Table des matières

Preface

This is a rough in-progress dump of the book. The grammar will probably be bad, there will be sections missing, but you get to watch me write the book and see how I do things.

You can also ask for help from me at help@learncodethehardway.org any time and I'll usually answer within 1 or 2 days.

This list is a discussion list, not an an announce-only list. It's for discussing the book and asking questions.

Finally, don't forget that I have Learn Python The Hard Way which you should read if you can't code yet. LCTHW will *not* be for beginners, but for people who have at least read LPTHW or know one other programming language.

Introduction: The Cartesian Dream Of C

Whatever I have up till now accepted as most true and assured I have gotten either from the senses or through the senses. But from time to time I have found that the senses deceive, and it is prudent never to trust completely those who have deceived us even once.

—Rene Descartes, Meditations On First Philosophy

If there ever were a quote that described programming with C, it would be this. To many programmers, this makes C scary and evil. It is the Devil, Satan, the trickster Loki come to destroy your productivity with his seductive talk of pointers and direct access to the machine. Then, once this computational Lucifer has you hooked, he destroys your world with the evil "segfault" and laughs as he reveals the trickery in your bargain with him.

But, C is not to blame for this state of affairs. No my friends, your computer and the Operating System controlling it are the real tricksters. They conspire to hide their true inner workings from you so that you can never really know what is going on. The C programming language's only failing is giving you access to what is really there, and telling you the cold hard raw truth. C gives you the red pill. C pulls the curtain back to show you the wizard. *C is truth.*

Why use C then if it's so dangerous? Because C gives you power over the false reality of abstraction and liberates you from stupidity.

What You Will Learn

The purpose of this book is to get you strong enough in C that you'll be able to write your own software in it, or modify someone else's code. At the end of the book we actually take code from a more famous book called `K&R C` and code review it using what you've learned. To get to this stage you'll have to learn a few things:

- The basics of C syntax and idioms.
- Compilation, make files, linkers.
- Finding bugs and preventing them.
- Defensive coding practices.
- Breaking C code.
- Writing basic Unix systems software.

By the final chapter you will have more than enough ammunition to tackle basic systems software, libraries, and other smaller projects.

How To Read This Book

This book is intended for programmers who have learned at least one other programming language. I refer you to Learn Python The Hard Way if you haven't learned a programming language yet. This book is meant for total beginners and works very well as a first book on programming. Once you've done those then you can come back and start this book.

For those who've already learned to code, this book may seem strange at first. It's not like other books where you read paragraph after paragraph of prose and then type in a bit of code here and there. Instead I have you coding right away and then I explain what you just did. This works better because it's easier to explain something you've already experienced.

Because of this structure, there are a few rules you *must* follow in this book:

- Type in all of the code. Do not copy-paste!
- Type the code in exactly, even the comments.
- Get it to run and make sure it prints the same output.
- If there are bugs fix them.
- Do the extra credit but it's alright to skip ones you can't figure out.
- Always try to figure it out first before trying to get help.

If you follow these rules, do everything in the book, and still can't code C then you at least tried. It's not for everyone, but the act of trying will make you a better programmer.

The Core Competencies

I'm going to guess that you come from a language for weaklings. One of those "usable" languages that lets you get away with sloppy thinking and half-assed hackery like Python or Ruby. Or, maybe you use a language like Lisp that pretends the computer is some purely functional fantasy land with padded walls for little babies. Maybe you've learned Prolog and you think the entire world should just be a database that you walk around in looking for clues. Even worse, I'm betting you've been using an IDE, so your brain is riddled with memory holes and you can't even type out an entire function's name without hitting CTRL-SPACE every 3 characters you type.

No matter what your background, you are probably bad at four skills:

Reading And Writing

This is especially true if you use an IDE, but generally I find programmers do too much "skimming" and have problems reading for comprehension. They'll skim code they need to understand in detail and think they understand it when they really don't. Other languages provide tools that also let them avoid actually writing any code, so when faced with a language like C they break down. Simplest thing to do is just understand *everyone* has this problem, and you can fix it by forcing yourself to slow down and be meticulous about your reading and writing. At first it'll feel painful and annoying, but take frequent breaks, and then eventually it'll be easy to do.

Attention To Detail

Everyone is bad at this, and it's the biggest cause of bad software. Other languages let you get away with not paying attention, but C demands your full attention because it is right in the machine and the machine is very picky. With C there is no "kind of similar" or "close enough", so you need to pay attention. Double check your work. Assume everything you write is wrong until you prove it's right.

Spotting Differences

A key problem people from other languages have is their brain has been trained to spot differences in *that* language, not in C. When you compare code you've written to my exercise code your eyes will jump right over characters you think don't matter or that aren't familiar. I'll be giving you strategies that force you to see your mistakes, but keep in mind that if your code is not *exactly*like the code in this book it is wrong.

Planning And Debugging

I love other easier languages because I can just hang out. I type the ideas I have into their interpreter and see results immediately. They're great for just hacking out ideas, but have you noticed that if you keep doing "hack until it works" eventually nothing works? C is harder on you because it requires you to plan out what you'll create first. Sure, you can hack for a bit, but you have to get serious much earlier in C than other languages. I'll be teaching you ways to plan out key parts of your program before you start coding, and this will hopefully make you a better programmer at the same time. Even just a little planning can smooth things out down the road.

Learning C makes you a better programmer because you are forced to deal with these issues earlier and more frequently. You can't be sloppy and half-assed about what you write or nothing will work. The advantage of C is it's a simple language you can figure out on your own, which makes it a great language for learning about the machine and getting stronger in these core programmer skills.

C is harder than some other languages, but that's only because C's not hiding things from you that those other languages try and fail to obfuscate.

License

Exercise 0: The Setup

In this chapter you get your system setup to do C programming. The good news for anyone using Linux or Mac OSX is that you are on a system designed *for* programming in C. The authors of the C language were also instrumental in the creation of the Unix operating system, and both Linux and OSX are based on Unix. In fact, the install will be incredibly easy.

I have some bad news for users of Windows: learning C on Windows is painful. You can write C code for Windows, that's not a problem. The problem is all of the libraries, functions, and tools are just a little "off" from everyone else in the C world. C came from Unix and is much easier on a Unix platform. It's just a fact of life that you'll have to accept I'm afraid.

I wanted to get this bad news out right away so that you don't panic. I'm not saying to avoid Windows entirely. I am however saying that, if you want to have the easiest time learning C, then it's time to bust out some Unix and get dirty. This could also be really good for you, since knowing a little bit of Unix will also teach you some of the idioms of C programming and expand your skills.

This also means that for everyone you'll be using the *command line*. Yep, I said it. You've gotta get in there and type commands at the computer. Don't be afraid though because I'll be telling you what to type and what it should look like, so you'll actually be learning quite a few mind expanding skills at the same time.

Linux

On most Linux systems you just have to install a few packages. For Debian based systems, like Ubuntu you should just have to install a few things using these commands:

```
$ sudo apt-get install build-essential
```

The above is an example of a command line prompt, so to get to where you can run that, find your "Terminal" program and run it first. Then you'll get a shell prompt similar to the `$` above and can type that command into it. *Do not type the ``$``, just the stuff after it.*

Here's how you would install the same setup on an RPM based Linux like Fedora:

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

Once you've run that, you should be able to do the first Exercise in this book and it'll work. If not then let me know.

Mac OSX

On Mac OSX the install is even easier. First, you'll need to either download the latest `XCode` from Apple, or find your install DVD and install it from there. The download will be massive and could take forever, so I recommend installing from the DVD. Also, search online for "installing xcode" for instructions on how to do it.

Once you're done installing XCode, and probably restarting your computer if it didn't make you do that, you can go find your Terminal program and get it put into your Dock. You'll be using Terminal a lot in the book, so it's good to put it in a handy location.

Windows

For Windows users I'll show you how to get a basic Ubuntu Linux system up and running in a virtual machine so that you can still do all of my exercises, but avoid all the painful Windows installation problems.

... have to figure this one out.

Text Editor

The choice of text editor for a programmer is a tough one. For beginners I tell them to just use Gedit since it's simple and works for code. However, it doesn't work in certain internationalized situations, and chances are you already have a favorite text editor if you've been programming for a while.

With this in mind, I want you to try out a few of the standard programmer text editors for your platform and then stick with the one that you like best. If you've been using GEdit and like it then stick with it. If you want to try something different, then try it out real quick and pick one.

The most important thing is *do not get stuck picking the perfect editor*. Text editors all just kind of suck in odd ways. Just pick one, stick with it, and if you find something else you like try it out. Don't spend days on end configuring it and making it perfect.

Some text editors to try out are:

- Gedit on Linux and OSX.
- TextWrangler on OSX.
- Nano which runs in Terminal and works nearly everywhere.
- Emacs and Emacs for OSX. Be prepared to do some learning though.
- Vim and MacVim

There is probably a different editor for every person out there, but these are just a few of the free ones that I know work. Try a few out, and maybe some commercial ones until you find one that you like.

WARNING: Do Not Use An IDE

An IDE, or "Integrated Development Environment" will turn you stupid. They are the worst tools if you want to be a good programmer because they hide what's going on from you, and your job is to know what's going on. They are useful if you're trying to get something done and the platform is designed around a particular IDE, but for learning to code C (and many other languages) they are pointless.

Note

If you've played guitar then you know what tablature is, but for everyone else let me explain. In music there's an established notation called the "staff notation". It's a generic, very old, and universal way to write down what someone should play on an instrument. If you play piano this notation is fairly easy to use, since it was created mostly for piano and composers.

Guitar however is a weird instrument that doesn't really work with notation, so guitarists have an alternative notation called "tablature". What tablature does is, rather than tell you the note to play, it tells you the fret and string you should play at that time. You could learn whole songs without ever knowing about a single thing you're playing. Many people do it this way, but if you want to know *what* you're playing, then tablature is pointless.

It may be harder than tablature, but traditional notation tells you how to play the *music* rather than just how to play the guitar. With traditional notation I can walk over to a piano and play the same song. I can play it on a bass. I can put it into a computer and design whole scores around it. With tablature I can just play it on a guitar.

IDEs are like tablature. Sure, you can code pretty quickly, but you can only code in that one language on that one platform. This is why companies love selling them to you. They know you're lazy, and since it only works on their platform they've got you locked in because you are lazy.

The way you break the cycle is you suck it up and finally learn to code without an IDE. A plain editor, or a programmer's editor like Vim or Emacs, makes you work with the code. It's a little harder, but the end result is you can work with *any* code, on any computer, in any language, and you know what's going on.

Exercise 1: Dust Off That Compiler

Here is a simple first program you can make in C:

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

{

    puts("Hello world.");



    return 0;

}
```

You can put this into a `ex1.c` then type:

```
$ make ex1

cc     ex1.c   -o ex1
```

Your computer may use a slightly different command, but the end result should be a file named `ex1` that you can run.

What You Should See

You can now run the program and see the output.

```
$ ./ex1

Hello world.
```

If you don't then go back and fix it.

How To Break It

In this book I'm going to have a small section for each program on how to break the program. I'll have you do odd things to the programs, run them in weird ways, or change code so that you can see crashes and compiler errors.

For this program, rebuild it with all compiler warnings on:

```
$ 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.

$
```

Now you are getting a warning that says the function "puts" is implicitly declared. The C compiler is smart enough to figure out what you want, but you should be getting rid of all compiler warnings when you can. How you do this is add the following line to the top of`ex1.c` and recompile:

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

Now do the make again like you just did and you'll see the warning go away.

Extra Credit

- Open the `ex1` file in your text editor and change or delete random parts. Try running it and see what happens.
- Print out 5 more lines of text or something more complex than hello world.
- Run `man 3 puts` and read about this function and many others.

Exercise 2: Make Is Your Python Now

In Python you ran programs by just typing `python` and the code you wanted to run. The Python interpreter would just run them, and import any other libraries and things you needed on the fly as it ran. C is a different beast completely where you have to *compile*your source files and manually stitch them together into a binary that can run on its own. Doing this manually is a pain, and in the last exercise you just ran `make` to do it.

In this exercise, you're going to get a crash course in GNU make, and you'll be learning to use it as you learn C. Make will for the rest of this book, be your Python. It will build your code, and run your tests, and set things up and do all the stuff for you that Python normally does.

The difference is, I'm going to show you smarter Makefile wizardry, where you don't have to specify every stupid little thing about your C program to get it to build. I won't do that in this exercise, but after you've been using "baby make" for a while, I'll show you "master make".

Using Make

The first stage of using make is to just use it to build programs it already knows how to build. Make has decades of knowledge on building a wide variety of files from other files. In the last exercise you did this already using commands like this:

```
$ make ex1

# or this one too

$ CFLAGS="-Wall" make ex1
```

In the first command you're telling make, "I want a file named ex1 to be created." Make then does the following:

- Does the file `ex1` exist already?
- No. Ok, is there another file that starts with `ex1`?
- Yes, it's called `ex1.c`. Do I know how to build `.c` files?
- Yes, I run this command `cc ex1.c   -o ex1` to build them.
- I shall make you one `ex1` by using `cc` to build it from `ex1.c`.

The second command in the listing above is a way to pass "modifiers" to the make command. If you're not familiar with how the Unix shell works, you can create these "environment variables" which will get picked up by programs you run. Sometimes you do this with a command like `export CFLAGS="-Wall"` depending on the shell you use. You can however also just put them before the command you want to run, and that environment variable will be set only while that command runs.

In this example I did `CFLAGS="-Wall" make ex1` so that it would add the command line option `-Wall` to the `cc` command that `make`normally runs. That command line option tells the compiler `cc` to report all warnings (which in a sick twist of fate isn't actually all the warnings possible).

You can actually get pretty far with just that way of using `make`, but let's get into making a `Makefile` so you can understand make a little better. To start off, create a file with just this in it:

```
CFLAGS=-Wall -g



clean:

   rm -f ex1
```

Save this file as `Makefile` in your current directory. Make automatically assumes there's a file called `Makefile` and will just run it. Also, *WARNING: Make sure you are only entering TAB characters, not mixtures of TAB and spaces.*

This `Makefile` is showing you some new stuff with make. First we set `CFLAGS` in the file so we never have to set it again, as well as adding the `-g` flag to get debugging. Then we have a section named `clean` which tells make how to clean up our little project.

Make sure it's in the same directory as your `ex1.c` file, and then run these commands:

```
$ make clean

$ make ex1
```

What You Should See

If that worked then you should see this:

```
$ 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'

$
```

Here you can see that I'm running `make clean` which tells make to run our `clean` target. Go look at the Makefile again and you'll see that under this I indent and then I put the shell commands I want `make` to run for me. You could put as many commands as you wanted in there, so it's a great automation tool.

Note

If you fixed `ex1.c` to have `#include <stdio.h>` then your output will not have the warning (which should really be an error) about puts. I have the error here because I didn't fix it.

Notice also that, even though we don't mention `ex1` in the `Makefile`, `make` still knows how to build it *plus* use our special settings.

How To Break It

That should be enough to get you started, but first let's break this make file in a particular way so you can see what happens. Take the line `rm -f ex1` and dedent it (move it all the way left) so you can see what happens. Rerun `make clean` and you should get something like this:

```
$ make clean

Makefile:4: *** missing separator.  Stop.
```

Always remember to indent, and if you get weird errors like this then double check you're consistently using tab characters since some make variants are very picky.

Extra Credit

- Create an `all: ex1` target that will build `ex1` with just the command `make`.
- Read `man make` to find out more information on how to run it.
- Read `man cc` to find out more information on what the flags `-Wall` and `-g` do.
- Research Makefiles online and see if you can improve this one even more.
- Find a `Makefile` in another C project and try to understand what it's doing.

Exercise 3: Formatted Printing

Keep that `Makefile` around since it'll help you spot errors and we'll be adding to it when we need to automate more things.

Many programming languages use the C way of formatting output, so let's try it:

```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;

}
```

Once you have that, do the usual `make ex3` to build it and run it. Make sure you *fix all warnings*.

This exercise has a whole lot going on in a small amount of code so let's break it down:

- First you're including another "header file" called `stdio.h`. This tells the compiler that you're going to use the "standard Input/Output functions". One of those is `printf`.
- Then you're using a variable named `age` and setting it to 10.
- Next you're using a variable `height` and setting it to 72.
- Then you use the `printf` function to print the age and height of the tallest 10 year old on the planet.
- In the `printf` you'll notice you're passing in a string, and it's a format string like in many other languages.
- After this format string, you put the variables that should be "replaced" into the format string by `printf`.

The result of doing this is you are handing `printf` some variables and it is constructing a new string then printing that new string to the terminal.

What You Should See

When you do the whole build you should see something like this:

```
$ make ex3

cc -Wall -g    ex3.c   -o ex3

$ ./ex3
```

```
I am 10 years old.

I am 72 inches tall.

$
```

Pretty soon I'm going to stop telling you to run `make` and what the build looks like, so please make sure you're getting this right and that it's working.

External Research

In the *Extra Credit* section of each exercise I may have you go find information on your own and figure things out. This is an important part of being a self-sufficient programmer. If you constantly run to ask someone a question before trying to figure it out first then you never learn to solve problems independently. This leads to you never building confidence in your skills and always needing someone else around to do your work.

The way you break this habit is to *force* yourself to try to answer your own questions first, and to confirm that your answer is right. You do this by trying to break things, experimenting with your possible answer, and doing your own research.

For this exercise I want you to go online and find out *all* of the `printf` escape codes and format sequences. Escape codes are `\n`or `\t` that let you print a newline or tab (respectively). Format sequences are the `%s` or `%d` that let you print a string or a integer. Find all of the ones available, how you can modify them, and what kind of "precisions" and widths you can do.

From now on, these kinds of tasks will be in the Extra Credit and you should do them.

How To Break It

Try a few of these ways to break this program, which may or may not cause it to crash on your computer:

- Take the `age` variable out of the first `printf` call then recompile. You should get a couple of warnings.
- Run this new program and it will either crash, or print out a really crazy age.
- Put the `printf` back the way it was, and then don't set `age` to an initial value by changing that line to `int age;` then rebuild and run again.

```
# 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.

$
```

Extra Credit

- Find as many other ways to break `ex3.c` as you can.
- Run `man 3 printf` and read about the other '%' format characters you can use. These should look familiar if you used them in other languages (`printf` is where they come from).
- Add `ex3` to your `Makefile`'s `all` list. Use this to `make clean all` and build all your exercises so far.
- Add `ex3` to your `Makefile`'s `clean` list as well. Now use `make clean` will remove it when you need to.

Exercise 4: Introducing Valgrind

It's time to learn about another tool you will live and die by as you learn C called `Valgrind`. I'm introducing `Valgrind` to you now because you're going to use it from now on in the "How To Break It" sections of each exercise. `Valgrind` is a program that runs your programs, and then reports on all of the horrible mistakes you made. It's a wonderful free piece of software that I use constantly while I write C code.

Remember in the last exercise that I told you to break your code by removing one of the arguments to `printf`? It printed out some funky results, but I didn't tell you why it printed those results out. In this exercise we're going to use `Valgrind` to find out why.

Note

These first few exercises are mixing some essential tools the rest of the book needs with learning a little bit of code. The reason is that most of the folks who read this book are not familiar with compiled languages, and definitely not with automation and helpful tools. By getting you to use `make` and `Valgrind` right now I can then use them to teach you C faster and help you find all your bugs early.

After this exercise we won't do many more tools, it'll be mostly code and syntax for a while. But, we'll also have a few tools we can use to really see what's going on and get a good understanding of common mistakes and problems.

Installing Valgrind

You could install `Valgrind` with the package manager for your OS, but I want you to learn to install things from source. This involves the following process:

- Download a source archive file to get the source.
- Unpack the archive to extract the files onto your computer.
- Run `./configure` to setup build configurations.
- Run `make` to make it build, just like you've been doing.
- Run `sudo make install` to install it onto your computer.

Here's a script of me doing this very process, which I want you to try to replicate:

```
# 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
```

Follow this, but obviously update it for new Valgrind versions. If it doesn't build then try digging into why as well.

Using Valgrind

Using `Valgrind` is easy, you just run `valgrind theprogram` and it runs your program, then prints out all the errors your program made while it was running. In this exercise we'll break down one of the error outputs and you can get an instant crash course in "Valgrind hell". Then we'll fix the program.

First, here's a purposefully broken version of the `ex3.c` code for you to build, now called `ex4.c`. For practice, type it in again:

```
#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;

}
```

You'll see it's the same except I've made two classic mistakes:

- I've failed to initialize the `height` variable.
- I've forgot to give the first `printf` the `age` variable.

What You Should See

Now we will build this just like normal, but instead of running it directly, we'll run it with `Valgrind` (see Source: "Building and running ex4.c with Valgrind"):

```
$ 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)

==3082==

==3082== Conditional jump or move depends on uninitialised value(s)

==3082==    at 0x4E7633B: 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 0x4E744C6: vfprintf (vfprintf.c:1613)

==3082==    by 0x4E7E6F9: printf (printf.c:35)

==3082==    by 0x40052B: main (ex4.c:11)

==3082==

I am 0 inches tall.

==3082==

==3082== HEAP SUMMARY:

==3082==     in use at exit: 0 bytes in 0 blocks

==3082==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated

==3082==

==3082== All heap blocks were freed -- no leaks are possible

==3082==

==3082== For counts of detected and suppressed errors, rerun with: -v

==3082== Use --track-origins=yes to see where uninitialised values come
from

==3082== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 4 from 4)

$
```

Note

If you run valgrind and it says something like `by 0x4052112: (below main) (libc-start.c:226)` instead of a line number in`main.c` then add run your valgrind command like this `valgrind --track-origins=yes ./ex4` to make it work. For some reason the Debian or Ubuntu version of `valgrind` does this but not other versions.

This one is huge because `Valgrind` is telling you exactly where every problem in your program is. Starting at the top here's what you're reading, line by line (line numbers are on the left so you can follow):

1

You do the usual `make ex4` and that builds it. Make sure the `cc` command you see is the same and has the `-g` option or your`Valgrind` output won't have line numbers.

2-6

Notice that the compiler is also yelling at you about this source file and it warns you that you have "too few arguments for format". That's where you forgot to include the `age` variable.

7

Then you run your program using `valgrind ./ex4`.

8

Then `Valgrind` goes crazy and yells at you for:

14-18

On line `main (ex4.c:11)` (read as "in the main function in file ex4.c at line 11) you have "Use of uninitialised value of size 8". You find this by looking at the error, then you see what's called a "stack trace" right under that. The line to look at first (ex4.c:11) is the bottom one, and if you don't see what's going wrong then you go up, so you'd try printf.c:35. Typically it's the bottom most line that matters (in this case, on line 18).

20-24

Next error is yet another one on line ex4.c:11 in the main function. `Valgrind` hates this line. This error says that some kind of if-statement or while-loop happened that was based on an uninitialized variable, in this case height.

25-35

The remaining errors are more of the same because the variable keeps getting used.

37-46

Finally the program exits and `Valgrind` tells you a summary of how bad your program is.

That is quite a lot of information to take in, but here's how you deal with it:

- Whenever you run your C code and get it working, rerun it under `Valgrind` to check it.
- For each error that you get, go to the source:line indicated and fix it. You may have to search online for the error message to figure out what it means.
- Once your program is "Valgrind pure" then it should be good, and you have probably learned something about how you write code.

In this exercise I'm not expecting you to fully grasp `Valgrind` right away, but instead get it installed and learn how to use it real quick so we can apply it to all the later exercises.

Extra Credit

- Fix this program using `Valgrind` and the compiler as your guide.
- Read up on `Valgrind` on the internet.
- Download other software and build it by hand. Try something you already use but never built for yourself.
- Look at how the `Valgrind` source files are laid out in the source directory and read its Makefile. Don't worry, none of that makes sense to me either.

Exercise 5: The Structure Of A C Program

You know how to use `printf` and have a couple basic tools at your disposal, so let's break down a simple C program line-by-line so you know how one is structured. In this program you're going to type in a few more things that you're unfamiliar with, and I'm going to lightly break them down. Then in the next few exercises we're going to work with these concepts.

```
#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;

}
```

Type this code in, make it run, and make sure you get *no Valgrind errors*. You probably won't but get in the habit of checking it.

What You Should See

This has pretty boring output, but the point of this exercise is to analyze the code:

```
$ make ex5

cc -Wall -g    ex5.c   -o ex5

$ ./ex5

You are 100 miles away.

$
```

Breaking It Down

There's a few features of the C language in this code that you might have only slightly figured out while you were typing code. Let's break this down line-by-line quickly, and then we can do exercises to understand each part better:

ex5.c:1

An `include` and it is the way to import the contents of one file into this source file. C has a convention of using `.h` extensions for "header" files, which then contain lists of functions you want to use in your program.

ex5.c:3

This is a multi-line `comment` and you could put as many lines of text between the `/*` and closing `*/` characters as you want.

ex5.c:4

A more complex version of the `main function` you've been using blindly so far. How C programs work is the operating system loads your program, and then runs the function named `main`. For the function to be totally complete it needs to return an `int` and take two parameters, an `int` for the argument count, and an array of `char *` strings for the arguments. Did that just fly over your head? Do not worry, we'll cover this soon.

ex5.c:5

To start the body of any function you write a `{` character that indicates the beginning of a "block". In Python you just did a `:` and indented. In other languages you might have a `begin` or `do` word to start.

ex5.c:6

A variable declaration and assignment at the same time. This is how you create a variable, with the syntax `type name = value;`. In C statements (except for logic) end in a `';'` (semicolon) character.

ex5.c:8

Another kind of comment, and it works like Python or Ruby comments where it starts at the `//` and goes until the end of the line.

ex5.c:9

A call to your old friend `printf`. Like in many languages function calls work with the syntax `name(arg1, arg2);` and can have no arguments, or any number.
The `printf` function is actually kind of weird and can take multiple arguments. We'll see that later.

ex5.c:11

A return from the main function, which gives the OS your exit value. You may not be familiar with how Unix software uses return codes, so we'll cover that as well.

ex5.c:12

Finally, we end the main function with a closing brace `}` character and that's the end of the program.

There's a lot of information in this break-down, so study it line-by-line and make sure you at least have a little grasp of what's going on. You won't know everything, but you can probably guess before we continue.

Extra Credit

- For each line, write out the symbols you don't understand and see if you can guess what they mean. Write a little chart on paper with your guess that you can use to check later and see if you get it right.
- Go back to the source code from the previous exercises and do a similar break-down to see if you're getting it. Write down what you don't know and can't explain to yourself.

Exercise 6: Types Of Variables

You should be getting a grasp of how a simple C program is structured, so let's do the next simplest thing which is making some variables of different types:

```
#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;

}
```

In this program we're declaring variables of different types and then printing them with different `printf` format strings.

What You Should See

Your output should look like mine, and you can start to see how the format strings for C are similar to Python and other languages. They've been around for a long time.

```
$ 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.

$
```

What you can see is we have a set of "types", which are ways of telling the C compiler what each variable should represent, and then format strings to match different types. Here's the breakdown of how they match up:

Integers

You declare Integers with the `int` keyword, and print them with `%d`.

Floating Point

Declared with `float` or `double` depending on how big they need to be (double is bigger), and printed with `%f`.

Character

Declared with `char`, written with a `'` (single-quote) character around the char, and then printed with `%c`.

String (Array of Characters)

Declared with `char name[]`, written with `"` characters, and printed with `%s`.

You'll notice that C makes a distinction between single-quote for `char` and double-quote for `char[]` or strings.

Note

When talking about C types, I will typically write in English char[] instead of the whole char SOMENAME[]. This is not valid C code, just a simpler way to talk about types when writing English.

How To Break It

You can easily break this program by passing the wrong thing to the printf statements. For example, if you take the line that prints my name, but put the `initial` variable before the `first_name` in the arguments, you'll get a bug. Make that change and the compiler will yell at you, then when you run it you might get a "Segmentation fault" like I did:

```
$ 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

$
```

Run this change under Valgrind too to see what it tells you about the error "Invalid read of size 1".

Extra Credit

- Come up with other ways to break this C code by changing the `printf`, then fix them.
- Go search for "printf formats" and try using a few of the more exotic ones.
- Research how many different ways you can write a number. Try octal, hexadecimal, and others you can find.
- Try printing an empty string that's just `""`.

Exercise 7: More Variables, Some Math

Let's get familiar with more things you can do with variables by declaring various `ints`, `floats`, `chars`, and `doubles`. We'll then use these in various math expressions so you get introduced to C's basic math.

```
#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;
}
```

Here's what's going on in this little bit of nonsense:

ex7.c:1-4

The usual start of a C program.

ex7.c:5-6

Declare an `int` and `double` for some fake bug data.

ex7.c:8-9

Print out those two, so nothing new here.

ex7.c:11

Declare a huge number using a new type `long` for storing big numbers.

ex7.c:12-13

Print out that number using `%ld` which adds a modifier to the usual `%d`. Adding 'l' (the letter ell) means "print this as a long decimal".

ex7.c:15-17

Just more math and printing.

ex7.c:19-21

Craft up a depiction of your bug rate compared to the bugs in the universe, which is a completely inaccurate calculation. It's so small though that we have to use `%e` to print it in scientific notation.

ex7.c:24

Make a character, with a special syntax `'\0'` which creates a 'nul byte' character. This is effectively the number 0.

ex7.c:25

Multiply bugs by this character, which produces 0 for how much you should care. This demonstrates an ugly hack you find sometimes.

ex7.c:26-27

Print that out, and notice I've got a `%%` (two percent chars) so I can print a '%' (percent) character.

ex7.c:28-30

The end of the `main` function.

This bit of source is entirely just an exercise, and demonstrates how some math works. At the end, it also demonstrates something you see in C, but not in many other languages. To C, a "character" is just an integer. It's a really small integer, but that's all it is. This means you can do math on them, and a lot of software does just that, for good or bad.

This last bit is your first glance at how C gives you direct access to the machine. We'll be exploring that more in later exercises.

What You Should See

As usual, here's what you should see for the output:

```
$ 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%.

$
```

How To Break It

Again, go through this and try breaking the `printf` by passing in the wrong arguments. See what happens when you try to print out that `nul_byte` variable too with `%s` vs. `%c`. When you break it, run it under `Valgrind` to see what it says about your breaking attempts.

Extra Credit

- Make the number you assign to `universe_of_defects` various sizes until you get a warning from the compiler.
- What do these really huge numbers actually print out?
- Change `long` to `unsigned long` and try to find the number that makes that one too big.
- Go search online to find out what `unsigned` does.
- Try to explain to yourself (before I do in the next exercise) why you can multiply a `char` and an `int`.

Exercise 8: Sizes And Arrays

In the last exercise you did math, but with a `'\0'` (nul) character. This may be odd coming from other languages, since they try to treat "strings" and "byte arrays" as different beasts. C however treats strings as just arrays of bytes, and it's only the different printing functions that know there's a difference.

Before I can really explain the significance of this, I have to introduce a few more concepts: `sizeof` and arrays. Here's the code we'll be talking about:

```
#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;

}
```

In this code we create a few arrays with different data types in them. Because arrays of data are so central to how C works, there's a huge number of ways to create them. For now, just use the syntax `type name[] = {initializer};` and we'll explore more. What this syntax means is, "I want an array of type that is initialized to {..}." When C sees this it does the following:

- Look at the type, in this first case it's `int`.
- Look at the `[]` and see that there's no length given.
- Look at the initializer, `{10, 12, 13, 14, 20}` and figure out that you want those 5 ints in your array.
- Create a piece of memory in the computer, that can hold 5 integers one after another.
- Take the name you want, `areas` and assign it this location.

In the case of `areas` it's creating an array of 5 ints that contain those numbers. When it gets to `char name[] = "Zed";` it's doing the same thing, except it's creating an array of 3 chars and assigning that to `name`. The final array we make is `full_name`, but we use the annoying syntax of spelling it out, one character at a time. To C, `name` and `full_name` are identical methods of creating a char array.

The rest of the file, we're using a keyword called `sizeof` to ask C how big things are in *bytes*. C is all about the size and location of pieces of memory and what you do with them. To help you keep that straight, it gives you `sizeof` so you can ask how big something is before you work with it.

This is where stuff gets tricky, so first let's run this and then explain further.

What You Should See

```
$ make ex8

cc -Wall -g    ex8.c   -o ex8

$ ./ex8

The size of an int: 4

The size of areas (int[]): 20

The number of ints in areas: 5

The first area is 10, the 2nd 12.

The size of a char: 1

The size of name (char[]): 4

The number of chars: 4
The size of full_name (char[]): 12

The number of chars: 12

name="Zed" and full_name="Zed A. Shaw"

$
```

Now you see the output of these different `printf` calls and start to get a glimpse of what C is doing. Your output could actually be totally different from mine, since your computer might have different size integers. I'll go through my output:

5

My computer thinks an `int` is 4 bytes in size. Your computer might use a different size if it's a 32-bit vs. 64-bit.

6

The `areas` array has 5 integers in it, so it makes sense that my computer requires 20 bytes to store it.

7

If we divide the size of `areas` by size of an `int` then we get 5 elements. Looking at the code, this matches what we put in the initializer.

8

We then did an array access to get `areas[0]` and `areas[1]` which means C is "zero indexed" like Python and Ruby.

9-11

We repeat this for the `name` array, but notice something odd about the size of the array? It says it's *4* bytes long, but we only typed "Zed" for 3 characters. Where's the 4th one coming from?

12-13

We do the same thing with `full_name` and notice it gets this correct.

13

Finally we just print out the `name` and `full_name` to prove that they actually are "strings" according to printf.

Make sure you can go through and see how these output lines match what was created. We'll be building on this and exploring more about arrays and storage next.

How To Break It

Breaking this program is fairly easy. Try some of these:

- Get rid of the `'\0'` at the end of `full_name` and re-run it. Run it under Valgrind too. Now, move the definition of `full_name` to the top of `main` before `areas`. Try running it under Valgrind a few times and see if you get some new errors. In some cases, you might still get lucky and not catch any errors.
- Change it so that instead of `areas[0]` you try to print `areas[10]` and see what Valgrind thinks of that.
- Try other versions of these, doing it to `name` and `full_name` too.

Extra Credit

- Try assigning to elements in the `areas` array with `areas[0] = 100;` and similar.
- Try assigning to elements of `name` and `full_name`.
- Try setting one element of `areas` to a character from `name`.
- Go search online for the different sizes used for integers on different CPUs.

Exercise 9: Arrays And Strings

In the last exercise you went through an introduction to creating basic arrays and how they map to strings. In this exercise we'll more completely show the similarity between arrays and strings, and get into more about memory layouts.

This exercise shows you that C stores its strings simply as an array of bytes, terminated with the `'\0'` (nul) byte. You probably clued into this in the last exercise since we did it manually. Here's how we do it in another way to make it even more clear by comparing it to an array of numbers:

```
#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;
}
```

In this code, we setup some arrays the tedious way, by assigning a value to each element. In `numbers` we are setting up numbers, but in `name` we're actually building a string manually.

What You Should See

When you run this code you should see first the arrays printed with their contents initialized to zero, then in its initialized form:

```
$ 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 each: Z e d

$
```

You'll notice some interesting things about this program:

- I didn't have to give all 4 elements of the arrays to initialize them. This is a short-cut that C has where, if you set just one element, it'll fill the rest in with 0.
- When each element of `numbers` is printed they all come out as 0.
- When each element of `name` is printed, only the first element 'a' shows up because the `'\0'` character is special and won't display.
- Then the first time we print `name` it only prints "a" because, since the array will be filled with 0 after the first 'a' in the initializer, then the string is correctly terminated by a `'\0'` character.
- We then setup the arrays with a tedious manual assignment to each thing and print them out again. Look at how they changed. Now the numbers are set, but see how the `name` string prints my name correctly?
- There's also two syntaxes for doing a string: `char name[4] = {'a'}` on line 6 vs. `char *another = "name"` on line 44. The first one is less common and the second is what you should use for string literals like this.

Notice that I'm using the same syntax and style of code to interact with both an array of integers and an array of characters, but that`printf` thinks that the `name` is just a string. Again, this is because to the C language there's no difference between a string and an array of characters.

Finally, when you make string literals you should usually use the `char *another = "Literal"` syntax. This works out to be the same thing, but it's more idiomatic and easier to write.

How To Break It

The source of almost all bugs in C come from forgetting to have enough space, or forgetting to put a `'\0'` at the end of a string. In fact it's so common and hard to get right that the majority of good C code just doesn't use C style strings. In later exercises we'll actually learn how to avoid C strings completely.

In this program the key to breaking it is to forget to put the `'\0'` character at the end of the strings. There's a few ways to do this:

- Get rid of the initializers that setup `name`.
- Accidentally set `name[3] = 'A';` so that there's no terminator.
- Set the initializer to `{'a','a','a','a'}` so there's too many 'a' characters and no space for the `'\0'` terminator.

Try to come up with some other ways to break this, and as usual run all of these under Valgrind so you can see exactly what is going on and what the errors are called. Sometimes you'll make these mistakes and even Valgrind can't find them, but try moving where you declare the variables to see if you get the error. This is part of the voodoo of C, that sometimes just where the variable is located changes the bug.

Extra Credit

- Assign the characters into `numbers` and then use `printf` to print them a character at a time. What kind of compiler warnings did you get?
- Do the inverse for `name`, trying to treat it like an array of `int` and print it out one `int` at a time. What does Valgrind think of that?
- How many other ways can you print this out?

- If an array of characters is 4 bytes long, and an integer is 4 bytes long, then can you treat the whole `name` array like it's just an integer? How might you accomplish this crazy hack?
- Take out a piece of paper and draw out each of these arrays as a row of boxes. Then do the operations you just did on paper to see if you get them right.
- Convert `name` to be in the style of `another` and see if the code keeps working.

Exercise 9: Arrays And Strings

In the last exercise you went through an introduction to creating basic arrays and how they map to strings. In this exercise we'll more completely show the similarity between arrays and strings, and get into more about memory layouts.

This exercise shows you that C stores its strings simply as an array of bytes, terminated with the `'\0'` (nul) byte. You probably clued into this in the last exercise since we did it manually. Here's how we do it in another way to make it even more clear by comparing it to an array of numbers:

```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;
}
```

In this code, we setup some arrays the tedious way, by assigning a value to each element. In `numbers` we are setting up numbers, but in `name` we're actually building a string manually.

What You Should See

When you run this code you should see first the arrays printed with their contents initialized to zero, then in its initialized form:

```
$ 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 each: Z e d
$
```

You'll notice some interesting things about this program:

- I didn't have to give all 4 elements of the arrays to initialize them. This is a short-cut that C has where, if you set just one element, it'll fill the rest in with 0.
- When each element of `numbers` is printed they all come out as 0.
- When each element of `name` is printed, only the first element 'a' shows up because the `'\0'` character is special and won't display.
- Then the first time we print `name` it only prints "a" because, since the array will be filled with 0 after the first 'a' in the initializer, then the string is correctly terminated by a `'\0'` character.
- We then setup the arrays with a tedious manual assignment to each thing and print them out again. Look at how they changed. Now the numbers are set, but see how the `name` string prints my name correctly?
- There's also two syntaxes for doing a string: `char name[4] = {'a'}` on line 6 vs. `char *another = "name"` on line 44. The first one is less common and the second is what you should use for string literals like this.

Notice that I'm using the same syntax and style of code to interact with both an array of integers and an array of characters, but that`printf` thinks that the `name` is just a string. Again, this is because to the C language there's no difference between a string and an array of characters.

Finally, when you make string literals you should usually use the `char *another = "Literal"` syntax. This works out to be the same thing, but it's more idiomatic and easier to write.

How To Break It

The source of almost all bugs in C come from forgetting to have enough space, or forgetting to put a `'\0'` at the end of a string. In fact it's so common and hard to get right that the majority of good C code just doesn't use C style strings. In later exercises we'll actually learn how to avoid C strings completely.

In this program the key to breaking it is to forget to put the `'\0'` character at the end of the strings. There's a few ways to do this:

- Get rid of the initializers that setup `name`.
- Accidentally set `name[3] = 'A';` so that there's no terminator.
- Set the initializer to `{'a','a','a','a'}` so there's too many 'a' characters and no space for the `'\0'` terminator.

Try to come up with some other ways to break this, and as usual run all of these under Valgrind so you can see exactly what is going on and what the errors are called. Sometimes you'll make these mistakes and even Valgrind can't find them, but try moving where you declare the variables to see if you get the error. This is part of the voodoo of C, that sometimes just where the variable is located changes the bug.

Extra Credit

- Assign the characters into `numbers` and then use `printf` to print them a character at a time. What kind of compiler warnings did you get?
- Do the inverse for `name`, trying to treat it like an array of `int` and print it out one `int` at a time. What does Valgrind think of that?
- How many other ways can you print this out?
- If an array of characters is 4 bytes long, and an integer is 4 bytes long, then can you treat the whole `name` array like it's just an integer? How might you accomplish this crazy hack?
- Take out a piece of paper and draw out each of these arrays as a row of boxes. Then do the operations you just did on paper to see if you get them right.
- Convert `name` to be in the style of `another` and see if the code keeps working.

Exercise 10: Arrays Of Strings, Looping

You can make an array of various types, and have the idea down that a "string" and an "array of bytes" are the same thing. The next thing is to take this one step further and do an array that has strings in it. We'll also introduce your first looping construct, the`for-loop` to help print out this new data structure.

The fun part of this is that there's been an array of strings hiding in your programs for a while now, the `char *argv[]` in the `main`function arguments. Here's code that will print out any command line arguments you pass it:

```
#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;

}
```

The format of a `for-loop` is this:

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

    CODE;

}
```

Here's how the `for-loop` works:

- The `INITIALIZER` is code that is run to setup the loop, in this case `i = 0`.
- Next the `TEST` boolean expression is checked, and if it's false (0) then `CODE` is skipped, doing nothing.
- The `CODE` runs, does whatever it does.
- After the `CODE` runs, the `INCREMENTER` part is run, usually incrementing something, like in `i++`.
- And it continues again with Step 2 until the `TEST` is false (0).

This `for-loop` is going through the command line arguments using `argc` and `argv` like this:

- The OS passes each command line argument as a string in the `argv` array. The program's name (./ex10) is at 0, with the rest coming after it.
- The OS also sets `argc` to the number of arguments in the `argv` array so you can process them without going past the end. Remember that if you give one argument, the program's name is the first, so argc is 2.
- The `for-loop` sets up with `i = 1` in the initializer.
- It then tests that `i` is less than `argc` with the test `i < argc`. Since initially $1 < 2$ it will pass.
- It then runs the code which just prints out the `i` and uses `i` to index into `argv`.
- The incrementer is then run using the `i++` syntax, which is a handy way of writing `i = i + 1`.
- This then repeats until `i < argc` is finally false (0) when the loop exits and the program continues on.

What You Should See

To play with this program you have to run it two ways. The first way is to pass in some command line arguments so that `argc` and`argv` get set. The second is to run it with no arguments so you can see that the first `for-loop` doesn't run since `i < argc` will be false.

```
$ 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

$
```

Understanding Arrays Of Strings

From this you should be able to figure out that in C you make an "array of strings" by combining the `char *str = "blah"` syntax with the `char str[] = {'b','l','a','h'}` syntax to construct a 2-dimensional array. The syntax `char *states[] = {...}`on line 14 is this 2-dimension combination, with each string being one element, and each character in the string being another.

Confusing? The concept of multiple dimensions is something most people never think about so what you should do is build this array of strings on paper:

- Make a grid with the index of each *string* on the left.
- Then put the index of each *character* on the top.
- Then, fill in the squares in the middle with what single character goes in that cell.
- Once you have the grid, trace through the code manually using this grid of paper.

Another way to figure this is out is to build the same structure in a programming language you are more familiar with like Python or Ruby.

How To Break It

- Take your favorite other language, and use it to run this program, but with as many command line arguments as possible. See if you can bust it by giving it way too many arguments.
- Initialize `i` to 0 and see what that does. Do you have to adjust `argc` as well or does it just work? Why does 0-based indexing work here?
- Set `num_states` wrong so that it's a higher value and see what it does.

Extra Credit

- Figure out what kind of code you can put into the parts of a `for-loop`.
- Look up how to use the `','` (comma) character to separate multiple statements in the parts of the `for-loop`, but between the`';'` (semicolon) characters.
- Read what a `NULL` is and try to use it in one of the elements of the `states` array to see what it'll print.
- See if you can assign an element from the `states` array to the `argv` array before printing both. Try the inverse.

Exercise 11: While-Loop And Boolean Expressions

You've had your first taste of how C does loops, but the boolean expression `i < argc` might have not been clear to you. Let me explain something about it before we see how a `while-loop` works.

In C, there's not really a "boolean" type, and instead any integer that's 0 is "false" and otherwise it's "true". In the last exercise the expression `i < argc` actually resulted in 1 or 0, not an explicit `True` or `False` like in Python. This is another example of C being closer to how a computer works, because to a computer truth values are just integers.

Now you'll take and implement the same program from the last exercise but use a `while-loop` instead. This will let you compare the two so you can see how one is related to another.

```
#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;

}
```

You can see from this that a `while-loop` is simpler:

```
while(TEST) {

    CODE;

}
```

It simply runs the `CODE` as long as `TEST` is true (1). This means that to replicate how the `for-loop` works we need to do our own initializing and incrementing of `i`.

What You Should See

The output is basically the same, so I just did it a little different so you can see another way it runs.

```
$ 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

$
```

How To Break It

In your own code you should favor `for-loop` constructs over `while-loop` because a `for-loop` is harder to break. Here's a few common ways:

- Forget to initialize the first `int i;` so have it loop wrong.
- Forget to initialize the second loop's `i` so that it retains the value from the end of the first loop. Now your second loop might or might not run.
- Forget to do a `i++` increment at the end of the loop and you get a "forever loop", one of the dreaded problems of the first decade or two of programming.

Extra Credit

- Make these loops count backward by using `i--` to start at `argc` and count down to 0. You may have to do some math to make the array indexes work right.
- Use a while loop to *copy* the values from `argv` into `states`.
- Make this copy loop never fail such that if there's too many `argv` elements it won't put them all into `states`.
- Research if you've really copied these strings. The answer may surprise and confuse you though.

Exercise 12: If, Else-If, Else

Something common in every language is the `if-statement`, and C has one. Here's code that uses an `if-statement` to make sure you enter only 1 or 2 arguments:

```
#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;

}
```

The format for the `if-statement` is this:

```
if(TEST) {
    CODE;

} else if(TEST) {

    CODE;

} else {

    CODE;

}
```

This is like most other languages except for some specific C differences:

- As mentioned before, the `TEST` parts are false if they evaluate to 0, and true otherwise.
- You have to put parenthesis around the `TEST` elements, while some other languages let you skip that.
- You don't need the `{}` braces to enclose the code, but it is *very* bad form to not use them. The braces make it clear where one branch of code begins and ends. If you don't include it then obnoxious errors come up.

Other than that, they work like others do. You don't need to have either `else if` or `else` parts.

What You Should See

This one is pretty simple to run and try out:

```
$ 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.

$
```

How To Break It

This one isn't easy to break because it's so simple, but try messing up the tests in the `if-statement`.

- Remove the `else` at the end and it won't catch the edge case.
- Change the `&&` to a `||` so you get an "or" instead of "and" test and see how that works.

Extra Credit

- You were briefly introduced to `&&`, which does an "and" comparison, so go research online the different "boolean operators".
- Write a few more test cases for this program to see what you can come up with.
- Go back to Exercises 10 and 11, and use `if-statements` to make the loops exit early. You'll need the `break` statement to do that. Go read about it.
- Is the first test really saying the right thing? To you the "first argument" isn't the same first argument a user entered. Fix it.

Exercise 13: Switch Statement

In other languages like Ruby you have a `switch-statement` that can take any expression. Some languages like Python just don't have a `switch-statement` since an `if-statement` with boolean expressions is about the same thing. For these languages,`switch-statements` are more alternatives to `if-statements` and work the same internally.

The `switch-statement` is actually entirely different and is really a "jump table". Instead of random boolean expressions, you can only put expressions that result in integers, and these integers are used to calculate jumps from the top of the `switch` to the part that matches that value. Here's some code that we'll break down to understand this concept of "jump tables":

```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;
}
```

In this program we take a single command line argument and print out all of the vowels in an incredibly tedious way to demonstrate a`switch-statement`. Here's how the `switch-statement` works:

- The compiler marks the place in the program where the `switch-statement` starts, let's call this location Y.
- It then evaluates the expression in `switch(letter)` to come up with a number. In this case the number will be the raw ASCII code of the letter in `argv[1]`.
- The compiler has also translated each of the `case` blocks like `case 'A':` into a location in the program that is that far away. So the code under `case 'A'` is at Y+'A' in the program.
- It then does the math to figure out where Y+letter is located in the `switch-statement`, and if it's too far then it adjusts it to Y+default.
- Once it knows the location, the program "jumps" to that spot in the code, and then continues running. This is why you have`break` on some of the `case` blocks, but not others.
- If `'a'` is entered, then it jumps to `case 'a'`, there's no break so it "falls through" to the one right under it `case 'A'` which has code and a `break`.
- Finally it runs this code, hits the break then exits out of the `switch-statement` entirely.

This is a deep dive into how the `switch-statement` works, but in practice you just have to remember a few simple rules:

- Always include a `default:` branch so that you catch any missing inputs.
- Don't allow "fall through" unless you really want it, and it's a good idea to add a comment `//fallthrough` so people know it's on purpose.
- Always write the `case` and the `break` before you write the code that goes in it.
- Try to just use `if-statements` instead if you can.

What You Should See

Here's an example of me playing with this, and also demonstrating various ways to pass the argument in:

```
$ 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

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

$
```

Remember that there's that `if-statement` at the top that exits with a `return 1;` when you don't give enough arguments. Doing a return that's not 0 is how you indicate to the OS that the program had an error. Any value that's greater than 0 can be tested for in scripts and other programs to figure out what happened.

How To Break It

It is *incredibly* easy to break a `switch-statement`. Here's just a few of the ways you can mess one of these up:

- Forget a `break` and it'll run two or more blocks of code you don't want it to run.
- Forget a `default` and have it silently ignore values you forgot.
- Accidentally put in variable into the `switch` that evaluates to something unexpected, like an `int` that becomes weird values.
- Use uninitialized values in the `switch`.

You can also break this program in a few other ways. See if you can bust it yourself.

Extra Credit

- Write another program that uses math on the letter to convert it to lowercase, and then remove all the extraneous uppercase letters in the switch.
- Use the `','` (comma) to initialize `letter` in the `for-loop`.
- Make it handle all of the arguments you pass it with yet another `for-loop`.
- Convert this `switch-statement` to an `if-statement`. Which do you like better?
- In the case for 'Y' I have the break outside the `if-statement`. What's the impact of this and what happens if you move it inside the `if-statement`. Prove to yourself that you're right.

Exercise 14: Writing And Using Functions

Until now you've just used functions that are part of the `stdio.h` header file. In this exercise you will write some functions and use some other functions.

```
#include <stdio.h>

#include <ctype.h>



// forward declarations

int can_print_it(char ch);

void print_letters(char arg[]);
```

```c
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);

        }

    }


    printf("\n");

}


int can_print_it(char ch)
```

```
{
    return isalpha(ch) || isblank(ch);
}



int main(int argc, char *argv[])
{
    print_arguments(argc, argv);
    return 0;
}
```

In this example you're creating functions to print out the characters and ASCII codes for any that are "alpha" or "blanks". Here's the breakdown:

ex14.c:2

Include a new header file so we can gain access to `isalpha` and `isblank`.

ex14.c:5-6

Tell C that you will be using some functions later in your program, without having to actually define them. This is a "forward declaration" and it solves the chicken-and-egg problem of needing to use a function before you've defined it.

ex14.c:8-15

Define the `print_arguments` which knows how to print the same array of strings that `main` typically gets.

ex14.c:17-30

Define the next function `print_letters` that is called *by* `print_arguments` and knows how to print each of the characters and their codes.

ex14.c:32-35

Define `can_print_it` which simply returns the truth value (0 or 1) of `isalpha(ch) || isblank(ch)` back to its caller`print_letters`.

ex14.c:38-42

Finally `main` simply calls `print_arguments` to make the whole chain of function calls go.

I shouldn't have to describe what's in each function because it's all things you've ran into before. What you should be able to see though is that I've simply defined functions the same way you've been defining `main`. The only difference is you have to help C out by telling it ahead of time if you're going to use functions it hasn't encountered yet in the file. That's what the "forward declarations" at the top do.

What You Should See

To play with this program you just feed it different command line arguments, which get passed through your functions. Here's me playing with it to demonstrate:

```
$ 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

$
```

The `isalpha` and `isblank` do all the work of figuring out if the given character is a letter or a blank. When I do the last run it prints everything but the '3' character, since that is a digit.

How To Break It

There's two different kinds of "breaking" in this program:

- Confuse the compiler by removing the forward declarations so it complains about `can_print_it` and `print_letters`.

- When you call `print_arguments` inside `main` try adding 1 to `argc` so that it goes past the end of the `argv` array.

Extra Credit

- Rework these functions so that you have fewer functions. For example, do you really need `can_print_it`?
- Have `print_arguments` figure how long each argument string is using the `strlen` function, and then pass that length to`print_letters`. Then, rewrite `print_letters` so it only processes this fixed length and doesn't rely on the `'\0'` terminator. You will need the `#include <string.h>` for this.
- Use `man` to lookup information on `isalpha` and `isblank`. Use the other similar functions to print out only digits or other characters.
- Go read about how different people like to format their functions. Never use the "K&R syntax" as it's antiquated and confusing, but understand what it's doing in case you run into someone who likes it.

Exercise 15: Pointers Dreaded Pointers

Pointers are famous mystical creatures in C that I will attempt to demystify by teaching you the vocabulary used to deal with them. They actually aren't that complex, it's just they are frequently abused in weird ways that make them hard to use. If you avoid the stupid ways to use pointers then they're fairly easy.

To demonstrate pointers in a way we can talk about them, I've written a frivolous program that prints a group of people's ages in three different ways:

```
#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");

    // setup the pointers to the start of the arrays
    int *cur_age = ages;
    char **cur_name = names;

    // second way using pointers
    for(i = 0; i < count; i++) {
        printf("%s is %d years old.\n",
                *(cur_name+i), *(cur_age+i));
    }

    printf("---\n");

    // third way, pointers are just arrays
    for(i = 0; i < count; i++) {
        printf("%s is %d years old again.\n",
                cur_name[i], cur_age[i]);
    }

    printf("---\n");

    // fourth way with pointers in a stupid complex way
    for(cur_name = names, cur_age = ages;
            (cur_age - ages) < count;
            cur_name++, cur_age++)
    {
        printf("%s lived %d years so far.\n",
                *cur_name, *cur_age);
    }

    return 0;
}
```

Before explaining how pointers work, let's break this program down line-by-line so you get an idea of what's going on. As you go through this detailed description, try to answer the questions for yourself on a piece of paper, then see if what you guessed was going on matches my description of pointers later.

ex15.c:6-10

Create two arrays, `ages` storing some `int` data, and `names` storing an array of strings.

ex15.c:12-13

Some variables for our `for-loops` later.

ex15.c:16-19

You know this is just looping through the two arrays and printing how old each person is. This is using `i` to index into the array.

ex15.c:24

Create a pointer that points at `ages`. Notice the use of `int *` to create a "pointer to integer" type of pointer. That's similar to `char *`, which is a "pointer to char", and a string is an array of chars. Seeing the similarity yet?

ex15.c:25

Create a pointer that points at names. A char * is already a "pointer to char", so that's just a string. You however need 2 levels, since names is 2-dimensional, that means you need char ** for a "pointer to (a pointer to char)" type. Study that too, explain it to yourself.

ex15.c:28-31

Loop through ages and names but instead use the pointers *plus an offset of i*. Writing *(cur_name+i) is the same as writingname[i], and you read it as "the value of (pointer cur_name plus i)".

ex15.c:35-39

This shows how the syntax to access an element of an array is the same for a pointer and an array.

ex15.c:44-50

Another admittedly insane loop that does the same thing as the other two, but instead it uses various pointer arithmetic methods:

ex15.c:44

Initialize our for-loop by setting cur_name and cur_age to the beginning of the names and ages arrays.

ex15.c:45

The test portion of the for-loop then compares the *distance* of the pointer cur_age from the start of ages. Why does that work?

ex15.c:46

The increment part of the for-loop then increments both cur_name and cur_age so that they point at the *next* element of thename and age arrays.

ex15.c:48-49

The pointers cur_name and cur_age are now pointing at one element of the arrays they work on, and we can print them out using just *cur_name and *cur_age, which means "the value of wherever cur_name is pointing".

This seemingly simple program has a large amount of information, and the goal is to get you to attempt figuring pointers out for yourself before I explain them. *Don't continue until you've written down what you think a pointer does.*

What You Should See

After you run this program try to trace back each line printed out to the line in the code that produced it. If you have to, alter theprintf calls to make sure you got the right line number.

```
$ 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.

$
```

Explaining Pointers

When you type something like `ages[i]` you are "indexing" into the array `ages`, and you're using the number that's held in `i` to do it. If `i` is set to 0 then it's the same as typing `ages[0]`. We've been calling this number `i` an "index" since it's a location inside `ages`that we want. It could also be called an "address", that's a way of saying "I want the integer in `ages` that is at address `i`".

If `i` is an index, then what's `ages`? To C `ages` is a location in the computer's memory where all of these integers start. It is *also* an address, and the C compiler will replace anywhere you type `ages` with the address of the very first integer in ages. Another way to think of `ages` is it's the "address of the first integer in ages". But, the trick is `ages` is an address inside the *entire computer*. It's not like `i` which was just an address inside `ages`. The `ages` array name is actually an address in the computer.

That leads to a certain realization: C thinks your whole computer is one massive array of bytes. Obviously this isn't very useful, but then C layers on top of this massive array of bytes the concept of *types* and *sizes* of those types. You already saw how this worked in previous exercises, but now you can start to get an idea that C is somehow doing the following with your arrays:

- Creating a block of memory inside your computer.
- "Pointing" the name `ages` at the beginning of that block.
- "Indexing" into the block by taking the base address of `ages` and getting the element that's `i` away from there.
- Converting that address at `ages+i` into a valid `int` of the right size, such that the index works to return what you want: the int at index `i`.

If you can take a base address, like `ages`, and then "add" to it with another address like `i` to produce a new address, then can you just make something that points right at this location all the time? Yes, and that thing is called a "pointer". This is what the pointers`cur_age` and `cur_name` are doing. They are variables pointing at the location where `ages` and `names` live in your computer's memory. The example program is then moving them around or doing math on them to get values out of the memory. In one instance, they just add `i` to `cur_age`, which is the same as what it does with `array[i]`. In the last `for-loop` though these two pointers are being moved on their own, without `i` to help out. In that loop, the pointers are treated like a combination of array and integer offset rolled into one.

A pointer is simply an address pointing somewhere inside the computer's memory, with a type specifier so you get the right size of data with it. It is kind of like a combined `ages` and `i` rolled into one data type. C knows where pointers are pointing, knows the data type they point at, the size of those types, and how to get the data for you. Just like `i` you can increment them, decrement them, subtract or add to them. But, just like `ages` you can also get values out with them, put new values in, and all the array operations.

The purpose of a pointer is to let you manually index into blocks or memory when an array won't do it right. In almost all other cases you actually want to use an array. But, there are times when you *have* to work with a raw block of memory and that's where a pointer comes in. A pointer gives you raw, direct access to a block of memory so you can work with it.

The final thing to grasp at this stage is that you can use either syntax for most array or pointer operations. You can take a pointer to something, but use the array syntax for accessing it. You can take an array and do pointer arithmetic with it.

Practical Pointer Usage

There are four primary useful things you do with pointers in C code:

- Ask the OS for a chunk of memory and use a pointer to work with it. This includes strings and something you haven't seen yet,`structs`.

- Passing large blocks of memory (like large structs) to functions with a pointer so you don't have to pass the whole thing to them.
- Taking the address of a function so you can use it as a dynamic callback.
- Complex scanning of chunks of memory such as converting bytes off a network socket into data structures or parsing files.

For nearly everything else you see people use pointers, they should be using arrays. In the early days of C programming people used pointers to speed up their programs because the compilers were really bad at optimizing array usage. These days the syntax to access an array vs. a pointer are translated into the same machine code and optimized the same, so it's not as necessary. Instead, you go with arrays every time you can, and then only use pointers as a performance optimization if you absolutely have to.

The Pointer Lexicon

I'm now going to give you a little lexicon to use for reading and writing pointers. Whenever you run into a complex pointer statement, just refer to this and break it down bit by bit (or just don't use that code since it's probably not good code):

`type *ptr`

"a pointer of type named ptr"

`*ptr`

"the value of whatever ptr is pointed at"

`*(ptr + i)`

"the value of (whatever ptr is pointed at plus i)"

`&thing`

"the address of thing"

`type *ptr = &thing`

"a pointer of type named ptr set to the address of thing"

`ptr++`

"increment where ptr points"

We'll be using this simple lexicon to break down all of the pointers we use from now on in the book.

Pointers Are Not Arrays

No matter what, you should never think that pointers and arrays are the same thing. They are not the same thing, even though C lets you work with them in many of the same ways. For example, if you do `sizeof(cur_age)` in the code above, you would get the size of the *pointer*, not the size of what it points at. If you want the size of the full array, you have to use the array's name, `age` as I did on line 12.

TODO: expand on this some more with what doesn't work on both the same.

How To Break It

You can break this program by simply pointing the pointers at the wrong things:

- Try to make `cur_age` point at `names`. You'll need to use a C cast to force it, so go look that up and try to figure it out.
- In the final `for-loop` try getting the math wrong in weird ways.
- Try rewriting the loops so they start at the end of the arrays and go to the beginning. This is harder than it looks.

Extra Credit

- Rewrite all the array usage in this program so that it's pointers.
- Rewrite all the pointer usage so they're arrays.
- Go back to some of the other programs that use arrays and try to use pointers instead.
- Process command line arguments using just pointers similar to how you did `names` in this one.
- Play with combinations of getting the value of and the address of things.
- Add another `for-loop` at the end that prints out the addresses these pointers are using. You'll need the `%p` format for `printf`.
- Rewrite this program to use a function for each of the ways you're printing out things. Try to pass pointers to these functions so they work on the data. Remember you can declare a function to accept a pointer, but just use it like an array.
- Change the `for-loops` to `while-loops` and see what works better for which kind of pointer usage.

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:

```
#include <stdio.h>

#include <assert.h>

#include <stdlib.h>

#include <string.h>



struct Person {

    char *name;

    int age;

    int height;

    int weight;
```

```c
};


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;

    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:

- I use `malloc` for "memory allocate" to ask the OS to give me a piece of raw memory.
- I pass to `malloc` the `sizeof(struct Person)` which calculates the total size of the struct, given all the fields inside it.
- 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.
- I initialize each field of `struct Person` using the `x->y` syntax, to say what part of the struct I want to set.
- 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:

- Create two people, `joe` and `frank`.
- 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.
- Age both of them by 20 years, with changes to their body too.
- Print each one after aging them.
- Finally destroy the structures so we can clean up correctly.

Go through this description carefully, and do the following:

- 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.
- Write a *comment* above each and every single line saying what the line does in English.
- Trace through each function call and variable so you know where it comes from in the program.
- Look up any symbols you don't know as well.

What You Should See

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

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:

- In the above code, you make a `struct` that has the fields you'd expect for a person: name, age, weight, height.
- Each of those fields has a type, like `int`.
- C then packs those together so they can all be contained in one single `struct`.
- 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.
- This lets you pass the whole cohesive grouping to other functions, as you did with `Person_print`.
- You can then access the individual parts of a `struct` by their names using `x->y` if you're dealing with a pointer.
- 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.

How To Break It

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

- 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`.
- 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.
- 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.
- This time, pass `NULL` to `Person_print` and see what `Valgrind` thinks of that.
- You should be figuring out that `NULL` is a quick way to crash your program.

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:

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