0:
            again: *to++ = *from++;

            case 7: *to++ = *from++;
            case 6: *to++ = *from++;
            case 5: *to++ = *from++;
            case 4: *to++ = *from++;
            case 3: *to++ = *from++;
            case 2: *to++ = *from++;
            case 1: *to++ = *from++;
                    if(--n > 0) goto again;
        }
    }

    return count;
}

```

```c
int valid_copy(char *data, int count, char expects)
{
    int i = 0;
    for(i = 0; i < count; i++) {
        if(data[i] != expects) {
            log_err("[%d] %c != %c", i, data[i], expects);
            return 0;
        }
    }

    return 1;
}


int main(int argc, char *argv[])
{
    char from[1000] = {'a'};
    char to[1000] = {'c'};
    int rc = 0;

    // setup the from to have some stuff
    memset(from, 'x', 1000);
    // set it to a failure mode
    memset(to, 'y', 1000);
    check(valid_copy(to, 1000, 'y'), "Not initialized right.");

    // use normal copy to
    rc = normal_copy(from, to, 1000);
    check(rc == 1000, "Normal copy failed: %d", rc);
    check(valid_copy(to, 1000, 'x'), "Normal copy failed.");

    // reset
    memset(to, 'y', 1000);

    // duffs version
    rc = duffs_device(from, to, 1000);
    check(rc == 1000, "Duff's device failed: %d", rc);
    check(valid_copy(to, 1000, 'x'), "Duff's device failed copy.");

    // reset
    memset(to, 'y', 1000);

    // my version
    rc = zeds_device(from, to, 1000);
    check(rc == 1000, "Zed's device failed: %d", rc);
    check(valid_copy(to, 1000, 'x'), "Zed's device failed copy.");

    return 0;
error:
    return 1;
}
```

这段代码中有 3 个版本的拷贝函数:

normal_copy 这是一段 *for-loop* 代码，它将字符串从一个数组复制到另一个数组中。

duffs_device 这就是称作“Duff 的设备”的脑力游戏，以 Tom Duff 命名，他就是这一段邪恶代码的始作俑者。

zeds_device 这是“Duff 的设备”的另一个版本，是用一个 goto 语句实现的，以方便你理解前面 *duffs_device* 函数中奇怪的 *do-while* 片段。

在继续往下读之前，先学习这三个函数。并试着去解释究竟发生了什么事情。

24.1 你应该看到的结果

这是一个没有输出的程序，程序运行完就退出了。并且在 valgrind 下运行也是无错的。

24.2 谜底

首先要明白的是 C 的语法是相当松散的，你可以把 *do-while* 的一半内容放到 *switch-statement* 中，另一半放到别的地方，代码还能正常工作。如果你看我那个版本的 *goto again* 部分，其中的细节其实更容易理解。不过你还是应该弄明白原始版本中的那部分是如何实现的。

其次要明白的是 *switch-statement* 的默认的一步到底的语义，正因为如此，你才能跳到某个 case，然后接着运行，直到 switch 语句结束。

最后的线索就是前面的 `count % 8` 和 *n* 的计算结果。

接下来，你要通过下面的方法去理解函数是如何工作的。

1. 将这些代码打印出来，这样你就能从纸面工作了。
2. 在纸上用表列出在 *switch-statement* 之前每一个变量的初始值。
3. 跟着 switch 的逻辑，跳转到对应的 case 中。
4. 更新变量，包括 *to*、*from*、以及它们指向的数组。
5. 选择 *while* 版本或我的 *goto* 版本进行跟踪。检查变量，并来回跟踪 *do-while* 或者 *again* 的定位标签。
6. 重复以上步骤并更新变量的值，直到你确认自己完全理解了这一部分代码是怎样工作的为止。

24.2.1 有必要这样做吗？

当你完全理解了它是如何工作。最后的问题是：为什么要这样做呢？这样做的目的是手动“展开循环”。大的循环执行起来可能会很慢，加快速度的方法之一就是找出循环中的固定步骤，把它放到循环外面顺次重复地写出来，并让重复的次数和循环的次数对应。比如：如果你知道一个循环要运行至少 20 次，那你可以将循环的内容在代码中写 20 次以代替这一段循环。

“Duff 的设备”是每次自动的让循环迭代 8 次。它能正常运行并非常巧妙，但是时至今日，好的编译器已经在帮你做这些事情了。这种技巧对你来说应该没多大用处，除非在极少数情况下，你可以证明这样做真能提高速度。

24.3 加分习题

1. 永远都别去使用这个技巧。
2. 去维基百科搜索“Duff Device”词条。看你能否从里边的代码发现错误。比较词条上的代码和我们的代码，仔细阅读并试图理解为什么我们运行维基百科上的代码会失败，而当时 Tom Duff 却能运行成功。
3. 创建一个宏集，让它能生产任何长度“duff 的设备”。比如，你想要一个有 32 条 case 语句的“Duff 设备”，你能一次性产生它们吗？你的宏一次能产生 8 条 case 语句吗？
4. 修改 *main* 函数，在里边放一些测速代码，看看三种实现方式哪种速度会比较快。
5. 阅读 *memcpy*、*memmove*、*memset* 这几个函数的文档，并比较它们的速度。
6. 永远都别去使用这个技巧！

Chapter 25

习题 24：输入、输出、文件

你已经可以用 *printf* 来打印了，这非常好。但是你肯定不满足于此。在这一节，你将会用 *fscanf* 和 *fgets* 来构建一个结构体以存贮某个人的信息。简单介绍过如何读取数据之后，你将会看到一个 C 语言所拥有的处理 IO 的完整函数列表。你应该已经见过而且用过其中的一部分，所以本节会帮助你回想起很多东西。

ex24.c

```c
#include <stdio.h>
#include "dbg.h"

#define MAX_DATA 100

typedef enum EyeColor {
    BLUE_EYES, GREEN_EYES, BROWN_EYES,
    BLACK_EYES, OTHER_EYES
} EyeColor;

const char *EYE_COLOR_NAMES[] = {
    "Blue", "Green", "Brown", "Black", "Other"
};

typedef struct Person {
    int age;
    char first_name[MAX_DATA];
    char last_name[MAX_DATA];
    EyeColor eyes;
    float income;
} Person;


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

```
{
    Person you = {.age = 0};
    int i = 0;
    char *in = NULL;

    printf("What's your First Name? ");
    in = fgets(you.first_name, MAX_DATA-1, stdin);
    check(in != NULL, "Failed to read first name.");

    printf("What's your Last Name? ");
    in = fgets(you.last_name, MAX_DATA-1, stdin);
    check(in != NULL, "Failed to read last name.");

    printf("How old are you? ");
    int rc = fscanf(stdin, "%d", &you.age);
    check(rc > 0, "You have to enter a number.");

    printf("What color are your eyes:\n");
    for(i = 0; i <= OTHER_EYES; i++) {
        printf("%d) %s\n", i+1, EYE_COLOR_NAMES[i]);
    }
    printf("> ");

    int eyes = -1;
    rc = fscanf(stdin, "%d", &eyes);
    check(rc > 0, "You have to enter a number.");

    you.eyes = eyes - 1;
    check(you.eyes <= OTHER_EYES && you.eyes >= 0, "Do it right, that's not an option.");

    printf("How much do you make an hour? ");
    rc = fscanf(stdin, "%f", &you.income);
    check(rc > 0, "Enter a floating point number.");

    printf("----- RESULTS -----\n");

    printf("First Name: %s", you.first_name);
    printf("Last Name: %s", you.last_name);
    printf("Age: %d\n", you.age);
    printf("Eyes: %s\n", EYE_COLOR_NAMES[you.eyes]);
    printf("Income: %f\n", you.income);

    return 0;
error:

    return -1;
}
```

这个程序表面看似简单，其实不然。这里面用到了函数 *fscanf*，它是用来做文件扫描的。*scanf* 相关的一系列函数与 *printf* 相关的一系列函数起的作用正相反。*printf* 按着指定格式输出数据，*scanf* 按着指定格式读取数据（或者扫描输入数据）。

文件的开头什么也没有，函数 *main* 做了如下一些事情：

ex24.c:24-28 创建一些必须的变量。

ex24.c:30-32 用 *fgets* 获取姓名中的名字，从输入获取一个字符串 (这里是从 *stdin* 获取字符串) 输入名字的时候不要太长，防止存储名字的缓冲区不够长导致内存溢出。

ex24.c:34-36 依然用 *fgets* 来获取 *you.last_name* 。

ex24.c:38-39 用 *fscanf* 从 *stdin* 获取一个整数，并且将这个整数存储在变量 *you.age*。可以看到用 *printf* 打印整数的时候使用了同样的格式表示符。为了让 *fscanf* 可以把输入的值存储在某个指针变量指定的位置上，你必须把 *address* 的地址取出来作为指针变量的值。这个例子生动的展示了如何把一串数据的内存地址作为输出参数。

ex24.c:41-45 通过遍历枚举 *EyeColor* 的全部数值，输出眼睛的颜色所具有的所有可能性。

ex24.c:47-50 继续 *fscanf*，给 *you.eyes* 赋值，同时确保这个数值是有真实有效的。这很重要，因为有的人会输入一个不在 *EYE_COLOR_NAMES* 数组范围之内的数字，导致一个段错误。

ex24.c:52-53 知道你挣了多少 *float* 收入知道你的到了多少收入 *you.income*。

ex24.c:55-61 打印所有的东西来确认一下程序运行无误。注意我们用 *EYE_COLOR_NAMES* 来打印真正的 *EyeColor* 的内容。

25.1 你将看到

当运行这个程序的时候，会看到你的输入被正确的处理。可以试试给一些奇怪的输入来检查一下代码是不是能够很好的保护自己而不是出现奇怪问题。

ex24 输出

```
$ make ex24
cc -Wall -g -DNDEBUG    ex24.c   -o ex24
$ ./ex24
What's your First Name? Zed
What's your Last Name? Shaw
How old are you? 37
What color are your eyes:
1) Blue
```

```
2) Green
3) Brown
4) Black
5) Other
> 1
How much do you make an hour? 1.2345
----- RESULTS -----
First Name: Zed
Last Name: Shaw
Age: 37
Eyes: Blue
Income: 1.234500
```

25.2 让程序出错

到目前为止一切顺利，其实这个练习的主要目的是看 *scanf* 是怎么被输入搞垮的。处理简单数字转换的时候还凑活，一旦处理字符串的时候就完全的垮掉了，因为很难在读取动作完成以前告诉 *scanf* 输入缓冲区有多大。函数 *gets*(不是 *fgets*) 也有同样的问题。这类函数完全无法知道输入的缓冲区有多大，过大的缓冲区会搞垮程序。把那行代码从用 *fgets* 替换成 `fscanf(stdin, "%50s", you.first_name)` 然后再运行一次程序就能看到仅靠 *fscanf* 和字符串就能搞垮我们的程序。

注意到程序似乎读取了太多内容而且好像你敲的回车似乎没工作了么？程序似乎没有按着你的期望方式工作，而且似乎还不如奇怪的由 *scanf* 引起的问题，仅仅是用 *fgets* 就可以。

下一步，把 *fgets* 替换成 *gets*，然后这样运行 *valgrind*: `valgrind ./ex24 < /dev/urandom`，这将把一些随机生成的垃圾丢给程序。这称作 Fuzzing 程序，这是一种有效的发现输入处理逻辑 bug 的方法。目前这种情况，是从 `/dev/urandom` 获得随机数并且喂给程序处理，然后看着它崩溃掉。在某些操作系统上，你可能需要多试几次，或者把 *MAX_DATA* 的定义改的小一点来让程序崩溃。

gets 确实非常差劲，以至于某些操作系统平台会在 *program* 运行 *gets* 的时候发出警告。你应该离这个函数越远越好，如果非要加个期限，我希望是一万年。最后，继续给 *you.eyes* 赋值并且把确保输入在合理范围内的代码删掉。然后输入一些有问题的数字比如 -1，或者 1000。在 Valgrind 里面再来一遍看看会发生什么。

25.3 I/O 函数

你应该为如下的函数列表创建索引卡，在上面写上函数名字，作用，以及类似变种。

1. fscanf
2. fgets
3. fopen
4. freopen

5. fdopen
6. fclose
7. fcloseall
8. fgetpos
9. fseek
10. ftell
11. rewind
12. fprintf
13. fwrite
14. fread

仔细浏览一遍这些函数，记住他们的区别和作用。比如有 *fscanf* 的卡片上，同样会有 *scanf*，*sscanf*，*vscanf*，卡片背面会有函数作用的说明。最后，可以用 *man* 读取卡片的相关信息。例如 `man fscanf` 就可以得到 *fscanf* 的帮助信息。

25.4 额外小作业

1. 不用 *fscanf* 完全重写一遍代码。你将会用到诸如 *atoi* 的函数来把输入的字符串转换成数字。
2. 把 *fscanf* 改成只用 *scanf* 看看有什么不同。
3. 把代码改成输入的名字会被回车以及任何空格截断的形式。
4. 使用 *scanf* 写一个函数，让它从文件中一次读入一个字符，但不要读到结束符后面。将函数写得尽可能通用化，让它接收字符串的大小作为参数，记得确认不论在任何情况下，字符串都以 `'\0'` 结尾。

Chapter 26

Exercise 25: Variable Argument Functions

In C you can create your own versions of functions like ***printf*** and ***scanf*** by creating a "variable argument function". These functions use the header **stdarg.h** and with them you can create nicer interfaces to your library. They are handy for certain types of "builder" functions, formatting functions, and anything that takes variable arguments.

Understanding "vararg functions" is *not* essential to creating C programs. I think I've used it maybe a 20 times in my code in the years I've been programming. However, knowing how a vararg function works will help you debug the ones you use and gives you more understanding of the computer.

ex25.c

```c
/** WARNING: This code is fresh and potentially isn't correct yet. */

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include "dbg.h"

#define MAX_DATA 100

int read_string(char **out_string, int max_buffer)
{
    *out_string = calloc(1, max_buffer + 1);
    check_mem(*out_string);

    char *result = fgets(*out_string, max_buffer, stdin);
    check(result != NULL, "Input error.");

    return 0;

error:
    if(*out_string) free(*out_string);
    *out_string = NULL;
    return -1;
}

int read_int(int *out_int)
{
    char *input = NULL;
    int rc = read_string(&input, MAX_DATA);
    check(rc == 0, "Failed to read number.");

    *out_int = atoi(input);

    free(input);
    return 0;

error:
    if(input) free(input);
    return -1;
}

int read_scan(const char *fmt, ...)
{
    int i = 0;
    int rc = 0;
    int *out_int = NULL;
    char *out_char = NULL;
    char **out_string = NULL;
    int max_buffer = 0;

    va_list argp;
    va_start(argp, fmt);

    for(i = 0; fmt[i] != '\0'; i++) {
        if(fmt[i] == '%') {
            i++;
            switch(fmt[i]) {
                case '\0':
                    sentinel("Invalid format, you ended with %%.");
                    break;

                case 'd':
                    out_int = va_arg(argp, int *);
                    rc = read_int(out_int);
                    check(rc == 0, "Failed to read int.");
                    break;

                case 'c':
                    out_char = va_arg(argp, char *);
                    *out_char = fgetc(stdin);
                    break;

                case 's':
                    max_buffer = va_arg(argp, int);
                    out_string = va_arg(argp, char **);
                    rc = read_string(out_string, max_buffer);
                    check(rc == 0, "Failed to read string.");
                    break;

                default:
                    sentinel("Invalid format.");
            }
        } else {
            fgetc(stdin);
        }

        check(!feof(stdin) && !ferror(stdin), "Input error.");
    }

    va_end(argp);
    return 0;

error:
    va_end(argp);
    return -1;
}



int main(int argc, char *argv[])
{
    char *first_name = NULL;
    char initial = ' ';
    char *last_name = NULL;
    int age = 0;

    printf("What's your first name? ");
    int rc = read_scan("%s", MAX_DATA, &first_name);
    check(rc == 0, "Failed first name.");
```

```c

    printf("What's your initial? ");
    rc = read_scan("%c\n", &initial);
    check(rc == 0, "Failed initial.");

    printf("What's your last name? ");
    rc = read_scan("%s", MAX_DATA, &last_name);
    check(rc == 0, "Failed last name.");

    printf("How old are you? ");
    rc = read_scan("%d", &age);

    printf("---- RESULTS ----\n");
    printf("First Name: %s", first_name);
    printf("Initial: '%c'\n", initial);
    printf("Last Name: %s", last_name);
    printf("Age: %d\n", age);

    free(first_name);
    free(last_name);
    return 0;
error:
    return -1;
}
```

This program is similar to the previous exercise, except I have written my own *scanf* style function that handles strings the way I want. The main function should be clear to you, as well as the two functions *read_string* and *read_int* since they do nothing new.

The varargs function is called *read_scan* and it does the same thing that *scanf* is doing using the *va_list* data structure and it's supporting macros and functions. Here's how it works:

1. I set as the last parameter of the function the keyword `...` which indicates to C that this function will take any number of arguments after the *fmt* argument. I could put many other arguments before this, but I can't put anymore after this.
2. After setting up some variables, I create a *va_list* variable and initialize it with *va_start*. This configures the gear in **stdarg.h** that handles variable arguments.
3. I then use a *for-loop* to loop through the format string *fmt* and process the same kind of formats that *scanf* has, but much simpler. I just have integers, characters, and strings.
4. When I hit a format, I use the *switch-statement* to figure out what to do.
5. Now, to *get* a variable from the *va_list argp* I use the macro *va_arg(argp, TYPE)* where TYPE is the exact type of what I will assign this function parameter to. The downside to this design is you're flying blind, so if you don't have enough parameters then oh well, you'll most likely crash.

6. The interesting difference from *scanf* is I'm assuming that people want *read_scan* to create the strings it reads when it hits a *'s'* format sequence. When you give this sequence, the function takes two parameters off the *va_list argp* stack: the max function size to read, and the output character string pointer. Using that information it just runs *read_string* to do the real work.
7. This makes *read_scan* more consistent than *scanf* since you *always* give an address-of \& on variables to have them set appropriately.
8. Finally, if it encounters a character that's not in the format, it just reads one char to skip it. It doesn't care what that char is, just that it should skip it.

26.1 What You Should See

When you run this one it's similar to the last one:

ex25.out

```
$ make ex25
cc -Wall -g -DNDEBUG    ex25.c   -o ex25
$ ./ex25
What's your first name? Zed
What's your initial? A
What's your last name? Shaw
How old are you? 37
---- RESULTS ----
First Name: Zed
Initial: 'A'
Last Name: Shaw
Age: 37
```

26.2 How To Break It

This program should be more robust against buffer overflows, but it doesn't handle the formatted input as well as *scanf*. To try breaking this, change the code that you forget to pass in the initial size for '%s' formats. Try also giving it more data than *MAX_DATA*, and then see how not using *calloc* in *read_string* changes how it works. Finally, there's a problem that fgets eats the newlines, so try to fix that using *fgetc* but leave out the '
0' that ends the string.

26.3 Extra Credit

1. Make double and triple sure that you know what each of the `out_` variables are doing. Most important is `out_string` and how it's a pointer to a pointer, so getting when you're setting the pointer vs. the contents is important. Break down each of the
2. Write a similar function to `printf` that uses the varargs system and rewrite `main` to use it.
3. As usual, read the man page on all of this so you know what it does on your platform. Some platforms will use macros and others use functions, and some have these do nothing. It all depends on the compiler and the platform you use.

Chapter 27

习题 26：第一个真正的程序

到这里正好是本书的一半，你也该参加一次期中考试了。这次考核我要求你把我专为本书写的一个叫 *devpkg* 的程序重写一遍，然后你要用一些关键方法改进代码，最重要的方法就是为它写一些单元测试。

Note 7 *WARNING: Beta Draft Content*

I wrote this exercise before writing some of the exercises you might need to complete this. If you are attempting this one now, please keep in mind that the software may have bugs, that you might have problems because of my mistakes, and that you might not know everything you need to finish it. If so, tell me at help@learncodethehardway.org and then wait until I finish the other exercises.

27.1 什么是 *devpkg*?

Devpkg 是一个简单的 C 程序，它的功能是用来安装别的软件。这个软件是我专为这本书写的，目的是教你学习真正的软件项目是怎样构架的，以及学习怎样重复使用别人写的 library。它使用了一个称作 The Apache Portable Runtime (APR) 可移植性 library，它里边有很多好用适用于很多平台（包括 Windows）的 C 函数。除此之外，它所做的就是从网上或者本地抓到代码，然后执行一下我们每个人都会的 `./configure ; make ; make install` 而已。

本节习题中你的任务是从源代码 build *devpkg*，完成我给你的每一个挑战，然后通过阅读源代码来理解 *devpkg* 的功能和原理。

27.1.1 我们要实现的功能

我们要做一个工具，它有三条命令:

devpkg -S	执行软件的全新安装。
devpkg -I	通过 URL 安装软件。
devpkg -L	列出所有安装了的软件。
devpkg -F	下载源代码以供手动 build。
devpkg -B	下载源代码，build 并且安装软件，即使在软件已被安装的情况下也会再装一遍。

我们要求 *devpkg* 能够识别绝大部分的 URL，识别出这个 URL 对应的是哪个项目，然后下载并且安装软件，最后记录下来它下载了哪个软件。我们还要求 *devpkg* 能够处理一份简单的需求列表 (dependency list)，这样它就能同时将当前要安装的软件依赖的软件也安装起来。

27.1.2 软件设计

我们将通过非常简单的设计来达到我们的目的：

使用外部命令 你的大部分工作将通过外部命令诸如 *curl*、*git*、*tar* 来完成。这样可以减少实现 *devpkg* 所需的代码量。

简单的文件数据库 要做复杂也不难，不过起始阶段你只要建立一个单个文件的简单数据库 `/usr/local/.devpkg/db`，用来记录所安装的软件即可。

只适用/usr/local 这里你也可以做得更高级，不过初始阶段我们就假设所有的东西都装到 `/usr/local` 好了，这也是大部分 Unix 下软件的标准安装路径。

configure, make, make install 我们假设绝大部分软件可以通过执行 *configure; make; make install* 来安装，而 *configure* 可能不是必须的一步。如果你要安装的软件不支持这些基本的安装方式，那你可以通过一个选项来修改安装命令，不过更多的东西 *devpkg* 就不去理会了。

用户可以是 root 我们将假设用户可以通过 sudo 命令得到 root 权限，不过在执行完安装命令以后他们将回到普通用户级别。

这样我们可以让程序的初始体积比较小，而功能也都能实现，这样我们就能顺利进行后面的学习，而以后你也能够进一步修改它。

27.1.3 Apache Portable Runtime

接下来你要做的是利用 Apache Portable Runtime (APR) 里一成套的可移植函数库以完成本项目。APR 不是必须项，就算不用它你也可以完成本程序，只不过你需要写更多的代码罢了。我要求你使用 APR 的原因是让你习惯于链接并使用别的库文件。最后要说的一点是，APR 在 *Windows* 下也能工作，所以你学的 APL 技能可以用在很多别的平台上面。

你需要下载 *apr-1.4.5* 和 *apr-util-1.3* 这两个函数库，同时阅读一下 APR 主站提供的文档。

下面是一个用来安装的 shell 脚本，你需要将这个脚本手动誊写一遍，然后运行它，直到它能毫不出错地安装 APR 为止。

APR 安装脚本

```
set -e

# go somewhere safe
cd /tmp

# get the source to base APR 1.4.5
curl -L -O http://download.nextag.com/apache//apr/apr-1.4.5.tar.gz
```

```

# extract it and go into the source
tar -xzvf apr-1.4.5.tar.gz
cd apr-1.4.5

# configure, make, make install
./configure
make
sudo make install

# reset and cleanup
cd /tmp
rm -rf apr-1.4.5 apr-1.4.5.tar.gz

# do the same with apr-util
curl -L -O http://download.nextag.com/apache//apr/apr-util-1.3.12.tar.gz

# extract
tar -xzvf apr-util-1.3.12.tar.gz
cd apr-util-1.3.12

# configure, make, make install
./configure --with-apr=/usr/local/apr
# you need that extra parameter to configure because
# apr-util can't really find it because...who knows.

make
sudo make install

#cleanup
cd /tmp
rm -rf apr-util-1.3.12* apr-1.4.5*
```

我要求你写这个脚本是因为 *devpkg* 实现的功能和这个脚本一样，只不过参数更多，功能也更完善而已。其实你可以用 shell 完全实现本项目，而且这样做代码量更少，不过在一本 C 语言的书里教你 shell 程序还是有些不合适吧？

运行这个脚本，如果有错就修改过来，直到它能正常工作为止。这样你就装好了实现剩余项目所需的函数库。

27.2 项目结构

你需要通过建立一些文件来开始一个新项目。以下是我通常建立新项目的方法：

项目骨架目录

```
mkdir devpkg
cd devpkg
touch README Makefile
```

27.2.1 其他需求

你应该已经装了 APR 和 APR-util，所以现在你还需要几个文件作为基本 dependency：

1. 习题 20 中的 **dbg.h**
2. **bstrlib.h** 和 **bstrlib.c**，它们来自 http://bstring.sourceforge.net/。下载并解压.zip 文件，把这两个文件复制出来即可。
3. 键入命令 `make bstrlib.o`，如果执行失败，请阅读下面的“解决 bstring 的问题”。

Note 8 *解决 bstring 的问题*

有些平台上面 bstring.c 会出现如下的错误：

```
bstrlib.c:2762: error: expected declaration specifiers or '...' before numeric
    ↪constant
```

这是因为 bstrlib.c 的作者用了一个有问题的 define 语句，这个语句有时不能正常工作。只要把这个 **#ifdef** 删掉并且重新编译即可。

上述步骤完成以后，你应该准备了以下文件：**Makefile**、**README**、**dbg.h**、**bstrlib.h**、**bstrlib.c**。现在你可以继续了。

27.3 Makefile

Makefile 是一个很好的开始着手点，这样你可以计划好要 build 的内容，以及你需要建立的代码文件。

Makefile

```
PREFIX?=/usr/local
CFLAGS=-g -Wall -I${PREFIX}/apr/include/apr-1  -I${PREFIX}/apr/include/apr-util-1
LDFLAGS=-lapr-1 -pthread -laprutil-1

all: devpkg

devpkg: bstrlib.o db.o shell.o commands.o

```

```
install: all
	install -d $(DESTDIR)/$(PREFIX)/bin/
	install devpkg $(DESTDIR)/$(PREFIX)/bin/

clean:
	rm -f *.o
	rm -f devpkg
	rm -rf *.dSYM
```

除了这个奇怪的 ?= 语法以外，这里基本没有你没见过的东西。这个表达式的意思是“如果 PREFIX 的值还没被设定，那么就将 PREFIX 设为该值”。

27.4 代码文件

从 Makefile 里我们可以看出 *devpkg* 需要以下四个 dependency：

bstrlib.o ，编译自你下载的 `bstlib.c` 和 `bstlib.h`。

db.o ，编译自 `db.c` 和 `db.h` 头文件。它包含我们即将需要的“数据库”功能。

shell.o ，编译自 `shell.c` 和 `shell.h`，它里边包含一些方便运行外部命令（例如 curl）的辅助函数。

commands.o ，编译自 `command.c` 和 `command.h`，它包含了实现 *devpkg* 命令行功能所须的各种函数。

devpkg 我们并没有特别提及它，不过其实它是这部分 Makefile 的 build target（就在最左边）。它是由 `devpkg.c` 编译而来，该文件包含了整个程序的 *main* 函数。

你现在的任务是创建上面提到的每个文件，键入必须的代码并确保正确。

27.4.1 数据库函数

我们有必要把安装过的软件的 URL 用某种方式记录下来，并且能够随时列出这些 URL，并通过这些记检查并跳过那些安装过的软件。我采用的是一个单个文件的数据库，并通过 `bstrlib.h` 这个头文件来实现它的功能。

首先创建 `db.h` 以供你实现必要的功能：

db.h

```
#ifndef _db_h
#define _db_h

#define DB_FILE "/usr/local/.devpkg/db"
#define DB_DIR "/usr/local/.devpkg"


```

Note 9 别被这场魔法秀忽悠了

读到这里也许你会想："神啊，Zed 怎么能这么牛，这么多东西随便搞搞就弄出来了？！我这辈子都做不到。"其实我并不是用我强悍的编程技术妙笔生花般地地写出这个 *devpkg* 结构的，我做的是下面这些事情：

1. 我写了一个短小的 README 用来梳理思路，从而决定了我要为 *devpkg* 实现什么样的功能。
2. 我创建了一个简单的 Bash 脚本（跟你刚写的那个类似），用以找出所有要实现的细节。
3. 我创建了一个.c 文件，并且花了几天时间去实现它的功能。
4. 待到功能基本实现，bug 基本没有了的时候，我接着就把一整个大文件分割成了这四个文件。
5. 分割完以后，我把里边的函数和数据结构该重命名的重命名，该完善的完善，从而让它们的逻辑更完整，而且更"美观"。
6. 最后，等我让这个重新构架过的程序成功运行起来之后，我又加了一些诸如 *-F* 和 *-B* 这样的命令行选项。

你读到的是我想要教你的顺序，不过其实这也不是我固定使用的顺序。有时我对主题了解比较明确，所以会花比较长的时间做更多计划，有时我会直接把想法写成代码看它是否能工作。有时我会写出一个项目再把它丢掉，然后重新设计并写出一个更好的来。一切都取决于我的个人经验带来的感觉，或者是编程的灵感。

如果你碰到一个"高手"说一个编程问题只有一种方法解决，那他是在骗你。要吗他们其实用了多种方法，要么他们其实不是高手。

```c
int DB_init();
int DB_list();
int DB_update(const char *url);
int DB_find(const char *url);

#endif
```

然后在 `db.c` 中实现这些函数，在你 build 它的时候，使用 *make* 来实现干净的编译，就跟你以前学过的一样。

db.c

```c
#include <unistd.h>
#include <apr_errno.h>
#include <apr_file_io.h>

#include "db.h"
#include "bstrlib.h"
#include "dbg.h"

static FILE *DB_open(const char *path, const char *mode)
{
    return fopen(path, mode);
}
```

```


static void DB_close(FILE *db)
{
    fclose(db);
}


static bstring DB_load()
{
    FILE *db = NULL;
    bstring data = NULL;

    db = DB_open(DB_FILE, "r");
    check(db, "Failed to open database: %s", DB_FILE);

    data = bread((bNread)fread, db);
    check(data, "Failed to read from db file: %s", DB_FILE);

    DB_close(db);
    return data;

error:
    if(db) DB_close(db);
    if(data) bdestroy(data);
    return NULL;
}


int DB_update(const char *url)
{
    if(DB_find(url)) {
        log_info("Already recorded as installed: %s", url);
    }

    FILE *db = DB_open(DB_FILE, "a+");
    check(db, "Failed to open DB file: %s", DB_FILE);

    bstring line = bfromcstr(url);
    bconchar(line, '\n');
    int rc = fwrite(line->data, blength(line), 1, db);
    check(rc == 1, "Failed to append to the db.");

    return 0;
```

```
error:
    if(db) DB_close(db);
    return -1;
}


int DB_find(const char *url)
{
    bstring data = NULL;
    bstring line = bfromcstr(url);
    int res = -1;

    data = DB_load(DB_FILE);
    check(data, "Failed to load: %s", DB_FILE);

    if(binstr(data, 0, line) == BSTR_ERR) {
        res = 0;
    } else {
        res = 1;
    }

error: // fallthrough
    if(data) bdestroy(data);
    if(line) bdestroy(line);

    return res;
}


int DB_init()
{
    apr_pool_t *p = NULL;
    apr_pool_initialize();
    apr_pool_create(&p, NULL);

    if(access(DB_DIR, W_OK | X_OK) == -1) {
        apr_status_t rc = apr_dir_make_recursive(DB_DIR,
                APR_UREAD | APR_UWRITE | APR_UEXECUTE |
                APR_GREAD | APR_GWRITE | APR_GEXECUTE, p);
        check(rc == APR_SUCCESS, "Failed to make database dir: %s", DB_DIR);
    }

    if(access(DB_FILE, W_OK) == -1) {
        FILE *db = DB_open(DB_FILE, "w");
        check(db, "Cannot open database: %s", DB_FILE);
        DB_close(db);
    }

    apr_pool_destroy(p);
    return 0;

error:
    apr_pool_destroy(p);
    return -1;
}


int DB_list()
{
    bstring data = DB_load(DB_FILE);
    check(data, "Failed to read load: %s", DB_FILE);

    printf("%s", bdata(data));
    bdestroy(data);
    return 0;

error:
    return -1;
}
```

挑战 1：代码审查

继续下一步之前，仔细阅读每一行代码，确认你输入的内容和书里的完全相同。练习倒着一行一行读回去，你还要检查每一处函数调用，确认你使用了 *check* 来检查返回值。最后，如果你有不认识或者不了解的函数，你需要到 APR 网站的文档中，或者在 `bstrlib.h` 和 `bstrlib.c` 这两个代码文件中寻找答案。

27.4.2 Shell 函数

devpkg 设计最关键的一个设计决策就是使用外部工具来实现大部分功能，例如 *curl*、*tar*、*git* 等程序。我们其实也能找到需要的函数库，在不借用外部工具的情况下实现这个项目，不过如果我们需要的只是这些程序的基本功能，那么使用函数库是没必要的，再说在 Unix 下多运行几条命令也不是啥丢人的事情。

为了达到目的，我将使用 `apr_thread_proc.h` 里的函数来运行外部程序，不过我还想创建一个简单的“模板”系统。我将使用 *struct Shell* 来存储运行程序必须的所有信息，不过我会在参数列表中留出“空位”，实际用到的参数将被填入到这些空位中。

从下面的 `shell.h` 文件中你可以看到我们需要使用的结构和命令。我用了 *extern* 来表示别的 `.c` 文件可以访问到我在 `shell.c` 中定义的变量。

shell.h

```c
#ifndef _shell_h
#define _shell_h

#define MAX_COMMAND_ARGS 100

#include <apr_thread_proc.h>

typedef struct Shell {
    const char *dir;
    const char *exe;

    apr_procattr_t *attr;
    apr_proc_t proc;
    apr_exit_why_e exit_why;
    int exit_code;

    const char *args[MAX_COMMAND_ARGS];
} Shell;

int Shell_run(apr_pool_t *p, Shell *cmd);
int Shell_exec(Shell cmd, ...);

extern Shell CLEANUP_SH;
extern Shell GIT_SH;
extern Shell TAR_SH;
extern Shell CURL_SH;
extern Shell CONFIGURE_SH;
extern Shell MAKE_SH;
extern Shell INSTALL_SH;

#endif
```

确认你一字不差地创建了 `shell.h`，而且获得了一样名称和数量的 *extern Shell* 变量。这些变量将被 *Shell_run* 函数和 *Shell_exec* 函数用以执行外部命令。我定义了这两个函数，然后在 `shell.c` 中创建了真正用到的变量。

shell.c

```c
#include "shell.h"
#include "dbg.h"
#include <stdarg.h>

```

```c
int Shell_exec(Shell template, ...)
{
    apr_pool_t *p = NULL;
    int rc = -1;
    apr_status_t rv = APR_SUCCESS;
    va_list argp;
    const char *key = NULL;
    const char *arg = NULL;
    int i = 0;

    rv = apr_pool_create(&p, NULL);
    check(rv == APR_SUCCESS, "Failed to create pool.");

    va_start(argp, template);

    for(key = va_arg(argp, const char *);
        key != NULL;
        key = va_arg(argp, const char *))
    {
        arg = va_arg(argp, const char *);

        for(i = 0; template.args[i] != NULL; i++) {
            if(strcmp(template.args[i], key) == 0) {
                template.args[i] = arg;
                break; // found it
            }
        }
    }

    rc = Shell_run(p, &template);
    apr_pool_destroy(p);
    va_end(argp);
    return rc;
error:
    if(p) {
        apr_pool_destroy(p);
    }
    return rc;
}

int Shell_run(apr_pool_t *p, Shell *cmd)
{
    apr_procattr_t *attr;
    apr_status_t rv;
    apr_proc_t newproc;

    rv = apr_procattr_create(&attr, p);
    check(rv == APR_SUCCESS, "Failed to create proc attr.");

    rv = apr_procattr_io_set(attr, APR_NO_PIPE, APR_NO_PIPE,
            APR_NO_PIPE);
    check(rv == APR_SUCCESS, "Failed to set IO of command.");

    rv = apr_procattr_dir_set(attr, cmd->dir);
    check(rv == APR_SUCCESS, "Failed to set root to %s", cmd->dir);

    rv = apr_procattr_cmdtype_set(attr, APR_PROGRAM_PATH);
    check(rv == APR_SUCCESS, "Failed to set cmd type.");

    rv = apr_proc_create(&newproc, cmd->exe, cmd->args, NULL, attr, p);
    check(rv == APR_SUCCESS, "Failed to run command.");

    rv = apr_proc_wait(&newproc, &cmd->exit_code, &cmd->exit_why, APR_WAIT);
    check(rv == APR_CHILD_DONE, "Failed to wait.");

    check(cmd->exit_code == 0, "%s exited badly.", cmd->exe);
    check(cmd->exit_why == APR_PROC_EXIT, "%s was killed or crashed", cmd->exe);

    return 0;

error:
    return -1;
}

Shell CLEANUP_SH = {
    .exe = "rm",
    .dir = "/tmp",
    .args = {"rm", "-rf", "/tmp/pkg-build", "/tmp/pkg-src.tar.gz",
        "/tmp/pkg-src.tar.bz2", "/tmp/DEPENDS", NULL}
};

Shell GIT_SH = {
    .dir = "/tmp",
    .exe = "git",
    .args = {"git", "clone", "URL", "pkg-build", NULL}
};

Shell TAR_SH = {
```

```c
    .dir = "/tmp/pkg-build",
    .exe = "tar",
    .args = {"tar", "-xzf", "FILE", "--strip-components", "1", NULL}
};

Shell CURL_SH = {
    .dir = "/tmp",
    .exe = "curl",
    .args = {"curl", "-L", "-o", "TARGET", "URL", NULL}
};

Shell CONFIGURE_SH = {
    .exe = "./configure",
    .dir = "/tmp/pkg-build",
    .args = {"configure", "OPTS", NULL},
};

Shell MAKE_SH = {
    .exe = "make",
    .dir = "/tmp/pkg-build",
    .args = {"make", "OPTS", NULL}
};

Shell INSTALL_SH = {
    .exe = "sudo",
    .dir = "/tmp/pkg-build",
    .args = {"sudo", "make", "TARGET", NULL}
};
```

从下往上倒着阅读 **shell.c** 里边的代码（这也是常见的 C 代码布局格式），你会看到我创建了实际用到的 *Shell* 变量，它们在 **shell.h** 中是以 *extern* 的形式声明的。它们就住在这里，但它们对于程序剩下的部分都是有效的。这样你就创建了一个住在 **.o** 文件中，但是可以在任何位置使用的全局变量。创造太多的全局变量不是一件好事情，不过在当前场合下还是比较合适的。

继续向上阅读我们就看到了 *Shell_run* 函数，这是一个“基础”函数，它会基于 *Shell* struct 中的内容执行一条指定的命令。这里使用了很多定义在 **apr_thread_proc.h** 中的函数，所以你需要去找出并学习每条函数的作用。比起直接调用 *system* 而言，这样似乎麻烦了许多，不过这样做的一个好处就是能让你更好地控制外部命令执行的一些细节。例如，在 *Shell* struct 中我们有一个 *.dir* 属性，它可以强制规定程序在某个指定的路径下运行。

最后，我写了 *Shell_exec* 这个函数，它是一个“可变参数”函数。这样的函数你已经见过了，不过请确认你明白了 **stdarg.h** 里的各个函数定义，并且知道怎样可以写出来这样的函数。在本节的挑战环节你将需要却分析这个函数。

挑战 2: 分析 Shell_exec

在这里除了和挑战 1 里一样的完整代码审查以外，你还要完整地分析 *Shell_exec* 函数，详细了解它的工作原理。你需要弄懂每一行，包括两个 *for-loop* 的原理，以及参数是怎样被替换的。

分析完以后，在 *struct Shell* 中添加一栏用以表示需要替换的 *args* 变量的个数。更新所有的命令，修正它们的参数个数，然后添加错误检查功能，以确认参数被正确替换。

27.4.3 命令函数

现在你要做的是让命令切实生效。这些命令将用到 APR、**db.h**、**shell.h** 中的函数来实现诸如下载以及 build 此类真正的功能。这些文件中的内容及其复杂，所以做这部分时要很小心。和之前一样，你要先完成 **commands.h** 文件，然后在 **commands.c** 中实现前面定义的函数。

commands.h

```c
#ifndef _commands_h
#define _commands_h

#include <apr_pools.h>

#define DEPENDS_PATH "/tmp/DEPENDS"
#define TAR_GZ_SRC "/tmp/pkg-src.tar.gz"
#define TAR_BZ2_SRC "/tmp/pkg-src.tar.bz2"
#define BUILD_DIR "/tmp/pkg-build"
#define GIT_PAT "*.git"
#define DEPEND_PAT "*DEPENDS"
#define TAR_GZ_PAT "*.tar.gz"
#define TAR_BZ2_PAT "*.tar.bz2"
#define CONFIG_SCRIPT "/tmp/pkg-build/configure"

enum CommandType {
    COMMAND_NONE, COMMAND_INSTALL, COMMAND_LIST, COMMAND_FETCH,
    COMMAND_INIT, COMMAND_BUILD
};


int Command_fetch(apr_pool_t *p, const char *url, int fetch_only);

int Command_install(apr_pool_t *p, const char *url, const char *configure_opts,
        const char *make_opts, const char *install_opts);

int Command_depends(apr_pool_t *p, const char *path);

int Command_build(apr_pool_t *p, const char *url, const char *configure_opts,
        const char *make_opts, const char *install_opts);

#endif
```

commands.h 里边没有什么新东西。你可以看到里边定义了一些各处都会用到的字符串，真正有意思的是接下来的 commands.c。

commands.c

```c
#include <apr_uri.h>
#include <apr_fnmatch.h>
#include <unistd.h>

#include "commands.h"
#include "dbg.h"
#include "bstrlib.h"
#include "db.h"
#include "shell.h"


int Command_depends(apr_pool_t *p, const char *path)
{
    FILE *in = NULL;
    bstring line = NULL;

    in = fopen(path, "r");
    check(in != NULL, "Failed to open downloaded depends: %s", path);

    for(line = bgets((bNgetc)fgetc, in, '\n'); line != NULL;
            line = bgets((bNgetc)fgetc, in, '\n'))
    {
        btrimws(line);
        log_info("Processing depends: %s", bdata(line));
        int rc = Command_install(p, bdata(line), NULL, NULL, NULL);
        check(rc == 0, "Failed to install: %s", bdata(line));
        bdestroy(line);
    }

    fclose(in);
    return 0;

error:
    if(line) bdestroy(line);
    if(in) fclose(in);
    return -1;
}

int Command_fetch(apr_pool_t *p, const char *url, int fetch_only)
{
    apr_uri_t info = {.port = 0};
    int rc = 0;
    const char *depends_file = NULL;
    apr_status_t rv = apr_uri_parse(p, url, &info);

    check(rv == APR_SUCCESS, "Failed to parse URL: %s", url);

    if(apr_fnmatch(GIT_PAT, info.path, 0) == APR_SUCCESS) {
        rc = Shell_exec(GIT_SH, "URL", url, NULL);
        check(rc == 0, "git failed.");
    } else if(apr_fnmatch(DEPEND_PAT, info.path, 0) == APR_SUCCESS) {
        check(!fetch_only, "No point in fetching a DEPENDS file.");

        if(info.scheme) {
            depends_file = DEPENDS_PATH;
            rc = Shell_exec(CURL_SH, "URL", url, "TARGET", depends_file, NULL);
            check(rc == 0, "Curl failed.");
        } else {
            depends_file = info.path;
        }

        // recursively process the devpkg list
        log_info("Building according to DEPENDS: %s", url);
        rv = Command_depends(p, depends_file);
        check(rv == 0, "Failed to process the DEPENDS: %s", url);

        // this indicates that nothing needs to be done
        return 0;

    } else if(apr_fnmatch(TAR_GZ_PAT, info.path, 0) == APR_SUCCESS) {
        if(info.scheme) {
            rc = Shell_exec(CURL_SH,
                    "URL", url,
                    "TARGET", TAR_GZ_SRC, NULL);
            check(rc == 0, "Failed to curl source: %s", url);
        }

        rv = apr_dir_make_recursive(BUILD_DIR,
                APR_UREAD | APR_UWRITE | APR_UEXECUTE, p);
        check(rv == APR_SUCCESS, "Failed to make directory %s", BUILD_DIR);

        rc = Shell_exec(TAR_SH, "FILE", TAR_GZ_SRC, NULL);
        check(rc == 0, "Failed to untar %s", TAR_GZ_SRC);
    } else if(apr_fnmatch(TAR_BZ2_PAT, info.path, 0) == APR_SUCCESS) {
        if(info.scheme) {
            rc = Shell_exec(CURL_SH, "URL", url, "TARGET", TAR_BZ2_SRC, NULL);
            check(rc == 0, "Curl failed.");
        }

        apr_status_t rc = apr_dir_make_recursive(BUILD_DIR,
                APR_UREAD | APR_UWRITE | APR_UEXECUTE, p);

        check(rc == 0, "Failed to make directory %s", BUILD_DIR);
        rc = Shell_exec(TAR_SH, "FILE", TAR_BZ2_SRC, NULL);
        check(rc == 0, "Failed to untar %s", TAR_BZ2_SRC);
    } else {
        sentinel("Don't now how to handle %s", url);
    }

    // indicates that an install needs to actually run
    return 1;
error:
    return -1;
}

int Command_build(apr_pool_t *p, const char *url, const char *configure_opts,
        const char *make_opts, const char *install_opts)
{
    int rc = 0;

    check(access(BUILD_DIR, X_OK | R_OK | W_OK) == 0,
            "Build directory doesn't exist: %s", BUILD_DIR);

    // actually do an install
    if(access(CONFIG_SCRIPT, X_OK) == 0) {
        log_info("Has a configure script, running it.");
        rc = Shell_exec(CONFIGURE_SH, "OPTS", configure_opts, NULL);
        check(rc == 0, "Failed to configure.");
    }

    rc = Shell_exec(MAKE_SH, "OPTS", make_opts, NULL);
    check(rc == 0, "Failed to build.");

    rc = Shell_exec(INSTALL_SH,
            "TARGET", install_opts ? install_opts : "install",
            NULL);
    check(rc == 0, "Failed to install.");

    rc = Shell_exec(CLEANUP_SH, NULL);
    check(rc == 0, "Failed to cleanup after build.");

    rc = DB_update(url);
    check(rc == 0, "Failed to add this package to the database.");

    return 0;

error:
    return -1;
}

int Command_install(apr_pool_t *p, const char *url, const char *configure_opts,
        const char *make_opts, const char *install_opts)
{
    int rc = 0;
    check(Shell_exec(CLEANUP_SH, NULL) == 0, "Failed to cleanup before building.");

    rc = DB_find(url);
    check(rc != -1, "Error checking the install database.");

    if(rc == 1) {
        log_info("Package %s already installed.", url);
        return 0;
    }

    rc = Command_fetch(p, url, 0);

    if(rc == 1) {
        rc = Command_build(p, url, configure_opts, make_opts, install_opts);
        check(rc == 0, "Failed to build: %s", url);
    } else if(rc == 0) {
        // no install needed
        log_info("Depends successfully installed: %s", url);
    } else {
        // had an error
        sentinel("Install failed: %s", url);
    }

    Shell_exec(CLEANUP_SH, NULL);
    return 0;

error:
    Shell_exec(CLEANUP_SH, NULL);
    return -1;
}
```

输入完成并编译成功以后，你就可以对它进行分析了。如果你完成了前面的挑战，那你应该看出 **shell.c** 中的函数是如何被用以运行 shell，以及参数是如何被替换的。如果你看不明白，那你需要回到前面，知道真正弄懂 *Shell_exec* 的工作原理为止。

挑战 3: 评判我的设计

和之前一样，做一次完整的代码审核，确认你写的和书里的完全一致。然后详细阅读每一个函数，弄明白它们的功能。试着去追踪每一个函数是怎样去调用你写在当前文件或者其它代码文件中的函数的。最后，确认你弄懂了你从 APR 调用的所有的函数。

确认文件内容正确，而且分析完成以后，回头重新审查一遍。这一次你就把我当成一个白痴，你可以批评我的设计方案，并且提出能够改善的地方。不要真去修改代码，创建一个 **notes.txt** 并写下你觉得可以改进的东西即可。

27.4.4 devpkg 的主函数

最后也是最重要的代码文件是 **devpkg.c**，不过这可能也是最简单的代码文件了。它就是 *main* 函数所在的地方。这里我们不需要 **.h** 文件，因为它已经包含了所有的头文件。当它和 **Makefile** 中包含的 **.o** 文件搭配编译的时候，我们就会编译出可执行文件 *devpkg*。输入该文件的代码并确认没有写错。

devpkg.c

```
#include <stdio.h>
#include <apr_general.h>
#include <apr_getopt.h>
#include <apr_strings.h>
#include <apr_lib.h>

#include "dbg.h"
#include "db.h"
#include "commands.h"

int main(int argc, const char const *argv[])
{
    apr_pool_t *p = NULL;
    apr_pool_initialize();
    apr_pool_create(&p, NULL);

    apr_getopt_t *opt;
    apr_status_t rv;

    char ch = '\0';
    const char *optarg = NULL;
    const char *config_opts = NULL;
    const char *install_opts = NULL;
    const char *make_opts = NULL;
    const char *url = NULL;
    enum CommandType request = COMMAND_NONE;


    rv = apr_getopt_init(&opt, p, argc, argv);

    while(apr_getopt(opt, "I:Lc:m:i:d:SF:B:", &ch, &optarg) == APR_SUCCESS) {
        switch (ch) {
            case 'I':
                request = COMMAND_INSTALL;
                url = optarg;
                break;

            case 'L':
                request = COMMAND_LIST;
                break;

            case 'c':
                config_opts = optarg;
                break;

            case 'm':
                make_opts = optarg;
                break;

            case 'i':
                install_opts = optarg;
                break;

            case 'S':
                request = COMMAND_INIT;
                break;

            case 'F':
                request = COMMAND_FETCH;
                url = optarg;
                break;

            case 'B':
                request = COMMAND_BUILD;
                url = optarg;
                break;
        }
    }

    switch(request) {
        case COMMAND_INSTALL:
            check(url, "You must at least give a URL.");
            Command_install(p, url, config_opts, make_opts, install_opts);
            break;

        case COMMAND_LIST:
            DB_list();
            break;

        case COMMAND_FETCH:
            check(url != NULL, "You must give a URL.");
            Command_fetch(p, url, 1);
            log_info("Downloaded to %s and in /tmp/", BUILD_DIR);
            break;

        case COMMAND_BUILD:
            check(url, "You must at least give a URL.");
            Command_build(p, url, config_opts, make_opts, install_opts);
            break;

        case COMMAND_INIT:
            rv = DB_init();
            check(rv == 0, "Failed to make the database.");
            break;

        default:
            sentinel("Invalid command given.");
    }


    return 0;

error:
    return 1;
}
```

挑战 4: README 和测试文件

对于这个代码文件的挑战是弄懂参数是什么意思，以及它们是怎样被处理的。然后你需要创建一个 README 文件来说明该程序的用法。在你写 README 的同时再写一个简单的 `test.sh`，让它去运行 *./devpkg* 以检查所有的命令是否都有效。在你的脚本最上方使用 `set -e`，这样脚本就会在第一次发生错误的时候就停止执行。

最后，在 valgrind 下面运行程序，并确认一切工作正常。这也是期中考试前的最后一件事情。

27.5 测验

你的最后一项挑战就是期中考试了，你要完成三件事情：

1. 将你的代码和我放在网上的代码比较，以 100 分为基数，每一处写错的地方扣掉 1 分。
2. 将你记在 notes.txt 中的代码或者功能方面可以改进的地方通过修改程序去一一实现。
3. 用你喜欢或者熟悉的另外一门语言写个新版本的 *devpkg*，和 C 语言版的比较，然后通过你学到的东西去改进 *C* 语言版的 *devpkg*。

通过下面的方法比较你的代码和我的代码:

```
cd ..  # get one directory above your current one
git clone git://gitorious.org/devpkg/devpkg.git devpkgzed
diff -r devpkg devpkgzed
```

上面的命令实现的是将我写的的版本克隆（clone）到本地 *devpkgzed* 文件夹下，然后使用 *diff* 将其和你写的版本进行比较。本书中的代码直接来自于你刚克隆的这个项目，所以如果哪行和你的不一样，那就说明出错了。

这次期中考试并没有及格不及格的说法，这样的练习方式是为了让你挑战自己，让自己尽可能做到精确细致。

Part II

进阶应用

Chapter 28

习题 27: 创造性与防御性编程

You have now learned most of the basics of C programming and are ready to start becoming a serious programmer. This is where you go from beginner to expert, both with C and hopefully with core computer science concepts. I will be teaching you a few of the core data structures and algorithms that every programmer should know, and then a few very interesting ones I've used in real software for years. 你已经学习了大部分 C 语言编程的基础知识，已经可以开始做一个真正的程序员了。这正是你从入门到专家的起点，依靠 C 还有计算机科学的核心概念。我会传教你一些每个程序员都应该知道的关键的数据结构和算法，然后就是一些我在这些年实际软件编程中觉得有意思的东西。

Before I can do that I have to teach you some basic skills and ideas that will help you make better software. Exercises 27 through 31 will teach you advanced concepts and feature more talking than code, but after those you'll apply what you learn to making a core library of useful data structures. 在开始之前，我不得不先教一些基础的技巧和思想，可以帮助你更好的编程。习题 27 到 32 会通过较多的文字稍少的代码来教你一些高级的概念和特性，但在这之后，你应该用你所学去做一个有用的数据结构的核心库。

The first step in getting better at writing C code (and really any language) is to learn a new mindset called "defensive programming". Defensive programming assumes that you are going to make many mistakes and then attempts to prevent them at every possible step. In this exercise I'm going to teach you how to think about programming defensively. 要写出更好的 C 代码（任何语言），第一步就是要学习一种叫做“防御性编程”的思维模式。防御性编程是设想你会出很多错误而且你应该在任何可能的步骤中尝试阻止错误发生。在本次习题中，我将教给你如何去思考防御性的编程。

28.1 The Creative Programmer Mindset

It's not possible to tell you how to be creative in a short exercise like this, but I will tell you that creativity involves taking risks and being open minded. Fear will quickly kill creativity, so the mindset I adopt, and many programmers adopt on, accident is designed to make me unafraid of taking chances and looking like an idiot: 在这样一个简短的练习中不可能叫你富有创造性，但是我会告诉你创造力包含了冒风险与发散思维。害怕失败或者出错会扼杀创造力，所以我采用的思维模式也是大多数程序员采取的思维模式就是事故或者失误的发生就是为了让我不惧风险并且看起来像个白痴：

1. 我一个错误也不会有。
2. 无所谓大家怎么在想。
3. 所有我脑海中的点子都是伟大的。

I only adopt this mindset temporarily, and even have little tricks to turn it on. By doing this I can come up with ideas, find creative solutions, open my thoughts to odd connections, and just generally invent weirdness without fear. In this mindset I will typically write a horrible first version of something just to get the idea out. 我是暂时这么想，而且是有技巧的让自己这么想。通过这样做，我可以让自己思维迸发，找到创新的解决方法，触类旁通毫无畏惧大胆思考。在这种思维模式下，我会很典型的写出一些糟糕的初版代码，只是为了将自己的想法表达出来。

However, when I've finished my creative prototype I will throw it out and get serious about making it solid. Where other people make a mistake is carrying the creative mindset into their implementation phase. This then leads to a very different destructive mindset that is the dark side of the creative mindset: 然而，当我完成了创新的原型我会丢掉这种心态开始认真的将它充实起来。而其他人会错误的将这种创造性思维模式带入到实现阶段。这会导致一种破坏性的心态，这是创新性思维模式的黑暗面：

1. 写出完美的软件是有可能的。
2. My brain tells me the truth, and it can't find any errors, therefore I have written perfect software. 我的大脑总是正确的，既然我没发现任何错误，那么我所写的就是完美的软件。
3. My code is who I am and people who criticize its perfection are criticizing me. 代码如人，批评我代码完美的人就是在批评我。

These are lies. You will frequently run into programmers who feel intense pride about what they've created, which is natural, but this pride gets in the way of their ability to objectively improve their craft. Because of pride and attachment to what they've written, they can continue to believe that what they write is perfect. As long as they ignore other people's criticism of their code they can protect their fragile ego and never improve. 这些都是谎言。你会经常遇到一些程序员对他们所创造的东西感到强烈的自豪，这很正常，但是这种自豪感阻碍了他们客观的提高自己手艺的能力。因为骄傲以及他们已经取得的成绩，他们一直相信他们写的代码是完美的。只要忽视了别人对他们代码的批评，他们就能保护脆弱的自我并且不再进步。

The trick to being creative *and* making solid software is to also be able to adopt a defensive programming mindset. 让自己富有创造力并写出扎实的软件的技巧也能够使用防御性编程的思维模式。

28.2 The Defensive Programmer Mindset 防御性编程思维模式

After you have a working creative prototype and you're feeling good about the idea, it's time to switch to being a defensive programmer. The defensive programmer basically hates your code and believes these things: 在你已经有了一个可以工作的创新性的原型并且你对这个主意感觉良好的时候，就到了该切换成一名防御性的程序员的时候了。防御性程序员基本上是讨厌你的代码并且深信以下几点：

1. Software has errors. 软件会出错。
2. You are not your software, yet you are are responsible for the errors. 人不代表代码，但是你要对代码出错负责。
3. You can never remove the errors, only reduce their probability. 你不可能消除错误，只能降低发生的概率。

This mindset lets you be honest about your work and critically analyze it for improvements. Notice that it doesn't say *you* are full of errors? It says your *code* is full of errors. This is a significant thing to understand because it gives you the power of objectivity for the next implementation. 这种思维模式让你诚实对待自己的工作并且能够批判性的分析进而改进。注意这不是说 你 充满错误，而是说的是你的代码 代码 充满错误。明白这一点非常的重要，因为它给了你接下来的实现以客观的力量。

Just like the creative mindset, the defensive programming mindset has a dark side as well. The defensive programmer is a paranoid who is afraid of everything, and this fear prevents them from possibly being wrong or making mistakes. That's great when you are trying to be ruthlessly consistent and correct, but it is murder on creative energy and concentration. 如同创造性的思维模式，防御性编程的思维模式也有其阴暗面。防御性编程者是一个处处担心的偏执狂，这种担心使得他们可以免于出错或者失误。当你需要绝对的统一以及保证正确的时候，这样做很好，但是它会扼杀创造力和专注力。

28.3 八个防御性编程的策略

Once you've adopted this mindset, you can then rewrite your prototype and follow a set of eight strategies I use to make my code as solid as I can. While I work on the "real" version I ruthlessly follow these strategies and try to remove as many errors as I can, thinking like someone who wants to break the software. 一旦你采用了这种思维模式，你就可以遵循我常用的这八个策略来重写你的原型使其更加的充实。当我在完成这个“真实”版本的时候，我会一边严格的遵循这些策略并且尽可能的减少错误，一边幻想有人在试图突破这个软件。

从不信任输入 Never trust the data you are given and always validate it. 从不信任得到的数据，并且总是进行验证。

预防错误 If an error is possible, no matter how probable, try to prevent it. 如果有可能出错，就算可能性再小，也要试图预防。

出错要尽早且有错误输出 Fail early, cleanly, and openly, stating what happened, where and how to fix it. 较早的凸显错误，出错信息要清晰开放，表明出了什么错，在哪里出错以及如何修正。

前提建档 Clearly state the pre-conditions, post-conditions, and invariants. 清晰的记录前提条件，后置条件以及不变量。

靠预防措施不靠文档说明 Do not do with documentation, that which can be done with code or avoided completely. 可以完全用代码解决或者避免的问题就不用靠文档去做说明。

自动化 全部自动化，尤其是测试。

简洁明了 Always simplify the code to the smallest, cleanest form that works without sacrificing safety. 要在不牺牲安全性的前提下简化代码到最精简最整洁形式。

质疑权威 Do not blindly follow or reject rules. 不要盲从或者盲目排斥任何规则。

These aren't the only ones, but they're the core things I feel programmers have to focus on when trying to make good solid code. Notice that I don't really say exactly how to do these. I'll go into each of these in more detail, and some of the exercises actually cover them extensively. 这些不是全部，但是这些是我觉得程序员在试图写出扎实代码时必须要关注的核心策略。需要注意我并没有详细说出如何去做。我会更详细的介绍每一个，并且有一些练习题很好的覆盖了这些策略。

28.4 Applying The Eight Strategies 应用八大策略

I'll now go through each of the eight strategies and give some basic advice and examples on how to use them in real code. This will help you understand these better since they may be vague or misinterpreted. 现在我就依次介绍这八个策略，就如何在实战中使用给出简单建议和例子。这样会让你加深理解，因为这几点可能不太明确或被误解。

28.4.1 Never Trust Input 从不信任输入

Explain untrusted inputs. Use a few examples from the real world, maybe the recent Rails and Github attack. Show a simple C examples using C strings taken from a socket. 解释什么是不被信任的输入。可以找一些真实的例子，比如最近的 Rails 和 Github 的攻击。用来自 socket 的 C string 做一个简单的 C 代码例子。

28.4.2 Prevent Errors 预防错误

Discuss the difference between a possible error and a probable error, then how humans are very bad at determining probability, therefore you should try to block all the possible errors you can. 讨论一下可能错误与概率错误的差别，以及人类在确定概率上是多么的糟糕，因此你应该尽可能阻止可能错误。

28.4.3 Fail Early And Openly 出错要尽早且有错误输出

Show how my awesome macros help you do this and discuss good error messages. Make sure they understand that you should try to explain how to maybe fix it, or report the defect. 展示了用我的宏来帮你做到这一点并讨论了优秀的错误信息。你应该尝试解释如何修正错误或者提出问题以确保他们明白。

28.4.4 Document Assumptions 前提建档

Explain design by contract and how you can use it to create pre-conditions, post-conditions, and invariants. Show a simple example of this with a function that has it all.

28.4.5 Prevention Over Documentation 靠预防措施不靠文档说明

Talk about how programmers think that a documented flaw means there is no flaw and that they should just remove the flaw.

28.4.6 Automate Everything 自动化

Discuss the advantages of automated testing and how I'll teach that in a later exercise.

28.4.7 Simplify And Clarify 简洁明了

Talk about how simpler code wins over more complex code because it reduces the probability of an error by reducing the number of branches and couplings in the code.

28.4.8 Question Authority 质疑权威

The final strategy is the most important because it breaks you out of the defensive programming mindset and lets you transition into the creative mindset. Defensive programming is authoritarian and it can be cruel. The job of this mindset is to make you follow rules because without them you'll miss something or get distracted.

This authoritarian attitude has the disadvantage of disabling independent creative thought. Rules are necessary for getting things done, but being a slave to them will kill your creativity.

This final strategy means you should question the rules you follow periodically and assume that they could be wrong, just like the software you are reviewing. What I will typically do is, after a session of defensive programming, I'll go take a non-programming break and let the rules go. Then I'll be ready to do some creative work or do more defensive coding if need to.

28.5 Order Is Not Important 秩序不重要

The final thing I'll say on this philosophy is that I'm not telling you to do this in a strict oder of "CREATE! DEFEND! CREATE! DEFEND!" At first you may want to do that, but I will actually do either in varying amounts depend on what I want to do, and I may even meld them together with no defined boundary.

I also don't think one mindset is better than another, or that there are strict separation between them. You need both creativity and strictness to do programming well, so work on both if you want to improve.

28.6 加分题

1. The code in the book up to this point (and for the rest of it) potentially violates these rules. Go back through and apply what you've learned to one exercise to see if you can improve it or find bugs.
2. Find an open source project and give some of the files a similar code review. Submit a patch that fixes a bug if you find it.

Chapter 29

Exercise 28: Intermediate Makefiles

In the next three Exercises you'll create a skeleton project directory to use in building your C programs later. This skeleton directory will be used in the rest of the book, and in this exercise I'll cover just the **`Makefile`** so you can understand it.

The purpose of this structure is to make it easy to build medium sized programs without having to resort to configure tools. If done right you can get very far with just GNU make and some small shell scripts.

29.1 The Basic Project Structure

The first thing to do is make a **`c-skeleton`** directory and then put a set of basic files and directories in it that many projects have. Here's my starter:

Initial C Project Skeleton

```
$ mkdir c-skeleton
$ cd c-skeleton/
$ touch LICENSE README.md Makefile
$ mkdir bin src tests
$ cp dbg.h src/   # this is from Ex20
$ ls -l
total 8
-rw-r--r--  1 zedshaw  staff     0 Mar 31 16:38 LICENSE
-rw-r--r--  1 zedshaw  staff  1168 Apr  1 17:00 Makefile
-rw-r--r--  1 zedshaw  staff     0 Mar 31 16:38 README.md
drwxr-xr-x  2 zedshaw  staff    68 Mar 31 16:38 bin
drwxr-xr-x  2 zedshaw  staff    68 Apr  1 10:07 build
drwxr-xr-x  3 zedshaw  staff   102 Apr  3 16:28 src
drwxr-xr-x  2 zedshaw  staff    68 Mar 31 16:38 tests
$ ls -l src
total 8
```

```
-rw-r--r--  1 zedshaw  staff  982 Apr  3 16:28 dbg.h
$
```

At the end you see me do an `ls -l` so you can see the final results.

Here's what each of these does:

LICENSE If you release the source of your projects you'll want to include a license. If you don't though, the code is copyright by you and nobody has rights to it by default.

README.md Basic instructions for using your project go here. It ends in `.md` so that it will be interpreted as markdown.

Makefile The main build file for the project.

bin/ Where programs that users can run go. This is usually empty and the Makefile will create it if it's not there.

build/ Where libraries and other build artifacts go. Also empty and the Makefile will create it if it's not there.

src/ Where the source code goes, usually `.c` and `.h` files.

tests/ Where automated tests go.

src/dbg.h I copied the `dbg.h` from Exercise 20 into `src/` for later.

I'll now break down each of the compontents of this skeleton project so you can understand how it works.

29.2 Makefile

The first thing I'll cover is the Makefile because from that you can understand how everything else works. The Makefile in this exercise is much more detailed than ones you've used so far, so I'm going to break it down after you type it in:

c-skeleton/Makefile

```
CFLAGS=-g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG $(OPTFLAGS)
LIBS=-ldl $(OPTLIBS)
PREFIX?=/usr/local

SOURCES=$(wildcard src/**/*.c src/*.c)
OBJECTS=$(patsubst %.c,%.o,$(SOURCES))

TEST_SRC=$(wildcard tests/*_tests.c)
TESTS=$(patsubst %.c,%,$(TEST_SRC))

TARGET=build/libYOUR_LIBRARY.a
SO_TARGET=$(patsubst %.a,%.so,$(TARGET))

```

```
# The Target Build
all: $(TARGET) $(SO_TARGET) tests

dev: CFLAGS=-g -Wall -Isrc -Wall -Wextra $(OPTFLAGS)
dev: all

$(TARGET): CFLAGS += -fPIC
$(TARGET): build $(OBJECTS)
        ar rcs $@ $(OBJECTS)
        ranlib $@

$(SO_TARGET): $(TARGET) $(OBJECTS)
        $(CC) -shared -o $@ $(OBJECTS)

build:
        @mkdir -p build
        @mkdir -p bin

# The Unit Tests
.PHONY: tests
tests: CFLAGS += $(TARGET)
tests: $(TESTS)
        sh ./tests/runtests.sh

valgrind:
        VALGRIND="valgrind --log-file=/tmp/valgrind-%p.log" $(MAKE)

# The Cleaner
clean:
        rm -rf build $(OBJECTS) $(TESTS)
        rm -f tests/tests.log
        find . -name "*.gc*" -exec rm {} \;
        rm -rf `find . -name "*.dSYM" -print`

# The Install
install: all
        install -d $(DESTDIR)/$(PREFIX)/lib/
        install $(TARGET) $(DESTDIR)/$(PREFIX)/lib/

# The Checker
BADFUNCS='[^_.>a-zA-Z0-9](str(n?cpy|n?cat|xfrm|n?dup|str|pbrk|tok|_)|stpn?cpy|a?sn?printf|byte
check:
        @echo Files with potentially dangerous functions.
```

```
	@egrep $(BADFUNCS) $(SOURCES) || true
```

Remember that you need to indent the Makefile consistently with tab characters. Your editor should know that and do the right thing, but if it doesn't then get a different text editor. No programmer should use an editor that fails at something so simple.

29.2.1 The Header

This makefile is designed to build a library we'll be working on later and to do so reliably on almost any platform by using special features of *GNU make*. I'll break down each part in sections, starting with the header.

Makefile:1 These are the usual *CFLAGS* that you set in all of your projects, but with a few others that may be needed to build libraries. *You may need to adjust these for different platforms.* Notice the *OPTFLAGS* variable at the end which lets people augment the build options as needed.

Makefile:2 Options used when linking a library, and allows someone else to augment the linking options using the *OPTLIBS* variable.

Makefile:3 Setting an *optional* variable called *PREFIX* that will only have this value if the person running the Makefile didn't already give a *PREFIX* setting. That's what the `?=` does.

Makefile:5 This fancy line of awesome *dynamically* creates the *SOURCES* variable by doing a *wildcard* search for all `*.c` files in the `src/` directory. You have to give both `src/**/*.c` and `src/*.c` so that GNU make will include the files in `src` and also the ones below it.

Makefile:6 Once you have the list of source files, you can then use the *patsubst* to take the *SOURCES* list of `*.c` files and make a *new* list of all the object files. You do this by telling *patsubst* to change all `%.c` extensions to `%.o` and then those are assigned to *OBJECTS*.

Makefile:8 Using the *wildcard* again to find all the test source files for the unit tests. These are separate from the library's source files.

Makefile:9 Then using the same *patsubst* trick to dynamically get all the *TEST* targets. In this case I'm stripping away the `.c` extension so that a full program will be made with the same name. Previously I had replaced the `.c` with .o so an object file is created.

Makefile:11 Finally, we say the ultimate target is `build/libYOUR_LIBRARY.a`, which you will change to be whatever library you are actually trying to build.

This completes the top of the Makefile, but I should explain what I mean by ”lets people augment the build”. When you run make you can do this:

A Changing A Make Build

```
# WARNING! Just a demonstration, won't really work right now.
# this installs the library into /tmp
$ make PREFIX=/tmp install
```

```
# this tells it to add pthreads
$ make OPTFLAGS=-pthread
```

If you pass in options that match the same kind of variables you have in your `Makefile`, then those will show up in your build. You can then use this to change how the `Makefile` runs. The first one alters the *PREFIX* so that it installs into `/tmp` instead. The second one sets *OPTFLAGS* so that the *-pthread* option is present.

29.2.2 The Target Build

Continuing with the breakdown of the `Makefile` I have actually building the object files and targets:

Makefile:14 Remember that the first target is what *make* will run by default when no target is given. In this case it's called *all:* and it gives `$(TARGET) tests` as the targets to build. Look up at the *TARGET* variable and you see that's the library, so *all:* will first build the library. The *tests* target is then further down in the *Makefile* and builds the unit tests.

Makefile:16 Another target for making "developer builds" that introduces a technique for changing options for just one target. If I do a "dev build" I want the *CFLAGS* to include options like *-Wextra* that are useful for finding bugs. If you place them on the target line as options like this, then give another line that says the original target (in this case *all*) then it will change the options you set. I use this for setting different flags on different platforms that need it.

Makefile:19 Builds the *TARGET* library, whatever that is, and also uses the same trick from line 15 of giving a target with just options changes to alter them for this run. In this case I'm adding *-fPIC* just for the library build using the `+=` syntax to add it on.

Makefile:20 Now the real target where I say first make the `build` directory, then compile all the *OBJECTS*.

Makefile:21 Runs the *ar* command which actually makes the *TARGET*. The syntax `$@ $(OBJECTS)` is a way of saying, "put the target for this Makefile source here and all the OBJECTS after that". In this case the `$@` maps back to the `$(TARGET)` on line 19, which maps to `build/libYOUR_LIBRARY.a`. It seems like a lot to keep track of this indirection, and it can be, but once you get it working this means you just change *TARGET* at the top and build a whole new library.

Makefile:22 Finally, to make the library you run *ranlib* on the *TARGET* and it's built.

Makefile:24-24 This just makes the `build/` or `bin/` directories if they don't exist. This is then referenced from line 19 when it gives the *build* target to make sure the `build/` directory is made.

You now have all the stuff you need to build the software, so we'll create a way to build and run unit tests to do automated testing.

29.2.3 The Unit Tests

C is different from other languages because it's easier to create one tiny little program for each thing you're testing. Some testing frameworks try to emulate the module concept other languages have and do dynamic loading, but this doesn't work well in C. It's also unnecessary because you can just make a single program that's run for each test instead.

I'll cover this part of the Makefile, and then later you'll see the contents of the `tests/` directory that make it actually work.

Makefile:29 If you have a target that's not "real", but there is a directory or file with that name, then you need to tag the target with *.PHONY:* so *make* will ignore the file and always run.

Makefile:30 I use the same trick for modifying the *CFLAGS* variable to add the *TARGET* to the build so that each of the test programs will be linked with the *TARGET* library. In this case it will add **build/libYOUR_LIBRARY.a** to the linking.

Makefile:31 Then I have the actual *tests:* target which depends on all the programs listed in the *TESTS* variable we created in the header. This one line actually says, "Make, use what you know about building programs and the current CFLAGS settings to build each program in *TESTS*."

Makefile:32 Finally, when all of the *TESTS* are built, there's a simple shell script I'll create later that knows how to run them all and report their output. This line actually runs it so you can see the test results.

Makefile:34-35 In order to be able to dynamically rerun the tests with Valgrind there's a *valgrind:* target that sets the right variable and runs itself again. This puts the valgrind logs into **/tmp/valgrind-*.log** so you can go look and see what might be going on. The **tests/runtests.sh** then knows to run the test programs under Valgrind when it sees this *VALGRIND* variable.

For the unit testing to work you'll need to create a little shell script that knows how to run the programs. Go ahead and create this **tests/runtests.sh** script:

tests/runtests.sh

```
echo "Running unit tests:"

for i in tests/*_tests
do
    if test -f $i
    then
        if $VALGRIND ./$i 2>> tests/tests.log
        then
            echo $i PASS
        else
            echo "ERROR in test $i: here's tests/tests.log"
            echo "------"
            tail tests/tests.log
            exit 1
        fi
    fi
done

echo ""
```

I'll be using this later when I cover how unit tests work.

29.2.4 The Cleaner

I now have fully working unit tests, so next up is making things clean when I need to reset everything.

Makefile:38 The *`clean:`* target starts things off whenever we need to clean up the project.

Makefile:39-42 This cleans out most of the junk that various compilers and tools leave behind. It also gets rid of the `build/` directory and uses a trick at the end to cleanly erase the weird `*.dSYM` directories Apple's XCode leaves behind for debugging purposes.

If you run into junk that you need to clean out, simply augment the list of things being deleted in this target.

29.2.5 The Install

After that I'll need a way to install the project, and for a `Makefile` that's building a library I just need to put something in the common *PREFIX* directory, which is usually `/usr/local/lib`.

Makefile:45 This makes *`install:`* depend on the *`all:`* target so that when you run `make install` it will be sure to build everything.

Makefile:46 I then use the program *`install`* to create the target `lib` directory if it doesn't exist. In this case I'm trying to make the install as flexible as possible by using two variables that are conventions for installers. *DESTDIR* is handed to make by installers that do their builds in secure or odd locations to build packages. *PREFIX* is used when people want the project to be installed in someplace other than `/usr/local`.

Makefile:47 After that I'm just using *`install`* to actually install the library where it needs to go.

The purpose of the *`install`* program is to make sure things have the right permissions set. When you run `make install` you usually have to do it as the root user, so the typical build process is `make && sudo make install`.

29.2.6 The Checker

The very last part of this `Makefile` is a bonus that I include in my C projects to help me dig out any attempts to use the "bad" functions in C. Namely the string functions and other "unprotected buffer" functions.

Makefile:50 Sets a variable which is a big regex looking for bad functions like *`strcpy`*.

Makefile:51 The *`check:`* target so you can run a check whenever you need.

Makefile:52 Just a way to print a message, but doing *`@echo`* tells *`make`* to not print the command, just its output.

Makefile:53 Run the *`egrep`* command on the source files looking for any bad patterns. The `|| true` at the end is a way to prevent *`make`* from thinking that *`egrep`* not finding things is a failure.

When you run this it will have the odd effect that you'll get an error when there is nothing bad going on.

29.3 What You Should See

I have two more exercises to go before I'm done building the project skeleton directory, but here's me testing out the features of the `Makefile`.

Checking The Makefile

```
$ make clean
rm -rf build
rm -f tests/tests.log
find . -name "*.gc*" -exec rm {} \;
rm -rf `find . -name "*.dSYM" -print`
$ make check
Files with potentially dangerous functions.
^Cmake: *** [check] Interrupt: 2

$ make
ar rcs build/libYOUR_LIBRARY.a
ar: no archive members specified
usage:  ar -d [-TLsv] archive file ...
        ar -m [-TLsv] archive file ...
        ar -m [-abiTLsv] position archive file ...
        ar -p [-TLsv] archive [file ...]
        ar -q [-cTLsv] archive file ...
        ar -r [-cuTLsv] archive file ...
        ar -r [-abciuTLsv] position archive file ...
        ar -t [-TLsv] archive [file ...]
        ar -x [-ouTLsv] archive [file ...]
make: *** [build/libYOUR_LIBRARY.a] Error 1
$ make valgrind
VALGRIND="valgrind --log-file=/tmp/valgrind-%p.log" make
ar rcs build/libYOUR_LIBRARY.a
ar: no archive members specified
usage:  ar -d [-TLsv] archive file ...
        ar -m [-TLsv] archive file ...
        ar -m [-abiTLsv] position archive file ...
        ar -p [-TLsv] archive [file ...]
        ar -q [-cTLsv] archive file ...
        ar -r [-cuTLsv] archive file ...
        ar -r [-abciuTLsv] position archive file ...
        ar -t [-TLsv] archive [file ...]
        ar -x [-ouTLsv] archive [file ...]
make[1]: *** [build/libYOUR_LIBRARY.a] Error 1
make: *** [valgrind] Error 2
$
```

When I run the `clean:` target that works, but because I don't have any source files in the **`src/`** directory none of the other commands really work. I'll finish that up in the next exercises.

29.4 Extra Credit

1. Try to get the **`Makefile`** to actually work by putting a source and header file in **`src/`** and making the library. You shouldn't need a *`main`* function in the source file.
2. Research what functions the *`check:`* target is looking for in the *`BADFUNCS`* regular expression it's using.
3. If you don't do automated unit testing, then go read about it so you're prepared later.

Chapter 30

Exercise 29: Libraries And Linking

A central part of any C program is the ability to link it to libraries that your operating system provides. Linking is how you get additional features for your program that someone else created and packaged on the system. You've been using some standard libraries that are automatically included, but I'm going to explain the different types of libraries and what they do.

First off, libraries are poorly designed in every programming language. I have no idea why, but it seems language designers think of linking as something they just slap on later. They are usually confusing, hard to deal with, can't do versioning right, and end up being linked differently everywhere.

C is no different, but the way linking and libraries are done in C is an artifact of how the Unix operating system and executable formats were designed years ago. Learning how C links things helps you understand how your OS works and how it runs your programs.

To start off there are two basic types of libraries:

static You've made one of these when you used *ar* and *ranlib* to creat the `libYOUR_LIBRARY.a` in the last exercise. This kind of library is nothing more than a container for a set of `.o` object files and their functions, and you can treat it like one big `.o` file when building your programs.

dynamic These typically end in `.so`, `.dll` or about 1 million other endings on OSX depending on the version and who happened to be working that day. Seriously though, OSX adds `.dylib`, `.bundle`, and `.framework` with not much distinction between the three. These files are built and then placed in a common location. When you run your program the OS dynamically loades these files and links them to your program on the fly.

I tend to like static libraries for small to medium sized projects because they are easier to deal with and work on more operating systems. I also like to put all of the code I can into a static library so that I can then link it to unit tests and to the file programs as needed.

Dynamic libraries are good for larger systems, when space is tight, or if you have a large number of programs that use common functionality. In this case you don't want to statically link all of the code for the common features to every program, so you put it in a dynamic library so that it is loaded only once for all of them.

In the previous exercise I laid out how to make a static library (a `.a` file), and that's what I'll use in the rest of the book. In this exercise I'm going to show you how to make a simple .so library, and how to dynamically load it with the Unix *dlopen* system. I'll have you do this manually so that you understand everything that's actually happening, then the Extra Credit will be to use the **c-skeleton** skeleton to create it.

30.0.1 Dynamically Loading A Shared Library

To do this I will create two source files. One will be used to make a **libex29.so** library, the other will be a program called *ex29* that can load this library and run functions from it.

libex29.c

```c
#include <stdio.h>
#include <ctype.h>
#include "dbg.h"


int print_a_message(const char *msg)
{
    printf("A STRING: %s\n", msg);

    return 0;
}


int uppercase(const char *msg)
{
    int i = 0;

    // BUG: \0 termination problems
    for(i = 0; msg[i] != '\0'; i++) {
        printf("%c", toupper(msg[i]));
    }

    printf("\n");

    return 0;
}

int lowercase(const char *msg)
{
    int i = 0;

    // BUG: \0 termination problems
    for(i = 0; msg[i] != '\0'; i++) {
        printf("%c", tolower(msg[i]));
    }

    printf("\n");

    return 0;
}

int fail_on_purpose(const char *msg)
{
    return 1;
}
```

There's nothing fancy in there, although there's some bugs I'm leaving in on purpose to see if you've been paying attention. You'll fix those later.

What we want to do is use the functions *dlopen*, *dlsym* and *dlclose* to work with the above functions.

ex29.c

```c
#include <stdio.h>
#include "dbg.h"
#include <dlfcn.h>

typedef int (*lib_function)(const char *data);


int main(int argc, char *argv[])
{
    int rc = 0;
    check(argc == 4, "USAGE: ex29 libex29.so function data");

    char *lib_file = argv[1];
    char *func_to_run = argv[2];
    char *data = argv[3];

    void *lib = dlopen(lib_file, RTLD_NOW);
    check(lib != NULL, "Failed to open the library %s: %s", lib_file, dlerror());

    lib_function func = dlsym(lib, func_to_run);
    check(func != NULL, "Did not find %s function in the library %s: %s", func_to_run, lib_file, dlerror())

    rc = func(data);
    check(rc == 0, "Function %s return %d for data: %s", func_to_run, rc, data);

    rc = dlclose(lib);
    check(rc == 0, "Failed to close %s", lib_file);

    return 0;

error:
    return 1;
}
```

I'll now break this down so you can see what's going on in this small bit of useful code:

ex29.c:5 I'll use this function pointer definition later to call functions in the library. This is nothing new, but make sure you understand what it's doing.

ex29.c:17 After the usual setup for a small program, I use the *`dlopen`* function to load up the library indicated by *`lib_file`*. This function returns a handle that we use later and works a lot like opening a file.

ex29.c:18 If there's an error, I do the usual check and exit, but notice at then end that I'm using *`dlerror`* to find out what the library related error was.

ex29.c:20 I use *`dlsym`* to get a function out of the *`lib`* by it's *string* name in *`func_to_run`*. This is the powerful part, since I'm dynamically getting a pointer to a function based on a string I got from the command line *`argv`*.

ex29.c:23 I then call the *`func`* function that was returned, and check its return value.

ex29.c:26 Finally, I close the library up just like I would a file. Usually you keep these open the whole time the program is running, so closing at the end isn't as useful, but I'm demonstrating it here.

30.1 What You Should See

Now that you know what this file does, here's a shell session of me building the **`libex29.so`**, *ex29* and then working with it. Follow along so you learn how these things are built manually.

Building And Using libex29.so

```
# compile the lib file and make the .so
$ cc -c libex29.c -o libex29.o
$ cc -shared -o libex29.so libex29.o

# make the loader program
$ cc -Wall -g -DNDEBUG ex29.c -ldl -o ex29

# try it out with some things that work
$ ex29 ./libex29.so print_a_message "hello there"
-bash: ex29: command not found
$ ./ex29 ./libex29.so print_a_message "hello there"
```

```
A STRING: hello there
$ ./ex29 ./libex29.so uppercase "hello there"
HELLO THERE
$ ./ex29 ./libex29.so lowercase "HELLO tHeRe"
hello there
$ ./ex29 ./libex29.so fail_on_purpose "i fail"
[ERROR] (ex29.c:23: errno: None) Function fail_on_purpose return 1 for data: i fail

# try to give it bad args
$ ./ex29 ./libex29.so fail_on_purpose
[ERROR] (ex29.c:11: errno: None) USAGE: ex29 libex29.so function data

# try calling a function that is not there
$ ./ex29 ./libex29.so adfasfasdf asdfadff
[ERROR] (ex29.c:20: errno: None) Did not find adfasfasdf
  function in the library libex29.so: dlsym(0x1076009b0, adfasfasdf): symbol not found

# try loading a .so that is not there
$ ./ex29 ./libex.so adfasfasdf asdfadfas
[ERROR] (ex29.c:17: errno: No such file or directory) Failed to open
    the library libex.so: dlopen(libex.so, 2): image not found
$
```

One thing that you may run into is that every OS, every version of every OS, and every compiler on every version of every OS, seems to want to change the way you build a shared library every other month that some new programmer thinks it's wrong. If the line I use to make the **`libex29.so`** file is wrong, then let me know and I'll add some comments for other platforms.

Note 10 *Irritating .so Ordering*

Sometimes you'll do what you think is normal and run this command `cc -Wall -g -DNDEBUG -ldl ex29.c -o ex29` thinking everthing will work, but nope. You see, on some platforms the order of where libraries goes makes them work or not, and for no real reason. On Debian or Ubuntu you have to do `cc -Wall -g -DNDEBUG ex29.c -ldl -o ex29` for no reason at all. It's just the way it is, so since this works on OSX I'm doing it here, but in the future, if you link against a dynamic library and it can't find a function, try shuffling things around.

The irritation here is there is an actual platform difference on nothing more than order of command line arguments. On no rational planet should putting an **`-ldl`** at one position be different from another. It's an option, and having to know these things is incredibly annoying.

30.2 How To Break It

Open **`libex29.so`** and edit it with an editor that can handle binary files. Change a couple bytes, then close it. Try to see if you can get the ***`dlopen`*** function to load it even though you've corrupted it.

30.3 Extra Credit

1. Were you paying attention to the bad code I have in the `libex29.c` functions? See how, even though I use a for-loop they still check for `'\0'` endings? Fix this so the functions always take a length for the string to work with inside the function.
2. Take the `c-skeleton` skeleton, and create a new project for this exercise. Put the `libex29.c` file in the `src/` directory. Change the Makefile so that it builds this as `build/libex29.so`.
3. Take the `ex29.c` file and put it in `tests/ex29_tests.c` so that it runs as a unit test. Make this all work, which means you have to change it so that it loads the `build/libex29.so` file and runs tests similar to what I did manually above.
4. Read the *`man dlopen`* documentation and read about all the related functions. Try some of the other options to *`dlopen`* beside *`RTLD_NOW`*.

Chapter 31

Exercise 30: Automated Testing

Automated testing is used frequently in other languages like Python and Ruby, but rarely used in C. Part of the reason comes from the difficulty of automatically loading and testing pieces of C code. In this chapter we'll create a very small little testing ”framework” and get your skeleton directory building an example test case.

The frameworks I'm going to use, and which you'll include in your **c-skeleton** skeleton is called ”minunit” which started with code from a tiny snippet of code by Jera Design. I then evolved it further, to be this:

tests/minunit.h

```
#undef NDEBUG
#ifndef _minunit_h
#define _minunit_h

#include <stdio.h>
#include <dbg.h>
#include <stdlib.h>

#define mu_suite_start() char *message = NULL

#define mu_assert(test, message) if (!(test)) { log_err(message); return message; }
#define mu_run_test(test) debug("\n-----%s", " " #test); \
    message = test(); tests_run++; if (message) return message;

#define RUN_TESTS(name) int main(int argc, char *argv[]) {\
    argc = 1; \
    debug("----- RUNNING: %s", argv[0]);\
        printf("----\nRUNNING: %s\n", argv[0]);\
        char *result = name();\
        if (result != 0) {\
            printf("FAILED: %s\n", result);\
        }\
        else {\
            printf("ALL TESTS PASSED\n");\
        }\
    printf("Tests run: %d\n", tests_run);\
        exit(result != 0);\
}


int tests_run;

#endif
```

There's mostly nothing left of the original, as now I'm using the **dbg.h** macros and I've created a large macro at the end for the boilerplate test runner. Even with this tiny amount of code we'll create a fully functioning unit test system you can use in your C code once it's combined with a shell script to run the tests.

31.1 Wiring Up The Test Framework

To continue this exercise, you should have your **src/libex29.c** working and that you completed the Exercise 29 extra credit where you got the **ex29.c** loader program to properly run. In Exercise 29 I had an extra credit to make it work like a unit test, but I'm going to start over and show you how to do that with **minunit.h**.

The first thing to do is create a simple empty unit test name **tests/libex29_tests.c** with this in it:

tests/libex29_tests.c.h

```
#include "minunit.h"

char *test_dlopen()
{

    return NULL;
}

char *test_functions()
{

    return NULL;
}

char *test_failures()
{
```

```
17
18      return NULL;
19  }
20
21  char *test_dlclose()
22  {
23
24      return NULL;
25  }
26
27  char *all_tests() {
28      mu_suite_start();
29
30      mu_run_test(test_dlopen);
31      mu_run_test(test_functions);
32      mu_run_test(test_failures);
33      mu_run_test(test_dlclose);
34
35      return NULL;
36  }
37
38  RUN_TESTS(all_tests);
```

This code is demonstrating the *RUN_TESTS* macro in `tests/minunit.h` and how to use the other test runner macros. I have the actual test functions stubbed out so that you can see how to structure a unit test. I'll break this file down first:

libex29_tests.c:1 Include the `minunit.h` framework.

libex29_tests.c:3-7 A first test. Tests are structured so they take no arguments and return a *char ** which is *NULL* on *success*. This is important because the other macros will be used to return an error message to the test runner.

libex29_tests.c:9-25 More tests that are the same as the first one.

libex29_tests.c:27 The runner function that will control all the other tests. It has the same form as any other test case, but it gets configured with some additional gear.

libex29_tests.c:28 Sets up some common stuff for a test with *mu_suite_start*.

libex29_tests.c:30 This is how you say what test to run, using the *mu_run_test* macro.

libex29_tests.c:35 After you say what tests to run, you then return *NULL* just like a normal test function.

libex29_tests.c:38 Finally, you just use the big *RUN_TESTS* macro to wire up the *main* method with all the goodies and tell it to run the *all_tests* starter.

That's all there is to running a test, now you should try getting just this to run within the project skeleton. Here's what it looks like when I do it:

First run of libex29_tests

```
$ make clean
rm -rf build src/libex29.o tests/libex29_tests
rm -f tests/tests.log
find . -name "*.gc*" -exec rm {} \;
rm -rf `find . -name "*.dSYM" -print`
$ make
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  -fPIC
    -c -o src/libex29.o src/libex29.c
src/libex29.c: In function â fail_on_purposeâ :
src/libex29.c:42: warning: unused parameter â msgâ
ar rcs build/libYOUR_LIBRARY.a src/libex29.o
ranlib build/libYOUR_LIBRARY.a
cc -shared -o build/libYOUR_LIBRARY.so src/libex29.o
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  build/libYOUR_LIBRARY.a
    tests/libex29_tests.c   -o tests/libex29_tests
sh ./tests/runtests.sh
Running unit tests:
----
RUNNING: ./tests/libex29_tests
ALL TESTS PASSED
Tests run: 4
tests/libex29_tests PASS

$
```

I first did a `make clean` and then I ran the build, which remade the template **`libYOUR_LIBRARY.a`** and **`libYOUR_LIBRARY.so`** files. Remember that you had to do this in the extra credit for Exercise 29, but just in case you didn't figure it out, here's the diff for the **`Makefile`** I'm using now:

Makefile changes for .so builds

```
diff --git a/code/c-skeleton/Makefile b/code/c-skeleton/Makefile
index 135d538..21b92bf 100644
--- a/code/c-skeleton/Makefile
+++ b/code/c-skeleton/Makefile
@@ -9,9 +9,10 @@ TEST_SRC=$(wildcard tests/*_tests.c)
 TESTS=$(patsubst %.c,%,$(TEST_SRC))

 TARGET=build/libYOUR_LIBRARY.a
+SO_TARGET=$(patsubst %.a,%.so,$(TARGET))
```

```
 # The Target Build
-all: $(TARGET) tests
+all: $(TARGET) $(SO_TARGET) tests

 dev: CFLAGS=-g -Wall -Isrc -Wall -Wextra $(OPTFLAGS)
 dev: all
@@ -21,6 +22,9 @@ $(TARGET): build $(OBJECTS)
         ar rcs $@ $(OBJECTS)
         ranlib $@

+$(SO_TARGET): $(TARGET) $(OBJECTS)
+        $(CC) -shared -o $@ $(OBJECTS)
+
 build:
         @mkdir -p build
         @mkdir -p bin
```

With those changes you should be now building everything and you can finally fill in the remaining unit test functions:

Final version of tests/libex29_tests.c

```c
#include "minunit.h"
#include <dlfcn.h>

typedef int (*lib_function)(const char *data);
char *lib_file = "build/libYOUR_LIBRARY.so";
void *lib = NULL;

int check_function(const char *func_to_run, const char *data, int expected)
{
    lib_function func = dlsym(lib, func_to_run);
    check(func != NULL, "Did not find %s function in the library %s: %s", func_to_run, lib_file, dlerror())

    int rc = func(data);
    check(rc == expected, "Function %s return %d for data: %s", func_to_run, rc, data);

    return 1;
error:
    return 0;
}

char *test_dlopen()
{
    lib = dlopen(lib_file, RTLD_NOW);
    mu_assert(lib != NULL, "Failed to open the library to test.");

    return NULL;
}

char *test_functions()
{
    mu_assert(check_function("print_a_message", "Hello", 0), "print_a_message failed.");
    mu_assert(check_function("uppercase", "Hello", 0), "uppercase failed.");
    mu_assert(check_function("lowercase", "Hello", 0), "lowercase failed.");

    return NULL;
}

char *test_failures()
{
    mu_assert(check_function("fail_on_purpose", "Hello", 1), "fail_on_purpose should fail.");

    return NULL;
}

char *test_dlclose()
{
    int rc = dlclose(lib);
    mu_assert(rc == 0, "Failed to close lib.");

    return NULL;
}

char *all_tests() {
    mu_suite_start();

    mu_run_test(test_dlopen);
    mu_run_test(test_functions);
    mu_run_test(test_failures);
    mu_run_test(test_dlclose);

    return NULL;
}

RUN_TESTS(all_tests);
```

Hopefully by now you can figure out what's going on, since there's nothing new in this except for the *check_function*

function. This is a common pattern where I see that I'll be doing a chunk of code repeatedly, and then simply automate it either by creating a function or a macro for it. In this case I'm going to run functions in the `.so` I load so I just made a little function to do it.

31.2 Extra Credit

1. This works but it's probably a bit messy. Clean the `c-skeleton` directory up so that it has all these files, but remove any of the code related to Exercise 29. You should be able to copy this directory over and kickstart new projects without much editing.
2. Study the `runtests.sh` and go read about *`bash`* syntax so you know what it does. Think you could write a C version of this script?

Chapter 32

Exercise 31: Debugging Code

I've already taught you about my awesome debug macros and you've been using them. When I debug code I use the *`debug()`* macro almost exclusively to analyze what's going on and track down the problem. In this exercise I'm going to teach you the basics of using gdb to inspect a simple program that runs and doesn't exit. You'll learn how to use gdb to attach to a running process, stop it, and see what's happening. After that I'll give you some little tips and tricks that you can use with gdb.

32.1 Debug Printing Vs. GDB Vs. Valgrind

I approach debugging primarily with a "scientific method" style, where I come up with possible causes and then rule them out or prove they cause the defect. The problem many programmers have though is their panic and rush to solve a bug makes them feel like this approach will "slow them down". In their rush to solve they fail to notice that they're really just flailing around and gathering no useful information. I find that logging (debug printing) forces me to solve a bug scientifically and it's also just easier to gather information in more situations.

In addition to that, I also have these reasons for using debug printing as my primary debugging tool:

1. You see an entire tracing of a program's execution with debug printing of variables which lets you track how things are going wrong. With gdb you have to place watch and debug statements all over for every thing you want and it's difficult to get a solid trace of the execution.

2. The debug prints can stay in the code, and when you need them you can recompile and they come back. With gdb you have to configure the same information uniquely for every defect you have to hunt down.

3. It's easier to turn on debug logging on a server that's not working right and then inspect the logs while it runs to see what's going on. System administrators know how to handle logging, they don't know how to use gdb.

4. Printing things is just easier. Debuggers are always obtuse and weird with their own quirky interface and inconsistencies. There's nothing complicated about `debug("Yo, dis right? %d", my_stuff);`.

5. Writing debug prints to find a defect forces you to actually analyze the code and use the scientific method. You can think of a debug usage as, "I hypothesize that the code is broken here." Then when you run it you get your hypothesis tested and if it's not broken then you can move to another part where it could be. This may seem like it takes longer, but it's actually faster because you go through a process of "differential diagnosis" and rule out possible

causes until you find the real one.

6. Debug printing works better with unit testing. You can actually just compile the debugs in all the time while you work, and when a unit test explodes just go look at the logs any time. With gdb you'd have to rerun the unit test under gdb and then trace through it to see what's going on.

7. With valgrind you get the equivalent of debug prints for many memory related errors, so you don't need to use something like gdb to find those defects anymore.

Despite all these reasons that I rely on *debug* over *gdb*, I still use *gdb* in a few situations and I think you should have any tool that helps you get your work done. Sometimes, you just have to connect to a broken program and poke around. Or, maybe you've got a server that's crashing and you can only get at core files to see why. In these and a few other cases, gdb is the way to go, and it's always good to have as many tools as possible to help solve problems.

I then break down when I use gdb vs. valgrind vs. debug printing like this:

1. Valgrind is used to catch all memory errors. I use gdb if valgrind is having problems or if using valgrind would slow the program down too much.

2. Print with debug to diagnose and fix defects related to logic or usage. This amounts to about 90% of the defects after you start using Valgrind.

3. Use gdb for the remaining "mystery weird stuff" or emergency situations to gather information. If Valgrind isn't turning anything up and I can't even print out the information I need, then I bust out gdb and start poking around. My use of gdb in this case is entirely to gather information. Once I have an idea of what's going on I go back to writing a unit test to cause the defect, and then do print statements to find out why.

32.2 A Debugging Strategy

This process will actually work with any debugging technique you're going to use, whether that's Valgrind, debug printing, or using a debugger. I'm going to describe it in terms of using *gdb* since it seems people skip this process the most when using debuggers, but use this for every bug until you only need it on the very difficult ones.

1. Start a little text file called **notes.txt** and use it as a kind of "lab notes" for ideas, bugs, problems, etc.

2. Before you use *gdb*, write out the bug you're going to fix and what could be causing it.

3. For each cause, write out the files and functions where you think the cause is coming from, or just write that you don't know.

4. Now start *gdb* and pick the first possible cause with good file:function possibles and set breakpoints there.

5. Use *gdb* to then run the program and confirm if that is the cause. The best way is to see if you can use the *set* command to either fix the program easily or cause the error immediately.

6. If this isn't the cause, then mark in the **notes.txt** that it wasn't and why. Move on to the next possible cause that's easiest to debug, and keep adding information you gather.

In case you haven't noticed, this is basically the scientific method. You write down a set of hypotheses, then you use debugging to prove or disprove them. This gives you insight into more possible causes and then eventually you find it. This process helps you avoid going over the same possible causes repeatedly even though you've found they aren't possible.

You can also do this with debug printing, the only difference is you actually write out your hypotheses in the source code where you think the problem is instead of the **notes.txt**. In a way, debug printing forces you to tackle bugs scientifically since you have to write out hypotheses as print statements.

32.3 Using GDB

The program I'll debug in this exercise is just a while-loop that doesn't terminate correctly. I'm putting a small *usleep* call in it so that there's something interesting to troll through as well.

ex31.c

```c
#include <unistd.h>

int main(int argc, char *argv[])
{
    int i = 0;

    while(i < 100) {
        usleep(3000);
    }

    return 0;
}
```

Compile this like normal and then start it under *gdb* like this: `gdb ./ex31`

Once it's running I want you to play around with these *gdb* commands to see what they do and how to use them.

help COMMAND Get a short help with COMMAND.

break file.c:(line|function) Sets a break point where you want to pause execution. You can give lines or function names to break at after the file.

run ARGS Runs the program, using the ARGS as arguments to the program.

cont Continues execution until a new breakpoint or error.

step Step through the code, but move *into functions*. Use this to trace into a function and see what it's doing.

next Just like *step*, but go *over functions* by just running them.

backtrace (or bt) Does a "backtrace", which dumps the trace of function calls leading to the current point in the program. Very useful for figuring out how you got there, since it also prints the parameters that were passed to each function. It's also similar to what Valgrind reports when you have a memory error.

set var X = Y Set variable X equal to Y.

print X Prints out the value of X, and you can usually use C syntax to access the values of pointers and contents of structs.

ENTER The ENTER key just repeats the last command.

quit Exits *gdb*

Those are the majority of commands I use with *gdb*. Your job is to now play with these and *ex31* so you can get familiar with the output.

Once you're familiar with *gdb* you'll want to play with it some more. Try using it on more complicated programs like *devpkg* to see if you can alter the program's execution or analyze what it's doing.

32.4 Process Attaching

The most useful thing about *gdb* is the ability to attach to a running program and debug it right there. When you have a crashing server or a GUI program, you can't usually start it under *gdb* like you just did. Instead, you have to start it, hope it doesn't crash right away, then attach to it and set a breakpoint. In this part of the exercise I'll show you how to do that.

After you exit *gdb* I want you to restart *ex31* if you stopped it, and then start another Terminal window so you can process attach to it. Process attaching is where you tell *gdb* to connect to a program that's already running so you can inspect it live. It stops the program and then you can walk through it, and when you're done it'll continue just like normal.

Here's a session of me doing it to *ex31*, stepping through it, then fixing the while-loop to make it exit.

ex31.sh-session

```
$ ps ax | grep ex31
10026 s000  S+     0:00.11 ./ex31
10036 s001  R+     0:00.00 grep ex31

$ gdb ./ex31 10026
GNU gdb 6.3.50-20050815 (Apple version gdb-1705) (Fri Jul  1 10:50:06 UTC 2011)
Copyright 2004 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 "x86_64-apple-darwin"...Reading symbols for shared libraries .. done

/Users/zedshaw/projects/books/learn-c-the-hard-way/code/10026: No such file or directory
Attaching to program: `/Users/zedshaw/projects/books/learn-c-the-hard-way/code/ex31', process 10026.
Reading symbols for shared libraries + done
Reading symbols for shared libraries ++........................ done
Reading symbols for shared libraries + done
0x00007fff862c9e42 in __semwait_signal ()

(gdb) break 8
Breakpoint 1 at 0x107babf14: file ex31.c, line 8.

(gdb) break ex31.c:11
Breakpoint 2 at 0x107babf1c: file ex31.c, line 12.

(gdb) cont
Continuing.

Breakpoint 1, main (argc=1, argv=0x7fff677aabd8) at ex31.c:8
8            while(i < 100) {

(gdb) p i
$1 = 0

(gdb) cont
Continuing.

Breakpoint 1, main (argc=1, argv=0x7fff677aabd8) at ex31.c:8
8            while(i < 100) {

(gdb) p i
$2 = 0

(gdb) list
3
4        int main(int argc, char *argv[])
5        {
6            int i = 0;
7
8            while(i < 100) {
9                usleep(3000);
10            }
11
12            return 0;

(gdb) set var i = 200

(gdb) p i
$3 = 200
```

```
62  (gdb) next
63
64  Breakpoint 2, main (argc=1, argv=0x7fff677aabd8) at ex31.c:12
65  12            return 0;
66
67  (gdb) cont
68  Continuing.
69
70  Program exited normally.
71  (gdb) quit
72  $
```

Note 11 *OSX Problems*

On OSX you may see a GUI prompt for the root password, and even after you give it you still get an error from *gdb* saying "Unable to access task for process-id XXX: (os/kern) failure." In that case stop both gdb and the *ex31* program, then start over and it should work as long as you successfully entered the root password.

I'll walk through this session and explain what I did:

gdb:1 I use *ps* to find out what the process id is of the *ex31* I want to attach.

gdb:5 I'm attaching using `gdb ./ex31 PID` replacing PID with the process id I have.

gdb:6-19 *gdb* prints out a bunch of information about it's license and then all the things it's reading. [1]

gdb:21 The program is attached and stopped at this point, so now I set a breakpoint at line 8 in the file with *break*. I'm assuming that I'm already in the file I want to break when I do this.

gdb:24 A better way to do a *break*, is give `file.c:line` format so you can be sure you did the right location. I do that in this *break*.

gdb:27 I use *cont* to continue processing until I hit a breakpoint.

gdb:30-31 The breakpoint is reached so *gdb* prints out variables I need to know about (*argc* and *argv*) and where it's stopped, then the line of code for the breakpoint.

gdb:33-34 I use the abbreviation for *print* "p" to print out the value of the *i* variable. It's 0.

gdb:36 Continue again to see if *i* changes.

gdb:42 Print out *i* again, and nope it's not changing.

gdb:45-55 Use *list* to see what the code is, and then I realize it's not exiting because I'm not incrementing *i*.

gdb:57 Confirm my hypothesis that *i* needs to change by using the *set* command to change it to be `i = 200`. This is one of the best features of *gdb* as it lets you "fix" a program really quick to see if you're right.

gdb:59 Print out *i* just to make sure it changed.

gdb:62 Use *next* to move to the next piece of code, and I see that the breakpoint at `ex31.c:12` is hit, so that means the while-loop exited. My hypothesis is correct, I need to make *i* change.

[1] Just in case you missed it that *gdb* really was the GNU debugger and just in case you didn't know it was doing all this stuff.

gdb:67 Use *`cont`* to continue and the program exits like normal.

gdb:71 I finally use *`quit`* to get out of *`gdb`*.

32.5 GDB Tricks

Here's a list of simple tricks you can do with GDB:

gdb –args Normally *`gdb`* takes arguments you give it and assumes they are for itself. Using *`--args`* passes them to the program.

thread apply all bt Dumps a backtrace for *all* threads. Very useful.

gdb –batch –ex r –ex bt –ex q –args Runs the program so that, if it bombs you get a backtrace.

? Got one? Leave it in the comments.

32.6 Extra Credit

1. Find a graphical debugger and compare using it to raw *`gdb`*. These are useful when the program you're looking at is local, but they are pointless if you have to debug a program on a server.
2. You can enable "core dumps" on your OS, and when a program crashes you'll get a core file. This core file is like a post-mortem of the program so you can load up what happened right at the crash and see what caused it. Change **`ex31.c`** so that it crashes after a few iterations, then try to get a core dump and analyze it.

Chapter 33

Exercise 32: Double Linked Lists

The purpose of this book is to teach you how your computer really works, and included in that is how various data structures and algorithms function. Computers by themselves don't do a lot of useful processing. To make them do useful things you need to structure the data and then organize processing on these structures. Other programming languages either include libraries that implement all of these structures, or they have direct syntax for them. C makes you implement all the data structures you need yourself, which makes it the perfect language to learn how they actually work.

My goal in teaching you these data structures and these algorithms is to help you do three things:

1. Understand what is really going on in Python, Ruby, or JavaScript code like: `data = {"name": "Zed}`
2. Get even better at C code by applying what you know to a set of solved problems using the data structures.
3. Learn a core set of data structures and algorithms so that you are better informed about what ones work best in certain situations.

33.1 What Are Data Structures

The name "data structure" is self-explanatory. It is an organization of data that fits a certain model. Maybe the model is designed to allow processing the data in a new way. Maybe it's just organized to store it on disk efficiently. In this book I'll follow a simple pattern for making data structures that works reliably:

1. Define a struct for the main "outer structure".
2. Define a struct for the contents, usually nodes with links between them.
3. Create functions that operate on these two.

There's other styles of data structures in C, but this pattern works well and is consistent for most data structures you'll make.

33.2 Making The Library

For the rest of this book you'll be creating a library that you can use when you're done with this book. This library will have the following elements:

1. Header (.h) files for each data structure.
2. Implementation (.c) files for the algorithms.
3. Unit tests that test all of them to make sure they keep working.
4. Documentation we'll autogenerate from the header files.

You already have the **c-skeleton** so use it to create a **liblcthw** project:

ex32.sh-session

```
$ cp -r c-skeleton liblcthw
$ cd liblcthw/
$ ls
LICENSE                Makefile       README.md       bin               build               src
$ vim Makefile
$ ls src/
dbg.h               libex29.c       libex29.o
$ mkdir src/lcthw
$ mv src/dbg.h src/lcthw
$ vim tests/minunit.h
$ rm src/libex29.* tests/libex29*
$ make clean
rm -rf build  tests/libex29_tests
rm -f tests/tests.log
find . -name "*.gc*" -exec rm {} \;
rm -rf `find . -name "*.dSYM" -print`
$ ls tests/
minunit.h       runtests.sh
$
```

In this session I'm doing the following:

1. Copy the **c-skeleton** over.
2. Edit the Makefile to change **libYOUR_LIBRARY.a** to **liblcthw.a** as the new *TARGET*.
3. Make the **src/lcthw** directory where we'll put our code.
4. Move the **src/dbg.h** into this new directory.
5. Edit **tests/minunit.h** so that it uses **#include <lcthw/dbg.h>** as the include.
6. Get rid of the source and test files we don't need for **libex29.***.

7. Clean up everything that's left over.

With that you're ready to start building the library, and the first data structure I'll build is the Double Linked List.

33.3 Double Linked Lists

The first data strucutre we'll add to **liblcthw** is a double linked list. This is the simplest data structure you can make, and it has useful properties for certain operations. A linked list works by nodes having pointers to their next or previous element. A "double linked list" contains pointers to both, while a "single linked list" only points at the next element.

Because each node has pointers to the next and previous, and because you keep track of the first and last element of the list, you can do some operations very quickly. Anything that involves inserting or deleting an element will be very fast. They are also easy to implement by most people.

The main disadvantage of a linked list is that traversing it involves processing every single pointer along the way. This means that searching, most sorting, or iterating over the elements will be slow. It also means that you can't really jump to random parts of the list. If you had an array of elements you could just index right into the middle of the list, but a linked list uses a stream of pointers. That means if you want the 10th element, you have to go through elements 1-9.

33.3.1 Definition

As I said in the introduction to this exercise, the process to follow is to first write a header file with the right C struct statements in it.

src/lcthw/list.h

```c
#ifndef lcthw_List_h
#define lcthw_List_h

#include <stdlib.h>

struct ListNode;

typedef struct ListNode {
    struct ListNode *next;
    struct ListNode *prev;
    void *value;
} ListNode;

typedef struct List {
    int count;
    ListNode *first;
17      ListNode *last;
18  } List;
19
20  List *List_create();
21  void List_destroy(List *list);
22  void List_clear(List *list);
23  void List_clear_destroy(List *list);
24
25  #define List_count(A) ((A)->count)
26  #define List_first(A) ((A)->first != NULL ? (A)->first->value : NULL)
27  #define List_last(A) ((A)->last != NULL ? (A)->last->value : NULL)
28
29  void List_push(List *list, void *value);
30  void *List_pop(List *list);
31
32  void List_shift(List *list, void *value);
33  void *List_unshift(List *list);
34
35  void *List_remove(List *list, ListNode *node);
36
37  #define LIST_FOREACH(L, S, M, V) ListNode *_node = NULL;\
38      ListNode *V = NULL;\
39      for(V = _node = L->S; _node != NULL; V = _node = _node->M)
40
41  #endif
```

The first thing I do is create two structs for the *ListNode* and the *List* that will contain those nodes. This creates the data structure I'll use in the functions and macros I define after that. If you read through these functions they seem rather simple. I'll be explaining them when I cover the implementation, but hopefully you can guess what they do.

How the data structure works is each *ListNode* has three components:

1. A value, which is a pointer to anything and stores the thing we want to put in the list.
2. A *ListNode *next* pointer which points at another ListNode that holds the next element in the list.
3. A *ListNode *prev* that holds the previous element. Complex right? Calling the previous thing "previous". I could have used "anterior" and "posterior" but only a jerk would do that.

The *List* struct is then nothing more than a container for these *ListNode* structs that have been linked together in a chain. It keeps track of the *count*, *first* and *last* element of the list.

Finally, take a look at **src/lcthw/list.h:37** where I define the *LIST_FOREACH* macro. This is a common idiom where you make a macro that generates iteration code so people can't mess it up. Getting this kind of processing right can be difficult with data structures, so writing macros helps people out. You'll see how I use this when I talk about the implementation.

33.3.2 Implemention

Once you understand that, you mostly understand how a double linked list works. It is nothing more than nodes with two pointers to the next and previous element of the list. You can then write the **src/lcthw/list.c** code to see how each operation is implemented.

src/lcthw/list.c

```c
#include <lcthw/list.h>
#include <lcthw/dbg.h>

List *List_create()
{
    return calloc(1, sizeof(List));
}

void List_destroy(List *list)
{
    LIST_FOREACH(list, first, next, cur) {
        if(cur->prev) {
            free(cur->prev);
        }
    }

    free(list->last);
    free(list);
}


void List_clear(List *list)
{
    LIST_FOREACH(list, first, next, cur) {
        free(cur->value);
    }
}


void List_clear_destroy(List *list)
{
    List_clear(list);
    List_destroy(list);
}


```

```
void List_push(List *list, void *value)
{
    ListNode *node = calloc(1, sizeof(ListNode));
    check_mem(node);

    node->value = value;

    if(list->last == NULL) {
        list->first = node;
        list->last = node;
    } else {
        list->last->next = node;
        node->prev = list->last;
        list->last = node;
    }

    list->count++;

error:
    return;
}

void *List_pop(List *list)
{
    ListNode *node = list->last;
    return node != NULL ? List_remove(list, node) : NULL;
}

void List_shift(List *list, void *value)
{
    ListNode *node = calloc(1, sizeof(ListNode));
    check_mem(node);

    node->value = value;

    if(list->first == NULL) {
        list->first = node;
        list->last = node;
    } else {
        node->next = list->first;
        list->first->prev = node;
        list->first = node;
    }

    list->count++;

error:
    return;
}

void *List_unshift(List *list)
{
    ListNode *node = list->first;
    return node != NULL ? List_remove(list, node) : NULL;
}

void *List_remove(List *list, ListNode *node)
{
    void *result = NULL;

    check(list->first && list->last, "List is empty.");
    check(node, "node can't be NULL");

    if(node == list->first && node == list->last) {
        list->first = NULL;
        list->last = NULL;
    } else if(node == list->first) {
        list->first = node->next;
    } else if (node == list->last) {
        list->last = node->prev;
    } else {
        ListNode *after = node->next;
        ListNode *before = node->prev;
        after->prev = before;
        before->next = after;
    }

    list->count--;
    result = node->value;
    free(node);

error:
    return result;
}
```

I then implement all of the operations on a double linked list that can't be done with simple macros. Rather than cover every tiny little line of this file, I'm going to give high-level overview of every operation in both the **`list.h`** and **`list.c`** file, then leave you to read the code.

list.h:List_count Returns the number of elements in the list, which is maintained as elements are added and removed.

list.h:List_first Returns the first element of the list, but does not remove it.

list.h:List_last Returns the last element of the list, but does not remove it.

list.h:LIST_FOREACH Iterates over the elements in the list.

list.c:List_create Simply creates the main *List* struct.

list.c:List_destroy Destroys a *List* and any elements it might have.

list.c:List_clear Convenience function for freeing the *values* in each node, not the nodes.

list.c:List_clear_destroy Clears and destroys a list. It's not very efficient since it loops through them twice.

list.c:List_push The first operation that demonstrates the advantage of a linked list. It adds a new element to the end of the list, and because that's just a couple of pointer assignments, does it very fast.

list.c:List_pop The inverse of *List_push*, this takes the last element off and returns it.

list.c:List_shift The other thing you can easily do to a linked list is add elements to the *front* of the list very fast. In this case I call that *List_shift* for lack of a better term.

list.c:List_unshift Just like *List_pop*, this removes the first element and returns it.

list.c:List_remove This is actually doing all of the removal when you do *List_pop* or *List_unshift*. Something that seems to always be difficult in data structures is removing things, and this function is no different. It has to handle quite a few conditions depending on if the element being removed is at the front; the end; both front and end; or middle.

Most of these functions are nothing special, and you should be able to easily digest this and understand it from just the code. You should definitely focus on how the *LIST_FOREACH* macro is used in *List_destroy* so you can understand how much it simplifies this common operation.

33.4 Tests

After you have those compiling it's time to create the test that makes sure they operate correctly.

src/lcthw/list.c

```
#include "minunit.h"
#include <lcthw/list.h>
#include <assert.h>

static List *list = NULL;
char *test1 = "test1 data";
char *test2 = "test2 data";
char *test3 = "test3 data";


char *test_create()
{
    list = List_create();
    mu_assert(list != NULL, "Failed to create list.");

    return NULL;
}


char *test_destroy()
{
    List_clear_destroy(list);

    return NULL;

}


char *test_push_pop()
{
    List_push(list, test1);
    mu_assert(List_last(list) == test1, "Wrong last value.");

    List_push(list, test2);
    mu_assert(List_last(list) == test2, "Wrong last value");

    List_push(list, test3);
    mu_assert(List_last(list) == test3, "Wrong last value.");
    mu_assert(List_count(list) == 3, "Wrong count on push.");

    char *val = List_pop(list);
    mu_assert(val == test3, "Wrong value on pop.");

    val = List_pop(list);
    mu_assert(val == test2, "Wrong value on pop.");

    val = List_pop(list);
    mu_assert(val == test1, "Wrong value on pop.");
    mu_assert(List_count(list) == 0, "Wrong count after pop.");

    return NULL;
}

```

```c
char *test_shift()
{
    List_shift(list, test1);
    mu_assert(List_first(list) == test1, "Wrong last value.");

    List_shift(list, test2);
    mu_assert(List_first(list) == test2, "Wrong last value");

    List_shift(list, test3);
    mu_assert(List_first(list) == test3, "Wrong last value.");
    mu_assert(List_count(list) == 3, "Wrong count on shift.");

    return NULL;
}

char *test_remove()
{
    // we only need to test the middle remove case since push/shift
    // already tests the other cases

    char *val = List_remove(list, list->first->next);
    mu_assert(val == test2, "Wrong removed element.");
    mu_assert(List_count(list) == 2, "Wrong count after remove.");
    mu_assert(List_first(list) == test3, "Wrong first after remove.");
    mu_assert(List_last(list) == test1, "Wrong last after remove.");

    return NULL;
}


char *test_unshift()
{
    char *val = List_unshift(list);
    mu_assert(val == test3, "Wrong value on unshift.");

    val = List_unshift(list);
    mu_assert(val == test1, "Wrong value on unshift.");
    mu_assert(List_count(list) == 0, "Wrong count after unshift.");

    return NULL;
}



```

```c
char *all_tests() {
    mu_suite_start();

    mu_run_test(test_create);
    mu_run_test(test_push_pop);
    mu_run_test(test_shift);
    mu_run_test(test_remove);
    mu_run_test(test_unshift);
    mu_run_test(test_destroy);

    return NULL;
}

RUN_TESTS(all_tests);
```

This test simply goes through every operation and makes sure it works. I use a simplification in the test where I create just one `List *list` for the whole program, then have the tests work on it. This saves the trouble of building a `List` for every test, but it could mean that some tests only pass because of how the previous test ran. In this case I try to make each test keep the list clear or actually use the previous test's results.

33.5 What You Should See

If you did everything right, then when you do a build and run the unit tests it should look like this:

Ex32 Session

```
$ make
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  -fPIC   -c -o src/lcthw/list.o src/lcthw/list.c
ar rcs build/liblcthw.a src/lcthw/list.o
ranlib build/liblcthw.a
cc -shared -o build/liblcthw.so src/lcthw/list.o
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  build/liblcthw.a    tests/list_tests.c   -o tests/list_te
sh ./tests/runtests.sh
Running unit tests:
----
RUNNING: ./tests/list_tests
ALL TESTS PASSED
Tests run: 6
tests/list_tests PASS
$
```

Make sure 6 tests ran, that it builds without warnings or errors, and that it's making the **`build/liblcthw.a`** and **`build/liblcthw.so`** files.

33.6 How To Improve It

Instead of breaking this, I'm going to tell you how to improve the code:

1. You can make `List_clear_destroy` more efficient by using `LIST_FOREACH` and doing both `free` calls inside one loop.
2. You can add asserts for preconditions that it isn't given a `NULL` value for the `List *list` parameters.
3. You can add invariants that check the list's contents are always correct, such as `count` is never `< 0`, and if `count > 0` then `first` isn't NULL.
4. You can add documentation to the header file in the form of comments before each struct, function, and macro that describes what it does.

These amount to going through the defensive programming practices I talked about and ”hardening” this code against flaws or improving usability. Go ahead and do these things, then find as many other ways to improve the code.

33.7 Extra Credit

1. Research double vs. single linked lists and when one is preferred over the other.
2. Research the limitations of a double linked list. For example, while they are efficient for inserting and deleting elements, they are very slow for iterating over them all.
3. What operations are missing that you can imagine needing? Some examples are copying, joining, splitting. Implement these operations and write the unit tests for them.

Chapter 34

Exercise 33: Linked List Algorithms

I'm going to cover two algorithms you can do on a linked list that involve sorting. I'm going to warn you first that if you need to sort the data, then don't use a linked list. These are horrible for sorting things, and there's much better data structures you can use if that's a requirement. I'm covering these two algorithms because they are slightly difficult to pull off with a linked list and get you thinking about manipulating them efficiently.

In the interest of writing this book, I'm going to put the algorithms in two different files **`list_algos.h`** and **`list_algos.c`** then write a test in **`list_algos_test.c`**. For now just follow my structure, as it does keep things clean, but if you ever work on other libraries remember this isn't a common structure.

In this exercise I'm going to also give you an extra challenge and I want you to try not to cheat. I'm going to give you the *unit test* first, and I want you to type it in. Then I want you to try and implement the two algorithms based on their descriptions in Wikipedia before seeing if your code is like my code.

34.0.1 Bubble And Merge Sort

You know what's awesome about the Internet? I can just link you to the Bubble Sort page and Merge Sort page on Wikipedia and tell you to read that. Man, that saves me a boat load of typing. Now I can tell you how to actually implement each of these using the pseudo-code they have there. Here's how you can tackle an algorithm like this:

1. Read the description and look at any visualizations it has.
2. Either draw the algorithm on paper using boxes and lines, or actually take a deck of numbered cards (like Poker Cards) and try to do the algorithm manually. This gives you a concrete demonstration of how the algorithm works.
3. Create the skeleton functions in your **`list_algos.c`** file and make a working **`list_algos.h`** file, then setup your test harness.
4. Write your first failing test and get everything to compile.
5. Go back to the Wikipedia page and copy-paste the pseudo-code (not the C code!) into the guts of the first function you're making.
6. Translate this pseudo-code into good C code like I've taught you, using your unit test to make sure it's working.
7. Fill out some more tests for edge cases like, empty lists, already sorted lists, etc.

8. Repeat for the next algorithm and test.

I just gave you the secret to figuring out most of the algorithms out there, that is until you get to some of the more insane ones. In this case you're just doing the Bubble and Merge Sorts from Wikipedia, but those will be good starters.

34.0.2 The Unit Test

Here is the unit test you should try to get passing:

tests/list_algos_tests.c

```c
#include "minunit.h"
#include <lcthw/list_algos.h>
#include <assert.h>
#include <string.h>

char *values[] = {"XXXX", "1234", "abcd", "xjvef", "NDSS"};
#define NUM_VALUES 5

List *create_words()
{
    int i = 0;
    List *words = List_create();

    for(i = 0; i < NUM_VALUES; i++) {
        List_push(words, values[i]);
    }

    return words;
}

int is_sorted(List *words)
{
    LIST_FOREACH(words, first, next, cur) {
        if(cur->next && strcmp(cur->value, cur->next->value) > 0) {
            debug("%s %s", (char *)cur->value, (char *)cur->next->value);
            return 0;
        }
    }

    return 1;
}

char *test_bubble_sort()
```

```c
{
    List *words = create_words();

    // should work on a list that needs sorting
    int rc = List_bubble_sort(words, (List_compare)strcmp);
    mu_assert(rc == 0, "Bubble sort failed.");
    mu_assert(is_sorted(words), "Words are not sorted after bubble sort.");

    // should work on an already sorted list
    rc = List_bubble_sort(words, (List_compare)strcmp);
    mu_assert(rc == 0, "Bubble sort of already sorted failed.");
    mu_assert(is_sorted(words), "Words should be sort if already bubble sorted.");

    List_destroy(words);

    // should work on an empty list
    words = List_create(words);
    rc = List_bubble_sort(words, (List_compare)strcmp);
    mu_assert(rc == 0, "Bubble sort failed on empty list.");
    mu_assert(is_sorted(words), "Words should be sorted if empty.");

    List_destroy(words);

    return NULL;
}

char *test_merge_sort()
{
    List *words = create_words();

    // should work on a list that needs sorting
    List *res = List_merge_sort(words, (List_compare)strcmp);
    mu_assert(is_sorted(res), "Words are not sorted after merge sort.");

    List *res2 = List_merge_sort(res, (List_compare)strcmp);
    mu_assert(is_sorted(res), "Should still be sorted after merge sort.");
    List_destroy(res2);
    List_destroy(res);

    List_destroy(words);
    return NULL;
}


```

```c
char *all_tests()
{
    mu_suite_start();

    mu_run_test(test_bubble_sort);
    mu_run_test(test_merge_sort);

    return NULL;
}

RUN_TESTS(all_tests);
```

I suggest that you start with the bubble sort and get that working, then move on to the merge sort. What I would do is lay out the function prototypes and skeletons that get all three files compiling, but not passing the test. Then just fill in the implementation until it starts working.

34.0.3 The Implementation

Are you cheating? In future exercises I will do exercises where I just give you a unit test and tell you to implement it, so it'll be good practice for you to not look at this code until you get your own working. Here's the code for the **list_algos.c** and **list_algos.h**:

src/lcthw/list_algos.h

```c
#ifndef lcthw_List_algos_h
#define lcthw_List_algos_h

#include <lcthw/list.h>

typedef int (*List_compare)(void *a, void *b);

int List_bubble_sort(List *list, List_compare cmp);

List *List_merge_sort(List *list, List_compare cmp);

#endif
```

src/lcthw/list_algos.c

```c
#include <lcthw/list_algos.h>
#include <lcthw/dbg.h>

```

```
inline void ListNode_swap(ListNode *a, ListNode *b)
{
    void *temp = a->value;
    a->value = b->value;
    b->value = temp;
}

int List_bubble_sort(List *list, List_compare cmp)
{
    int sorted = 1;

    if(List_count(list) <= 1) {
        return 0;  // already sorted
    }

    do {
        sorted = 1;
        LIST_FOREACH(list, first, next, cur) {
            if(cur->next) {
                if(cmp(cur->value, cur->next->value) > 0) {
                    ListNode_swap(cur, cur->next);
                    sorted = 0;
                }
            }
        }
    } while(!sorted);

    return 0;
}

inline List *List_merge(List *left, List *right, List_compare cmp)
{
    List *result = List_create();
    void *val = NULL;

    while(List_count(left) > 0 || List_count(right) > 0) {
        if(List_count(left) > 0 && List_count(right) > 0) {
            if(cmp(List_first(left), List_first(right)) <= 0) {
                val = List_unshift(left);
            } else {
                val = List_unshift(right);
            }

            List_push(result, val);
```

```c
        } else if(List_count(left) > 0) {
            val = List_unshift(left);
            List_push(result, val);
        } else if(List_count(right) > 0) {
            val = List_unshift(right);
            List_push(result, val);
        }
    }

    return result;
}

List *List_merge_sort(List *list, List_compare cmp)
{
    if(List_count(list) <= 1) {
        return list;
    }

    List *left = List_create();
    List *right = List_create();
    int middle = List_count(list) / 2;

    LIST_FOREACH(list, first, next, cur) {
        if(middle > 0) {
            List_push(left, cur->value);
        } else {
            List_push(right, cur->value);
        }

        middle--;
    }

    List *sort_left = List_merge_sort(left, cmp);
    List *sort_right = List_merge_sort(right, cmp);

    if(sort_left != left) List_destroy(left);
    if(sort_right != right) List_destroy(right);

    return List_merge(sort_left, sort_right, cmp);
}
```

The bubble sort isn't too bad to figure out, although it is really slow. The merge sort is much more complicated, and honestly I could probably spend a bit more time optimizing this code if I wanted to sacrifice clarity.

34.1 What You Should See

If everything works then you should get something like this:

Ex33 Session

```
$ make clean all
rm -rf build src/lcthw/list.o src/lcthw/list_algos.o tests/list_algos_tests tests/list_tests
rm -f tests/tests.log
find . -name "*.gc*" -exec rm {} \;
rm -rf `find . -name "*.dSYM" -print`
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  -fPIC   -c -o src/lcthw/list.o src/lcthw/list.c
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  -fPIC   -c -o src/lcthw/list_algos.o src/lcthw/list_algos
ar rcs build/liblcthw.a src/lcthw/list.o src/lcthw/list_algos.o
ranlib build/liblcthw.a
cc -shared -o build/liblcthw.so src/lcthw/list.o src/lcthw/list_algos.o
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  build/liblcthw.a    tests/list_algos_tests.c   -o tests/l
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  build/liblcthw.a    tests/list_tests.c   -o tests/list_te
sh ./tests/runtests.sh
Running unit tests:
----
RUNNING: ./tests/list_algos_tests
ALL TESTS PASSED
Tests run: 2
tests/list_algos_tests PASS
----
RUNNING: ./tests/list_tests
ALL TESTS PASSED
Tests run: 6
tests/list_tests PASS
$
```

After this exercise I'm not going to show you this output unless it's necessary to show you how it works. From now on you should know that I ran the tests and they all passed and everything compiled.

34.2 How To Improve It

Going back to the description of the algorithms, there's several ways to improve these implementations, and there's a few obvious ones:

1. The merge sort does a crazy amount of copying and creating lists, find ways to reduce this.
2. The bubble sort Wikipedia description mentions a few optimizations, implement them.
3. Can you use the *List_split* and *List_join* (if you implemented them) to improve merge sort?

4. Go through all the defensive programming checks and improve the robustness of this implementation, protecting against bad `NULL` pointers, and create an optional debug level invariant that does what `is_sorted` does after a sort.

34.3 Extra Credit

1. Create a unit test that compares the performance of the two algorithms. You'll want to look at `man 3 time` for a basic timer function, and you'll want to run enough iterations to at least have a few seconds of samples.
2. Play with the amount of data in the lists that need to be sorted and see if that changes your timing.
3. Find a way to simulate filling different sized random lists and measuring how long they take, then graph it and see how it compares to the description of the algorithm.
4. Try to explain why sorting linked lists is a really bad idea.
5. Implement a `List_insert_sorted` that will take a given value, and using the `List_compare`, insert the element at the right position so that the list is always sorted. How does using this method compare to sorting a list after you've built it?

Chapter 35

Exercise 34: Dynamic Array

This is an array that grows on its own and has most of the same features as a linked list. It will usually take up less space, run faster, and has other beneficial properties. This exercise will cover a few of the disadvantages like very slow removal from the front, with a solution (just do it at the end).

A dynamic array is simply an array of `void **` pointers that is pre-allocated in one shot and that point at the data. In the linked list you had a full struct that stored the `void *value` pointer, but in a dynamic array there's just a single array with all of them. This means you don't need any other pointers for next and previous records since you can just index into it directly.

To start, I'll give you the header file you should type up for the implementation:

src/lcthw/darray.h

```c
#ifndef _DArray_h
#define _DArray_h
#include <stdlib.h>
#include <assert.h>
#include <lcthw/dbg.h>

typedef struct DArray {
    int end;
    int max;
    size_t element_size;
    size_t expand_rate;
    void **contents;
} DArray;

DArray *DArray_create(size_t element_size, size_t initial_max);

void DArray_destroy(DArray *array);

void DArray_clear(DArray *array);

int DArray_expand(DArray *array);

int DArray_contract(DArray *array);

int DArray_push(DArray *array, void *el);

void *DArray_pop(DArray *array);

void DArray_clear_destroy(DArray *array);

#define DArray_last(A) ((A)->contents[(A)->end - 1])
#define DArray_first(A) ((A)->contents[0])
#define DArray_end(A) ((A)->end)
#define DArray_count(A) DArray_end(A)
#define DArray_max(A) ((A)->max)

#define DEFAULT_EXPAND_RATE 300


static inline void DArray_set(DArray *array, int i, void *el)
{
    check(i < array->max, "darray attempt to set past max");
    array->contents[i] = el;
error:
    return;
}

static inline void *DArray_get(DArray *array, int i)
{
    check(i < array->max, "darray attempt to get past max");
    return array->contents[i];
error:
    return NULL;
}

static inline void *DArray_remove(DArray *array, int i)
{
    void *el = array->contents[i];

    array->contents[i] = NULL;

    return el;
```

```c
}

static inline void *DArray_new(DArray *array)
{
    check(array->element_size > 0, "Can't use DArray_new on 0 size darrays.");

    return calloc(1, array->element_size);

error:
    return NULL;
}

#define DArray_free(E) free((E))

#endif
```

This header file is showing you a new technique where I put `static inline` functions right in the header. These function definitions will work similar to the `#define` macros you've been making, but they're cleaner and easier to write. If you need to create a block of code for a macro and you don't need code generation, then use a `static inline` function.

Compare this technique to the `LIST_FOREACH` that *generates* a proper for-loop for a list. This would be impossible to do with a `static inline` function because it actually has to generate the inner block of code for the loop. The only way to do that is with a callback function, but that's not as fast and is harder to use.

I'll then change things up and have you create the unit test for `DArray`:

tests/darray_tests.c

```c
#include "minunit.h"
#include <lcthw/darray.h>

static DArray *array = NULL;
static int *val1 = NULL;
static int *val2 = NULL;

char *test_create()
{
    array = DArray_create(sizeof(int), 100);
    mu_assert(array != NULL, "DArray_create failed.");
    mu_assert(array->contents != NULL, "contents are wrong in darray");
    mu_assert(array->end == 0, "end isn't at the right spot");
    mu_assert(array->element_size == sizeof(int), "element size is wrong.");
    mu_assert(array->max == 100, "wrong max length on initial size");

    return NULL;
}

char *test_destroy()
{
    DArray_destroy(array);

    return NULL;
}

char *test_new()
{
    val1 = DArray_new(array);
    mu_assert(val1 != NULL, "failed to make a new element");

    val2 = DArray_new(array);
    mu_assert(val2 != NULL, "failed to make a new element");

    return NULL;
}

char *test_set()
{
    DArray_set(array, 0, val1);
    DArray_set(array, 1, val2);

    return NULL;
}

char *test_get()
{
    mu_assert(DArray_get(array, 0) == val1, "Wrong first value.");
    mu_assert(DArray_get(array, 1) == val2, "Wrong second value.");

    return NULL;
}

char *test_remove()
{
    int *val_check = DArray_remove(array, 0);
    mu_assert(val_check != NULL, "Should not get NULL.");
    mu_assert(*val_check == *val1, "Should get the first value.");
    mu_assert(DArray_get(array, 0) == NULL, "Should be gone.");
    DArray_free(val_check);

    val_check = DArray_remove(array, 1);
    mu_assert(val_check != NULL, "Should not get NULL.");
    mu_assert(*val_check == *val2, "Should get the first value.");
    mu_assert(DArray_get(array, 1) == NULL, "Should be gone.");
    DArray_free(val_check);

    return NULL;
}

char *test_expand_contract()
{
    int old_max = array->max;
    DArray_expand(array);
    mu_assert((unsigned int)array->max == old_max + array->expand_rate, "Wrong size after expand.");

    DArray_contract(array);
    mu_assert((unsigned int)array->max == array->expand_rate + 1, "Should stay at the expand_rate at least.

    DArray_contract(array);
    mu_assert((unsigned int)array->max == array->expand_rate + 1, "Should stay at the expand_rate at least.

    return NULL;
}

char *test_push_pop()
{
    int i = 0;
    for(i = 0; i < 1000; i++) {
        int *val = DArray_new(array);
        *val = i * 333;
        DArray_push(array, val);
    }

    mu_assert(array->max == 1201, "Wrong max size.");

    for(i = 999; i >= 0; i--) {
        int *val = DArray_pop(array);
        mu_assert(val != NULL, "Shouldn't get a NULL.");
        mu_assert(*val == i * 333, "Wrong value.");
        DArray_free(val);
    }

    return NULL;
```

```c
}


char * all_tests() {
    mu_suite_start();

    mu_run_test(test_create);
    mu_run_test(test_new);
    mu_run_test(test_set);
    mu_run_test(test_get);
    mu_run_test(test_remove);
    mu_run_test(test_expand_contract);
    mu_run_test(test_push_pop);
    mu_run_test(test_destroy);

    return NULL;
}

RUN_TESTS(all_tests);
```

This shows you how all of the operations are used, which then makes implementing the *DArray* much easier:

src/lcthw/darray.c

```c
#include <lcthw/darray.h>
#include <assert.h>


DArray *DArray_create(size_t element_size, size_t initial_max)
{
    DArray *array = malloc(sizeof(DArray));
    check_mem(array);
    array->max = initial_max;
    check(array->max > 0, "You must set an initial_max > 0.");

    array->contents = calloc(initial_max, sizeof(void *));
    check_mem(array->contents);

    array->end = 0;
    array->element_size = element_size;
    array->expand_rate = DEFAULT_EXPAND_RATE;

    return array;
```

```

error:
    if(array) free(array);
    return NULL;
}

void DArray_clear(DArray *array)
{
    int i = 0;
    if(array->element_size > 0) {
        for(i = 0; i < array->max; i++) {
            if(array->contents[i] != NULL) {
                free(array->contents[i]);
            }
        }
    }
}

static inline int DArray_resize(DArray *array, size_t newsize)
{
    array->max = newsize;
    check(array->max > 0, "The newsize must be > 0.");

    void *contents = realloc(array->contents, array->max * sizeof(void *));
    // check contents and assume realloc doesn't harm the original on error
    check_mem(contents);

    array->contents = contents;

    return 0;
error:
    return -1;
}

int DArray_expand(DArray *array)
{
    size_t old_max = array->max;
    check(DArray_resize(array, array->max + array->expand_rate) == 0,
            "Failed to expand array to new size: %d",
            array->max + (int)array->expand_rate);

    memset(array->contents + old_max, 0, array->expand_rate + 1);
    return 0;

```

```
error:
    return -1;
}

int DArray_contract(DArray *array)
{
    int new_size = array->end < (int)array->expand_rate ? (int)array->expand_rate : array->end;

    return DArray_resize(array, new_size + 1);
}


void DArray_destroy(DArray *array)
{
    if(array) {
        if(array->contents) free(array->contents);
        free(array);
    }
}

void DArray_clear_destroy(DArray *array)
{
    DArray_clear(array);
    DArray_destroy(array);
}

int DArray_push(DArray *array, void *el)
{
    array->contents[array->end] = el;
    array->end++;

    if(DArray_end(array) >= DArray_max(array)) {
        return DArray_expand(array);
    } else {
        return 0;
    }
}

void *DArray_pop(DArray *array)
{
    check(array->end - 1 >= 0, "Attempt to pop from empty array.");

    void *el = DArray_remove(array, array->end - 1);
    array->end--;

    if(DArray_end(array) > (int)array->expand_rate && DArray_end(array) % array->expand_rate) {
        DArray_contract(array);
    }

    return el;
error:
    return NULL;
}
```

This shows you another way to tackle complex code. Instead of diving right into the `.c` implementation, look at the header file, then read the unit test. This gives you an "abstract to concrete" understanding how the pieces work together and making it easier to remember.

35.1 Advantages And Disadvantages

A *DArray* is better when you need to optimize these operations:

1. Iteration. You can just use a basic for-loop and *DArray_count* with *DArray_get* and you're done. No special macros needed, and it's faster because you aren't walking pointers.
2. Indexing. You can use *DArray_get* and *DArray_set* to access any element at random, but with a *List* you have to go through N elements to get to N+1.
3. Destroying. You just free the struct and the *contents* in two operations. A *List* requires a series of *free* calls and also walking every element.
4. Cloning. You can also clone it in just two operations (plus whatever it's storing) by copying the struct and *contents*. A list again requires walking the whole thing and copying every *ListNode* plus its value.
5. Sorting. As you saw, *List* is horrible if you need to keep the data sorted. A *DArray* opens up a whole class of great sorting algorithms because now you can access elements randomly.
6. Large Data. If you need to keep around a lot of data, then a *DArray* wins since it's base *contents* takes up less memory than the same number of *ListNode* structs.

The *List* however wins on these operations:

1. Insert and remove on the front (what I called shift). A *DArray* needs special treatment to be able to do this efficiently, and usually has to do some copying.
2. Splitting or joining. A *List* can just copy some pointers and it's done, but with a *DArray* you have to do copying of the arrays involved.
3. Small Data. If you only need to store a few elements, then typically the storage will be less in a *List* than a generic *DArray* because the *DArray* needs to expand the backing store to accommodate future inserts, but a *List* only makes what it needs.

With this, I prefer to use a *DArray* for most of the things you see other people use a *List*. I reserve using *List* for any data structure that requires small number of nodes that are inserted and removed from either end. I'll show you two similar data structures called a *Stack* and *Queue* where this is important.

35.2 How To Improve It

As usual, go through each function and operation and add the defensive programming checks, pre-conditions, invariants, and anything else you can find to make the implementation more bulletproof.

35.3 Extra Credit

1. Improve the unit tests to cover more of the operations and test that using a for-loop to iterate works.
2. Research what it would take to implement bubble sort and merge sort for DArray, but don't do it yet. I'll be implementing DArray algorithms next and you'll do this then.
3. Write some performance tests for common operations and compare them to the same operations in *List*. You did some of this, but this time, write a unit test that repeatedly does the operation in question, then in the main runner do the timing.

Chapter 36

Exercise 35: Dynamic Array Algorithms

Covers quick sort, heap sort, and radix sort for dynamic arrays, then has a test driven exercise to implement a pre-sorted dynamic array variant.

36.1 What You Should See

36.2 How To Break It

36.3 Extra Credit

Chapter 37

Exercise 36: Hash Maps

Hash maps are used frequently in modern programming languages as they have many advantages over traditional binary trees. This exercise will introduce a very basic implementation, then discuss some of the issues with hash maps like ordering, flooding buckets, and key hash algorithms.

37.1 What You Should See

37.2 How To Break It

37.3 Extra Credit

Chapter 38

Exercise 37: Hash Map Algorithms

Implements 3 of the more popular hashing algorithms for hash maps, then test driven exercises to make them selectable on construction.

38.1 What You Should See

38.2 How To Break It

38.3 Extra Credit

Chapter 39

Exercise 38: Safer Strings

In this exercise we make a miniature version of the better string (bstring) library you've previously used. This version of the library has the majority of the functions you'll use from bstring and mostly copies its API.

The purpose of this exercise is to finally teach you why C strings are bad, and how to fix them.

39.1 What You Should See

39.2 How To Break It

39.3 Extra Credit

Chapter 40

Exercise 39: String Algorithms

Covers algorithms for searching through strings such as boyer-moore-horspool.

40.1 What You Should See

40.2 How To Break It

40.3 Extra Credit

Chapter 41

Exercise 40: Binary Trees

Now that we've taken a break to focus on strings, I'll teach you the last major data struct you should know.

The binary tree is the simplest tree based data structure and while it has been replaced by Hash Maps in many languages is still useful for many applications. Variants on the binary tree exist for very useful things like database indexes, search algorithm structures, and even graphics processing.

This exercise will create a simple binary tree and then discuss a few variants.

41.1 What You Should See

41.2 How To Break It

41.3 Extra Credit

Chapter 42

Exercise 41: Binary Tree Algorithms

Covers a binary search algorithm, depth first traversal, and leaves breadth first traversal for a test driven exercise.

42.1 What You Should See

42.2 How To Break It

42.3 Extra Credit

Chapter 43

Exercise 42: Stacks and Queues

Stacks are used usually in processing tree-like data. They are heavily used in computer architectures (function calls use stacks), parsing, and anything that involves reordering information and collapsing it.

Queues are useful for anything involving communication between parts in an ordered fashion. This exercise will implement a basic queue using a linked list.

43.1 What You Should See

43.2 How To Break It

43.3 Extra Credit

Chapter 44

Exercise 43: Stack and Queue Algorithms

A test driven exercise to create a simple library for generic queueing of data between parts of a piece of software. This will involve filling a binary tree with data and then queueing it to another function in sorted order for processing.

A test driven exercise to make an RPN calculator with stacks or maybe just process a binary tree.

44.1 What You Should See

44.2 How To Break It

44.3 Extra Credit

Chapter 45

Exercise 44: Ring Buffer

Ring buffers are incredibly useful when processing asynchronous IO. They allow one side to receive data in random intervals of random sizes, but feed cohesive chunks to another side in set sizes or intervals.

This will implement a simple naive variant of a ring buffer, and then have a test driven exercise to make a real complete ring buffer.

45.1 What You Should See

45.2 How To Break It

45.3 Extra Credit

Chapter 46

Exercise 45: Ring Buffer Algorithms

A simple exercise to create a select based async IO library that feeds through a ring buffer to any number of waiting handlers.

46.1 What You Should See

46.2 How To Break It

46.3 Extra Credit

Chapter 47

Exercise 46: Suffix Arrays

A suffix array is a nothing more than a heap sorted list of all the suffices in a string. They are very handy for finding sub-sequences in strings based on other strings and are used in things like searching for genomic similarities. A suffix array is also a very simple algorithm to implement in a naive form.

This exercise will make a very simple suffix array to play with in the next exercise.

47.1 What You Should See

47.2 How To Break It

47.3 Extra Credit

Chapter 48

Exercise 47: Suffix Array Algorithms

A test driven exercise to allow finding the longest sub sequence, prefix, and suffix from a given string.

48.1 What You Should See

48.2 How To Break It

48.3 Extra Credit

Chapter 49

Exercise 48: Ternary Search Tree

My favorite algorithm, a TST creates a binary tree of the characters in a set of strings, then sets the terminating character of each string to a piece of data. This effectively creates a kind of ”suffix array hash map” or ”character binary tree”. It’s very useful in quickly matching a given string to a set of strings, such as in web application routing.

I show a simplified implementation used in Mongrel and Mongrel2 and how it works.

49.1 What You Should See

49.2 How To Break It

49.3 Extra Credit

Chapter 50

Exercise 49: Ternary Search Tree Algorithms

Test driven exercise to implement prefix matching and suffix matching algorithms.

Maybe also approximate matching.

50.1 What You Should See

50.2 How To Break It

50.3 Extra Credit

Chapter 51

Exercise 50: A Tiny Virtual Machine Part 1

51.1 What You Should See

51.2 How To Break It

51.3 Extra Credit

Chapter 52

Exercise 51: A Tiny Virtual Machine Part 2

Create a tiny virtual machine similar to:

https://github.com/GenTiradentes/tinyvm

Use the library just made for the algorithms and implement the entire thing in just a few hundred lines of code (after all the libs of course).

52.1 What You Should See

52.2 How To Break It

52.3 Extra Credit

Chapter 53

Exercise 52: A Tiny Virtual Machine Part 3

Final part of implementing the tiny virtual machine is getting it to run a set of test programs. Make it compatible with GenTirades TinyVM format and then they can use that to test it.

Have test driven exercises for adding a feature like dynamic library loading.

53.1 What You Should See

53.2 How To Break It

53.3 Extra Credit

Chapter 54

Next Steps

After you read this book you should...

Chapter 55

Deconstructing *"K&R C"*

When I was a kid I read this awesome book called "The C Programming Language" by the language's creators, Brian Kernighan and Dennis Ritchie. This book taught me and many people of my generation, and a generation before, how to write C code. You talk to anyone, whether they know C or not, and they'll say, "You can't beat *"K&R C"* . It's the best C book." It is an established piece of programmer lore that is not soon to die.

I myself believed that until I started writing this book. You see, *"K&R C"* is actually riddled with bugs and bad style. Its age is no excuse. These were bugs when they wrote the first printing, and the 42nd printing. I hadn't actually realized just how bad most of the code was in this book and recommended it to many people. After reading through it for just an hour I decided that it needs to be taken down from its pedestal and relegated to history rather than vaunted as state of the art.

I believe it is time to lay this book to rest, but I want to use it as an exercise for you in finding hacks, attacks, defects, and bugs by going through *"K&R C"* to break all the code. That's right, you are going to destroy this sacred cow for me, and you're going to have no problem doing it. When you are done doing this, you will have a finely honed eye for defect. You will also have an informed opinion of the book's actual quality, and will be able to make your own decisions on how to use the knowledge it contains.

In this chapter we will use all the knowledge you've gained from this book, and spend it reviewing the code in *"K&R C"* . What we will do is take many pieces of code from the book, find all the bugs in it, and write a unit test that *exercises* the bugs. We'll then run this test under Valgrind to get statistics and data, and then we'll fix the bugs with a redesign.

This will obviously be a long chapter so I'm going to only do a handful of these and then I'm going have you do the rest. I'll provide a guide that is each page, with the code on it, and hints to the bugs that it has. Your job is to then tear that piece of code apart and try to think like an attacker trying to break the code.

Note 12 *Warning For The Fanboys*

As you read this, if you feel that I am being disrespectful to the authors, then that's not my intent. I respect the authors more than anything you know and owe them a debt of gratitude for writing their book. My criticisms here are both for educational purposes of teaching people *modern* C code, and to destroy the belief in their work as a item of worship that cannot be questioned.

However, if when you read this you have feelings of me insulting *you* then just stop reading. You will gain nothing from this chapter but personal grief because you've attached your identity to *"K&R C"* and my criticisms will only be taken personally.

55.1 An Overall Critique Of Correctness

The primary problem *"K&R C"* has is its view of ”correctness” comes from the first system it was used on: *Unix*. In the world of Unix software programs have a particular set of properties:

1. Programs are started and then exit, making resource allocation easier.
2. Most functions are only called by other parts of the same program in set ways.
3. The inputs to the program are limited to ”expert” restricted users.

In the context of this 1970’s computing style, *"K&R C"* is actually correct. As long as only trusted people run complete cohesive programs that exit and clean up all their resources then their code is fine.

Where *"K&R C"* runs into problems is when the functions or code snippets are taken *out of the book* and used in other programs. Once you take many of these code snippets and try use them in some other program they fall apart. They then have blatant buffer overflows, bugs, and problems that a beginner will trip over.

Another problem is that software these days doesn’t exit right away, but instead it stays running for long periods of time because they’re servers, desktop applications and mobile applications. The old style of ”leaving the cleanup to the OS” doesn’t work in the modern world the way it did back in the day.

The final problem though is that no software lives in a vacuum anymore. Software is now frequently attacked by people over network connections in an attempt to gain special privilege or simple street cred. The idea that ”nobody will ever do that” is dead, and actually that’s probably the first thing somebody will do.

The best way to summarize the problem of *"K&R C"* ”correctness” is with an example from English. Imagine if you have the pair of sentences, ”Jack and Jill went up the hill. He fell down.” Well, from context clues you know that ”He” means Jack. However, if you have that sentence on its own it’s not clear who ”He” is. Now, if you put that sentence at the end of another sentence you can get an unclear pronoun reference: ”Jack and Frank went up the hill. He fell down.” Which ”He” are we talking about in that sentence?

This is how the code in *"K&R C"* works. As long as that code is not used in other programs without serious analysis of the entire software then it works. The second you take many of the functions out and try to use them in other systems they fall apart. And, what’s the point of a book full of code you can’t actually use in your own programs?

55.1.1 A First Demonstration Defect

The following *copy* function is found in the very first chapter and is an example of copying two strings. Here’s a new source file to demonstrate the defects in this function.

exercise-1.9-1.c

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

#define MAXLINE 10 // in the book this is 1000
```

```

void copy(char to[], char from[])
{
    int i;

    i = 0;
    while((to[i] = from[i]) != '\0')
        ++i;
}

int main(int argc, char *argv[])
{
    int i;

    // use heap memory as many modern systems do
    char *line = malloc(MAXLINE);
    char *longest = malloc(MAXLINE);

    assert(line != NULL && longest != NULL && "memory error");

    // initialize it but make a classic "off by one" error
    for(i = 0; i < MAXLINE; i++) {
        line[i] = 'a';
    }

    // cause the defect
    copy(longest, line);

    free(line);
    free(longest);

    return 0;
}
```

In the above example, I'm doing something that is fairly common: switching from using stack allocation to heap allocation with `malloc`. What happens is, typically `malloc` returns memory from the heap, and so the bytes after it are not initialized. Then you see me use a loop to accidentally initialize it wrong. This is a common defect, and one of the reasons we avoided classic style C strings in this book. You could also have this bug in programs that read from files, sockets, or other external resources. It is a *very* common bug, probably the most common in the world.

Before the switch to heap memory, this program probably ran just fine because the stack allocated memory will probably have a `'\0'` character at the end on accident. In fact, it would appear to run fine almost always since it just runs and exits quickly.

What's the effect of running this new program with `copy` used wrong?

exercise-1.9-1.c Valgrind Failures

```
$ make 1.9-1
cc     1.9-1.c   -o 1.9-1
$ ./1.9-1
$
$ valgrind ./1.9-1
==2162== Memcheck, a memory error detector
==2162== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==2162== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
==2162== Command: ./1.9-1
==2162==
==2162== Invalid read of size 1
==2162==    at 0x4005C0: copy (in
    ↪/home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
==2162==    by 0x400651: main (in
    ↪/home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
==2162==  Address 0x51b104a is 0 bytes after a block of size 10 alloc'd
==2162==    at 0x4C2815C: malloc (vg_replace_malloc.c:236)
==2162==    by 0x4005E6: main (in
    ↪/home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
==2162==
==2162== Invalid write of size 1
==2162==    at 0x4005C3: copy (in
    ↪/home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
==2162==    by 0x400651: main (in
    ↪/home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
==2162==  Address 0x51b109a is 0 bytes after a block of size 10 alloc'd
==2162==    at 0x4C2815C: malloc (vg_replace_malloc.c:236)
==2162==    by 0x4005F4: main (in
    ↪/home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
==2162==
==2162== Invalid read of size 1
==2162==    at 0x4005C5: copy (in
    ↪/home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
==2162==    by 0x400651: main (in
    ↪/home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
==2162==  Address 0x51b109a is 0 bytes after a block of size 10 alloc'd
==2162==    at 0x4C2815C: malloc (vg_replace_malloc.c:236)
==2162==    by 0x4005F4: main (in
    ↪/home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
==2162==
==2162==
==2162== HEAP SUMMARY:
==2162==     in use at exit: 0 bytes in 0 blocks
==2162==   total heap usage: 2 allocs, 2 frees, 20 bytes allocated
==2162==
==2162== All heap blocks were freed -- no leaks are possible
==2162==
==2162== For counts of detected and suppressed errors, rerun with: -v
==2162== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 4 from 4)
$
```

As you've already learned, Valgrind will show you all of your sins in full color. In this case, a perfectly harmless seeming program has a ton of "Invalid read of size 1". If you kept running it you'd find other errors pop up at random.

Now, in the context of the entire program in the original *"K&R C"* example, this function will work correctly. However, the second this function is called with *`longest`* and *`line`* uninitialized, initialized wrong, without a trailing `'\0'` character, then you'll hit difficult to debug errors.

This is the failing of the book. While the code works in the book, it does *not* work in many other situations leading to difficult to spot defects, and those are the worst kind of defects for a beginner (or expert). Instead of code that only works in this delicate balance, we will strive to create code that has a higher probability of working in any situation.

55.1.2 Why copy() Fails

Many people have looked at this copy function and thought that it is not defective. They claim that, as long as it's used correctly, it is correct. One person even went so far as to say, "It's not defective, it's just unsafe." Odd, since I'm sure this person wouldn't get into a car if the manufacturer said, "Our car is not defective, it's just unsafe."

However, there is a way to formally prove that this function is defective by enumerating the possible inputs and then seeing if any of them cause the while loop to never terminate.

What we'll do is have two strings, A and B, and figure out what copy() does with them:

1. `A = {'a','b','\0'}; B = {'a', 'b', '\0'};  copy(A,B);`
2. `A = {'a','b'}; B = {'a', 'b', '\0'};  copy(A,B);`
3. `A = {'a','b','\0'}; B = {'a', 'b'};  copy(A,B);`
4. `A = {'a','b'}; B = {'a', 'b'};  copy(A,B);`

This is all the basic permutations of strings that can be passed to the function based on whether they are terminated with a `'\0'` or not. To be complete I'm covering all possible permutations, even if they seem irrelevant. You may think there's no need to include permutations on A, but as you'll see in the analysis, not including A fails to find buffer overflows that are possible.

We can then go through each case and determine if the while loop in copy() terminates:

1. while-loop finds `'\0'` in B, copy fits in A, terminates.

2. while-loop finds `'\0'` in B, overflows A, terminates.
3. while-loop does not find `'\0'` in B, overflows A, does not terminate.
4. while-loop does not find `'\0'` in B, overflows A, does not terminate.

This provides a formal proof that the function is defective because there are possible inputs that causes the while-loop to run forever or overflow the target. If you were to try and use this function safely, you would need to follow all paths to its usage, and confirm that the data is correct along every path. That gives every path to this function a 50% to 75% chance it will fail with just the inputs above. You could find some more permutations of failure but these are the most basic ones.

Let's now compare this to a copy function that knows the lengths of all the inputs to see what it's probability of failure is:

1. `A = {'a','b','\0'}; B = {'a', 'b', '\0'};  safercopy(2, A, 2, B);`
2. `A = {'a','b'}; B = {'a', 'b', '\0'};  safercopy(2, A, 2, B);`
3. `A = {'a','b','\0'}; B = {'a', 'b'};  safercopy(2, A, 2, B);`
4. `A = {'a','b'}; B = {'a', 'b'};  safercopy(2, A, 2, B);`

Also assume that the safercopy() function uses a for-loop that does not test for a `'\0'` only, but instead uses the given lengths to determine the amount to copy. With that we can then do the same analysis:

1. for-loop processes 2 characters of A, terminates.
2. for-loop processes 2 characters of A, terminates.
3. for-loop processes 2 characters of A, terminates.
4. for-loop processes 2 characters of A, terminates.

In every case the for-loop variant with string length given as arguments will terminate no matter what. To really test the for-loop variant we'd need to add some permutations for differing lengths of strings A and B, but in every case the for-loop will always stop because it will only go through a fixed previously known finite number of characters.

That means the for-loop will never loop forever, and as long as it handles all the possible differing lengths of A and B, never overflow either side. The only way to break safercopy() is to lie about the lengths of the strings, but even then it will *still always terminate*. The worst possible scenario for the safercopy() function is that you are given an erroneous length for one of the strings and that string does not have a `'\0'` properly, so the function buffer overflows.

This shows exactly why the copy() function is defective, because it does not terminate cleanly for most possible inputs, and is only reliable for one of the conditions: B terminated and A the right size. It also shows why a for-loop variant with a given fixed length for each input is superior.

Finally, the significance of this is that I've effectively done a formal proof (well, mostly formal) that shows what you should be doing to analyze code. Each function has to stand on its own and not have any defects such as while-loops that do not terminate. In the above discussion I've shown that the original *"K&R C"* is defective, and fatally so since there is no way to fix it given the inputs. There's no way from just a pointer to ask if a string is properly formed since the only way to test that is to scan it, and scanning it runs into this same problem.

55.1.3 But, That's Not A C String

Some folks then defend this function (despite the proof above) by claiming that the strings in the proof aren't C strings. They want to apply an artful dodge that says "the function is not defective because you aren't giving it the right inputs", but I'm saying the function is defective because most of the *possible* inputs cause it to crash the software.

The problem with this mindset is there's no way to confirm that a C string is valid. Imagine you wanted to write a little `assert_good_string` function that checks if a C string is correctly terminated before using it. This function needs to go to the end of the string and see if there's a `'\0'` terminator. How does it do this? This function would also have to scan the target function to confirm that it ended in `'\0'`, which means it has the same problem as copy() because the input may not be terminated.

This may seem silly, but people actually do this with strlen(). They take an input and think that they just have to run strlen() on the input to confirm that it's the right length, but strlen() itself has the same fatal flaw because it has to scan and if the string isn't terminated it will also overflow.

This means any attempt to fix the problem using just C strings also has this problem. The only way to solve it is to include the length of every string and use that to scan it.

If you can't validate a C string in your function, then your only choice is to do full code reviews manually. This introduces human error and no matter what you do the error will happen.

55.1.4 Just Don't Do That

Another argument in favor of this copy() function is when the proponents of *"K&R C"* state that you are "just supposed to not use bad strings". Despite the mountains of empirical evidence that this is impossible in C code, they are basically correct and that's what I'm teaching in this exercise. But, instead of saying "just don't do that by checking all possible inputs", I'm advocating "just don't do that by not using this kind of function". I'll explain further.

In order to confirm that all inputs to this function are valid I have to go through a code review process that involves this:

1. Find all the places the copy() function is called.
2. Trace backwards from that call point to where the inputs are created.
3. Confirm that the data is created correctly.
4. Follow the path from the creation point of the data to where it's used and confirm that no line of code alters the data.
5. Repeat this for all paths and all branches, including all loops and if-statements involving the data.

In my experience this is only possible in small programs like the little ones that *"K&R C"* has. In real software the number of possible branches you'd need to check is much too high for most people to validate, especially in a team environment where individuals have varying degrees of capability. A way to quantify this difficulty is that each branch in the code leading to a function like copy() has a 50-70% chance of causing the defect.

However, if you can use a different function and avoid all of these checks then doesn't that mean the copy() function is defective by comparison? These people are right, the solution is to "just not do that" by just not using the copy() function. You can change the function to one that includes the sizes of the two strings and the problem is solved. If that's the case then the people who think "just don't do that" have just proved that the function is defective, because the simpler way to "not do that" is to use a better function.

If you think copy() is valid as long as you avoid the errors I outline, and if safercopy() avoids the errors, then safercopy() is superior and copy() is defective by comparison.

55.1.5 Stylistic Issues

A more minor critique of the book is that the style is not only old, but just error prone and annoyingly "clever". Take the code you just saw again and look at the *`while-loop`* in *`copy`*. There's no reason to write this loop this way, as the compiler can just as easily work with a *`for-loop`* and without the clever triple-equality trick. The original code also has a while-loop without braces, but an if-statement with braces, which leads to even more confusion:

Braces Are Free, Use Them

```c
/* bad use of while loop with compound if-statement */
while ((len = getline(line, MAXLINE)) > 0)
    if (len > max) {
        max = len;
        copy(longest, line);
    }
if (max > 0) /* there was a line */
    printf("%s", longest);
```

This code is *incredibly* error prone because you can't easily tell where the pair of if-statements and the while-loop are paired. A quick glance makes it seem like this while-loop will loop both if-statements, but it doesn't. In modern C code you would instead just use braces all the time and avoid the confusion completely.

While the book could be forgiven for this because of its age, it has been republished in this form *42 times*, and it was updated for the ANSI standard. At some point in its history you'd think the authors or some publisher ghostwriter could have been bothered to update the book's style. However, this is the problem with sacred cows. Once they become idols of worship people are reluctant to question them or modify them.

In the rest of this chapter though we will be modernizing the code in *"K&R C"* to fit the style you've been learning throughout this book. It will be more verbose, but it will be clearer and less error prone because of this slight increase in verbosity.

55.2 Chapter 1 Examples

Now we begin...