

Head First C

Wouldn't it be dreamy if there were a book on C that was better than having a root canal at the dentist? I guess it's just a fantasy...



David Griffiths
Dawn Griffiths

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Head First C

by David Griffiths and Dawn Griffiths

Copyright © 2011 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators: Kathy Sierra, Bert Bates

Editor: Brian Sawyer

Cover Designers:

Production Editor:

Proofreader:

Indexer:

Page Viewers: Mum and Dad, Carl

Printing History:

December 2011: First Edition.

Mum and Dad →



← Carl

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First C*, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

No kittens were harmed in the making of this book. Really.



This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-1-449-39991-7

[C]

To Brian Kernighan and Dennis Ritchie for inventing C.

Authors of Head First C



David Griffiths



Dawn Griffiths

David Griffiths began programming at age 12, when he saw a documentary on the work of Seymour Papert. At age 15, he wrote an implementation of Papert's computer language LOGO. After studying Pure Mathematics at University, he began writing code for computers and magazine articles for humans. He's worked as an agile coach, a developer, and a garage attendant, but not in that order. He can write code in over 10 languages and prose in just one, and when not writing, coding, or coaching, he spends much of his spare time travelling with his lovely wife—and co-author—Dawn.

Before writing *Head First C*, David wrote two other *Head First* books: *Head First Rails* and *Head First Programming*.

You can follow him on Twitter at:

<http://twitter.com/dogriffiths>

Dawn Griffiths started life as a mathematician at a top UK university where she was awarded a first-class honours degree in mathematics. She went on to pursue a career in software development, and has over 15 years experience working in the IT industry.

Before joining forces with David on *Head First C*, Dawn wrote two other *Head First* books (*Head First Statistics* and *Head First 2D Geometry*) and has also worked on a host of other books in the series.

When Dawn's not working on *Head First* books, you'll find her honing her Tai Chi skills, running, making bobbin lace or cooking. She also enjoys traveling and spending time with her husband, David.

Table of Contents (Summary)

	Intro	xxiii
1	Diving in: <i>Getting started with C</i>	1
2	Memory and pointers: <i>What are you pointing at?</i>	37
3	Do one thing and do it well: <i>Creating small tools</i>	95
4	Break it down, build it up: <i>Using multiple source files</i>	149
5	Structs, unions, and bitfields: <i>Rolling your own structures</i>	
6	Data structures and dynamic memory: <i>Connecting your custom data types</i>	
7	Reusable utilities: <i>Turning your functions up to 11</i>	
8	Dynamic libraries: <i>Hot, sweappable code</i>	
9	Creating new processes: <i>Process mojo</i>	
10	Using multiple source files: <i>Doing more than one thing at once</i>	
11	Sockets and asynchronous I/O: <i>Talking to the network</i>	
12	Inter-process communication: <i>Living in a community</i>	

Table of Contents (the real thing)

Intro

Your brain on C. Here *you* are trying to *learn* something, while here your *brain* is doing you a favor by making sure the learning doesn't *stick*. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how *do* you trick your brain into thinking that your life depends on knowing C?

Who is this book for?	xxiv
We know what you're thinking	xxv
Metacognition	xxvii
Bend your brain into submission	xxix
Read me	xxx
The technical review team	xxxii
Acknowledgments	xxxiii

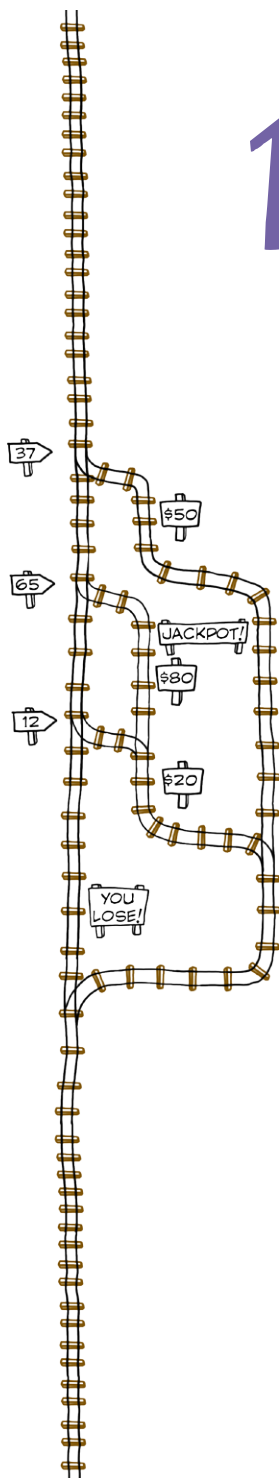
getting started with C

1

Diving in

Want to get inside the computers head?

Need to write **high-performance code** for a new game? Program an **Arduino**? Or use that advanced **third-party library** in your iPhone app? If so, then C's there to help. C works at a **much lower level** than most other languages, so understanding C gives you a much better idea of **what's really going on**. C can even help you better understand other languages as well. So dive in, grab your compiler, and get started in no time.



memory and pointers

What are you pointing at?

2

If you want to kick butt with C, you need to understand how C handles memory.

The C language gives you a lot more *control* over how your program uses the **computer's memory**. In this chapter, you'll strip back the covers and see exactly what happens when you **read and write variables**. You'll learn **how arrays work**, how to avoid some **nasty memory SNAFUs**, and most of all, you'll see how **mastering pointers and memory addressing** is key to becoming a kick-ass C programmer.



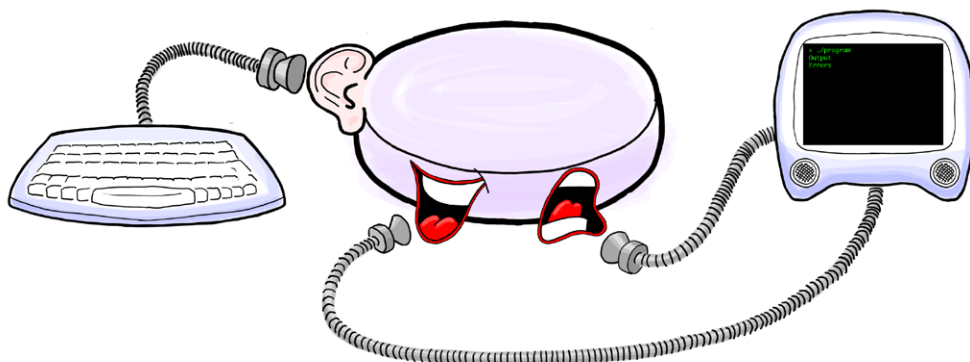
creating small tools

Do one thing and do it well

3

Every operating system includes small tools.

Small tools perform **specialized small tasks**, such as reading and writing files, or filtering data. If you want to perform more complex tasks, you can even *link several tools together*. But how are these small tools built? In this chapter, you'll look at the building blocks of creating small tools. You'll learn how to control **command-line options**, how to manage **streams of information**, and **redirection**, getting toolled up in no time.



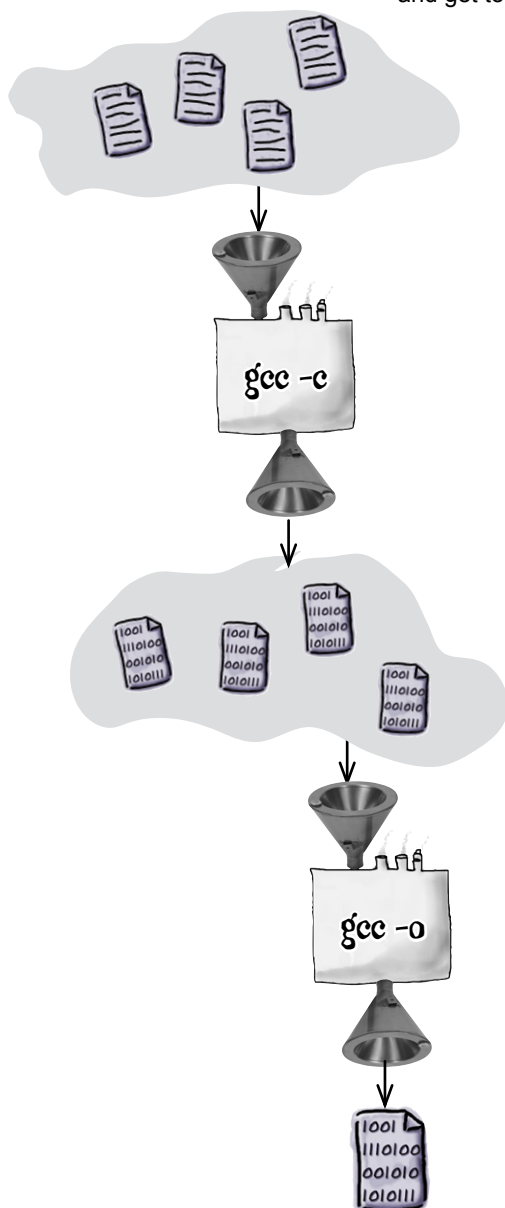
using multiple source files

Break it down, build it up

4

If you create a big program, you don't want a big source file.

Can you imagine how difficult and time-consuming a single source file for an enterprise level program would be to maintain? In this chapter, you'll learn how C allows you to break your source code into **small manageable chunks** and then rebuild them into **one huge program**. Along the way, you'll learn a bit more about **data-type subtleties**, and get to meet your new best friend: **make**.



struts, unions, and bitfields

5

Rolling your own structures

Most things in life are more complex than a simple number.

So far we've looked at the basic data-types of the C language, but what if you want to go beyond numbers and pieces of text, and **model things in the real world**?

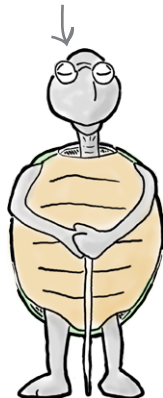
Structs allow you to model **real-world complexities** by writing your own structures.

We'll show you how to **combine the basic data-types** into structs, and even **handle life's uncertainties** with **unions**. And if you're after a simple yes or no, **bitfields** may be just what you need.

This is Myrtle...



...but her clone is sent to the function.



Turtle "t".



data structures and dynamic memory

6

Connecting your custom data types

Once you've created custom data types, the next thing to do is connect them together.

This chapter begins by looking in more detail at why most coders pass struct pointers rather than structs themselves. Then, you'll use struct pointers to connect custom data types into large, complex data structures to model real world data. To make the data structures cope with flexible amounts of data, you'll finally look at how to dynamically allocate memory on the heap and ways of tidying away memory when we're done with it.



reusable utilities

7 Turn your functions up to 11

Basic functions are great, but sometimes you need more.

Earlier chapters have looked at basic functions, but what if you need even more power and flexibility to achieve what you want? Topics in the chapter include:

rubber functions, or how to have a flexible number of arguments, and how passing functions as parameters can multiply your code's IQ. By the end of this chapter, you will be able to write more flexible, reusable, and powerful utilities.

dynamic libraries

8

Hot, swappable code

You don't always need to use everything.

You don't pack your swimsuit if you're going to Alaska. And you don't write programs that load code unless they need it. This chapter will show you how to split your problem into dynamically loaded libraries. By the end of the chapter, you will be able to create dynamic libraries that can switch at runtime, making their applications more dynamic and configurable.

creating new processes

Process mojo

9

Every operating system includes small tools.

Programs often need to create and manage other processes. This chapter will teach you how to spawn new processes and how to communicate with them once they're running. By the end of this chapter, you will understand how to use `fork()` and `exec()` calls to spawn/replace processes and how to use signals to communicate with other processes.

using multiple source files

10

Doing more than one thing at once

Programs often need to several things at the same time.

POSIX threads can boost the performance of your code by spinning off several pieces of code to run in parallel. But... be careful! Threads are powerful tools, but you don't want them crashing them into each other. In this chapter, you'll learn how to put up traffic signs and lane markers that will prevent a code pile-up. By the end, you will know how to create POSIX threads and how to use synchronization mechanisms to protect the integrity of sensitive data.

sockets and asynchronous i/o

11

Talking to the network

Many programs need to talk to programs on a different machine.

You've learned how to use I/O to communicate with files and how processes on the same machine can communicate with each other. Now you're going to reach to the rest of the world and see how we can write C programs that can talk to other programs across the network and across the world. By the end of this chapter, you will be able to create programs that behave as servers and programs that behave as clients.

12

inter-process communication

Living in the community

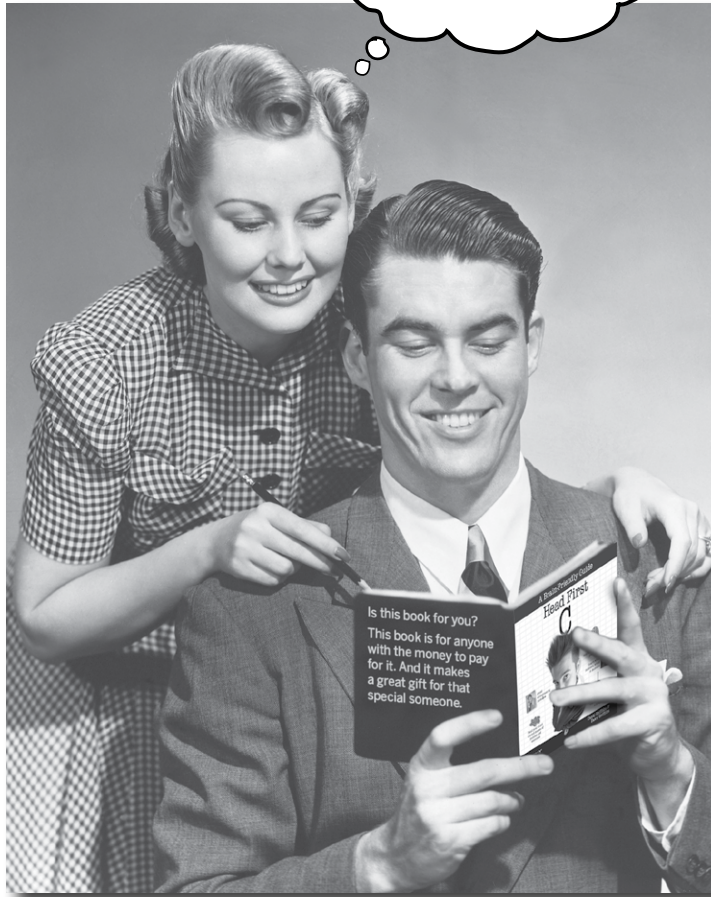
Programs need to work together.

Thankfully, the C language provides a set of tools that makes this possible. Two programs need to share live data? Well, they can share memory. Two programs need to talk to each other? Try connecting them with a pair of pipes. These tools allow programs to communicate and cooperate. But like any civilized conversation, rules need to be observed. In this chapter, you'll learn how **locking mechanisms** like semaphores can prevent dog fights and keep your computer a civilized, well-ordered, and stable place to be.

how to use this book

Intro

I can't believe they put *that* in a C book.



In this section we answer the burning question:
"So why DID they put that in a C book?"

Who is this book for?

If you can answer “yes” to all of these:

- 1 Do you already know how to program in another programming language?
- 2 Do you want to master C, create the next big thing in software, make a small fortune, and retire to your own private island?
- 3 Do you prefer actually doing things and applying the stuff you learn over listening to someone in a lecture rattle on for hours on end?

← OK, maybe that one's a little far-fetched. But, you gotta start somewhere, right?

this book is for you.

Who should probably back away from this book?

If you can answer “yes” to any of these:

- 1 Are you looking for a quick introduction or reference book to C?
- 2 Would you rather have your toenails pulled out by 15 screaming monkeys than learn something new? Do you believe a C book should cover *everything* and if it bores the reader to tears in the process then so much the better?

this book is **not** for you.



[Note from marketing: this book is for anyone with a credit card... we'll accept a check, too.]

We know what you're thinking

“How can *this* be a serious C book?”

“What’s with all the graphics?”

“Can I actually *learn* it this way?”

We know what your *brain* is thinking

Your brain craves novelty. It’s always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain’s *real* job—recording things that *matter*. It doesn’t bother saving the boring things; they never make it past the “this is obviously not important” filter.

How does your brain *know* what’s important? Suppose you’re out for a day hike and a tiger jumps in front of you, what happens inside your head and body?

Neurons fire. Emotions crank up. *Chemicals surge.*

And that’s how your brain knows...

This must be important! Don't forget it!

But imagine you’re at home, or in a library. It’s a safe, warm, tiger-free zone. You’re studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, ten days at the most.

Just one problem. Your brain’s trying to do you a big favor. It’s trying to make sure that this *obviously* non-important content doesn’t clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never have posted those “party” photos on your Facebook page. And there’s no simple way to tell your brain, “Hey brain, thank you very much, but no matter how dull this book is, and how little I’m registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around.”

Your brain thinks
THIS is important.



Great. Only 464
more dull, dry,
boring pages.

Your brain thinks
THIS isn't worth
saving.



We think of a “Head First” reader as a learner.

So what does it take to *learn* something? First, you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know what turns your brain on.

Some of the Head First learning principles:

Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). It also makes things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another page, and learners will be up to *twice* as likely to solve problems related to the content.

Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner party companion, or a lecture?

Get the learner to think more deeply. In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

Get—and keep—the reader's attention. We've all had the “I really want to learn this but I can't stay awake past page one” experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.

Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, “what the...?”, and the feeling of “I Rule!” that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that “I'm more technical than thou” Bob from engineering *doesn't*.

Metacognition: thinking about thinking

If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

But we assume that if you're holding this book, you really want to learn how to program. And you probably don't want to spend a lot of time. If you want to use what you read in this book, you need to *remember* what you read. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

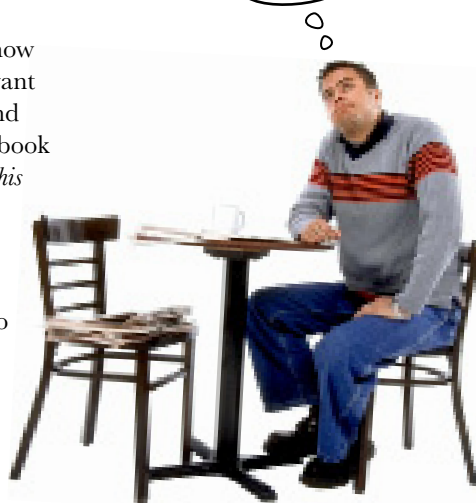
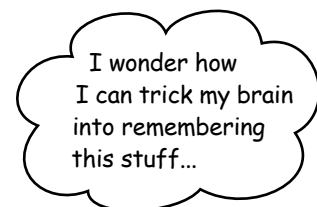
So just how **DO** you get your brain to treat programming like it was a hungry tiger?

There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dulllest of topics if you keep pounding the same thing into your brain. With enough repetition, your brain says, "This doesn't *feel* important to him, but he keeps looking at the same thing *over* and *over* and *over*, so I suppose it must be."

The faster way is to do **anything that increases brain activity**, especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to make sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

But pictures and conversational style are just the beginning...



Here's what WE did:

We used **pictures**, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really *is* worth a thousand words. And when text and pictures work together, we embedded the text *in* the pictures because your brain works more effectively when the text is *within* the thing the text refers to, as opposed to in a caption or buried in the text somewhere.

We used **redundancy**, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.

We used concepts and pictures in **unexpected** ways because your brain is tuned for novelty, and we used pictures and ideas with at least *some emotional content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little **humor, surprise, or interest**.

We used a personalized, **conversational style**, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included more than 80 **activities**, because your brain is tuned to learn and remember more when you **do** things than when you *read* about things. And we made the exercises challenging-yet-do-able, because that's what most people prefer.

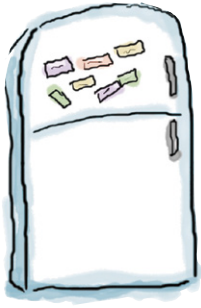
We used **multiple learning styles**, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.

We include content for **both sides of your brain**, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

And we included **stories** and exercises that present **more than one point of view**, because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.

We included **challenges**, with exercises, and by asking **questions** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That **you're not spending one extra dendrite** processing a hard-to-understand example, or parsing difficult, jargon-laden, or overly terse text.

We used **people**. In stories, examples, pictures, etc., because, well, because *you're* a person. And your brain pays more attention to *people* than it does to *things*.



Cut this out and stick it on your refrigerator.

Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.

1 **Slow down. The more you understand, the less you have to memorize.**

Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

2 **Do the exercises. Write your own notes.**

We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

3 **Read the "There are No Dumb Questions"**

That means all of them. They're not optional sidebars, **they're part of the core content!** Don't skip them.

4 **Make this the last thing you read before bed. Or at least the last challenging thing.**

Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

5 **Talk about it. Out loud.**

Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

6 **Drink water. Lots of it.**

Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

7 **Listen to your brain.**

Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

8 **Feel something.**

Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

9 **Write a lot of code!**

There's only one way to learn to program in C: **write a lot of code.** And that's what you're going to do throughout this book. Coding is a skill, and the only way to get good at it is to practice. We're going to give you a lot of practice: every chapter has exercises that pose a problem for you to solve. Don't just skip over them—a lot of the learning happens when you solve the exercises. We included a solution to each exercise—don't be afraid to **peek at the solution** if you get stuck! (It's easy to get snagged on something small.) But try to solve the problem before you look at the solution. And definitely get it working before you move on to the next part of the book.

Read Me

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of learning whatever it is we're working on at that point in the book. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

We assume you're new to C, but not to programming.

We assume that you've already done some programming. Not a lot, but we'll assume you've already seen things like loops and variables in some other language, like JavaScript. C is actually a pretty advanced language, so if you've never done any programming *at all*, then you might want to read some other book before you start on this one. We'd suggest starting with *Head First Programming*.

You need to install a C compiler on your computer.

Throughout the book we'll be using the *Gnu Compiler Collection* (gcc) because it's free and, well, we think it's just a pretty darned good compiler. You'll need to make sure you have gcc installed on your machine. The good news is, if you have a *Linux* computer, then you should already have gcc. If you're using a Mac, you'll need to install the Xcode/Developer tools. You can either download these from the Apple *App Store* or by downloading them from Apple. If you're on a Windows machine you have a couple options. *Cygwin* (<http://www.cygwin.com>) gives you a complete simulation of a *UNIX* environment, including gcc. But if you want to create programs that will work on Windows plain-and-simple, then you might want to install the *Minimalist GNU for Windows* (MingW) from <http://www.mingw.org>.

All the code in this book is intended to run across *all* these operating systems and we've tried hard not to write anything that will only work on one type of computer. Occasionally there will be some differences, but we'll make sure to point those out to you. .

We begin by teaching some basic C concepts, then we start putting C to work for you right away.

We cover the fundamentals of C in Chapter 1. That way, by the time you make it all the way to Chapter 2, you are creating programs that actually do something real, useful, and—gulp!—fun. The rest of the book then builds on your C skills turning you from *C newbie* to *coding ninja master* in no time.

The activities are NOT optional.

The exercises and activities are not add-ons; they're part of the core content of the book. Some of them are to help with memory, some are for understanding, and some will help you apply what you've learned. *Don't skip the exercises.*

The redundancy is intentional and important.

One distinct difference in a Head First book is that we want you to *really* get it. And we want you to finish the book remembering what you've learned. Most reference books don't have retention and recall as a goal, but this book is about *learning*, so you'll see some of the same concepts come up more than once.

The examples are as lean as possible.

Our readers tell us that it's frustrating to wade through 200 lines of an example looking for the two lines they need to understand. Most examples in this book are shown within the smallest possible context, so that the part you're trying to learn is clear and simple. Don't expect all of the examples to be robust, or even complete—they are written specifically for learning, and aren't always fully-functional.

The Brain Power exercises don't have answers.

For some of them, there is no right answer, and for others, part of the learning experience of the Brain Power activities is for you to decide if and when your answers are right. In some of the Brain Power exercises, you will find hints to point you in the right direction.

The technical review team

Acknowledgments

Our editor:

Many thanks to **Brian Sawyer** for asking us to write this book in the first place. Brian believed in us every step of the way, gave us the freedom to try out new ideas, and didn't panic too much when deadlines loomed.

Brian Sawyer



The O'Reilly team:

A big thank you goes to the lovely **Karen Shaner** made us feel at home in Boston, and was always there to help us track down elusive images. Thanks also to **Laurie Petrycki** for keeping us well-fed and well-motivated.

Family, Friends and colleagues:

We've made a lot of friends on our *Head First* journey. A special thanks goes to **Lou Barr**, **Brett McLaughlin**, and **Sanders Kleinfeld** for teaching us so much.

David: My thanks to **Andy Parker**, **Joe Broughton**, **Carl Jacques**, and **Simon Jones** and the many other friends who have heard so little from me whilst I was busy scribbling away.

Dawn: Work on this book would have been a lot harder without my amazing support network of family and friends. Special thanks go to Mum and Dad, Carl, Steve, Gill, Jacqui, Joyce, and Paul. I've truly appreciated all your support and encouragement.

The without-whom list:

Our technical review team did an excellent job of keeping us straight and making sure what we covered was spot on.

Finally, our thanks to **Kathy Sierra** and **Bert Bates** for creating this extraordinary series of books

Safari® Books Online



When you see a Safari® icon on the cover of your favorite technology book that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at <http://safari.oreilly.com>.

1 getting started with c

Diving in



Don't you just love the deep blue C? Come on in, the water's lovely!

Want to get inside the computers head?

Need to write **high-performance code** for a new game? Program an **Arduino**? Or use that advanced **third-party library** in your iPhone app? If so, then C's there to help. C works at a **much lower level** than most other languages, so understanding C gives you a much better idea of **what's really going on**. C can even help you better understand other languages as well. So dive in, grab your compiler, and get started in no time.

C is a language for small, fast programs

The C language is designed to create small, fast programs. It's lower-level than most other languages; that means *it creates code that's a lot closer to what machines really understand.*

The way C works

Computers really only understand one language - machine code, a binary stream of 1s and 0'. You convert your C code into machine code with the aid of a **compiler**.



1

Source

You start off by creating a source file. The source file contains human-readable C code.

2

Compile

You run your source code through a compiler. The compiler checks for errors, and once it's happy, it compiles the source code.

3

Output

The compiler creates a new file called an *executable*. This file contains machine code, a stream of 1's and 0's that the computer understands. And that's the program you can run.

C is used where speed and space are important. Most operating systems are written in C. Most other computer languages are also written in C. And most games software is written in C.



Sharpen your pencil

Try to guess what each of these code fragments do.

Describe what you think the code does.



```
int card_count = 11;
if (card_count > 10)
    puts("The deck is hot. Increase bet.");
```

.....


```
int c = 10;
while (c > 0) {
    puts("I must not write code in class");
    c = c - 1;
}
```

.....


```
/* Assume name shorter than 20 chars. */
char ex[20];
puts("Enter boyfriend's name: ");
scanf("%s", ex);
printf("Dear %s.\n\n\tYou're history.\n", ex);
```

.....


```
char suit = 'H';
switch(suit) {
case 'C':
    puts("Clubs");
    break;
case 'D':
    puts("Diamonds");
    break;
case 'H':
    puts("Hearts");
    break;
default:
    puts("Spades");
}
```

.....

Sharpen your pencil

Solution

```
int card_count = 11;
if (card_count > 10)
    puts("The deck is hot. Increase bet.");
```

← An integer is a whole number.

← This displays a string on the command prompt or terminal.

Create an integer variable and set it to 11.
 Is the count more than 10?
 If so, display a message on the command prompt.

```
int c = 10;
while (c > 0) {
    puts("I must not write code in class");
    c = c - 1;
}
```

← The braces define a block statement.

Create an integer variable and set it to 10.
 As long as the value is positive...
 ...display a message...
 ...and decrease the count.
 The end of the code that should be repeated

```
/* Assume name shorter than 20 chars. */
char ex[20];
puts("Enter boyfriend's name: ");
scanf("%s", ex);
printf("Dear %s.\n\n\tYou're history.\n", ex);
```

← This means "store everything the user types into the ex array".

← This will insert this string of characters here in place of the %s.

This is a comment.
 Create an array of 20 characters.
 Display a message on the screen.
 Store what the user enters into the array.
 Display a message including the text entered

```
char suit = 'H';
switch(suit) {
case 'C':
    puts("Clubs");
    break;
case 'D':
    puts("Diamonds");
    break;
case 'H':
    puts("Hearts");
    break;
default:
    puts("Spades");
}
```

← A switch statement checks a single variable for different values.

Create a character variable, store the letter 'H'
 Look at the value of the variable.
 Is it 'C'?
 If so, display the word "Clubs".
 Then skip past the other checks.
 Is it 'D'?
 If so, display the word "Diamonds".
 Then skip past the other checks.
 Is it 'H'?
 If so, display the word "Hearts".
 Then skip past the other checks.
 Otherwise...
 Display the word "Spades"
 This is the end of tests.

But what does a complete C program look like?

In order to create a full program, you need to enter your code into a *C source file*. C source files can be created by any text editor and their filenames usually end with `.c`. ← This is just a convention, but you should follow it.

Let's have a look at a typical C source file.

1 C programs normally begin with a comment.

The comment describes the purpose of the code in the file, and maybe some license or copyright information. There's no absolute need to include a comment here - or anywhere else in the file - but it's good practice and what most C programmers will expect to find.

```
/*
 * Program to calculate the number of cards in the shoe.
 * This code is released under the Vegas Public License.
 * (c)2014, The College Blackjack Team.
 */
```

2 Next comes the includes section.

C is a very, very small language and it can do almost nothing without the use of *external libraries*. You will need to tell the compiler what external code to use by including header files for the relevant libraries. The header you will see more than any other is **stdio.h**. The `stdio` library contains code that allows you to read and write data from and to the terminal.

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

```
int main()
{
    int decks;
    puts("Enter a number of decks");
    scanf("%i", &decks);
    if (decks < 1) {
        puts("That is not a valid number of decks");
        return 1;
    }
    printf("There are %i cards\n", (decks * 52));
    return 0;
}
```

3 The last thing you find in a source file are the functions.

All C code runs inside functions. The most important function you will find in any C program is called the **main() function**. The `main()` function is the start point for all of the code in your program.

So let's look at the main() function in a little more detail.



The main() Function Up Close

The computer will start running your program from the `main()` function. The name is important - if you don't have a function called `main`, your program won't be able to start.

The main function has a **return type** of `int`. So what does this mean? Well - when the computer runs your program it will need to have some way of deciding if the program ran successfully or not. It does this by checking the *return value* of the main function. If the main function returns 0, this means that the program was successful. If it returns any other value, it means that there was a problem.

This is the return type - it should always be `int` for the main function.

Because it's called 'main' the program will start here.

If we had any parameters, they'd be mentioned here.

The body of the function is always surrounded by braces.

```
int main()
{
    int decks;
    puts("Enter a number of decks");
    scanf("%i", &decks);
    if (decks < 1) {
        puts("That is not a valid number of decks");
        return 1;
    }
    printf("There are %i cards\n", (decks * 52));
    return 0;
}
```

The function name comes after the return type. That's followed by the function parameters if there are any. Finally we have the *function body*. The function body **must** be surrounded by *braces*.



Geek Bits

The `printf()` function is used to display **formatted output**. It replaces format characters with the values of variables, like this:

1st parameter will be inserted here as a string.

```
printf("%s says the count is %i", "Ben", 21);
```

2nd parameter will be inserted here as an integer.

You can include as many parameters as you like when you call the `printf()` function - but make sure you have a matching %-format character for each one.

If you want to check the exit status of a program type

```
echo %ErrorLevel%
```

in Windows or

```
echo $?
```

in Linux or the Mac



Code Magnets

The College Blackjack Team were working on some code on the dorm fridge, but someone mixed their magnets up! Can you re-assemble the code from the magnets?

```

/*
 * Program to evaluate face values.
 * Released under the Vegas Public License.
 * (c)2014 The College Blackjack Team.
 */
.....

.....

....._main()
{
    char card_name[3];
    puts("Enter the card_name: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {

        .....

    } else if (card_name[0] == ..... ) {
        val = 10;

    } ..... (card_name[0] == ..... ) {

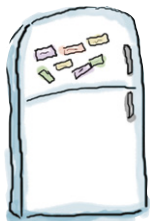
        .....

    } else {
        val = atoi(card_name);
    }
    printf("The card value is: %i\n", val);
    ..... 0;
}

```

<stdlib.h> ;
 ; val = 11
 int 'J'
 #include 'A'

return
 else #include
 if val = 10
 <stdio.h>



Code Magnets Solution

The College Blackjack Team were working on some code on the dorm fridge, but someone mixed their magnets up! Can you re-assemble the code from the magnets?

```

/*
 * Program to evaluate face values.
 * Released under the Vegas Public License.
 * (c)2014 The College Blackjack Team.
 */

#include <stdio.h>

#include <stdlib.h>

int main()
{
    char card_name[3];
    puts("Enter the card_name: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    printf("The card value is: %i\n", val);
    return 0;
}

```

there are no Dumb Questions

Q: What does `card_name[0]` mean?

A: It's the first character that the user typed. So if they type "10", `card_name[0]` would be '1'.

Q: Do you always write comments using `/*` and `*/`?

A: If your compiler also supports C++ then you can begin a comment with `///
//`. The compiler treats the rest of that line as a comment.

Q: C++? So that's not a C comment then?

A: No. Although most compilers will cope with it.

But how will we run the program?

C is a *compiled language*. That means the computer will not interpret the code directly. Instead you will need to convert - or *compile* - the human-readable source code into machine-readable *machine code*.

To compile the code you need a program called a **compiler**. One of the most popular C compilers is the *GNU Compiler Collection* or **gcc**. gcc is available on a lot of operating systems and it can compile lots of languages other than C. Best of all - it's completely free.

Here's how you can compile and run the program using gcc:

- 1 Save the code from the Code Magnets exercise on the opposite page in a file called `cards.c`.



cards.c

← C source files usually end `.c`.

- 2 Compile with `gcc cards.c -o cards` at a command prompt or terminal.

Compile `cards.c`
to a file called `cards`.

```
File Edit Window Help Compile
> gcc cards.c -o cards
>
```



cards.c



cards

↑
This will be `cards.exe`
if you're on Windows.

- 3 Run by typing `cards` on Windows, or `./cards` on Mac and Linux machines.

```
File Edit Window Help Compile
> ./cards
Enter the card_name:
```



Geek Bits

You can compile and run your code in a single step like this:

⚡ here means 'and then if it's successful, do this..'

```
gcc test.c -o test && ./test
```

← You should put
"test" instead
of "./test" on a
Windows machine.

This command will only run the new program if it compiles successfully. If there's a problem with the compile, it will skip running the program and simply display the errors on the screen.



You should create the `cards.c` file and compile it now. We'll be working on it more and more as the chapter progresses.



TEST DRIVE

Let's see if the program compiles and runs. Open up a command prompt or terminal on your machine and try it out.

This line compiles the code and creates the cards program.

This line runs the program. If you're on Windows don't type the ./

Running the program again.

The user enters the name from a card...

...and the program displays the corresponding value.

```
File Edit Window Help 21
> gcc cards.c -o cards
> ./cards
Enter the card_name:
Q
The card value is: 10
> ./cards
Enter the card_name:
A
The card value is: 11
> ./cards
Enter the card_name:
7
The card value is: 7
```

Remember - you can combine the compile and run steps together (turn back a page to see how).


The program works!

Congratulations! You have compiled and run a C program. The `gcc` compiler took the human-readable source code from `cards.c` and converted it into computer-readable *machine code* in the `cards` program. If you are using a Mac or Linux machine, the compiler will have created the machine code in a file called **cards**. But on Windows, all programs need to have a `.exe` extension, so the file will be called **cards.exe**.

there are no Dumb Questions

Q: Why do I have to prefix the program with `./` when I run it on Linux and the Mac?

A: On Unix-style operating systems, programs are only run if you specify the directory where they live or if their directory is listed in the `PATH` environment variable.



Wait - I don't get it. When we ask the user what the name of the card is we're using an array of characters. An **array of characters**???? Why? Can't we use a **string** or something???

The C language doesn't support strings out of the box.

C is a lower-level than most other languages and so instead of strings, C normally uses something similar: *an array of single characters*. If you've programmed in other languages you've probably met an array before. An array is just a list of things given a single name. So `card_name` is just a variable name we use to refer to the list of characters entered at the command prompt. We defined `card_name` to be a *2 character array* so we can refer to the first and second character as `char_name[0]` and `char_name[1]`. To see how this works, let's take a deeper dive into the memory of the computer's memory and see how C handles text...

← But there are a number of C extension-libraries that do give you strings.



Strings Way Up Close

Strings are just character arrays. When C sees a string like this:

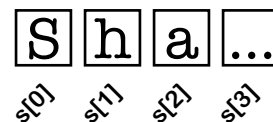
```
s="Shatner"
```

it reads it like it was just an array of separate characters:

```
s = { 'S', 'h', 'a', ... }
```

← This is how you define an array in C.

Each of the characters in the string is just an element in an array, which is why you can refer to the individual characters in the string by using an index, like `s[0]` and `s[1]`.



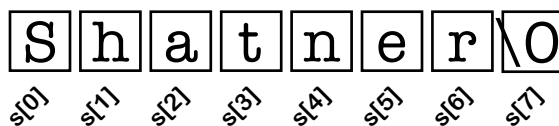
Don't fall off the end of the string

But what happens when C wants to read the contents of the string? Say it wants to print it out. Now in a lot of languages the computer keeps a pretty close track on the size of an array, but C is lot lower-level than most languages and it can't always work out exactly *how long* an array is. If C is going to display a string on the screen, it needs to know when it gets to the end of the character array. And it does this by adding a **sentinal character**.

The sentinel character is an additional character at the end of the string that has the value `'\0'`. Whenever the computer needs to read the contents of the string, it goes through the elements of the character array one at a time, until it reaches `'\0'`. That means that when the computer see this:

```
s="Shatner"
```

it actually stores it in memory like this:



← `'\0'` is the character with ASCII value 0.



That's why in our code we had to define the `card_name` like this:

```
char card_name[3];
```

The `card_name` string is only ever going to record 1 or 2 characters, but because strings end in a *sentinal character* we have to allow for an extra character in the array.

there are no
Dumb Questions

Q: Why are the characters numbered from 0? Why not 1?

A: The index is an offset - it's a measure of how far the character is from the first character

Q: Why?

A: The computer will store the characters in consecutive bytes of memory. It can use the index to calculate the location of the character. If it knows that `c[0]` is at memory location 1,000,000 then it can quickly calculate that `c[96]` is at 1,000,000 + 96.

Q: Why does it need a sentinel character? Doesn't it know how long the string is?

A: Usually it doesn't. C is not very good at keeping track of how long arrays are - and a string is just an array.

Q: It doesn't know how long arrays are???

A: No. Sometimes the compiler can work out the length of an array by analyzing the code, but usually C relies on you to keep track of your arrays.

Q: Does it matter if I use single quotes of double quotes.

A: Yes. Single quotes are used for individual characters, but double quotes are always used for strings.

Q: So should I define my strings using quotes (") or as explicit arrays of characters?

A: Usually you will define strings using quotes. They are called literal strings and they are easier to type.

Q: Are there any differences between literal strings and character arrays

A: Only one - literal strings are constant

Q: What does that mean?

A: It means that you can't change the individual characters once they are created

Q: What will happen if I try?

A: It will depend upon the compiler, but gcc will usually display a Bus Error

Q: What the heck's a bus error?

A: C will store literal strings in memory in a different way. A bus error just means that your program can't update that piece of memory.

Two types of command

So far every command we've seen has fallen into one of two categories:

Do something

Most of the commands in C are statements. Simple statements are *actions*, they *do* things and they *tell us* things. We've met statements that define variables, that read input from the keyboard or display data to the screen.

```
split_hand(); ← This is a simple statement.
```

Sometimes we group statements together to create *block statements*. Block statements are groups of commands surrounded by braces.

These commands form a block statement because they are surrounded by braces.

```
{
    deal_first_card();
    deal_second_card();
    cards_in_hand = 2;
}
```



Do you need braces?

Block statements allow you to treat a *whole set* of statements as if they were a single statement. You see them a lot in conditions and loops.

But most C programs omit the braces if you just run a single line of code. So instead of writing:

```
if (x == 2) {
    puts("Do something");
}
```

most C programmers write:

```
if (x == 2)
    puts("Do something");
```

Do something only if something is true

Control statements such as `if` check a condition before running the code:

```
if (value_of_hand <= 16) ← This is the condition.
    hit(); ← Run this statement if the condition is true.
else
    stand(); ← Run this statement if the condition is false.
```

if statements typically need to do more than one thing when a condition is true, so they are often used with block statements:

```
if (dealer_card == 6) {
    double_down();
    hit();
}
```

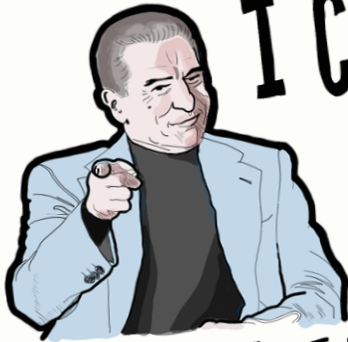
BOTH of these commands will run if the condition is true. The commands are grouped inside a single block statement.

Here's the code so far

```
/*
 * Program to evaluate face values.
 * Released under the Vegas Public License.
 * (c)2014 The College Blackjack Team.
 */
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    puts("Enter the card_name: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    printf("The card value is: %i\n", val);
    return 0;
}
```

I've had a thought.
Could this check if
a card value is in a
particular range? That
might be handy...





I CAN MAKE YOU RICH JUST LIKE ME!

The Eddie Rich blackjack correspondence school

Hey - how's it going? You look to me like a smart guy. And I know - 'cause I'm a smart guy too! Listen - I'm onto a sure thing here, and Eddie's a nice guy, so Eddie's going to let you in on it. See - I'm an expert in card counting. The Capo di tutti capi. What's card counting, you say? Well - to me, it's a career!

Seriously, card counting is way of improving the odds when you play blackjack. In blackjack, if there are plenty of high-value cards left in the shoe, then the odds are slanted in favor of the player - that's you!

Card counting helps you keep track of the number of high-value cards left. Say

you start off with a count of 0. Then the dealer leads with a Queen - that's a high card. That's one less available in the deck, so you reduce the count by one:

It's a queen ==> count - 1

But if it's a low card, like a 4, the count goes up by one:

It's a four ==> count + 1

High cards are 10s and the face cards (Jack, Queen, King). Low cards are 3s, 4s, 5s and 6s.

You keep doing this for every low card and every high card until the count gets real

high, then you lay on cash in your next bet and bad-a-bing! Soon you'll have more money than my third wife!

If you'd like to learn more, then enroll today in my Blackjack Correspondence School. Learn more about card counting as well as:

- * How to use the Kelly Criterion to maximize the value of your bet
- * How to avoid getting whacked by a pit boss
- * How to get cannoli stains off a silk suit
- * Things to wear with plaid

For more information, contact Cousin Vinny c/o the Blackjack Correspondence School

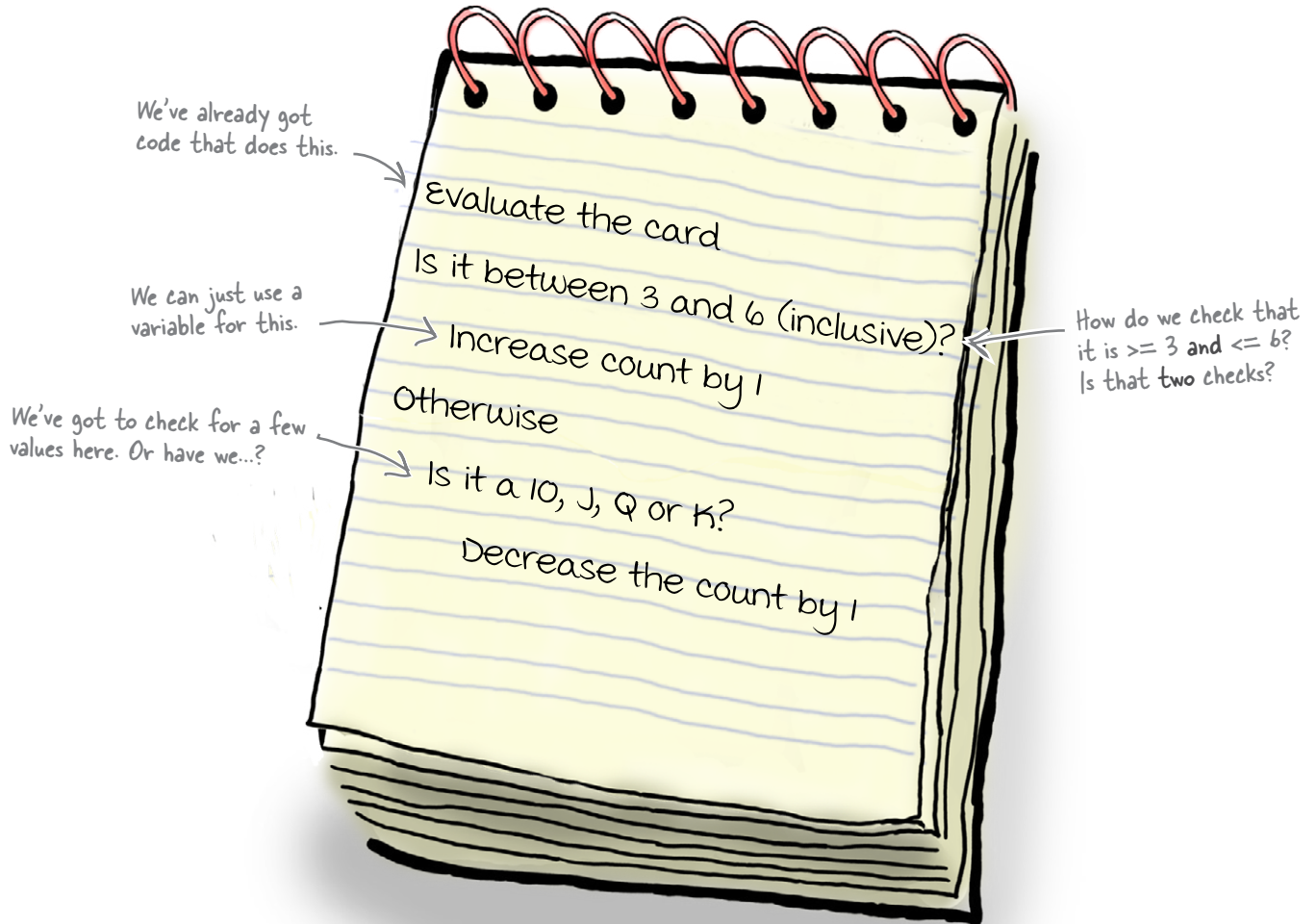
Wig Straps FOR MEN

STARBUZZ HOTEL

Buy an EDSEL

Card-counting? In C?

Card counting is a way to increase your chances of winning at blackjack. By keeping a running count as the cards are dealt, a player can work out the best time to place large bets and the best time to place small bets. Even though it's a powerful technique, it's really quite simple.



How difficult would this be to write in C? We've looked at how to make a single test, but the card-counting algorithm needs to check multiple conditions - we need to check that a number is ≥ 3 as well as checking that it's ≤ 6 .

We need a set of operations that will allow us to combine conditions together.

There's more to booleans than equals...

So far we've looked at `if` statements that check if a single condition is true, but what if we want to check several conditions? Or check if a single condition is *not* true?

`&&` checks two conditions are true

The *and* operator (`&&`) evaluates to true, only if **both** conditions given to it are true.

```
if ((dealer_up_card == 6) && (hand == 11))
    double_down();
```

Both of these conditions need to be true for this piece of code to run

The *and* operator is efficient: if the first condition is false then the computer won't bother evaluating the second condition. It knows that if the first condition is false, then the whole condition must be false.

`||` checks one of two conditions is true

The *or* operator (`||`) evaluates to true, if **either** condition given to it is true.

```
if (cupcakes_in_fridge || chips_on_table)
    eat_food();
```

Either can be true.

If the first condition is true, the computer won't bother evaluating the second condition. It knows that if the first condition is true, the *whole condition* must be true.

`!` flips the value of a condition

`!` is the *not* operator. It reverses the value of a condition.

```
if (!brad_on_phone)
    answer_phone();
```

! means "not"



Geek Bits

In C, boolean values are represented by numbers. To C, the number 0 is the value for false. But what's the value for true? Anything that is not equal to 0 is treated as true. So there is nothing wrong in writing C code like this:

```
int people_moshing = 34;
if (people_moshing)
    take_off_glasses();
```

In fact, C programs often use this as a short-hand way of checking if something is not zero



Exercise

You are going to modify the program so that it can be used for card-counting. It will need to display one message if the value of the card is from 3 to 6. It will need to display a different message if the card is a 10, Jack, Queen or a King.

```
int main()
{
    char card_name[3];
    puts("Enter the card_name: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    /* Check if the value is 3 to 6 */
    if .....
        puts("Count has gone up");
    /* Otherwise check if the card was 10, J, Q or K */
    else if .....
        puts("Count has gone down");
    return 0;
}
```



The Polite Guide to Standards

The ANSI C standard has no value for true and false. C programs treat the value 0 as false, and any other value as true. The C99 standard does allow you to use the words true and false in your programs - but the compiler treats them as the values "1" and "0" anyway.



Exercise Solution

You are going to modify the program so that it can be used for card-counting. It will need to display one message if the value of the card is from 3 to 6. It will need to display a different message if the card is a 10, Jack, Queen or a King.

```
int main()
{
    char card_name[3];
    puts("Enter the card_name: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    /* Check if the value is 3 to 6 */
    if ((val > 2) && (val < 7))
        puts("Count has gone up");
    /* Otherwise check if the card was 10, J, Q or K */
    else if (val == 10)
        puts("Count has gone down");
    return 0;
}
```

There are a few ways of writing this condition.

Did you spot that you just needed a single condition for this?

there are no Dumb Questions

Q: Why not just "|" and "&"?

A: You can use "&" and "|" if you want. The "&" and "|" operators will **always evaluate both conditions**, but "&&" and "||" can often skip the second condition.

Q: So why do the "&" and "|" operators exist?

A: Because they do more than simply evaluate logical conditions. They perform bitwise operations on the individual bits of a number.

Q: Huh? What do you mean?

A: Well $6 \& 4 == 4$, because if you checked which binary digits are common to 6 (110 in binary) and 4 (100 in binary), you get 4 (100).



TEST DRIVE

Let's see what happens when we compile and run the program now:

This line compiles
and runs the code. →

```
File Edit Window Help FiveOfSpades
> gcc cards.c -o cards && ./cards
Enter the card_name:
0
Count has gone down

> ./cards
Enter the card_name:
8

> ./cards
Enter the card_name:
3
Count has gone up

>
```

We run it a
few times to
check that the
different value
ranges work. →

The code works. By combining multiple conditions with a boolean operator we are able to check for a range of values rather than a single value. You now have the basic structure in place for a card counter.

The computer says the
card was low. The count
went up! Raise the bet!
Raise the bet!

← Stealthy communication device.





The Compiler Exposed

This week's interview:
What Has gcc Ever Done For Us?

Head First: May I begin by thanking you gcc for finding time in your very busy schedule to speak to us.

gcc: That's not a problem my dear boy. A pleasure to help.

Head First: gcc, you can speak many languages, is that true?

gcc: I am fluent in over six million forms of communication...

Head First: Really?

gcc: Just teasing. But I do speak many languages. C, obviously, but also C++ and Objective C. I can get by in Pascal, Fortran, PL/I and so forth. Oh - and I have a smattering of Go...

Head First: And on the hardware side, you can produce machine code for many, many platforms?

gcc: Virtually any processor. Generally when a hardware engineer creates a new type of processor one of the first things they want to do is get some form of me running on them.

Head First: How have you achieved some incredible flexibility?

gcc: My secret, I suppose, is that there are two sides to my personality. I have a front-end, a part of me that understands some type of source code

Head First: Written in a language such as C?

gcc: Exactly. My front end can convert that language into an intermediate code. All of my language front-ends produce the same sort of code.

Head First: You say there are two sides to your personality?

gcc: I also have a back-end - a system for converting that intermediate code into machine code that is understandable on many platforms. Add to that my knowledge of the particular executable file formats for just about every operating system you've ever heard of...

Head First: And yet, you are often described as a mere translator. Do you think that's fair? Surely that's not all you are?

gcc: Well of course I do a little more than simple translation. For example I can often spot errors in code.

Head First: Such as?

gcc: Well I can check the obvious things such as code like mis-spelling variable names. But I also look for subtler things, such as the redefinition of variables. Or I can warn the programmer if they choose to name variables after existing functions and so on.

Head First: So you check code quality as well then?

gcc: Oh yes. And not just quality - but also performance. If I discover a section of code inside a loop that could work easily as well outside a loop I can very quietly move it.

Head First: You do rather a lot?

gcc: I like to think I do. But in a quiet way.

Head First: gcc - thank you.



BE the Compiler

Each of the C files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile, and if not, why not. For extra bonus points, say what you think the output of each compiled file will be when run, and whether you think the code is working as intended.

A

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1)
        card = card - 1;
    if (card < 7)
        puts("Small card");
    else {
        puts("Ace!");
    }
    return 0;
}
```

B

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Small card");
    }
    else
        puts("Ace!");
    return 0;
}
```

C

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1)
        card = card - 1;
    if (card < 7)
        puts("Small card");
    else {
        puts("Ace!");
    }
    return 0;
}
```

D

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Small card");
    }
    else
        puts("Ace!");

    return 0;
}
```



BE the Compiler Solution

Each of the C files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile, and if not, why not. For extra bonus points, say what you think the output of each compiled file will be when run, and whether you think the code is working as intended.

A

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

```
int main()
```

```
{
    int card = 1;
    if (card > 1)
        card = card - 1;
    if (card < 7)
        puts("Small card");

    else {
        puts("Ace!");
    }
    return 0;
}
```

The code compiles. The program displays "Small card". But it doesn't work properly because the else is attached to the wrong if.

B

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

```
int main()
```

```
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Small card");
    }
    else
        puts("Ace!");
}
return 0;
}
```

Code compiles. The program displays nothing and is not really working properly because the else matches to the wrong if.

C

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

```
int main()
```

```
{
    int card = 1;
    if (card > 1)
        card = card - 1;
    if (card < 7)
        puts("Small card");
    else {
        puts("Ace!");
    }
    return 0;
}
```

The code compiles. The program displays "Ace!" and is properly written.

D

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

```
int main()
```

```
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Small card");
    }
    else
        puts("Ace!");

    return 0;
}
```

The code won't compile because the braces are not matched.

What's the code like now?

```
int main()
{
    char card_name[3];
    puts("Enter the card_name: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    /* Check if the value is 3 to 6 */
    if ((val > 2) && (val < 7))
        puts("Count has gone up");
    /* Otherwise check if the card was 10, J, Q or K */
    else if (val == 10)
        puts("Count has gone down");
    return 0;
}
```

Hmmm... is there something we can do with that sequence of if statements? They're all checking the same value - `card_name[0]` - and most of them are setting the `val` variable to 10. I wonder if there's a more efficient way of saying that in C?

C programs often need to check the same value several times and then perform very similar pieces of code for each case.

Now you can just use a sequence of `if` statements - and that will probably be just fine. But C gives you an alternative way of writing this kind of logic.

C can perform logical tests with the switch statement.



Pulling the ol' switcheroo

Sometimes when you're writing conditional logic, you need to check the value of the same variable over and over again. To prevent you have to write lots and lots of `if` statements, the C language gives you another option: the **switch** statement

The switch statement is kind of like an `if` statement, except it can test for multiple values of a *single variable*:

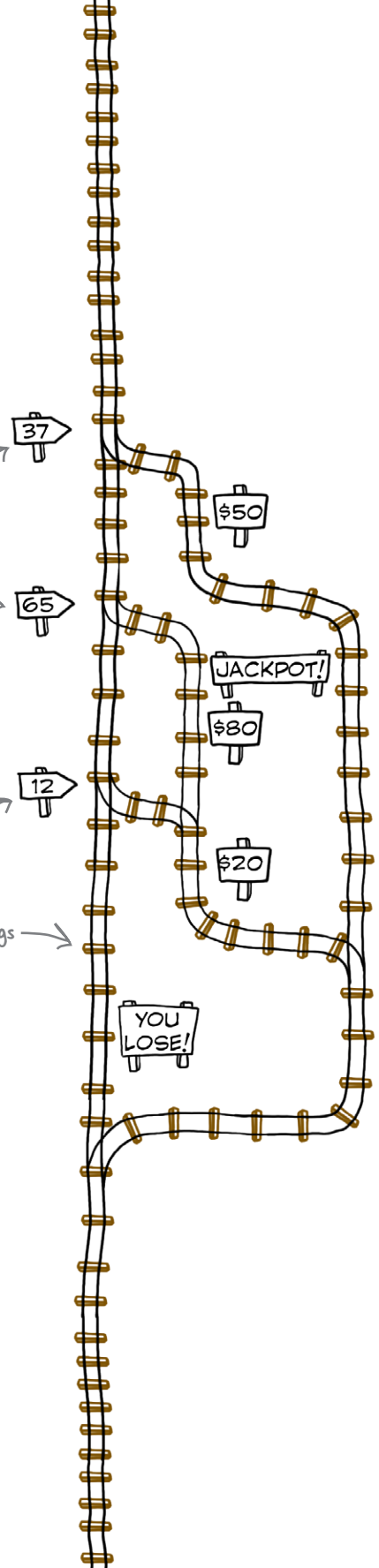
```
switch(train) {
case 37:
    winnings = winnings + 50;
    break;
case 65:
    puts("Jackpot!");
    winnings = winnings + 80;
case 12:
    winnings = winnings + 20;
    break;
default:
    winnings = 0;
}
```

← If the train == 37, add 50 to the winnings then skip to the end.

← If the train == 65 then add 80 to the winnings AND THEN also add 20 to the winnings, then skip to the end.

← If the train == 12 then just add 25 to the winnings.

← For any other value of train, set the winnings back to ZERO.



When the computer hits a `switch` statement, it checks the value it was given, and then looks for a matching case. When it finds one, it runs *all* of the code that follows it until it reaches a `break` statement. **The computer keeps going until it is told to break out of the switch statement.**



Sharpen your pencil Solution

Here's the code rewritten using a switch statement.

```
int val = 0;
if (card_name[0] == 'K') {
    val = 10;
} else if (card_name[0] == 'Q') {
    val = 10;
} else if (card_name[0] == 'J') {
    val = 10;
} else if (card_name[0] == 'A') {
    val = 11;
} else {
    val = atoi(card_name);
}
```

```
int val = 0;
switch(card_name[0]) {
    case 'K':
    case 'Q':
    case 'J':
        val = 10;
        break;
    case 'A':
        val = 11;
        break;
    default:
        val = atoi(card_name);
}
```



BULLET POINTS

- switch statements can replace a sequence of if statements
- switch statements check a single value
- The computer will start to run the code at the first matching case statement
- It will continue to run until it reaches a break or gets to the end of the switch statement
- Check that you've included breaks in the right place - otherwise your switches will be buggy.

there are no Dumb Questions

Q: Why would I use a switch statement instead of an if?

A: If you are performing multiple checks on the same variable then you might want to use a switch statement.

Q: What are the advantages of using a switch statement.

A: There are several. Firstly: clarity. It is clear that an entire block of code is processing a single variable. That's not so obvious if you just have a sequence of if statements. Secondly, you can use fall-through logic to re-use sections of code for different cases.

Q: Does the switch statement have to check a variable? Can't it check a value.

A: Yes it can. The switch statement will simply check that two values are equal.

Q: Can I check strings in a switch statement.

A: No - you can't use a switch statement to check a string of characters or any kind of array. The switch statement will only check a single value.

Sometimes once is not enough...

We've learned a lot about the C language, but there are still some things we need to cover. We've seen how to write programs for a lot of different situations, but there is one really fundamental thing that we haven't really look at yet. What if you want your program to do something *again and again and again*?

Using while loops in C

Loops are a special type of control statement. A control statement decides *if* a section of code will be run, but a loop statement decides *how many times* a piece of code will be run.

The most basic kind of loop in C is the while loop. A while loop runs code *over and over and over* so long as some condition remains true.

```

while (<some condition>) {
    ... /* Do something here */
}

```

Checks the condition before running the body

The body is between the braces.

... /* Do something here */ ← If you only have one line in the body, you don't need the braces.

When it gets to the end of the body, the computer checks if the loop-condition is still true. If it is, the body code runs again.

But what if you need to stop looping somewhere inside the loop body?

```

while (more_balls)
    keep_juggling();

```



You use break to breakout...

A while loop checks the condition *before* it runs the loop body. But what if somewhere inside the code you decide that you don't need to run the loop any more?

Fortunately there's a way of skipping out of the loop immediately - we can use the `break` statement.

```
while(feeling_hungry) {
    eat_cake();
    if (feeling_queasy) {
        /* Break out of the while loop */
        break;
    }
    drink_coffee();
}
```

"break" skips out of the loop immediately.

A `break` statement will break you straight out of the current loop, skipping whatever follows it in the loop body. `break`s can be useful as they're sometimes the simplest and best way to end a loop. But you might want to avoid using too many because they can make the code a little harder to read.

...and continue to continue

If you want to skip the rest of the loop body and go back to the start of the loop, then the `continue` statement is your friend:

```
while(feeling_hungry) {
    if (not_lunch_yet) {
        /* Go back to the loop condition */
        continue;
    }
    eat_cake();
}
```

"continue" takes you back to the start of the loop.

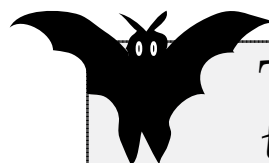
Let's test out your new-found loop-mojo.



Watch it!

The break statement is used to break out of loops and also inside switch statements.

Be careful that you know what you're breaking out of when you're breaking out.



Tales from the Crypt

breaks don't break if statements

On January 15th, 1990 AT&T's long distance telephone system crashed and 60,000 people lost their phone service. The cause? A developer working on the C code used in the exchanges tried to use a `break` to break out of an `if` statement. But `break`s don't break out of `ifs`. Instead the program skipped an entire section of code and introduced a bug that interrupted 70 million phone calls over 9 hours...



Mixed Messages

A short C program is listed below. One block of the program is missing. Your challenge is to **match the candidate block of code** (on the left), **with the output** that you'd see if the block were inserted. Not all of the lines of output will be used, and some of the lines of output might be used more than once. Draw lines connecting the candidate blocks of code with their matching command-line output.

```
#include <stdio.h>

int main()
{
    int x = 0;
    int y = 0;
    while (x < 5) {
        
        printf("%i%i ", x, y);
        x = x + 1;
    }
    return 0;
}
```

Candidate code goes here.

Candidates:

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if (y > 4)
    y = y - 1;
```

```
x = x + 1;
y = y + x;
```

```
if (y < 5) {
    x = x + 1;
    if (y < 3)
        x = x - 1;
}
y = y + 2;
```

Possible output:

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

Match each candidate with one of the possible outputs.



Mixed Messages Solution

A short C program is listed below. One block of the program is missing. Your challenge is to **match the candidate block of code** (on the left), **with the output** that you'd see if the block were inserted. Not all of the lines of output will be used, and some of the lines of output might be used more than once. Draw lines connecting the candidate blocks of code with their matching command-line output.

```

#include <stdio.h>

int main()
{
    int x = 0;
    int y = 0;
    while (x < 5) {
        
        printf("%i%i ", x, y);
        x = x + 1;
    }
    return 0;
}
    
```

Candidate code goes here.

Candidates:	Possible output:
<code>y = x - y;</code>	22 46
<code>y = y + x;</code>	11 34 59
<code>y = y + 2;</code> <code>if (y > 4)</code> <code> y = y - 1;</code>	02 14 26 38
<code>x = x + 1;</code> <code>y = y + x;</code>	02 14 36 48
<code>if (y < 5) {</code> <code> x = x + 1;</code> <code> if (y < 3)</code> <code> x = x - 1;</code> <code>}</code> <code>y = y + 2;</code>	00 11 21 32 42
	11 21 32 42 53
	00 11 23 36 410
	02 14 25 36 47



Exercise

Now that you know how to create `while` loops, modify the program to keep a running count of the card game. Display the count after each card and end the program if the player types 'Q'. Display an error message if the player enters types a bad card value like 11 or 24.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    int count = 0;
    while (.....) {
        puts("Enter the card_name: ");
        scanf("%2s", card_name);
        int val = 0;
        switch(card_name[0]) {
            case 'K':
            case 'Q':
            case 'J':
                val = 10;
                break;
            case 'A':
                val = 11;
                break;
            case 'X':
                .....
            default:
                val = atoi(card_name);
                .....
                .....
                .....
                .....
        }
        if ((val > 2) && (val < 7)) {
            count++;
        } else if ((val > 9) && (val < 11)) {
            count--;
        }
        printf("Current count: %i\n", count);
    }
    return 0;
}
```

You need to stop if they enter Q.

What will you do here?

You need display an error if the val is not in the range 1 to 10. You should also skip the rest of the loop body and try again.



Exercise Solution

Now that you know how to create `while` loops, modify the program to keep a running count of the card game. Display the count after each card and end the program if the player types 'Q'. Display an error message if the player enters types a bad card value like 11 or 24.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    int count = 0;
    while ( card_name[0] != 'X' ) {
        puts("Enter the card_name: ");
        scanf("%2s", card_name);
        int val = 0;
        switch(card_name[0]) {
            case 'K':
            case 'Q':
            case 'J':
                val = 10;
                break;
            case 'A':
                val = 11;
                break;
            case 'X':
                continue;
            default:
                val = atoi(card_name);
                if ((val < 1) || (val > 10)) {
                    puts("I don't understand that value!");
                    continue;
                }
        }
        if ((val > 2) && (val < 7)) {
            count++;
        } else if ((val > 9) && (val < 11)) {
            count--;
        }
        printf("Current count: %i\n", count);
    }
    return 0;
}
```

We need to check if the first character was a 'Q'.

break wouldn't break us out of the loop, because we're inside a switch statement. We need a `continue` to go back and check the loop condition again.

This is just one way of writing this condition.

We need another `continue` here because we want to keep looping.



TEST DRIVE

Now the card counting program is finished, it's time to take it for a spin. What do you think? Will it work?

Remember - you don't need
"/." if you're on Windows.

This will compile
and run the
program.

```
File Edit Window Help GoneLoopy
> gcc card_counter.c -o card_counter && ./card_counter
Enter the card_name:
4
Current count: 1
Enter the card_name:
K
Current count: 0
Enter the card_name:
3
Current count: 1
Enter the card_name:
5
Current count: 2
Enter the card_name:
23
I don't understand that value!
Enter the card_name:
6
Current count: 3
Enter the card_name:
5
Current count: 4
Enter the card_name:
3
Current count: 5
Enter the card_name:
X
```

We now check
if it looks
like a correct
card value.

The count is
increasing!

By betting big when
the count was high - I
made a fortune!

The card counting program works!

You've completed your first C program. By using the power of C statements, loops and conditions you've created a full functioning card counter.

Great job!

Disclaimer: Using a computer for card-counting is illegal in a lot of states, and those casino guys can get kinda gnarly. So don't do it, OK?





Your C Toolbox

You've got Chapter 1 under your belt and now you've added C basics to your tool box.

For a complete list of tooltips in the book, see Appendix X.

Simple statements are commands

Block statements are surrounded by { and }

switch statements efficiently check for multiple values of a variable

#include includes external code for things like input and output

if statements run code if something is true

Every program needs a main function

You need to compile your C program before you run it

You can combine conditions together with `&&` and `||`

You can use the `if` operator on the command line to only run your program if it compiles

gcc is the most popular C compiler

Your source files should have a name ending in ".c"

`-o` specifies the output file

2 memory and pointers

What are you pointing at?



If want to kick butt with C, you need to understand how C handles memory.

The C language gives you a lot more *control* over how your program uses the **computer's memory**. In this chapter, you'll strip back the covers and see exactly what happens when you **read and write variables**. You'll learn **how arrays work**, how to avoid some **nasty memory SNAFUs**, and most of all, you'll see how **mastering pointers and memory addressing** is key to becoming a kick-ass C programmer.

The Head First Lounge has a new champagne bar

Things have always been pretty swinging down in the Head First Lounge. But the guys are cracking open a whole new crate of fun with the Head First Lounge Champagne Bar.

Thing is, this has given the guys a little problem...

Sometimes we kind of lose track of the champagne we have in stock. It's hard to keep count when you're tending bar and dancing the merengue, Baby!

In order to prevent the guys running out of stock, they need a stock-taking program. It's a little like some of the code you've written so far. Here's what the program needs to do:

We'll start with 30 bottles
- that's about 180 glasses
while there's still some fizz:
Display the current stock
Enter the number of glasses
ordered
Adjust the stock
Then just go round again





Code Magnets

The guys started to write the code on the fridge door. Unfortunately they discovered some tequila inside the fridge, the magnets got a little mixed up. Now the program no longer works. See if you can reassemble the magnets and get the program working again.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int stock = 180;
    char order_string[3];
    int order;
    while(stock > 0) {

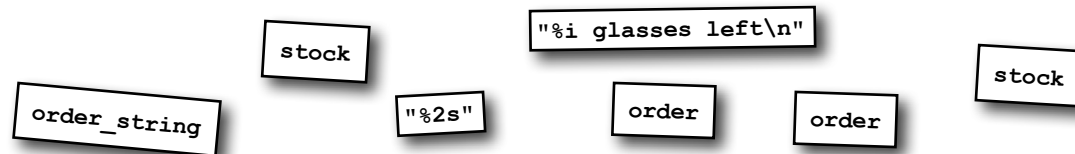
        printf( ..... , ..... );

        scanf( ..... , ..... );

        order = atoi( ..... );

        stock = ..... - ..... ;

        printf("You ordered %i glasses\n", ..... );
    }
    puts("We're out of stock, baby!");
    return 0;
}
```





Code Magnets Solution

The guys started to write the code on the fridge door. Unfortunately they discovered some tequila inside the fridge, the magnets got a little mixed up. Now the program no longer works. See if you can reassemble the magnets and get the program working again.

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    int stock = 180;
    char order_string[3];
    int order;
    while(stock > 0) {
        printf( "%i glasses left\n", stock );
        scanf( "%2s", order_string );
        order = atoi( order_string );
        stock = stock - order;
        printf("You ordered %i glasses\n", order );
    }
    puts("We're out of stock, baby!");
    return 0;
}

```

We begin with 180 glasses in stock.

This string will allow use to enter a number of up to 2 digits.

Display the remaining glasses in stock.

Read a 2-character string for the order.

Convert that string into a number called 'order'.

Subtract order from the stock.

Once we run out of champagne, break the bad news.



— TEST DRIVE —

OK - it's time to compile the code and take it for a test drive.

We begin by compiling the code. →

This is the initial stock. →

Every time we enter an order... →

...the stock goes down... →

...until... →

...chug, chug, chug... →

Bummer. →

```

File Edit Window Help Hic
> gcc champagne.c -o champagne && ./champagne
180 glasses left
20
You ordered 20 glasses
160 glasses left
10
You ordered 10 glasses
150 glasses left
60
You ordered 60 glasses
90 glasses left
90
You ordered 90 glasses
We're out of stock, baby!
>

```

← This will run the program after compiling it.

Great news!

It looks like the program works. It takes the champagne orders one at a time, and once an order uses up all of the remaining glasses, it tells the bar tender.

Wow, this will be really useful. We'll be able to order up some more from the cellar as soon as we run out. Just in time for the Friends of Italian Opera party tonight!

Let's go to the party!

Not a moment too soon! Some of these Italian Opera buffs can turn kinda nasty if they can't get a drink.



Then someone ordered 100 glasses...

Everything was going well until...

Suddenly there was no more champagne! Next thing I know, this big opera buff gets angry... it all goes dark... and I wake up in the infirmary.



It looks like there's a bug in the code. Somewhere. Speaking to the guys, and assembling the events from the police reports, it looks like the program did something like this:

```
File Edit Window Help Hic
> ./champagne
180 glasses left
20
You ordered 20 glasses
160 glasses left
60
You ordered 60 glasses
100 glasses left
100
You ordered 10 glasses
90 glasses left
You ordered 0 glasses
90 glasses left
40
```

The program started OK. →

The first few orders went fine. →

There were 100 glasses left when someone ordered 100. →

But why did the order only go down by 10? →

And what's this extra order for "0"?? No one ordered 0...!?!? →

This was the big guy's order - but by this time all the drink had gone. The guys behind the bar entered a world of pain... →

Something very strange was happening in the program. The stock level had dropped to 100 glasses at just the moment that someone ordered 100 glasses. That *should* have reduced the stock level to 0... but instead it only reduced it by 10.



Can you figure out what happened? It looks like the problems started when the program thought an order for 100 was an order for only 10 glasses. What do **you** think happened?

Let's see what's happening

Let's take a little closer look at that `scanf()` line:

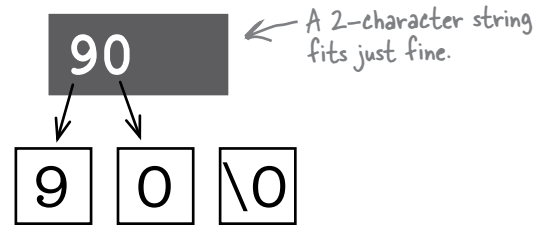
```
char order_string[3];
...
scanf("%2s", order_string);
```

We're going to enter a 2-character string into `order_string`.

`order_string` is 2 characters + the string terminator.

`scanf()` reads the characters that the user enters, and then stores them into a the 3-character string called `order_string`. Remember - strings always need an extra *termination character*, so a 3-character array is used to store **2 characters**.

To make sure that the user doesn't enter more characters than the array can hold, we pass the `scanf()` function a format string with the value `%2s` - which means *only accept two characters*.



But what if we enter more than two characters?

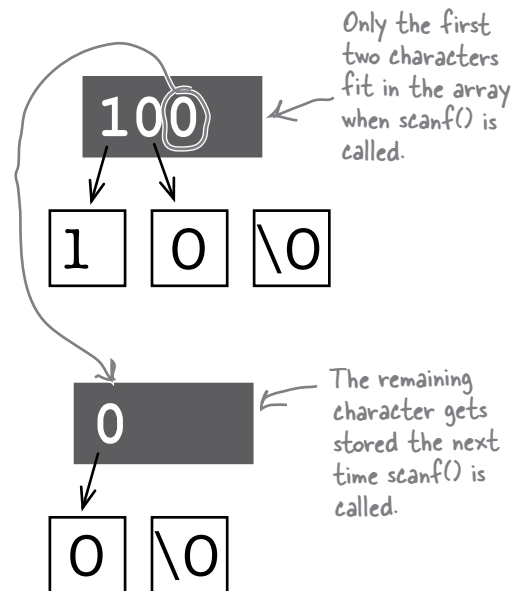
When someone enters a value with more than two characters such as 100, we have a problem. `scanf()` has been told to only accept a maximum of two characters. **It only puts the first two characters in the `order_string` array.**

And that's the problem

Even though the user entered "100", the `order_string` array was set to "10". But what did `scanf()` do with the final "0" in "100"? Well - it saved it up until the next time it was called and then it put it into the `order_string` array.

That's why the stock was adjusted by 10 instead of 100, and why there was that mysterious order for "0" - even though nothing was typed in the second time.

So the problem is caused because the `order_string` array only allows the user to enter up to 2-characters in it. Question is - how do we fix it?



Cubicle conversation



I really don't see what the problem is.

Frank: Well the string is not long enough to accept a number over 99.

Joe: No - what I mean is, I don't see *why* that is such an issue. If they get an order for more than 99 glasses, can't they just split it?

Jill: You mean if the is 140, then have two orders of 80 and 60?

Joe: Yeah - exactly.

Frank: That's changing the way the program works. I think the users will want to enter the real order number. But it's no problem. We just need to make the array one character longer. That means they can enter values up to 999.

Jill: Sure - that will fix it in this case. But what about generally? What if some other program needs people to enter numbers with 6 digits, 7 digits or whatever.

Frank: That's a good point. It would be nice if there was a way to say "Just get the user to enter an integer"

Joe: Well we could if we knew what values an int variable accepted. That's where the value will be stored after all.

Jill: Oh yeah - in that order variable.

Frank: Well to do that we'd just need to know how big a number we can store in an int variable.

Jill: That shouldn't be too hard to figure out. So we *could* fix the program for now by just making the `order_string` array one character longer, like this:

```
char order_string[4];  
...  
scanf("%3i", order_string);
```

But if we want a fix that will work for **all** programs, we could size the string so that it can cope with *any* number that will fit into an int variable. We could do that if we knew how big int numbers can be.

So how big is an int?

Remember we said that C was a little more *low-level* than most other languages? When you program in C, you have to think a little more about the hardware you are using than if you were using a language, say, Java. And the hardware is really important because **the maximum size of an integer depends upon the machine you are using.**

On some machines, an integer is stored as **4 bytes**. That will let you store numbers between the values:

10000000 00000000 00000000 00000000 ← That's -2147483648.
 and There are 4 bytes - which is 32 bits. These numbers are in binary.
 01111111 11111111 11111111 11111111 ← That's +2147483647.

The problem is that if your machine uses 16-bits for an `int` or 64-bits or even 128-bits, then you might not know exactly what range of numbers you can use.

Fortunately C gives you a little help. First of all, you can use the `sizeof()` operator to check how many bytes an integer takes up. Secondly, you can look at special values called `INT_MIN` and `INT_MAX` to find out what range of numbers you can use:

```
#include <stdio.h>
#include <limits.h> ← You need limits.h to get the INT_MIN and INT_MAX values

int main()
{
    printf
```

Are you kidding me? We just want someone to enter a number from the command line and then store it in an int. I don't care how big the string needs to be. Just give me an int!

That's a good point - we *actually* want an int value and we shouldn't have to get too hung up on the number of characters the user types in.

Fortunately the `scanf()` function has a way of letting the user enter `ints` directly - without needing to say how many characters it might take.



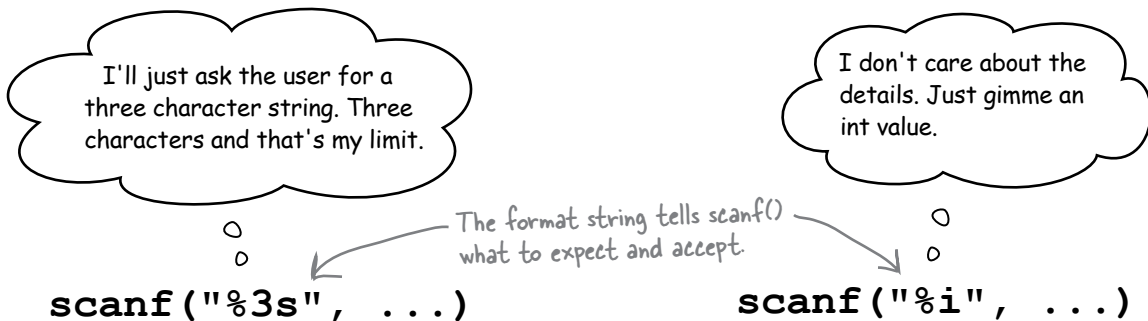
Don't worry if you don't know a lot about binary numbers.

You just need to know that the more bits you can use to store a number, then the greater the range of numbers you can store.



scanf() lets you enter numbers directly

So far we've only used `scanf()` to enter *string values*. But it can do so much more. Remember that the first value we pass to `scanf()` is a **format string** that says what kind of data the user will be entering. By changing the format string, we can tell `scanf()` to accept some other data type, like an *integer*.

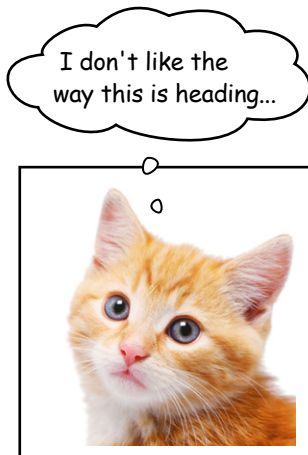


That means that all of that *complex* information about how *long* integers are, how many characters the user might enter on the command and all that other stuff is dealt with by `scanf()`.

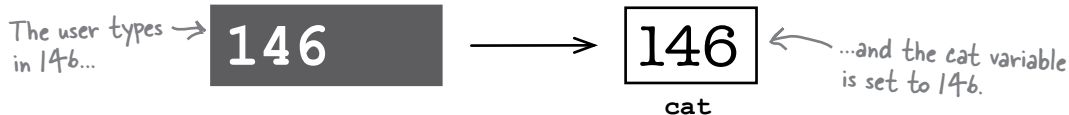
So let's say we want to enter a value into a variable called `cat`:

For an integer you need to put a "`%i`" at the front of the variable name.

```
int cat;
puts("Enter the number of ways to skin a cat");
scanf("%i", &cat);
printf("Number of ways = %i", cat);
```



Once the `scanf()` function has been called, the `cat` variable will set to the number the user entered. You can see that this is a lot simpler than storing things in a string first. We don't need to know how many characters to deal with. We don't have to check ranges, or think about binary numbers or worry about what machine we are running on, or any of that stuff.





Exercise

Now that you know how to enter numbers directly into `int` variables, take another look at the program and see if you can fix it to avoid the ordering problem.

```
#include <stdio.h> ← Use a pencil to modify the code.
#include <stdlib.h>
int main()
{
    int stock = 180;
    char order_string[3];
    int order;
    while(stock > 0) {
        printf("%i glasses left\n", stock);
        scanf("%2s", order_string);
        order = atoi(order_string);
        stock = stock - order;
        printf("You ordered %i glasses\n", order);
    }
    puts("We're out of stock, baby!");
    return 0;
}
```

So - do you think you
could fix my program now?
It's another Opera night
tonight and **I'm** on the bar...





Exercise Solution

Now that you know how to enter numbers directly into `int` variables, take another look at the program and see if you can fix it to avoid the ordering problem.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int stock = 180;
    char order_string[31]; ← We no longer need the order_string.
    int order;
    while(stock > 0) {
        printf("%i glasses left\n", stock);
        scanf("%2s", order_string); ← These two lines read the string,
        order = atoi(order_string); ← so they can go.
        scanf("%i", &order); ← Now we just need this line to read the
        stock = stock - order; ← user input direct into the order variable.
        printf("You ordered %i glasses\n", order);
    }
    puts("We're out of stock, baby!");
    return 0;
}
```

there are no Dumb Questions

Q: So integers are different size on different machines? Why's that?

A: `ints` are used for quick general purpose calculations. In order to make the calculations run as quickly as possible, C uses the same kinds of numbers that the processor uses.

Q: Does that really make a big difference?

A: It can. If numbers are the same size as the registers in a central processing unit, then it might be possible to avoid some slow processes, like looking up values from memory.

Q: What's the down side to have `ints` of different sizes.

A: Portability can be a problem.

Q: So what I really want to use an integer of a particular size. Can I do that?

A: Yes. If you include the `<inttypes.h>` file, then you will have types like `int8_t` for 8 bits and `int16_t` for 16 bits and so on.



TEST DRIVE

So let's take the fixed code out on the road. And we already have a great test case. If we run the exact same orders, we can see if we've fixed the bug.

When we entered 100 before, the code thought we'd entered 10. →
This time it knows it's 100. →

```
File Edit Window Help Hic
> ./champagne
180 glasses left
20
You ordered 20 glasses
160 glasses left
60
You ordered 60 glasses
100 glasses left
100
You ordered 100 glasses
We're out of stock, baby!
```

It works!

By using the extra formats `scanf()` supports we are able to accept different data types straight from the command line. Not only is our code shorter and easier to read and more understandable...

Hey, not so fast! Understandable - you say? **Understandable??**
Oh sure the code's shorter and it's nice that I don't have to get bogged down in a lot of the details. But - you're missing something... I don't **understand** why we have to put a "&" at the front of order?!?! Why do we do that? Did I miss a memo or something?

There's a very good reason why we need to use the & symbol when we use scanf for something other than a string.

It's because `scanf()` only accepts **pointers** to data.

Pointers can be one of the most hardest things to learn when you first start using C. To see what a pointer *is* and exactly how they work, we need to dig a little deeper into how `scanf` works for numbers and arrays.

Use the "&" operator and the "%i" format to enter ints directly with `scanf`.





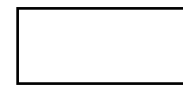
Using scanf For Numbers Up Close

To see how `scanf()` can read integers from the command line, let's look at the following piece of code in more detail:

```
int x;
scanf("%i", &x);
printf("You entered the value %i\n", x);
```

1 Create the storage.

The first thing the code needs to do is create the `x` variable. What is a variable? It's really just a storage location. Let's imagine the machine allocates memory address 4,000,000 for the `x` variable. Whenever the program needs to know the value of `x`, it will just look at the contents of this memory address.



This will be where we store the `x` variable.

4,000,000

It's at address 4,000,000.

2 scanf looks at the format string.

When the `scanf()` function starts to run it will look at the *format string* it was given and decide how it needs to treat the input from the user.

This is the format string.

```
scanf("%i", ...)
```

`%i` means the input will be treated as an int.



Hmm.... Looks like the user is going to enter the digits of an integer.

3 scanf finds out where to store the result.

But what will happen when the user enters a number? The `scanf()` function will need to store the resulting `int` somewhere in memory. The second parameter - `&x` - is the **value of the address of `x`**. The `&`-operator finds the memory used by a variable - so `&x` has the value **4,000,000** because that is the memory address where `x` is stored.

So when I find out the number I need to store it at location 4,000,000.



`&x` == the address where `x` is stored.

4 The user types in a number at the command line.

The `scanf()` function will now wait until the user has entered some characters at the command line and hit [RETURN].

The user types in some characters at the command line.

1234



5 scanf stores the number in memory.

Once the `scanf()` function has the characters, it then needs to convert them into an integer and then store them in memory:

These are the characters that were entered by the user.

1, 2, 3, 4

The computer converts them to 1234.

1234

1234

4,000,000

The number 1234 is stored at memory location 4,000,000.

6 printf displays the result.

The `printf()` function then displays the result. `printf` is passed two values:

- * The format string "You entered the value %i\n"
- * The *value* of `x`

Remember - when the computer sees the variable `x` it will just look at the contents of memory location 4,000,000.

OK - so I have to print this formatted string with the value.... 1234.



File Edit Window Help

You entered the value 1234

The variable `x` now has the value 1234.



Geek Bits

The *address* of a variable is also known as a **pointer** to the variable. That's because it *points* to the variable in memory in the same way that your home address *points* to you. Pointers are used a lot of C programs and they can be very confusing when you first start to program in C.

But how come we didn't have to use the &-operator when we asked `scanf()` to read a string for us?



Using scanf For Text Up Close

To see how `scanf()` works with text, let's look at this example code:

```
char username[9];
scanf("%8s", username);
printf("Your username is '%s'", username);
```

1 Allocate space for a new array.

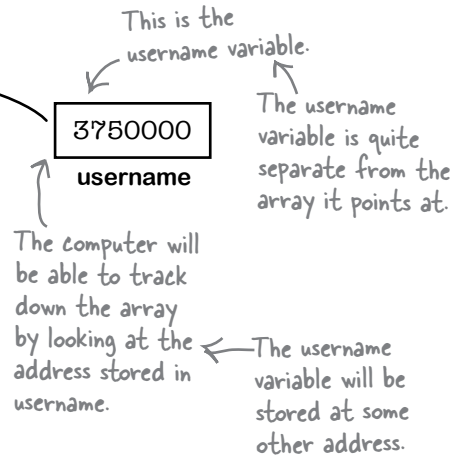
The computer will begin by allocating a section of memory for the array. Let's say it starts to allocate space from address **3,750,000**. It can't just allocate a single memory location because it needs to leave space for 9 characters in the array, so the computer will actually memory from location 3,750,000 to 3,750,008.



2 Create a new variable called "username".

But the computer isn't done with that first line of code yet. It's allocated space for the array, but it hasn't created the variable yet. Now you might think the array is the variable, but it's not. The `username` variable will actually be created at an entirely separate location.

But if the array variable is separate from the array itself, then what will value of the variable be? Well - the `username` variable will actually contain **the address of the first element of the array**



3 The scanf() function starts running.

The `scanf()` function is now called. It's passed two things:



The format string will tell the computer that it will need to read a string of up to 8 characters. And the `username` variable will have the value **3,750,000**. This is the address where `scanf()` will start to store the string

We're passing `username` and not `&username` because the `username` variable **already contains an address**.



So I need to read a string and store it starting at location 3,750,000.

4 Read the characters from the command line.

The `scanf()` function will now wait for the user to enter the some characters at the keyboard and hit [RETURN].

The user types in characters at the command line.

Geronimo



5 scanf stores the number in memory.

Now that `scanf` has read the characters from the command line it will store them away in memory, beginning at location 3,750,000.

G	e	r	o	n	i	m	o	\0
3,750,000	3,750,001	3,750,002	3,750,003	3,750,004	3,750,005	3,750,006	3,750,007	3,750,008



6 printf displays the result.

Now the `printf` function is called. It's also passed two things:

The format string.

The address of the array.

```
printf("Your comment was '%s'", comment);
```

Because `printf` knows that it is going to print a string, it doesn't treat the second parameter as a simple number. Instead it will treat the second parameter as **starting memory address**.



I'll print the contents of 3,750,000 - that's a 'G'. At 3,750,001 there's an 'e'....

File Edit Window Help

Your username is Geronimo



Don't worry if pointers seem a little strange when you first meet them.

So even though `scanf()` looks different for numbers and text, in both cases the second parameter needs to be a pointer - that is, it needs to be an address.

Pointers are one of the trickiest things in C, but there's no rush. Spend a bit of time going through these past few pages and make sure you're comfortable with pointers before you continue



Using scanf For Text Way Up Close

When you're reading text with the `scanf()` function, there's one thing you need to be careful about: **by default it stops at whitespace.**

Let's say someone changed the previous example code to allow for longer strings to be entered:

This will allow for a longer string.

```
scanf("%50s", username);
```

Great - now the user can enter their full name.

The code will work great... right up until someone enters a string that contains a space.

The user types in characters at the command line, including a space.

Geronimo Schwartz



If *that* happens the `scanf()` function will do something that's kinda annoying:

So that's a G, e, r, o, n, i, m, o... and... Oh - a space. The piece of text must have ended.

That's right - even though we told the computer to accept up to 50 characters, it will stop as soon as it hits some whitespace.

Now the good news is that `scanf` doesn't throw away the rest of the text that it read from the keyboard. It **buffers it**. That means if you call the `scanf()` function a *second time*, it won't need to ask the user for more text, it will simply carry on reading where it left off.



```
scanf("%50s", username);
```

← This will read the text "Geronimo".

```
scanf("%50s", username2);
```

← If we call it a second time, username2 will be set to "Schwartz".

You can even shorten the code to enter *several strings at once* by using a **pattern** like this:

```
scanf("%50s %50s", first_name, last_name);
printf("First name: %s\nLast name: %s\n", first_name, last_name);
```

Both `scanf()` and `printf()` allow you to pass as many variables as you like to them.

The user can now enter two strings at once...
...and `scanf` stores them separately.

```
File Edit Window Help
> John Smith
First name: John
Last name: Smith
```

OK - but what if you really really just want to enter a whole chunk of text, spaces and all?

Well you *could* use `scanf`, using a **regular expression**. You can use a piece of code like this:

```
scanf("%79[^\n]", line);
```

← This means "Read everything that's not a NEWLINE"

This code will read all of the characters from the keyboard buffer until it reaches a NEWLINE character. It works because `%79[^\n]` literally means **read up to 79 characters, so long as they're not NEWLINES**.

But if this seems kinda complicated - you're right.

The `scanf()` function was designed for *structured text*. That's the kind of text that's used for computer languages or data files. The kind of text where you know exactly what kind of strings you are going to find. But it's not so good for *unstructured text*. That's why the `fgets()` function was invented. `fgets()` asks the user for a complete line of text and it *reads all of it*. That's exactly the sort of thing you need for free-format text:

We'll create an array for our text.

The first value in `fgets` is the name of the array.

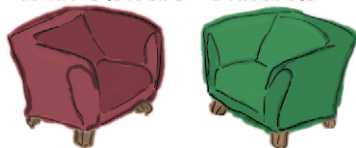
```
char line[80];
fgets(line, 80, stdin);
printf("Your quote was: %s", line);
```

The second value in `fgets` is the size of the array - if our array is 80 chars, then this needs to be exactly 80.

`stdin` just means "Read it from the keyboard" rather than a file.

For free text input use `fgets()`

Fireside Chats



Tonight's talk: **Array and Pointer talk through their differences**

Array:

Ah - pointer! Come in! Sit ye down. Sit ye down.

All the better for seeing you, old friend. Tell me, how are things?

Really? How frightful.

What?

Well yes. I suppose I am the real reason that you exist.

Nothing, nothing. I don't mean to offend. It's just that pointers really only exist to provide access to arrays don't they.

Really?

Pointer:

Array, my dear and venerable colleague. How are you?

Well you know. Mustn't grumble. Oh - one thing. I was in a program the other day and I was mistaken for you.

I know. Many coders confuse us I think because... well. How can I put this delicately? You... you don't really have a name.

A name. I mean if anyone ever needs to get in touch with you, they need go through me.

I'm sorry my dear and very *old* friend, I'm afraid I don't quite follow.

Well - no. That's not true. Actually I have several other friends.

Yes. Yes - I often provide access to other data types. I am frequently called upon for functions such as `scanf` that need to update data that is passed to them.

Array:

True, true. I was forgetting. But you are... how can I put it... just an *intermediary* aren't you?

Well I am the data that people are interested in.

Please - I didn't mean to offend.

I'm so sorry. I can't quite catch your drift. As so many people have said - you are **awfully confusing**.

Yeah - you and whose array? You're just a memory address! You don't have any proper content!

You might talk big but you're only a word long.

You cheeky mare! Just wait till I...

Pointer:

"Intermediary" you say, my very eminent and *ancient* compatriot. How so?

And who, pray, am I? The cat's mother?

Whenever anyone creates an array in memory they need *me* to get access to it. Without me, you'd be useless. Cocker.

You're looking for a punch up the bracket.

Well... I know where you live, mate.

Fatso!

**At this point we draw a close to today's discussion
between such eminent figures.**

there are no Dumb Questions

Q: So a pointer variable contains a number?

A: Yes - it contains the address of a memory location.

Q: So if I look at it's value I'll just see an int or something?

A: Well not exactly. C gives a pointer variable a particular type, like "pointer to char". That means if you try to read the numeric address directly the compiler will give you a warning. You code shouldn't care what the actual address is, so long as your pointer is pointing at the right data.

Q: If "quote" is a variable pointing to the start of an array, I can see that the computer can use it to find quote[0] - the first element. But how does it find quote[1], and quote[2] and so on?

A: The computer knows what type of data a pointer points to, so it knows how many bytes of memory something like a char occupies. To read the value of quote[1] it will just read the contents of (quote + 1).

Q: So the computer records the start of an array in a variable. Where does it record the length?

A: It doesn't.

Q: So how do I know when I get to the end of the array?

A: You either have to keep your own record of the length somewhere, or you need to mark the last item in the array with some special value - just like strings end with a special '\0' value.

Q: Can I print out the actual address of a variable?

A: Yes. If you have an int called 'x', then use `printf("%p", &x);`

Q: What does the "%p" format string mean?

A: It means that the value you are printing is a pointer-type.

Q: If pointers are just addresses, does that mean they are ints or something?

A: No. A pointer is a distinct type. In fact there's a different pointer for different data types. So there are char-pointers that contain the addresses of chars and int-pointers that point at ints and so on.



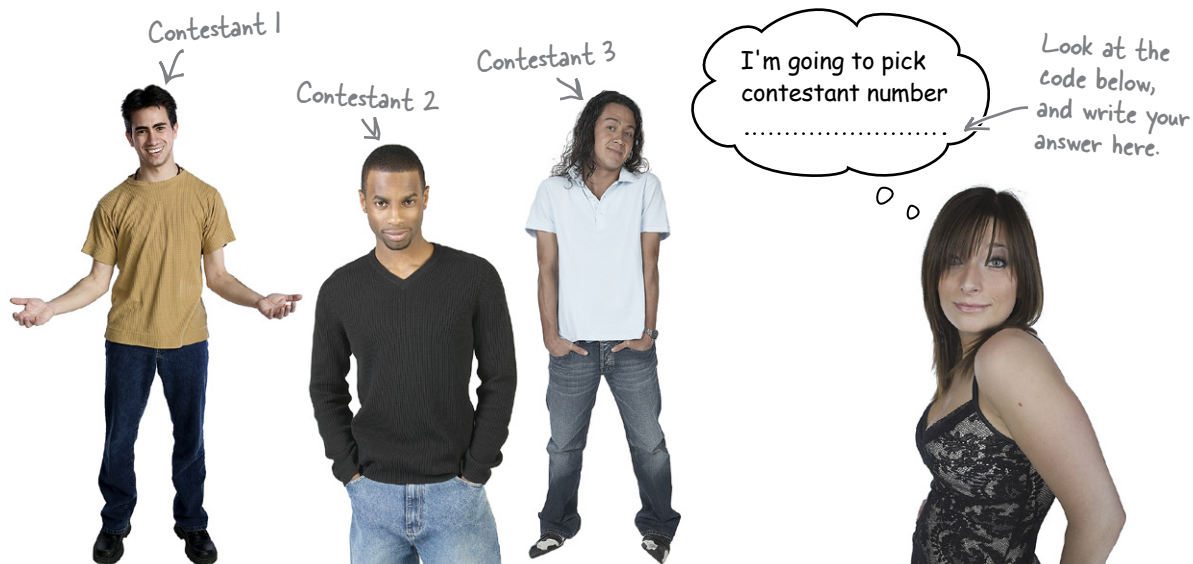
BULLET POINTS

- `scanf()` lets you enter strings with the "%s" format.
- `scanf()` lets you enter int-values with the "%i" format.
- The `&`-operator returns the address of a variable, like `&order`.
- `char q[20]` does two things - it *creates an array* and it *creates a pointer variable* containing the address of the array.
- The address of a piece of data is called a **pointer** to the data.
- A variable that contains a pointer is called a **pointer variable**.
- The name of an array is actually just a pointer variable to the first item in the array.
- If you are reading an int called `x`, you need to use `scanf("%i", &x)`.
- If you are reading a string called `q`, you need to use `scanf("%s", q)`.
- Both `&x` and `q` are addresses.

THE MATING GAME

We have a classic trio of bachelors ready to play The Mating Game today.

Tonight's lucky lady is going to pick one of these fine contestants. Who will she choose?



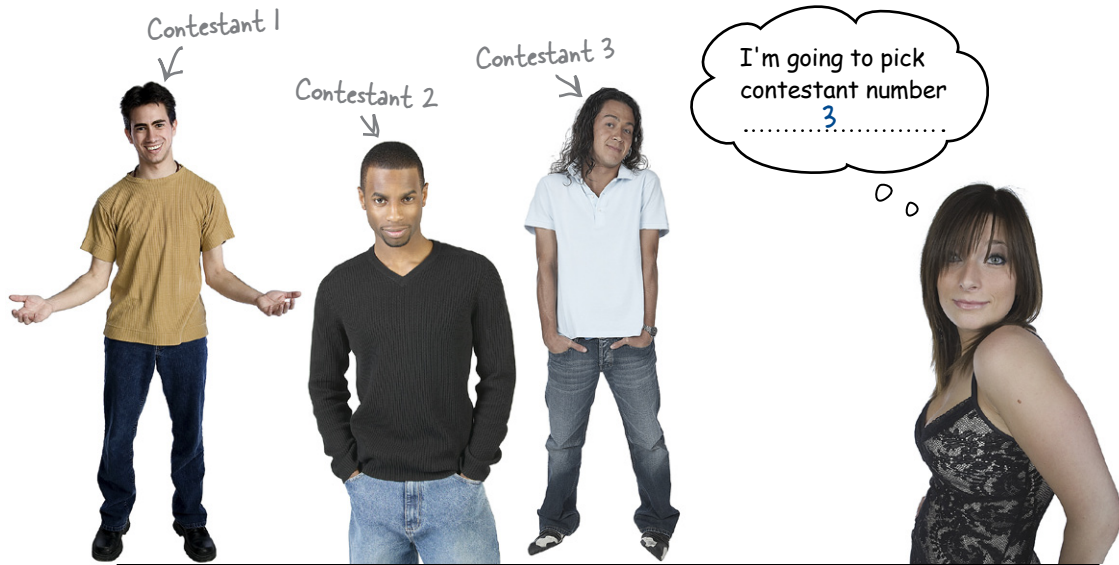
```
#include <stdio.h>

int main()
{
    int contestants[] = {1, 2, 3};
    int choice = contestants[0];
    contestants[0] = 2;
    contestants[1] = contestants[2];
    contestants[2] = choice;
    choice = contestants[1];
    printf("I'm going to pick contestant number %i\n", choice);
    return 0;
}
```

THE MATING GAME SOLUTION

We have a classic trio of bachelors ready to play The Mating Game today.

Tonight's lucky lady is going to pick one of these fine contestants. Who will she choose?



```
#include <stdio.h>

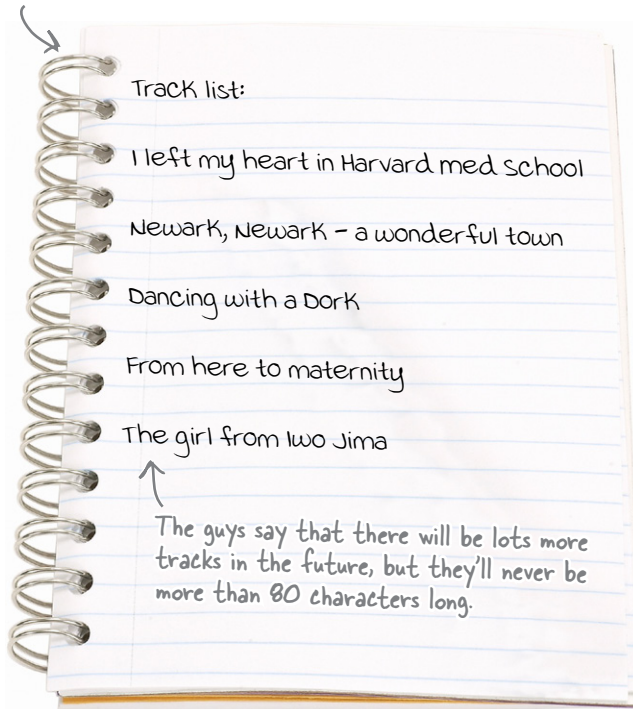
int main()
{
    int contestants[] = {1, 2, 3};
    int choice = contestants[0];
    contestants[0] = 2;
    contestants[1] = contestants[2];
    contestants[2] = choice;
    choice = contestants[1];
    printf("I'm going to pick contestant number %i\n", choice);
    return 0;
}
```

Frank Desperately seeking Susan

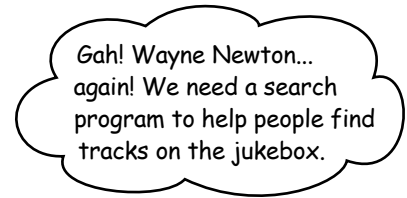
There are so many tracks on the retro jukebox that people can't find the music they are looking for. To help the customers, the guys in the Head First Lounge want you to write another program.

This is the track list:

Tracks from the new album "Little Known Sinatra".



The list is likely to get longer, so there's just the first few tracks for now. What you'll need to do is write a C program that will ask the user which track they are looking for, and then get it to search through all of the tracks and display any that match.



There'll be lots of strings in this program. How do you think you can record that information in C?

You need to create an array of arrays

There are several track names that you need to record. You can record several things at once in an array. But remember - *each string is itself an array*. That means you need to create an array of arrays - like this:

This first set of brackets is for the array of all strings.

The compiler can tell that we have 5 strings, so we don't need a number between these brackets.

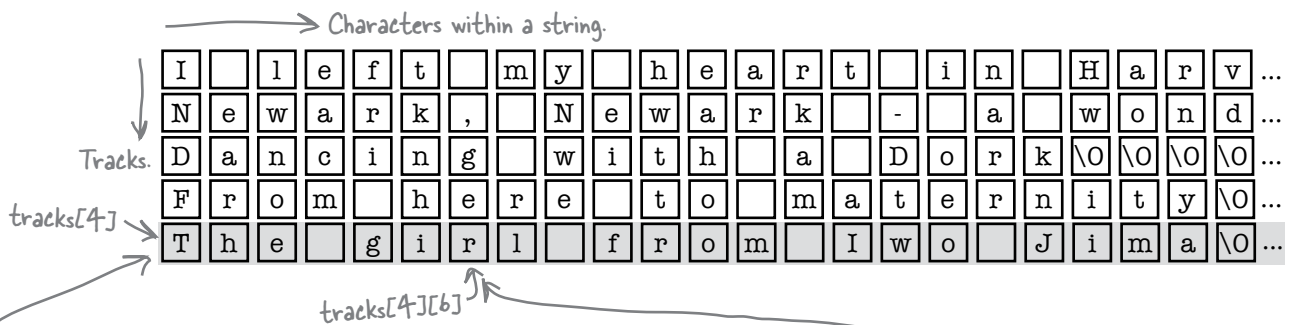
The second set of brackets is used for each individual string.

We know that track names will never get longer than 80 characters, so we'll set this to 81.

```
char tracks[][81] = {
    "I left my heart in Harvard Med School",
    "Newark, Newark - a wonderful town",
    "Dancing with a Dork",
    "From here to maternity",
    "The girl from Iwo Jima",
};
```

Each string is an array, so this is an array of arrays.

The array of arrays looks something like this in memory:



That means that we will be able to find an individual track name like this:

This has this value. → This is the 5th string. ← Remember - arrays begin at zero.

```
tracks[4] → "The girl from Iwo Jima"
```

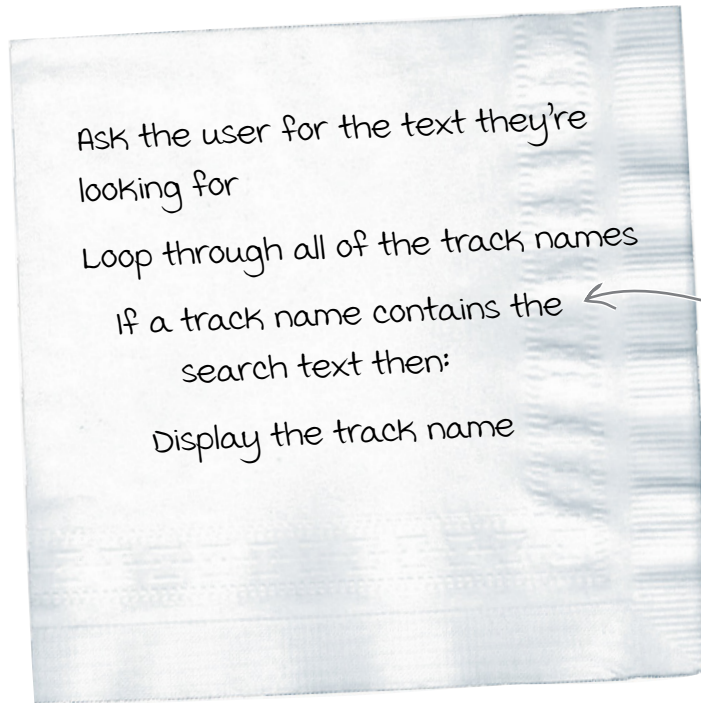
But we can also read the individual characters of each of the strings if we want to:

```
tracks[4][6] → 'r' ← This is the 7th character in the 5th string.
```

So now we know how to record the data in C, what do we need to do with it?

...then find strings containing the search text

The Guys have helpfully given you a spec:



Hmmm... how do we find out if a track name contains some text?

Well, we know how to record the tracks. We also know how to read the value of an individual track name - so it shouldn't be too difficult to loop through each of them. We even know how to ask the user for a piece of text to search for.

But how we look to see if the track name contains a given piece of text?

That could be pretty gnarly code to write. Fortunately we won't have to - we'll use the **library**.

Using string.h

The **C standard library** is a bunch of useful code that you get for free when you install a C compiler. The library code does useful stuff like opening files, or doing math, or managing memory. Now chances are you won't want to use the *whole* of the standard library at once, so the library is broken up into several sections, and each one has a **header** file. The header file lists all of the functions that live in a particular section of the library.

So far we have only really used the `stdio.h` header file. `stdio.h` lets you use the standard *input/output* functions like `printf` and `scanf`.

But the standard library also contains code to *process strings*. String processing is required by a lot of the programs and by using the string code in the standard library is tested, stable and fast.



You can include the string code into your program using the `string.h` header file. You can add it at the top of your program, just like we include `stdio.h`.

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

We'll use both `stdio.h` and `string.h` in our jukebox program.

But what do the string functions look like, and which do we need to use?

* WHAT'S MY PURPOSE? *

See if you can match up each string.h function with the description of what it does.

strchr

Concatenate two strings together

strcmp

Find the location of a string inside another string

strstr

Find the location of a character inside a string

strlen

Find the length of a string

strcmp

Compare two strings together

strcpy

Copy one string to another

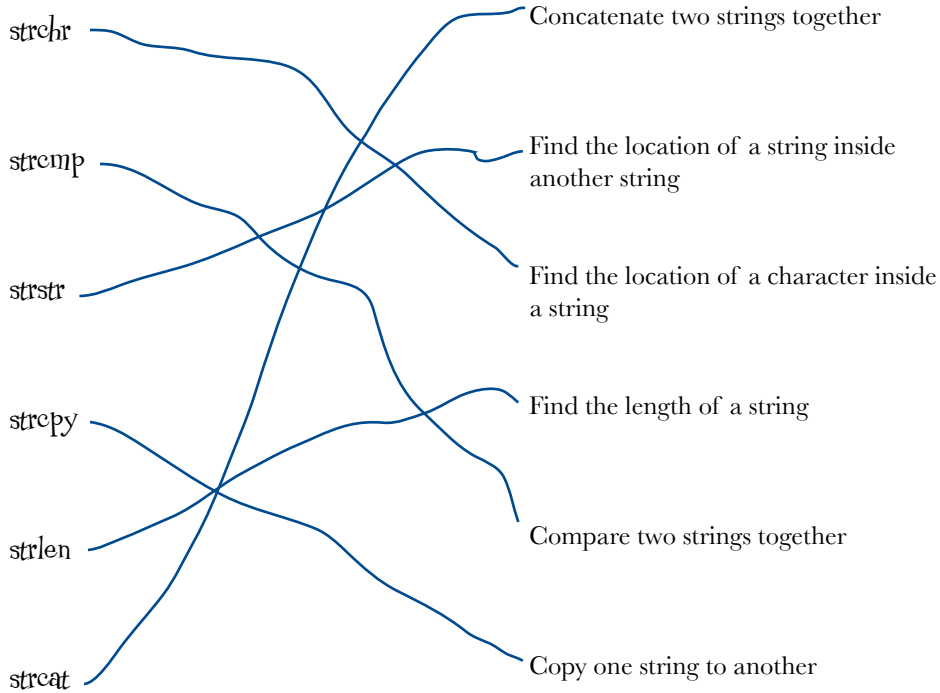
Sharpen your pencil

Which of the functions above should you use for the jukebox program? Write your answer below.

.....

WHAT'S MY PURPOSE? SOLUTION

See if you can match up each string.h function with the description of what it does.



Sharpen your pencil Solution

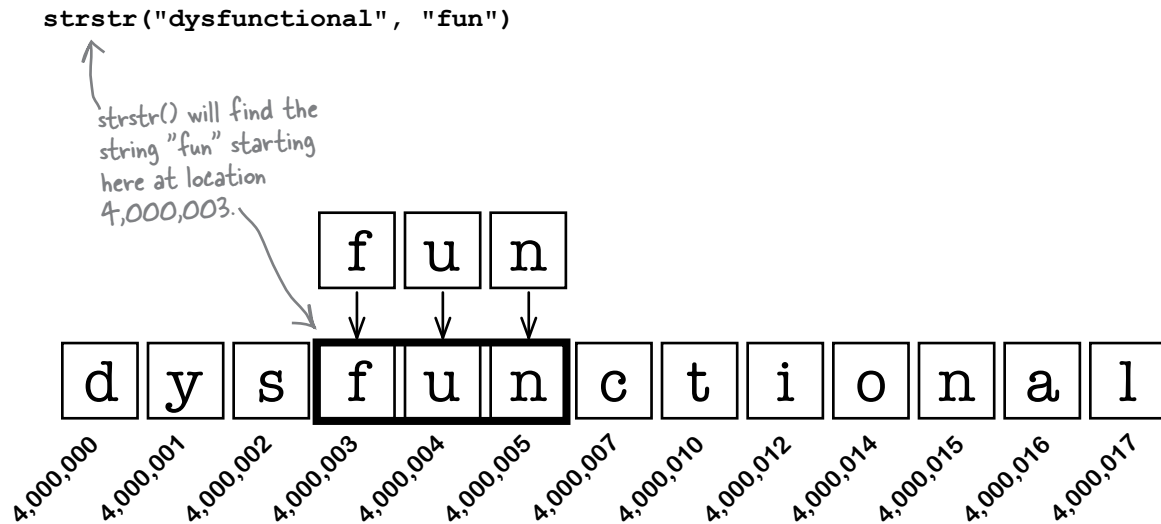


Which of the functions above should you use for the jukebox program? Write your answer below.

.....
strstr

Using the strstr function

So how exactly does the `strstr()` function work? Let's look at an example. Let's say we are looking for the string "fun" inside a larger string "dysfunctional". We'd call it like this:



The `strstr()` function will search for the second string in the first string. If it finds the string then it will return the address of the located string in memory. In the example here, the function might find that the "fun" substring begins at memory location 4,000,003.

But what if the `strstr()` can't find the substring? What then? In that case, `strstr()` returns the value 0. Can you think why that is? Well - if you remember, C treats zero as *false*. That means you can use `strstr` to check for the existence of one string inside another, like this:

```
char s0[] = "dysfunctional";
char s1[] = "fun";
if (strstr(s0, s1))
    puts("I found the fun in dysfunctional!");
```

Let's see how we can use `strstr()` in the jukebox program.

Pool Puzzle



The guys in the lounge had already started to write the code to search through the track list, but - oh no! - some of the paper they were writing the code on has fallen into the pool. Do you think you can select the correct pieces of code to complete the search function? It's been a while since the pool was cleaned - so be warned: some of the code in the pool might not be needed for this program.

Note: The guys have slipped in a couple new pieces of code they found in a book somewhere.

Hey look - we're creating a separate function. Presumably when we get round to writing the main() function it will call this.

"void" just means this function won't return a value.

```
void find_track(char search_for[])
```

```
{
```

```
    int i;
```

This is the "for loop". We'll look at this in more detail in a while, but for now you just need to know that it will run this piece of code 5 times.

```
    for (i = 0; i < 5; i++) {
```

```
        if ( ..... ( ..... , ..... ) )
```

This is where we're checking to see if the search term is contained in the track name.

If the track name matches our search, we'll display it here.

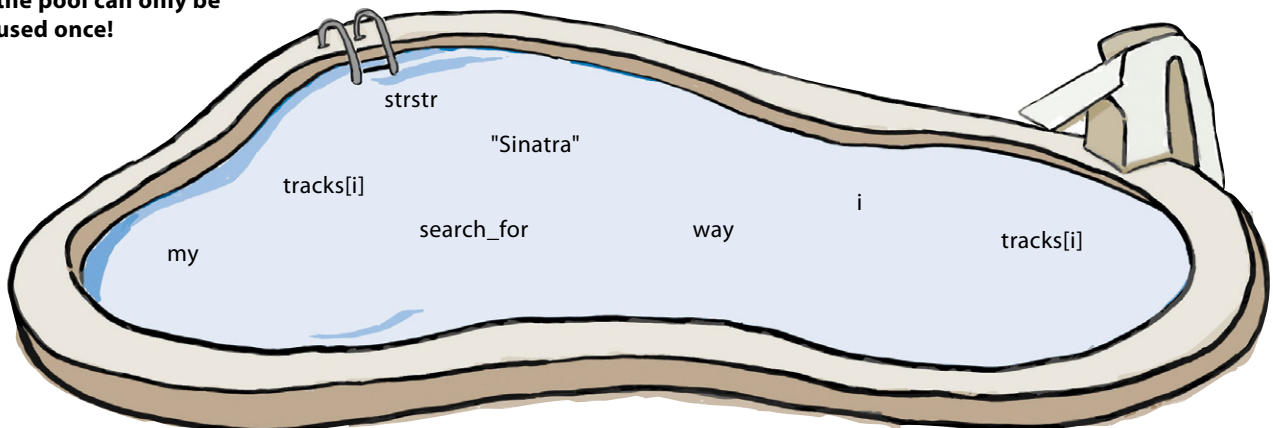
```
            printf("Track %i: '%s'\n", ..... , ..... );
```

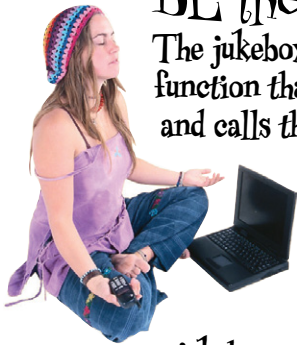
We're going to be printing out two values here.

One value will need to be an integer.

The other will be a string.

Note: each thing from the pool can only be used once!





BE the Compiler

The jukebox program needs a `main()` function that reads input from the user and calls the `find_track()` function on the opposite page. Your job is to play like you're the compiler and say which of the following `main()` functions is the one you need for the jukebox program.

```
int main()
{
    char search_for[80];
    printf("Search for: ");
    scanf("%79s", search_for);
    find_track();
    return 0;
}
```

```
int main()
{
    char search_for[80];
    printf("Search for: ");
    scanf("%79s", search_for);
    find_track(search_for);
    return 0;
}
```

```
int main()
{
    char search_for[80];
    printf("Search for: ");
    scanf("%80s", search_for);
    find_track(search_for);
    return 0;
}
```

```
int main()
{
    char search_for[80];
    printf("Search for: ");
    scanf("%79s", search_for);
    find_track(search_for);
}
```

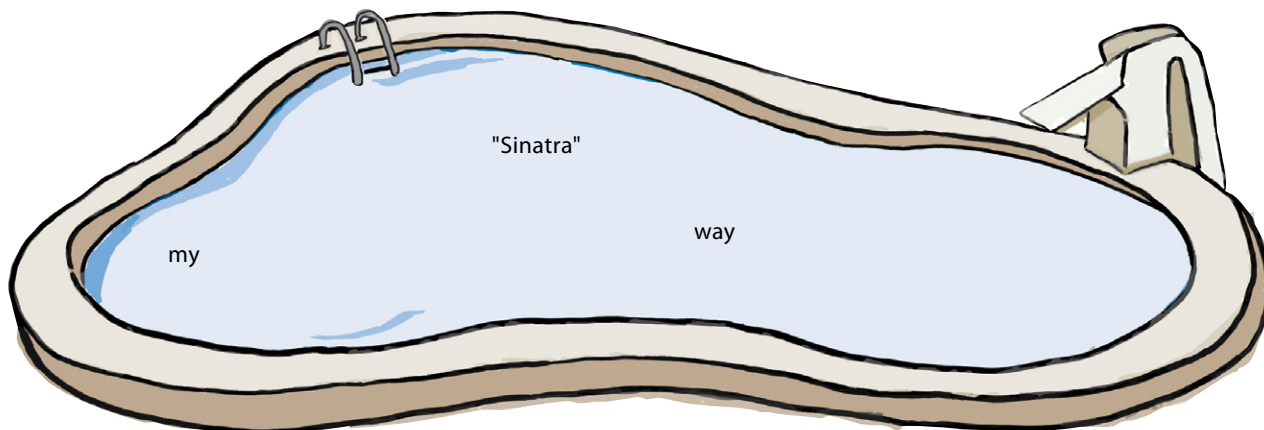
Pool Puzzle Solution

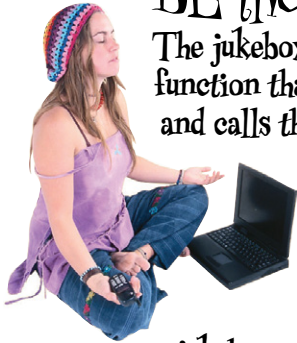


The guys in the lounge had already started to write the code to search through the track list, but - oh no! - some of the paper they were writing the code on has fallen into the pool. Do you think you can select the correct pieces of code to complete the search function? It's been a while since the pool was cleaned - so be warned: some of the code in the pool might not be needed for this program.

Note: The guys have slipped in a couple new pieces of code they found in a book somewhere.

```
void find_track(char search_for[])
{
    int i;
    for (i = 0; i < 5; i++) {
        if ( ..... strstr( tracks[i] , search_for ) )
            printf("Track %i: '%s'\n", ..... i ..... , ..... tracks[i] ..... );
    }
}
```





BE the Compiler Solution

The jukebox program needs a `main()` function that reads input from the user and calls the `find_track()` function on the opposite page. Your job is to play like you're the compiler and say which of the following `main()` functions is the one you need for the jukebox program.

```
int main()
{
    char search_for[80];
    printf("Search for: ");
    scanf("%79s", search_for);
    find_track(); ← find_track() is being
    return 0;      called without passing
}                 the search term.
```

This is the correct `main()` function

```
int main()
{
    char search_for[80];
    printf("Search for: ");
    scanf("%79s", search_for);
    find_track(search_for);
    return 0;
}
```

This version would allow the user to enter an 81 character string – that's one too many for the 80 character `search_for` array.

```
int main()
{
    char search_for[80];
    printf("Search for: ");
    scanf("%80s", search_for);
    find_track(search_for);
    return 0;
}
```

```
int main()
{
    char search_for[80];
    printf("Search for: ");
    scanf("%79s", search_for);
    find_track(search_for);
}
```

Oops! This one doesn't have a return value. That will cause the compiler to generate a warning.

It's time for a code review

Let's bring the code together and review what we've got so far:

```

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

char tracks[][80] = {
    "I left my heart in Harvard Med School",
    "Newark, Newark - a wonderful town",
    "Dancing with a Dork",
    "From here to maternity",
    "The girl from Iwo Jima",
};

void find_track(char search_for[])
{
    int i;
    for (i = 0; i < 5; i++) {
        if (strstr(tracks[i], search_for))
            printf("Track %i: '%s'\n", i, tracks[i]);
    }
}

int main()
{
    char search_for[80];
    printf("Search for: ");
    scanf("%79s", search_for);
    find_track(search_for);
    return 0;
}

```

We still need to `stdio.h` for the `printf()` and `scanf()` functions.

We will also need the `string.h` header so we can search with the `strstr()` function

We'll set the `tracks` array outside of the `main()` and `find_track()` functions - that way, the `tracks` will be usable everywhere in the program.

This is our new `find_track()` function - we'll need to declare it here before we call it from `main()`.

`i++` means "increase the value of `i` by 1".

This code will display all the matching tracks.

We're asking for the search text here.

Now we call our new `find_track()` function and display the matching tracks.

And this is our `main()` function - which is the starting point of the program.

It's important that we assemble the code in this order. The headers are included at the top, so that the compiler will have all the correct functions before it compiles our code. Then we'll define the `tracks` *before* we write the functions. This is called putting the `tracks` array in **global scope**. A global variable is one that lives outside any particular function. Global variables like `tracks` are available to all of the functions in the program.

Finally, we have the functions: `find_track` first, followed by `main()`. The `find_track()` function needs to come first, *before* we call it from `main()`.

OK - let's see if the program works.



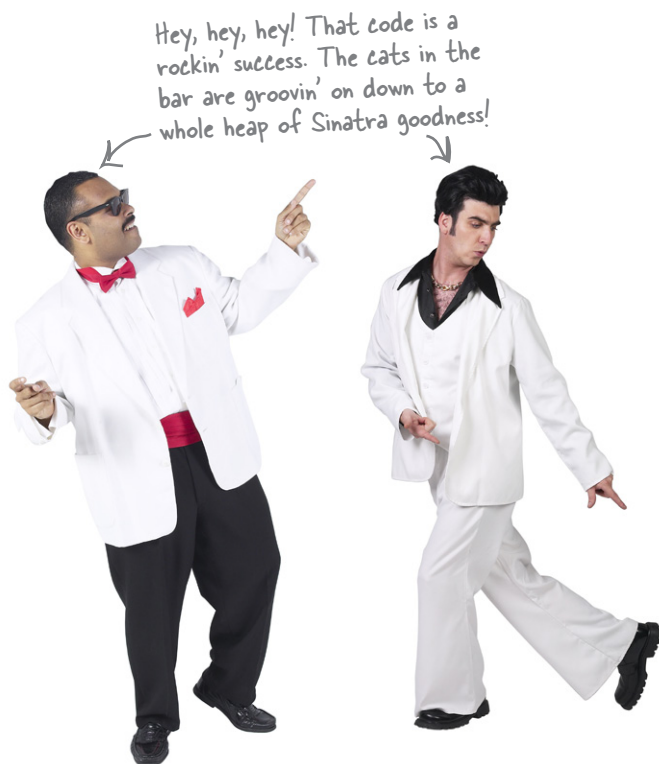
TEST DRIVE

It's time to fire up the terminal and see if the code works.

```
File Edit Window Help MyWeigh
> gcc text_search.c -o text_search && ./text_search
Search for: town
Track 1: 'Newark, Newark - a wonderful town'
>
```

And the great news is - the program works!

Even though this program is a little longer than any code we've written so far, it's actually doing a lot more. It creates an array of strings and then uses the string library to search through all of them to find the music track that the user was looking for.



Geek Bits

For more information about the functions available in `string.h` see <http://linux.die.net/man/3/string>

If you are using a Mac or a Linux machine, you can find out more about each of the `string.h` functions like `strstr` by typing:

```
man strstr
```

But what was that "for" all about?

There was one piece of code that we used in the code before really looking at how it worked - the **for** loop.

Now if you've used another C-like language, like JavaScript or Java or C#, then the for-loop is probably already familiar to you. But if not here's how to use it in a little more detail.

The for-loop is used to run a piece of code several times - just like the while-loop does. In fact, there's no real need to ever use a for-loop. Theoretically, you could just stick to while loops. So why do for-loops exist? What are they for?

A lot of programs contain loops that follow a particular pattern, something like this:

```

int counter = 1;
while (counter < 11) {
    printf("%1 green bottles, hanging on a wall\n");
    counter++;
}

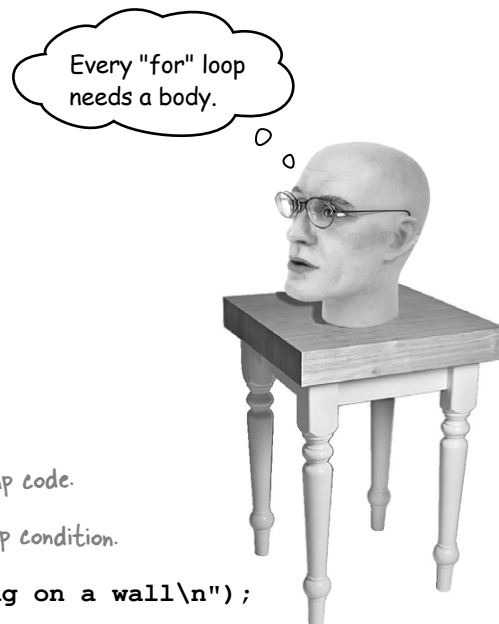
```

← This is the loop startup code.

← This is the loop condition.

← Remember - counter++ means "increase the counter variable by one" ..

← This is the loop update code that runs at the end of the loop body to update a counter.



Loops like this have code that prepares variables ready for the loop, some sort of condition that is checked each time the loop runs and finally some sort of code at the end of the loop that updates a counter or something similar.

Because this pattern is so common, the designers of C created the for-loop to make it a little more concise. Here is that same piece of code written with a for-loop:

```

int counter;
for (counter = 1; counter < 11; counter++) {
    printf("%1 green bottles, hanging on a wall\n");
}

```

← This initializes the loop variable.

← This is the code that will run after each loop.

← This is the text condition checked before the loop runs each time.

← Because there's only one line in the loop-body, we could actually have skipped these braces.

for-loops are actually used a *lot* in C - as much, if not more than while loops. Not only do they make the code slightly shorter, but they're easier for other C programmers to read because all of the code that controls the loop - the stuff that controls the value of the `counter` variable - is now contained in the for-statement and is taken out of the loop-body.

there are no
Dumb Questions

Q: Why is the list of tracks defined as `tracks[][80]`? Why not `tracks[5][80]`?

A: You *could* have defined it that way, but the compiler can tell there are 5 items in the list, so you can skip the `[5]` and just put `[]`.

Q: But in that case why couldn't we just say `tracks[][]`?

A: The track names are all different lengths, so you will need to tell the compiler how long to make each item in the array.

Q: Does that mean each string in the tracks array is 80 characters then?

A: The program will *allocate* 80 characters for each string, even though each of them is much smaller.

Q: So the tracks array takes 80 times 5 characters = 400 characters worth of space in memory?

A: Yes.

Q: What happens if I forget to include a header file like `string.h`?

A: For some header files, the compiler will give you a warning and then include them anyway. For other header files the compiler will simply give a compiler error.

Q: Why did we put the tracks array definition outside of the functions?

A: We put it into global scope. Global variables can be used by all functions in the program.

Q: Now that we've created two functions, how does the computer know which one to run first?

A: The program will always run the `main()` function first.

Q: Why do I have to put the `find_track()` function before `main()`?

A: C needs to know what parameters a function takes and what its return type is before it can be called.

Q: What would happen if I put the functions in a different order?

A: In this case you'd just get a few warnings.



BULLET POINTS

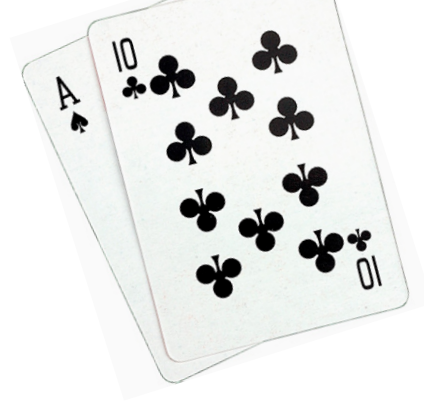
- You can create an array of arrays with char strings`[..][..]`.
- The first set of brackets are used to access the outer-array.
- The second set of brackets are used to access the details of each of the inner arrays.
- The `string.h` header file gives you access to a set of string manipulation functions in the C Standard Library.
- You can create several functions in a C program, but the computer will always run `main()` first.
- The for-loop is a more compact way of writing certain kinds of loops.

Anyone for three-card monte?

In the back-room of the Head First Lounge there's a game of three-card monte going on. Someone shuffles three cards around and you have to watch carefully and decide where you think the Queen card went. Of course, being the Head First Lounge, they're not using real cards - they're using *code*. Here's the program they're using:

```
#include <stdio.h>

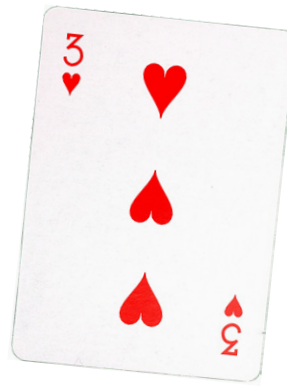
int main()
{
    char* cards = "JQK";
    char a_card = cards[2];
    cards[2] = cards[1];
    cards[1] = cards[0];
    cards[0] = cards[2];
    cards[2] = cards[1];
    cards[1] = a_card;
    puts(cards);
    return 0;
}
```



↑
Find the Queen.

The code is designed to shuffle the letters in the 3-letter string "JQK". Remember - in C, a string is just an array of characters. The program switches the characters around and then displays what the string looks like.

The players place their bets on where they think the 'Q' letter will be, then the code is compiled and run.



But what's with the *?

Take a closer look at the code. There's something we haven't used before:

```
char* cards = "JQK";
```

The `*` means that the `cards` variable is designed to store a **pointer** to a character in memory. Remember - a *pointer* is just another word for an address. By saying that `cards` is a `char*` we are telling the compiler that we are only ever going to store addresses in it.

Every time we've created strings before we've always declared the string with array notation like this:

```
char cards[] = "JQK";
```

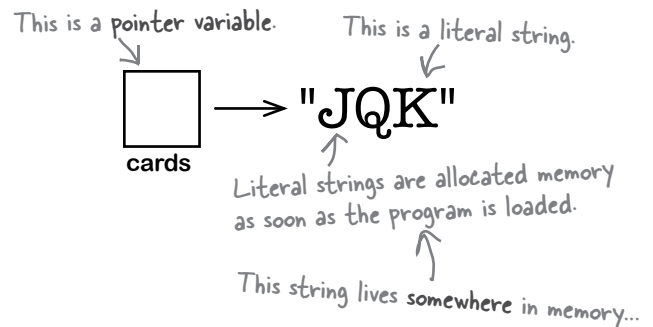
So what's the difference? Well, remember when you declare an array like this, you are actually doing *two* things:

- 1 Creating a new array of characters.
- 2 Creating a variable called `cards` that contains the address of the first character of the array.

If we use the `*`-notation, we actually skip the first step. By declaring `cards` as a variable of type `char*`, we are just creating a new **pointer variable** called `cards` and pointing it to the start of the "JQK" string.

But - wait a minute. How come we don't have to create the string array? Well the truth is that whenever a C program runs it automatically loads any *literal strings* the program uses into memory. So by the time the `main()` function gets called in our program, the string "JQK" *somewhere* in memory.

```
char* cards = "JQK";
```



Literal strings are automatically created in memory when the program starts.



Now that you know what the pointer-notation means in the code - it's time for you to try and work out what the code actually does. Where do you think the 'Q' is?

Oops... there's a memory problem...

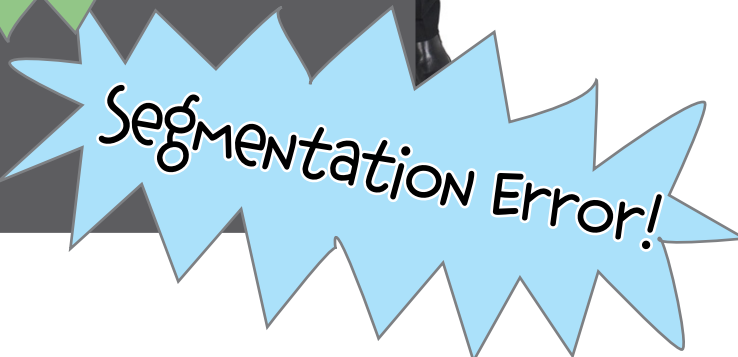
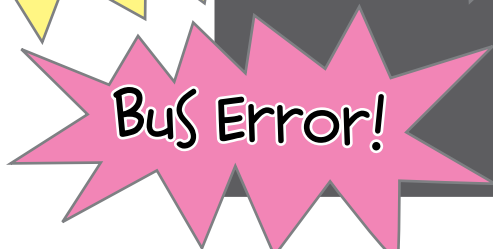
It seems there's a problem with the card shark's code. When the code is compiled and run on the Lounge's notebook computer, this happens:

```
File Edit Window Help PlaceBet
> gcc -Wall monte.c -o monte && ./monte
bus error
```



What's more, if the guys try the same code on different machines and operating systems, they get a whole bunch of different errors:

```
File Edit Window Help HolyCrap
> gcc -Wall monte.c -o monte && ./monte
monte.exe has stopped working
```



What's wrong with the code?

* WHAT'S YOUR HUNCH? *

It's time to use your **intuition**. Don't over-analyze. Just **take a guess**.
Read through these possible answers and select *only* the one you think is correct.

What do **you** think the problem is?

The string can't be updated	
We're swapping characters outside the string	
The string isn't in memory	
Something else	

* WHAT'S YOUR HUNCH? * * SOLUTION *

It's time to use your **intuition**. Don't over-analyze. Just **take a guess**.
Read through these possible answers and select *only* the one you think is correct.

What do **you** think the problem is?

The string can't be updated	✓
We're swapping characters outside the string	
The string isn't in memory	
Something else	

Literal strings can never be updated

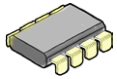
A variable that points to a literal string, can't be used to change the contents of the string:

```
char* cards = "JQK"; ← This variable can't modify this string.
```

But if you create an array from a literal string, then you **can** modify it:

```
char cards[] = "JQK";
```

It all comes down to how C uses memory...



In memory: `char* cards="JQK";`

To see why this line of code causes a memory error, we need to dig into the memory of the computer and see exactly what the computer will do.

1 The computer loads the literal string.

When the computer loads the program into memory, it puts all of the constant values - like the string literal "JQK" into the constant memory block. This section of memory is **read only**.

2 The program creates the cards variable on the stack.

The stack is the section of memory that the computer uses for local variables - variables inside functions. The cards variable will live here.

3 The cards variable is set to the address of "JQK".

The cards variable will contain the address of the **literal string "JQK"**. Remember - this is a string that lives in **read-only memory**.

4 The computer tries to change the string.

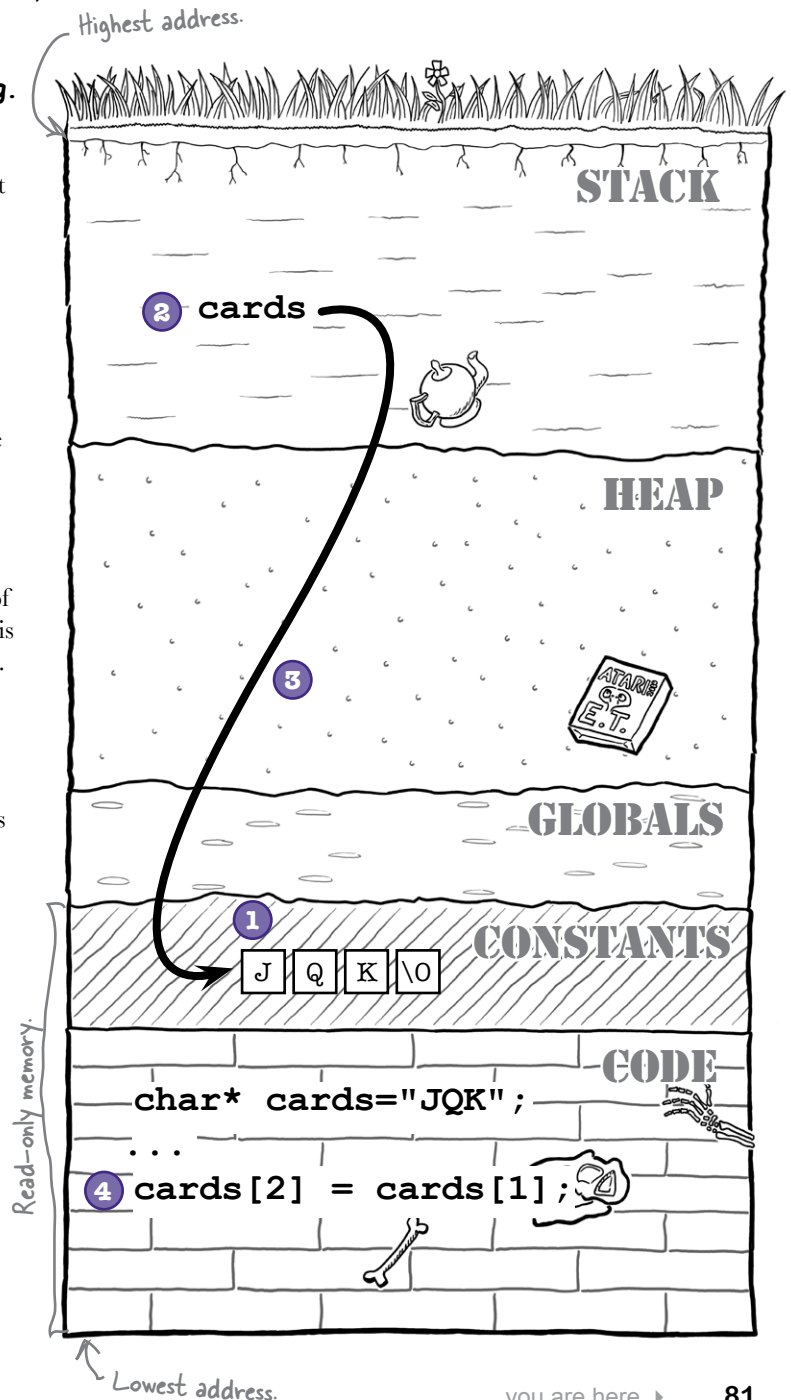
When the program tries to change the contents of the string pointed to by the cards variable - it can't; the string is read-only.

I can't update that, buddy - it's in the constant memory block so it's read-only.



So the problem is that literal strings like "JQK" are held in read only memory. They're constants.

But if that's the problem, how do we fix it?



If you're going to change a string, make a copy

The truth is that if we want to change the contents of a string, we need to make sure that we make our *own copy* of it. If we create a copy of the string in an area of the memory that's *not* read-only, we won't have a problem.

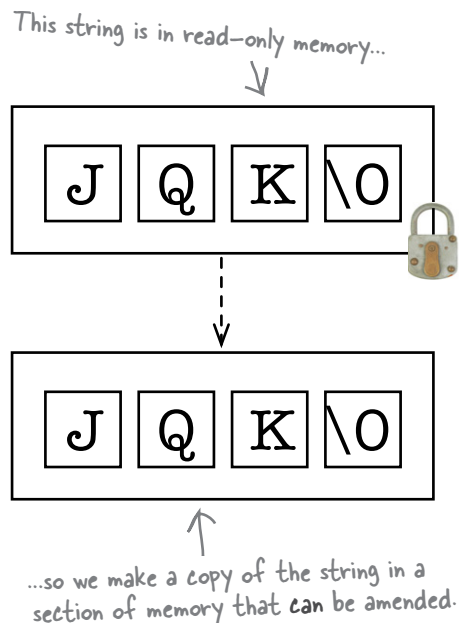
And how do we make a copy? Well - we just create the string as we've always done before: as a **new array**.

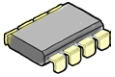
```
char s[] = "JQK";
```

Remember - when we create an array like this, we are actually doing two things:

- 1 We're creating a new array in memory and setting the values to 'J', 'Q', 'K' and the termination character '\0', and
- 2 We're creating a new *pointer variable* called `cards` with the address of the array.

To see how this code fixes the problem, let's take a look at what it does to memory.





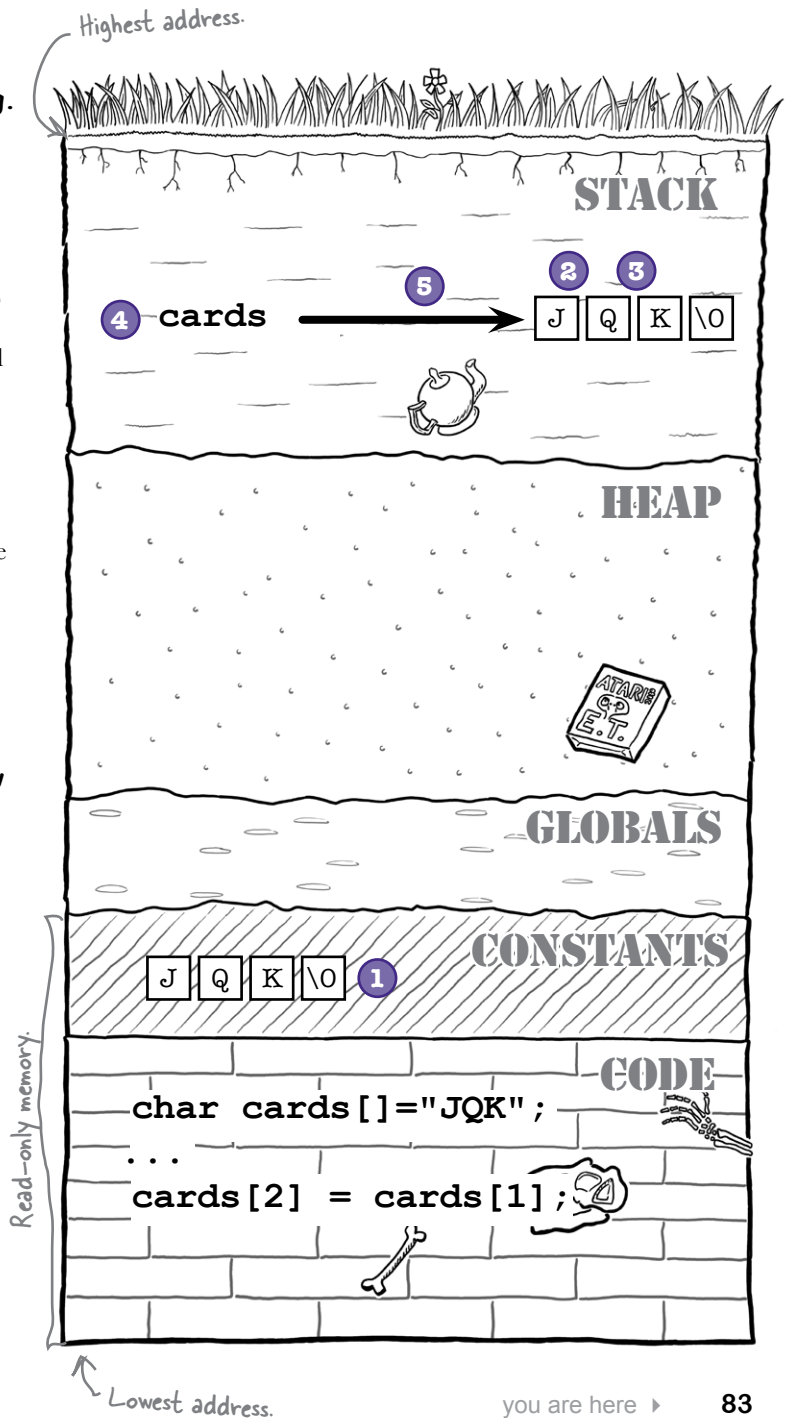
In memory: `char cards[]="JQK";`

We've already seen what happens with the *broken code*, but what about our new code? Let's take a look:

- 1 **The computer loads the literal string.**
As before, when the computer loads the program into memory, it stores the constant values like the string "JQK" into read only memory.
- 2 **The program creates a new array on the stack.**
We're declaring an array, so the program will create one large enough to store the "JQK" string - 4 characters worth.
- 3 **The program initializes the array.**
But as well as allocating the space, the program will also **copy the contents** of the literal string "JQK" into the stack memory. .
- 4 **The program creates the cards variable on the stack.**
Just the same as before.
- 5 **The cards variable points at the new array.**
But now, rather than pointing at the original *literal string*, it points at the copy in the array that is sitting in the stack.

So the difference this time, is that the cards *pointer variable* is now pointing at a string in memory that it *can* change. That should mean that if anything tries to change the cards string, there shouldn't be a problem.

Let's try it and see.





TEST DRIVE

So if we construct a **new array** in the code - let's see what happens:

```
#include <stdio.h>

int main()
{
    char cards[] = "JQK";
    char a_card = cards[2];
    cards[2] = cards[1];
    cards[1] = cards[0];
    cards[0] = cards[2];
    cards[2] = cards[1];
    cards[1] = a_card;
    puts(cards);
    return 0;
}
```

```
File Edit Window Help Where'sTheLady?
> gcc monte.c -o monte && ./monte
QKJ
```

Yes! The Queen
was the first
card. I knew it...



The code works! Our cards variable now points to a string in an unprotected section of memory, so we are free to modify it's contents.



Geek Bits

One way to avoid this problem in the future is to never write code that sets a simple char pointer to a literal string value like:

```
char * s = "Some string";
```

There's nothing wrong with setting a pointer to a string literal - the problems only happen when you try to modify a literal string. Instead, if you want to set a pointer to a literal, always make sure you use the const keyword:

```
const char * s = "some string";
```

That way, if the compiler sees some code that uses tries to modify the string, it will give you a compile error:

```
s[0] = 'S';
monte.c:7: error: assignment of read-only location
```

there are no
Dumb Questions

Q: Why didn't the compiler just tell me I couldn't change the string?

A: Because we'd declare the `cards` as a simple `"char *"`, the compiler didn't know that the variable would always be pointing at a literal string.

Q: Why are string literals stored in read only memory?

A: Because they are designed to be constant. If you write a function to print "Hello World", you don't want some other part of the program modifying the "Hello World" literal string.

Q: Do all operating systems enforce the read-only rule?

A: The vast majority do. Some versions of GCC on Cygwin actually allow you to modify a literal string without complaining. But it is *always* wrong to do that.

Q: What does `const` actually mean? Does it make the string read-only?

A: Literal strings are read-only anyway. The `const` modifier means that you the compiler will complain if you try to modify an array with that particular variable.

Q: Do the different memory segments always appear in the same order in memory?

A: They will always appear in the same order for a given operating system. But different operating systems can vary the order slightly. For example, Windows doesn't place the code in the lowest memory addresses.



BULLET POINTS

- If you see a `***` in a variable declaration, it means the variable will be a pointer.
- String literals are stored in read-only memory.
- If you want to modify a string, you need to make a copy in a new array.
- You can make declare a char-pointer as `const char *` to prevent the code using it to modify a string.

The Case of the Magic Bullet

He was scanning his back catalog of Guns 'n' Ammo into Delicious Library when there was a knock at the door and she walked in. 5' 6", blonde with a good laptop bag and cheap shoes. He could tell she was a code jockey. "You've gotta help me.. you gotta clear his name! Jimmy was innocent I tells you. Innocent!" He passed her a tissue to wipe the tears from her baby blues and led her to a seat.

It was the old story. She'd met a guy, who knew a guy. Jimmy Blomstein worked tables at the local Starbuzz and spent his weekends cycling and working on his taxidermy collection. He hoped one day to save up enough for an elephant. But he'd fallen in with the wrong crowd. The Masked Robber had met Jimmy in the morning for coffee and they'd both been alive:

```
char masked_raider[] = "Alive";
char * jimmy = masked_raider;

printf("Masked raider is %s, Jimmy is %s\n", masked_raider, jimmy);
```

Five Minute
Mystery



```
File Edit Window Help
Masked raider is Alive, Jimmy is Alive
```

Then, that afternoon, the Masked Raider had gone off to pull a heist. Like a hundred heists he'd pulled before. But this time, he'd not reckoned on the crowd of G-Men enjoying their weekly 3-card Monte session in the backroom of the Head First Lounge. You get the picture. A rattle of gunfire, a scream and moments later the villain was laying on the sidewalk, creating a public health hazard:

```
masked_raider[0] = 'D';
masked_raider[1] = 'E';
masked_raider[2] = 'A';
masked_raider[3] = 'D';
masked_raider[4] = '!';
```

Problem is, when Toots here goes in to check in with her boyfriend at the coffee shop, she's told he's served his last Orange Frappe Mochaccino:

```
printf("Masked raider is %s, Jimmy is %s\n", masked_raider, jimmy);
```

```
File Edit Window Help
Masked raider is DEAD!, Jimmy is DEAD!
```

So what gives? How come a single magic bullet killed Jimmy and the Masked Raider? What do you think happened?



Code Magnets

The guys are working on a new piece of code for a game. They've created a function that will display a string backwards on the screen. Unfortunately some of the fridge magnets have moved out of place. Do you think you can help them to re-assemble the code?

```

void print_reverse(char* s)
{
    size_t len = strlen(s);
    char* t = ..... + ..... - 1;

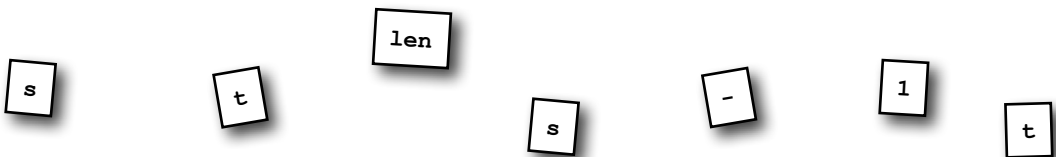
    while ( ..... >= ..... ) {
        printf("%c", *t);

        t = ..... ..... .....;
    }
    puts("");
}

```

size_t is just an integer used for storing the sizes of things. →

← This works out the length of a string - so strlen("ABC") == 3.





Code Magnets Solution

The guys are working on a new piece of code for a game. They've created a function that will display a string backwards on the screen. Unfortunately some of the fridge magnets have moved out of place. Do you think you can help them to re-assemble the code?

```

void print_reverse(char* s)
{
    size_t len = strlen(s);

    char* t = ... s ... + len - 1;

    while ( ... t ... >= ... s ... ) {
        printf("%c", *t);

        t = ... t ... - 1;
    }
    puts("");
}

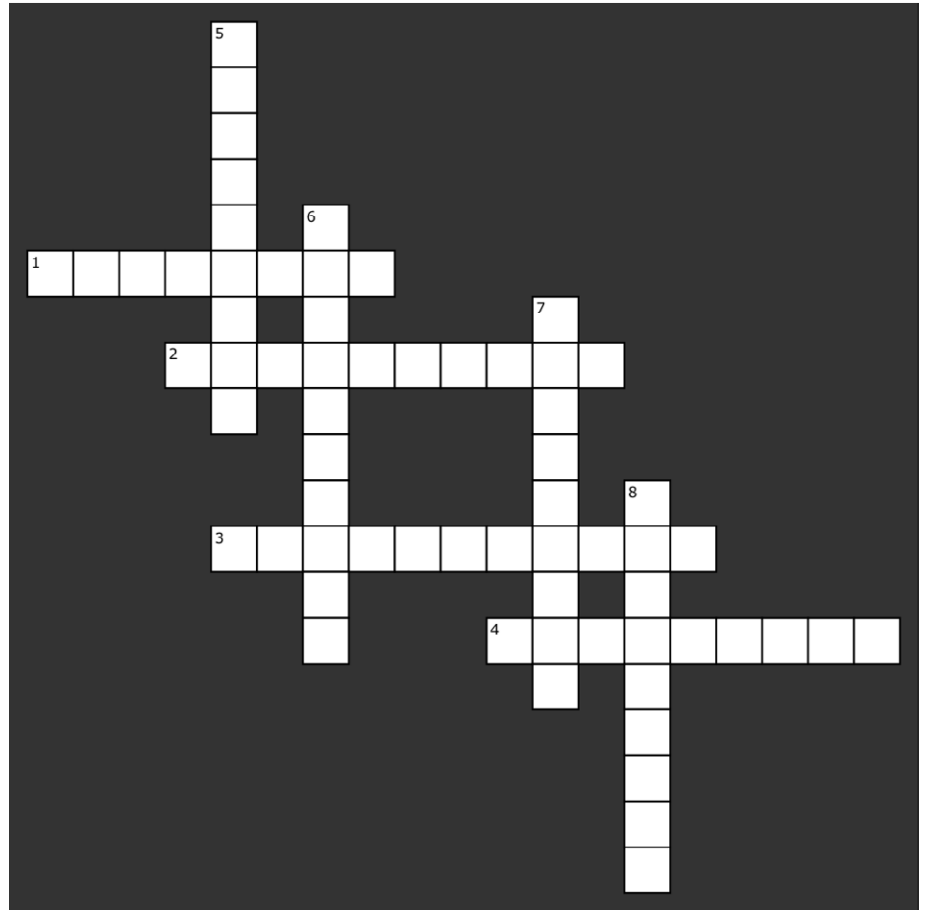
```

← Calculating addresses like this is called pointer arithmetic.



C-Cross

Now that the guys have the `print_reverse` function working, they've used it to create a crossword. The answers are displayed by the output lines in the code:



Across

```
int main()
{
    char* juices[] = {
        "dragonfruit", "waterberry", "sharonfruit", "uglifruit",
        "rumberry", "kiwifruit", "mulberry", "strawberry",
        "blueberry", "blackberry", "starfruit"
    };
    char* a;
    1 puts(juices[6]);
    2 print_reverse(juices[7]);
    a = juices[2];
    juices[2] = juices[8];
    juices[8] = a;
    3 puts(juices[8]);
    4 print_reverse(juices[(18 + 7) / 5]);
}
```

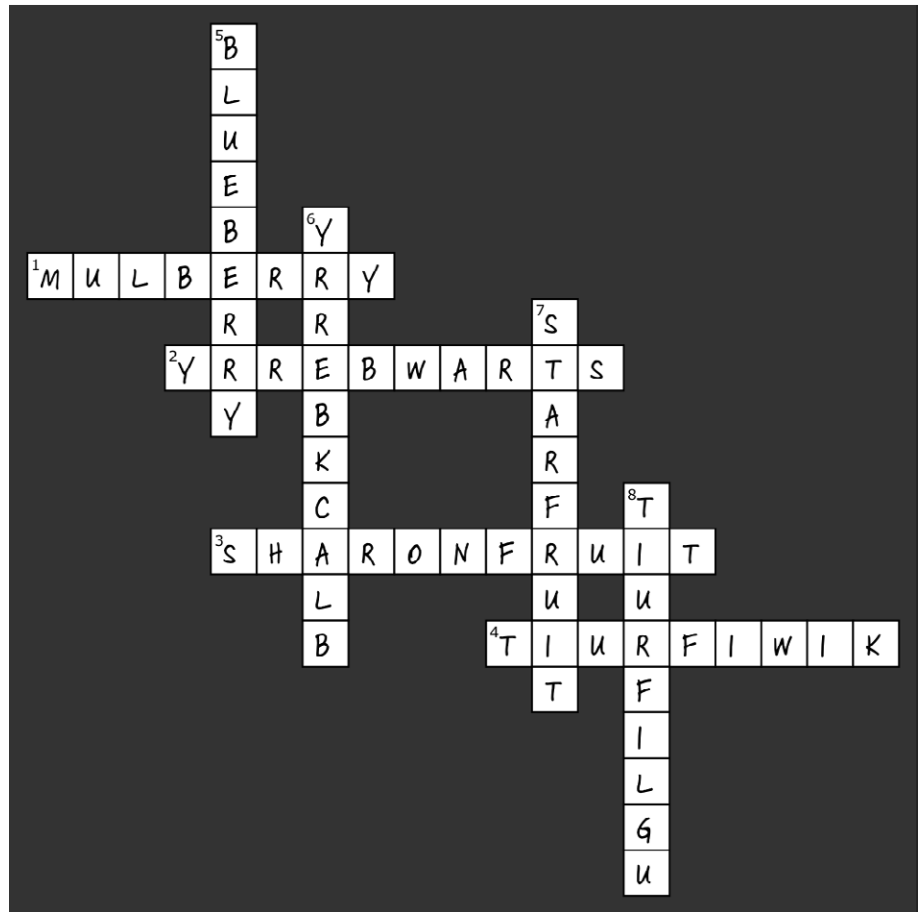
Down

```
5 puts(juices[2]);
6 print_reverse(juices[9]);
  juices[1] = juices[3];
7 puts(juices[10]);
8 print_reverse(juices[1]);
  return 0;
}
```



C-Cross Solution

Now that the guys have the `print_reverse` function working, they've used it to create a crossword. The answers are displayed by the output lines in the code:



Across

```
int main()
{
    char* juices[] = {
        "dragonfruit", "waterberry", "sharonfruit", "uglifruit",
        "rumberry", "kiwifruit", "mulberry", "strawberry",
        "blueberry", "blackberry", "starfruit"
    };
    char* a;
    1 puts(juices[6]);
    2 print_reverse(juices[7]);
    a = juices[2];
    juices[2] = juices[8];
    juices[8] = a;
    3 puts(juices[8]);
    4 print_reverse(juices[(18 + 7) / 5]);
}
```

Down

```
5 puts(juices[2]);
6 print_reverse(juices[9]);
  juices[1] = juices[3];
7 puts(juices[10]);
8 print_reverse(juices[1]);
  return 0;
}
```


The Case of the Magic Bullet

How come a single magic bullet killed Jimmy and the Masked Raider?

Jimmy, the mild-mannered barista was mysteriously gunned-down at the same time as arch-fiend The Masked Raider:

```
int main()
{
    char masked_raider[] = "Alive";
    char * jimmy = masked_raider;
    printf("Masked raider is %s, Jimmy is %s\n", masked_raider, jimmy);
    masked_raider[0] = 'D';
    masked_raider[1] = 'E';
    masked_raider[2] = 'A';
    masked_raider[3] = 'D';
    masked_raider[4] = '!';
    printf("Masked raider is %s, Jimmy is %s\n", masked_raider, jimmy);
}
```

Note from marketing. Ditch the product placement for the Brain Booster Drink - the deal fell through.

It took the detective a while to get to the bottom of the mystery. While he was waiting he took a long refreshing drink from a Head First Brain Booster Fruit Beverage. He sat back in his seat and looked across the desk at her blue, blue eyes. She was like a rabbit caught in the headlights of an on-coming truck, and he knew that he was at the wheel.

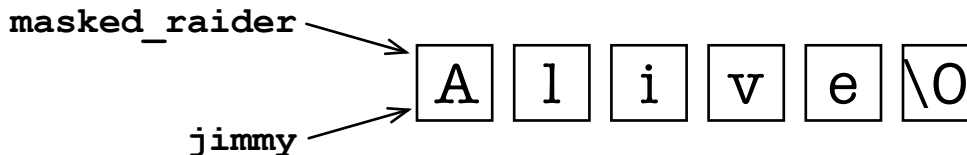


Five Minute
Mystery
Solved

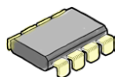
"I'm afraid I got some bad news for you. Jimmy and The Masked Raider... were one and the same man!"

"No!"

She took a sharp intake of breathe and raised her hand to her mouth. "Sorry sister. I have to say it how I see it. Just look at the memory usage". He drew a diagram:



"jimmy and masked_raider are just aliases for the same memory address. They're both pointing to the same place. When the masked_raider stopped the bullet, so did Jimmy. Add to that this invoice from the San Francisco elephant sanctuary and this order for 15 tons of packing material - and it's an open and shut case."



Memory memorizer

Stack.

This is the section of memory used for **local variable storage**. Every time you call a function, all of the function's local variables get created on the stack. It's called the stack because it's like a stack of plates - variables get added to the stack when you enter a function, and get taken off the stack when you leave. Weird thing is, the stack actually works upside down. It starts at the top of memory - and **grows downwards**.

Heap.

This is a section of memory we haven't really used yet. The heap is for **dynamic memory** - pieces of data that get created when the program is running and then hang around a long time. We'll see later in the book how we'll use the Heap.

Globals.

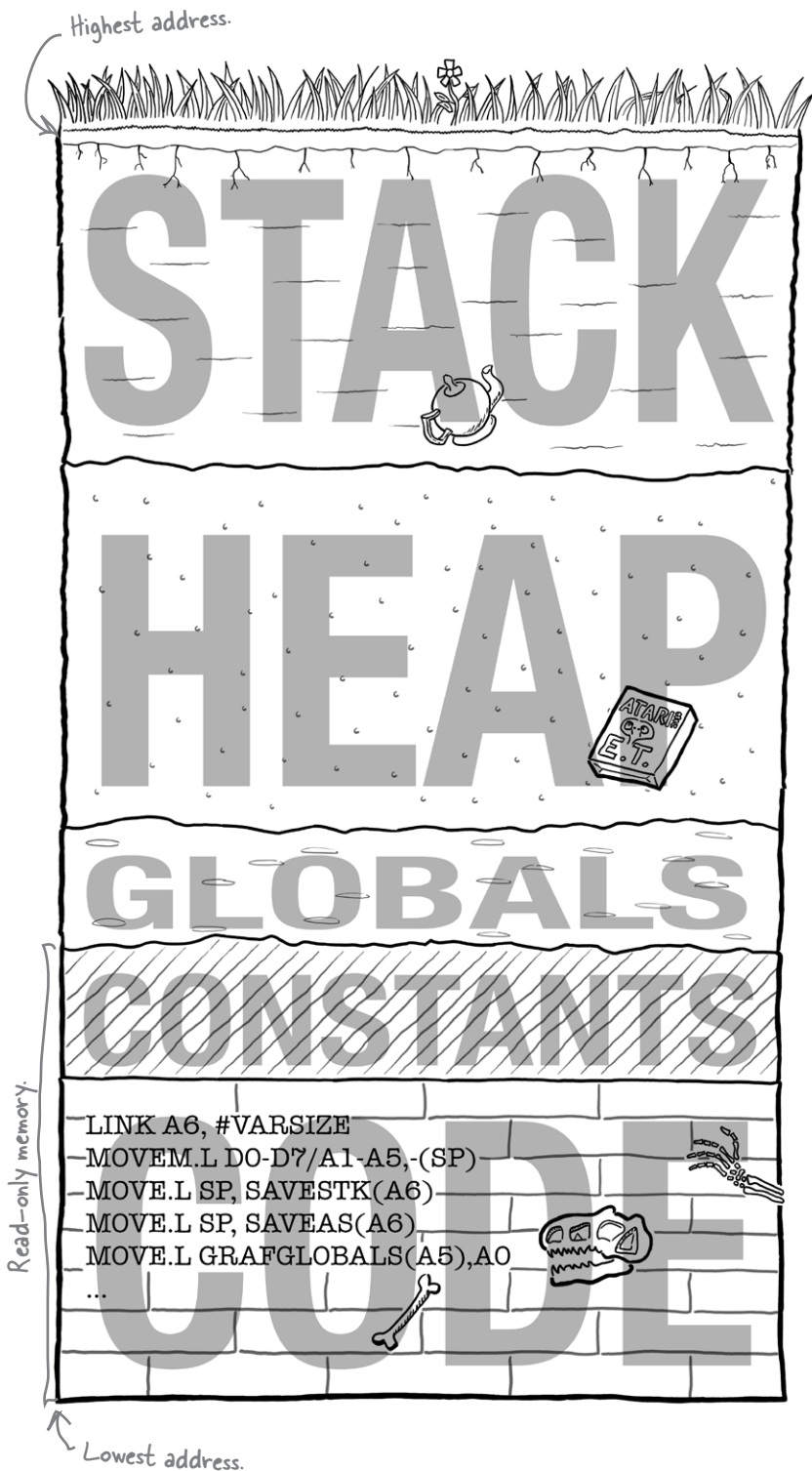
Remember when we put the list of songs from jukebox *outside* of any function? That's what a global variable is - it's a variable that lives outside of all of the functions and is visible to all of them. Globals get created when the program first runs and you can update them freely. But that's unlike...

Constants.

Constants are *also* created when the program first runs, but they are stored in **read-only** memory. Constants are things like *literal strings* that you will need when the program is running, but you'll never want them to change.

Code.

Finally the code segment. A lot of operating systems place the code right down in the lowest memory addresses. The code segment is also read-only. This is the part of the memory where the actual assembled code gets loaded.





Your C Toolbox

You've got Chapter 2 under your belt and now you've added pointers and memory to your tool box. For a complete list of tooltips in the book, see Appendix X.

`scanf("%i", &x)`
will allow a user to enter a number `x` directly.

ints are different sizes on different machines.

`&x` returns the address of `x`.

`&x` is called a pointer to `x`.

The `string.h` header contains useful string functions.

`strstr(a, b)` will return the address of string `b` in string `a`.

A char pointer variable `x` is declared as `char * x`.

Literal strings are stored in read-only memory.

Local variables are stored on the stack.

Initialize a new array with a string and it will copy it.

3 creating small tools

*Do one thing
and do it well*



Every operating system includes small tools.

Small tools perform **specialized small tasks**, such as reading and writing files, or filtering data. If you want to perform more complex tasks, you can even *link several tools together*. But how are these small tools built? In this chapter, you'll look at the building blocks of creating small tools. You'll learn how to control **command-line options**, how to manage **streams of information**, and **redirection**, getting toolled up in no time.

Small tools can solve big problems

A **small tool** is a C program that does *one* task and *does it well*. It might display the contents of a file on the screen or list the processes running on the computer. Or it might display the first 10 lines of a file or send it to the printer. Most operating systems come with a whole set of small tools that you can run from the command prompt or the terminal. Sometimes when you have a *big* problem to solve, you can break it down into a series of *small* problems, and then writing small tools for each of them.

A small tool does one task and does it well.

← Operating systems like Linux are mostly made up of hundreds and hundreds of small tools.

Someone's written me a map web application, and I'd love to publish my route data with it. Trouble is the format of the data coming from my GPS is wrong.



This is the data from the cyclist's GPS. It's a comma-separated format

This is a latitude.

This is a longitude.

```
42.363400,-71.098465,Speed = 21
42.363327,-71.097588,Speed = 23
42.363255,-71.096710,Speed = 17
...
```

This is the data format the map needs. It's in JavaScript Object Notation, or JSON.

The data's the same, but the format's a little different.

```
data=[
  {latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},
  {latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},
  {latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
  ...
]
```

If one small part of your program needs to convert data from one format to another, that's the perfect kind of task for a small tool.



Pocket Code

Hey - who hasn't taken a code print out on a long ride only to find that it soon becomes... unreadable? Sure - we all have. But with a little thought you should be able to piece together the original version of some code.

This program can read comma-separated data from the command line and then display it in JSON format. See if you can figure out what the missing code is.

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

```
int main()
```

```
{
```

```
    float latitude;
```

```
    float longitude;
```

```
    char info[80];
```

```
    int started = .....;
```

```
    puts("data=");
```

```
    while (scanf("%f,%f,%79[^\n]", ..... , ..... , ..... ) == 3) {
```

```
        if (started)
```

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

```
        else
```

```
            started = .....; ← Be careful how you set "started".
```

```
            printf("{latitude: %f, longitude: %f, info: '%s'}", ..... , ..... , ..... );
```

```
        }
```

```
    puts("\n");
```

```
    return 0;
```

```
}
```

We're using scanf to enter more than one piece of data.

What will these values be? Remember - scanf always uses pointers.

The scanf() function returns the number of values it was able to read.

This is just a way of saying "Give me every character up to the end of the line".

What values need to be displayed?



Pocket Code Solution

Hey - who hasn't taken a code print out on a long ride only to find that it soon becomes... unreadable? Sure - we all have. But with a little thought you should be able to piece together the original version of some code.

This program can read comma-separated data from the command line and then display it in JSON format. See if you can figure out what the missing code is.

```
#include <stdio.h>

int main()
{
    float latitude;
    float longitude;
    char info[80];
    int started = 0;

    puts("data=");
    while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3) {
        if (started)
            printf(",\n");
        else
            started = 1;
        printf("{latitude: %f, longitude: %f, info: '%s'}", latitude, longitude, info);
    }
    puts("\n");
    return 0;
}
```

We need to begin with "started" set to 0 - which means false.

Did you remember the "&"s on the number variables? scanf needs pointers.

We'll only display a comma if we've already displayed a previous line.

Once the loop has started, we can set started to 1 - which is true.

We don't need "&"s here because printf is using the values, not the addresses of the numbers.



TEST DRIVE

So what happens when we compile and run this code? What will it do?

This is the data that's printed out.

This is the data you type in.

The input and the output are mixed up.

```
File Edit Window Help JSON
> ./geo2json
data=[
42.363400,-71.098465,Speed = 21
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'}42.363327,-71.097588,Speed = 23
,
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'}42.363255,-71.096710,Speed = 17
,
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'}42.363182,-71.095833,Speed = 22
,
...
...
...
{latitude: 42.363182, longitude: -71.095833, info: 'Speed = 22'}42.362385,-71.086182,Speed = 21
,
{latitude: 42.362385, longitude: -71.086182, info: 'Speed = 21'}^D
]
>
```

Several more hours worth of typing...

In the end you need to press CTRL-D just to stop the program.

The program let's you enter GPS data at the keyboard and then it displays the JSON-formatted data on the screen. Problem is, the *input* and the *output* are all *mixed up together*. Also - there's a **lot of data**. If you are writing a small tool, you don't want to type the data in - you want to get large amounts of data by reading a **file**.

Also - how is the JSON data going to be used? Surely it can't be much use on the *screen*?

So is the program running OK? Is it doing the right thing? **Do we need to change the code?**

We really don't want the output on the screen. We need it in a file so we can use it with the mapping application. Here - let me show you...

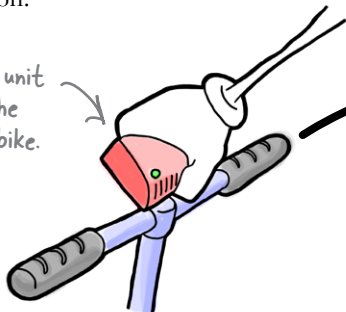


Here's how the program should work

- 1 Take the GPS from the bike and download the data.

It creates a file called `gpsdata.csv` with one line of data for every location.

This is the GPS unit used to track the location of the bike.



The data is downloaded into this file.

`gpsdata.csv`

- 2 The `geo2json` tool needs to read the contents of the `gpsdata.csv` line by line...

This is our `geo2json` tool.



Reading this file.

- 3 ...and then write that data in JSON format into a file called `output.json`.

Writing this file.

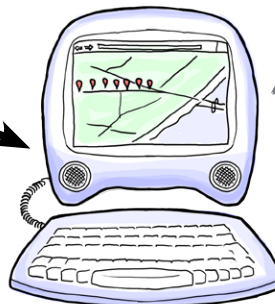


Our tool will write data to this file.



`output.json`

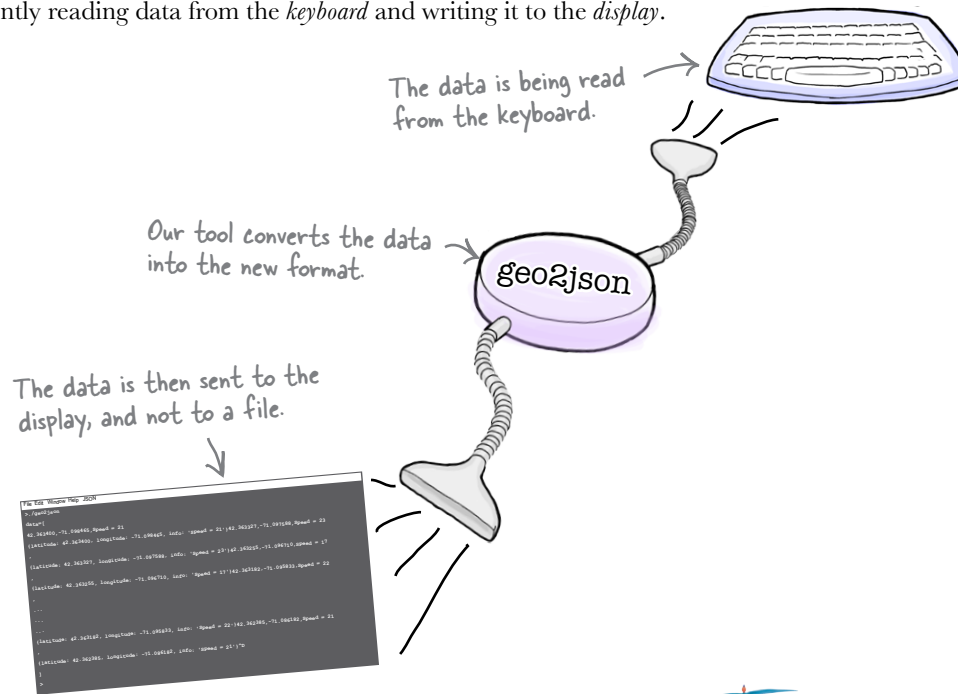
- 4 The web page that contains the map application reads the `output.json` file. It displays all of the locations on the map.



The mapping application reads the data from `output.json` and displays in on a map inside a web page.

But we're not using files...

The problem is, instead of reading and writing files, our program is currently reading data from the *keyboard* and writing it to the *display*.



But that isn't good enough. The user won't want to type in all of the data if it's already stored in a file somewhere. And if the data in JSON format is just displayed on the screen, there's no way the map within the web page will be able to read.

We need to make our program work with **files**. But how do we do that? If we want to use *files* instead of the keyboard and the display, what code will we have to change? Will we have to change any code at all?



Is there a way of making our program use files without changing code? Without even *re-compiling* it?



Geek Bits

Tools that read data line by line, process it and write it out again are called **filters**. If you have a Unix machine, or you've installed Cygwin on Windows, you already have a few filter tools installed:

head - this tool displays the first few lines of a file

tail - and this filter displays the lines at the end of a file

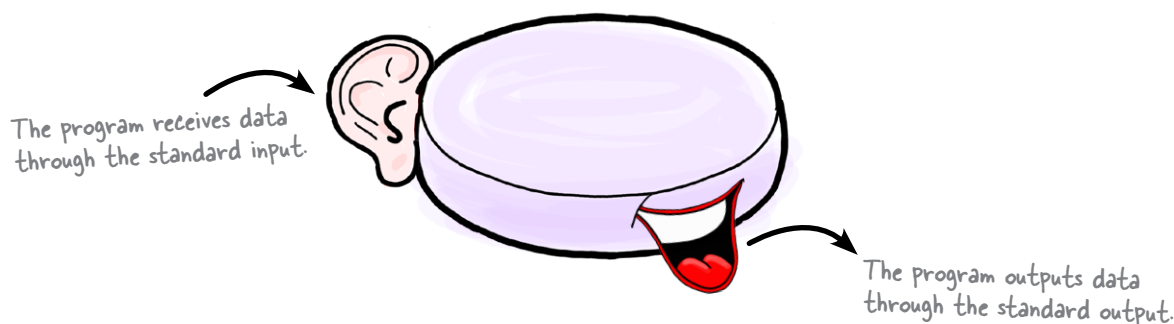
sed - the *stream editor* lets you do things like search and replace text

We'll see later how filters can be combined together to form **filter chains**.

We can use redirection

We're using `scanf()` and `printf()` to read from the keyboard and write to the display. But the truth is they don't talk *directly* to the keyboard and display. Instead they use the **standard input and standard output data streams**.

A data stream is exactly what it sounds like it is: a stream of data flowing into or out of a program. The *standard input* and *standard output* are created by the operating system when the program runs.



The operating system controls how data gets into and out of the standard input and output. If you run a program from the command prompt or terminal, the operating system will send all of the key strokes from the keyboard into the standard input stream. If the operating system reads any data out of the standard output stream, by default it will send that data to the display.

The `scanf()` and `printf()` don't know - or care - where the data comes from or goes to. They just read and write standard input and the standard output.

Now this might sound like it's kind of complicated. After all, why not just have your program talk direct to the keyboard and screen? Wouldn't that be simpler?

Well - there's a very good reason why operating systems communicate with programs using the standard input and the standard output:

You can redirect the standard data streams so that they read and write data somewhere else - such as to and from files.

You can redirect the Standard Input with <...

Instead of entering data at the keyboard we can use the "<" operator to read the data from a file.

```
42.363400,-71.098465,Speed = 21
42.363327,-71.097588,Speed = 23
42.363255,-71.096710,Speed = 17
42.363182,-71.095833,Speed = 22
42.363110,-71.094955,Speed = 14
42.363037,-71.094078,Speed = 16
42.362965,-71.093201,Speed = 18
42.362892,-71.092323,Speed = 22
42.362820,-71.091446,Speed = 17
42.362747,-71.090569,Speed = 23
42.362675,-71.089691,Speed = 14
42.362602,-71.088814,Speed = 19
42.362530,-71.087936,Speed = 16
42.362457,-71.087059,Speed = 16
42.362385,-71.086182,S
```

This is the file containing the data from the GPS device.

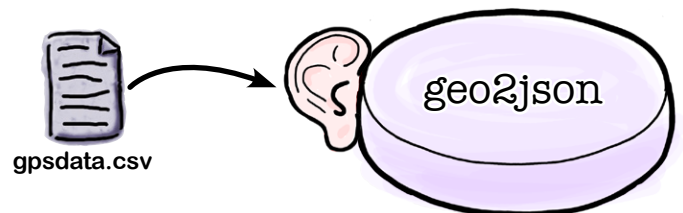
This is telling the operating system to send the data from the file into Standard Input of the program.

We don't have to type the GPS data in - so we don't see it mixed up with the output.

Now we just see the JSON data coming from the program.

```
File Edit Window Help MindYourStreams
> ./geo2json < gpsdata.csv
data=[
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
{latitude: 42.363182, longitude: -71.095833, info: 'Speed = 22'},
{latitude: 42.363110, longitude: -71.094955, info: 'Speed = 14'},
{latitude: 42.363037, longitude: -71.094078, info: 'Speed = 16'},
...
...
{latitude: 42.362385, longitude: -71.086182, info: 'Speed = 21'}
]
>
```

The "<" operator tells the operating system that the standard input of the program should be connected to the gpsdata.csv file instead of the keyboard. So we can send the program data from a file. Now we just need to redirect its **output**.



...and redirect the Standard Output with >

To redirect the standard output to a file, we need to use the > operator:

Now we are redirecting both the Standard Input stream and the Standard Output stream.

```
File Edit Window Help MindYourStreams
> ./geo2json < gpsdata.csv > output.json
>
```

The output of the program will now be written to output.json.

There's no output on the display at all - it's all gone to the output.json file.

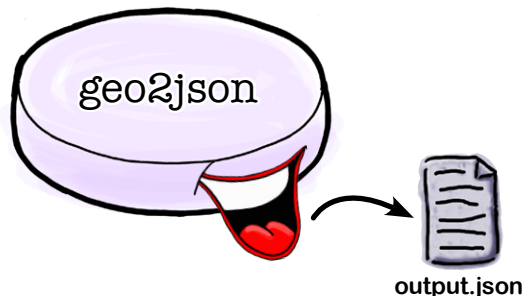
```
data=[
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
{latitude: 42.363182, longitude: -71.095833, info: 'Speed = 22'},
{latitude: 42.363110, longitude: -71.094955, info: 'Speed = 14'},
{latitude: 42.363037, longitude: -71.094078, info: 'Speed = 16'},
{latitude: 42.362965, longitude: -71.093201, info: 'Speed = 18'},
{latitude: 42.362892, longitude: -71.092323, info: 'Speed = 22'},
{latitude: 42.362820, longitude: -71.091446, info: 'Speed = 17'},
{latitude: 42.362747, longitude: -71.090569, info: 'Speed = 23'},
{latitude: 42.362675, longitude: -71.089691, info: 'Speed = 14'},
{latitude: 42.362602, longitude: -71.088814, info: 'Speed = 19'},
{latitude: 42.362530, longitude: -71.087936, info: 'Speed = 16'},
{latitude: 42.362457, longitude: -71.087059, info: 'Speed = 16'},
{latitude: 42.362385, longitude: -71.086182, info: 'Speed = 21'}
]
```



output.json

Because we've redirected the Standard Output, we don't see any data appearing on the screen at all. But the program has now created a file called output.json.

The output.json file is the one we needed to create for the mapping application. Let's see if it works.



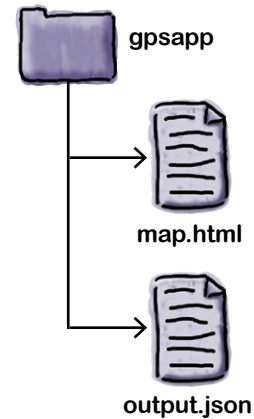
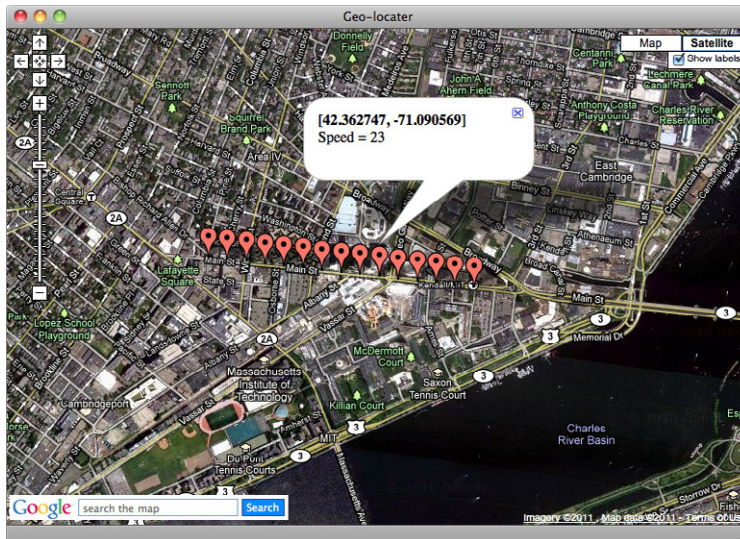


— Test Drive —

Now it's time to see if the new data file we've created can be used to plot the location data on a map. We'll take a copy of the web page containing the mapping program and put it into the same folder as the `output.json` file. Then we need to open the web page in a browser:



Download the web page from:
<http://oreillyhfc.appspot.com/map.html>



This is the web page that contains the map.

This is the file that our program created.

The map works.

The map inside the web page is able to read the data from our output file.



But there's a problem with some of the data...

Our program seems to be able to read GPS data and format it correctly for the mapping application. But after a few days a problem creeps in.



So what happened here? The problem is that there was some **bad data** in the GPS data file:

```
...  
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},  
{latitude: 423.63182, longitude: -71.095833, info: 'Speed = 22'},  
...
```

The decimal point is in the wrong place in this number.

But the `geo2json` program doesn't do any checking of the data it reads - it just reformats the numbers and sends them to the output.

That should be easy to fix. We need to validate the data





Exercise

We need to add some code to the geo2json program that will check for bad latitude and longitude values. We don't need anything fancy. If a latitude or longitude falls outside the expected numeric, just display an error message and exit the program with an error status of 2:

```
#include <stdio.h>

int main()
{
    float latitude;
    float longitude;
    char info[80];
    int started = 0;

    puts("data=");
    while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3) {
        if (started)
            printf(",\n");
        else
            started = 1;

        .....
        .....
        .....
        .....
        .....
        .....
        .....
        .....
        .....
        .....

        printf("{latitude: %f, longitude: %f, info: '%s'}", latitude, longitude, info);
    }
    puts("\n");
    return 0;
}
```

If the latitude is < -90 or > 90 then error with status 2. If the longitude is < -180 or > 180 then error with status 2.



Exercise Solution

We need to add some code to the geo2json program that will check for bad latitude and longitude values. We don't need anything fancy. If a latitude or longitude falls outside the expected numeric, just display an error message and exit the program with an error status of 2:

```
#include <stdio.h>

int main()
{
    float latitude;
    float longitude;
    char info[80];
    int started = 0;

    puts("data=[");
    while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3) {
        if (started)
            printf(",\n");
        else
            started = 1;
        if ((latitude < -90.0) || (latitude > 90.0)) {
            printf("Invalid latitude: %f\n", latitude);
            return 2;
        }
        if ((longitude < -180.0) || (longitude > 180.0)) {
            printf("Invalid longitude: %f\n", longitude);
            return 2;
        }

        printf("{latitude: %f, longitude: %f, info: '%s'}", latitude, longitude, info);
    }
    puts("\n");
    return 0;
}
```

These lines check that the latitude and longitude are in the correct range.

These lines display simple error messages.

These lines will exit from the main method with an error status of 2.



TEST DRIVE

OK - so we now have the code in place to check that the latitude and longitude are in range. But will it be enough to make our program cope with bad data? Let's see.

We'll compile the code and then run the bad data through the program:

This line will recompile the program.

Then we run the program again with the bad data.

WTF??? No error message?

This means "Welcome to Finland"...

```
File Edit Window Help MindYourStreams
> gcc geo2json.c -o geo2json
> ./geo2json < gpsdata.csv > output.json
>
```

We'll save the output in the output.json file.

And where did all the points go?

Hmmm... that's odd. We added the error checking code but when we run the program nothing *appears* to be different. But now no points appear on the map at all. What gives?



Study the code. What do **you** think happened? Is the code doing what we asked it to? Why weren't there any error messages? Why did the mapping program think that the entire `output.json` was corrupt?

CODE DECONSTRUCTION

The mapping program is complaining about the output.json file, so let's open her up and see what's inside:

← This is the output.json file.

```
data=[
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
Invalid latitude: 423.631805
```

Oh - the error message was also redirect to the output file.



Once you open the file you can see *exactly* what happened. The program saw that there was a problem with some of the data and it exited straight away. It didn't process any more data and it *did* output an error message. Problem is, because we were **redirecting the standard output** into the `output.json`, that meant we were also redirecting the error message. So the program ended silently and we never saw what the problem was.

Now, we *could* have checked the exit status of the program, but really we want to be able to see the error messages.

But how can we still display error messages if we are redirecting the output?



Geek Bits

If our program finds a problem in the data, it exits with a status of 2. But how can we check that error status after the program has finished? Well - it depends what operating system you are using. If you are running on a Mac, Linux, some other kind of Unix machine or if you are using Cygwin on a Windows machine, you can display the error status like this:

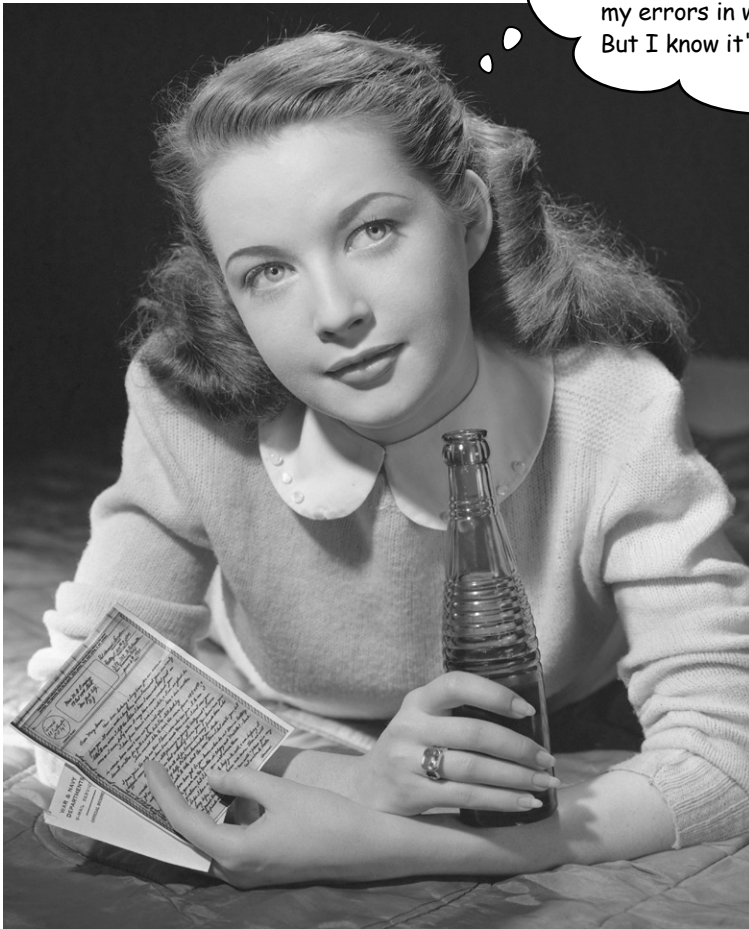
```
File Edit Window Help
$ echo $?
2
```

If you are using the Command Prompt in Windows, then it's a little different:

```
File Edit Window Help
C:\> echo %ERRORLEVEL%
2
```

Both commands do the same thing - they display the number returned by the program when it finished.

Wouldn't it be dreamy if there was a special output stream for errors so that I didn't have to mix my errors in with standard output? But I know it's just a fantasy...



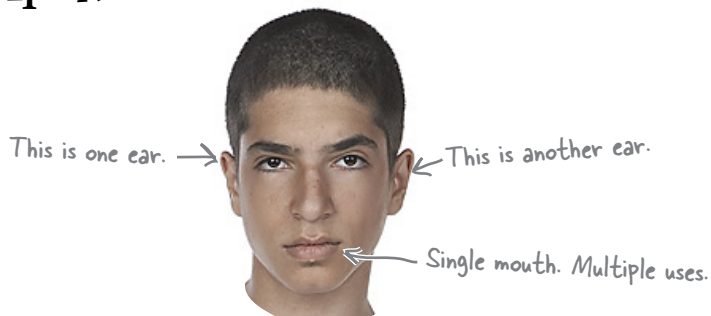
Introducing the Standard Error data stream

The **standard output** data stream is the *default* way of outputting data from a program. But what if something *exceptional* happens like an error? You'll probably want to deal with things like error messages a little differently from the usual output.

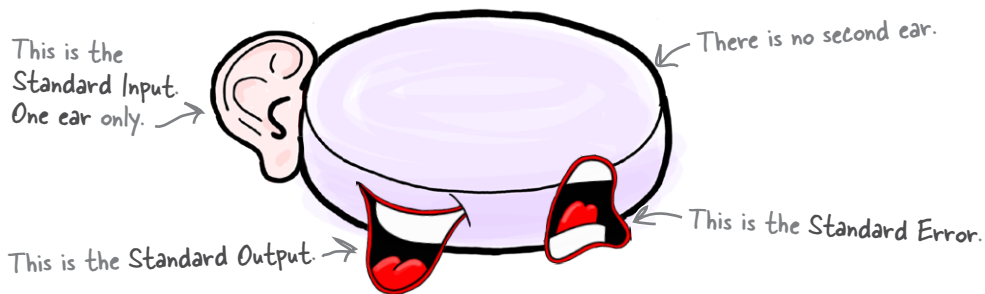
That's why the **Standard Error** data stream was invented. The Standard Error is a *second output data stream* that was created for sending error messages.

Human beings generally have two ears and one mouth, but processes are wired a little differently. Every process has **one ear** - the Standard Output - and **two mouths** - the Standard Output and the Standard Error.

Human



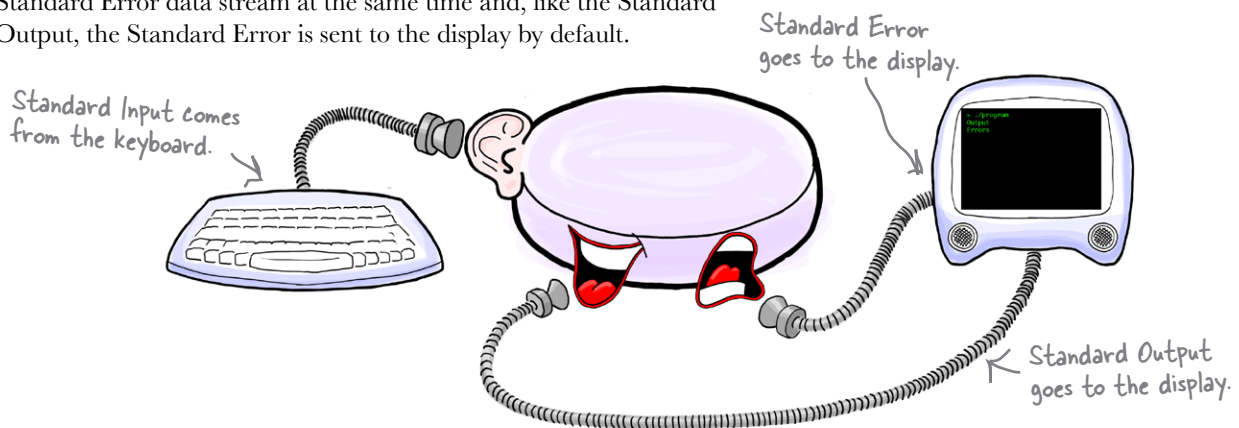
Process



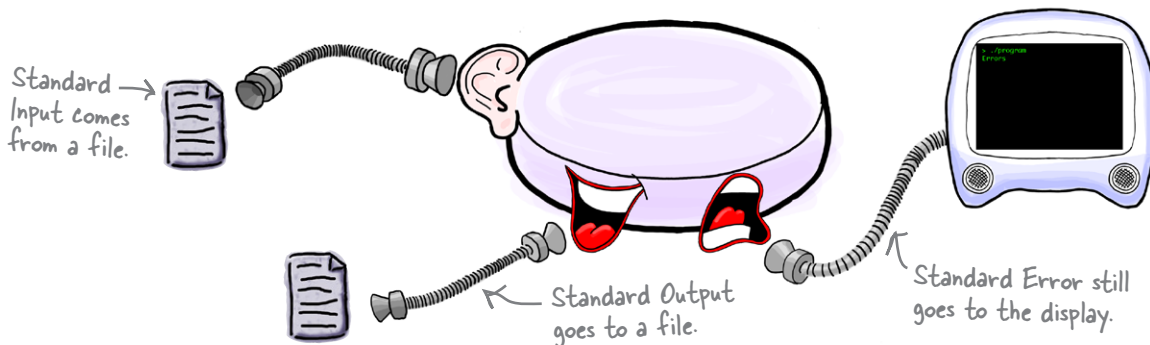
Let's see how the operating system sets these data streams up.

By default the Standard Error is sent to the display

Remember we said that when a new process is created, the operating system points the Standard Input at the keyboard and the Standard Output at the screen? Well, the operating system creates a Standard Error data stream at the same time and, like the Standard Output, the Standard Error is sent to the display by default.



That means that if someone redirects the Standard Input and Standard Output so they use files, the Standard Error will continue to send data to the display.



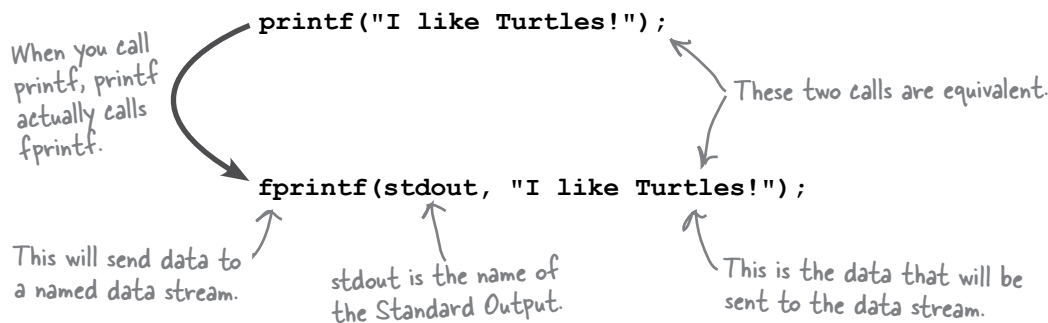
And that's really cool because it means that even if the standard output is redirected somewhere else, by default **any messages sent down the Standard Error will still be visible on the screen.**

So we can fix the problem of our hidden error messages by simply displaying them on the Standard Error.

But how do we do that?

fprintf prints to a file descriptor

We've already seen that the `printf()` function sends data to the Standard Error. What we *didn't* tell you is that the `printf()` function is just a version of a more general function called `fprintf()`:



The `fprintf()` function allows you to choose which data stream you want to send text to. You can tell `fprintf()` to print to `stdout` (the Standard Output) or `stderr` (the Standard Error).

there are no Dumb Questions

Q: There's a `stdout` and a `stderr`. Is there a `stdin`?

A: Yes - and as you probably guessed - it refers to the Standard Input.

Q: Can I print to it?

A: No - the Standard Input can't be printed to.

Q: Can I read from it?

A: Yes - using `fscanf()` - which is just like `scanf()` except you can specify the data stream.

Q: Can I redirect the Standard Error?

A: Yes. "`>`" redirects the Standard Output. But "`2>`" redirects the Standard Error.

Q: So I could write "`geo2json 2> errors.txt`"?

A: Yes.

Let's update the code to use fprintf

With just a couple small changes we can get our error messages printing on the standard error:

```
#include <stdio.h>

int main()
{
    float latitude;
    float longitude;
    char info[80];
    int started = 0;

    puts("data=[");
    while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3) {
        if (started)
            printf(",\n");
        else
            started = 1;
        if ((latitude < -90.0) || (latitude > 90.0)) {
printf("Invalid latitude: %f\n", latitude);
            fprintf(stderr, "Invalid latitude: %f\n", latitude);
            return 2;
        }
        if ((longitude < -180.0) || (longitude > 180.0)) {
printf(stderr, "Invalid longitude: %f\n", longitude);
            fprintf(stderr, "Invalid longitude: %f\n", longitude);
            return 2;
        }
        printf("{latitude: %f, longitude: %f, info: '%s'}", latitude, longitude, info);
    }
    puts("\n");
    return 0;
}
```

Instead of "printf" we use "fprintf".

We need to specify "stderr" as the first parameter.

That means that the code should now work in exactly the same way, *except* the error messages should appear on the Standard Error instead of the Standard Output.

Let's run the code and see.



TEST DRIVE

If we recompile the program and then run the corrupted GPS data through it again, this happens.

```
File Edit Window Help ControlErrors
> gcc geo2json.c -o geo2json
> ./geo2json-page21 < gpsdata.csv > output.json
Invalid latitude: 423.631805
```

That's excellent. This time, even though we are redirecting the Standard Output into the `output.json` file, the error message is still visible on the screen.

The Standard Error data stream was created with exactly this in mind: to separate the error messages from the usual output. But remember - the `stderr` and `stdout` are both just output data streams. And there's nothing to prevent you using them for anything.

Let's try out your new found Standard Input and Standard Error skills.



BULLET POINTS

- The `printf()` sends data to the *Standard Output*.
- The Standard Output goes to the display by default.
- You can *redirect* the output of the Standard Output to a file using `>` on the command line.
- `scanf()` reads data from the *Standard Input*.
- The Standard Input reads data from the keyboard by default.
- You can redirect the Standard Input to read a file by using `<` on the command line.
- The Standard Error is reserved for outputting error messages.
- You can redirect the Standard error using `2>`.

TOP SECRET

We have reason to believe that the following program has been used in the transmission of secret messages:

```
#include <stdio.h>

int main()
{
    char word[10];
    int i = 0;
    while (scanf("%9s", word) == 1) {
        i = i + 1;
        if (i % 2)
            fprintf(stdout, "%s\n", word);
        else
            fprintf(stderr, "%s\n", word);
    }
    return 0;
}
```

i % 2 means "The remainder left when you divide by 2"

We have intercepted a file called `secret.txt` and a scrap of paper with instructions:

THE BUY SUBMARINE
SIX WILL EGGS
SURFACE AND AT
SOME NINE MILK PM

`secret.txt` ↗

Run with:
`secret_messages < secret.txt > message1.txt 2> message2.txt`

`>` will redirect the Standard Output.

`2>` will redirect the Standard Error.

Your mission is to decode the two secret messages. Write your answers below.

Message 1

Message 2

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

TOP SECRET - SOLVED

We have reason to believe that the following program has been used in the transmission of secret messages:

```
#include <stdio.h>

int main()
{
    char word[10];
    int i = 0;
    while (scanf("%9s", word) == 1) {
        i = i + 1;
        if (i % 2)
            fprintf(stdout, "%s\n", word);
        else
            fprintf(stderr, "%s\n", word);
    }
    return 0;
}
```

We have intercepted a file called `secret.txt` and a scrap of paper with instructions:

THE BUY SUBMARINE
SIX WILL EGGS
SURFACE AND AT
SOME NINE MILK PM

`secret.txt` ↗

Run with:
`secret_messages < secret.txt > message1.txt a> message2.txt`

Your mission is to decode the two secret messages. Write your answers below.

Message 1

THE
SUBMARINE
WILL
SURFACE
AT
NINE
PM

Message 2

BUY
SIX
EGGS
AND
SOME
MILK



The Operating System Exposed

This week's interview:
Does the Operating System Matter?

Head First: Operating System - we're so pleased you have found time for us today.

O/S: Time-sharing - it's what I'm good at.

Head First: Now you've agreed to appear under conditions of anonymity, is that right.

O/S: Don't Ask/Don't Tell. Just call me O/S.

Head First: Does it matter what kind of O/S you are?

O/S: A lot of people get pretty heated over which operating system to use. But for simple C programs, we all behave pretty much the same way.

Head First: Because of the C Standard Library?

O/S: Yeah - if you're writing C then the basics are the same everywhere. Like I always say - we're all the same with the lights out. Know what I'm saying?

Head First: Oh - of course. Now you are in charge of loading programs into memory?

O/S: I turn them into processes - that's right.

Head First: Important job?

O/S: I like to think so. You can't just throw a program into memory and let it struggle, you know? There's a whole bunch of setup. I need to allocate memory for them and connect them to their standard data streams so they can use things like displays and keyboards.

Head First: Like you just did for the geo2json program?

O/S: That guy's a real tool.

Head First: Oh - I'm sorry.

O/S: No - I mean he's a real tool - a simple text based program.

Head First: Ah - I see. And do you deal with a lot of tools?

O/S: Ain't that life? It depends on the operating system. UNIX-style systems use a lot of tools to get the work done. Windows uses them less, but they're still important.

Head First: Creating small tools that work together is almost a philosophy, isn't it?

O/S: It's a way of life. Sometimes when you've got a big problem to solve, it can be easier to break it down into a set of simpler tasks.

Head First: Then write a tool for each task?

O/S: Exactly. Then use the operating system - that's me - to connect the tools together.

Head First: Are there any advantages to that approach?

O/S: The big one is simplicity. If you have a set of small programs, they are easier to test. The other thing is that once you've built a tool you can use it in other projects.

Head First: Any downsides?

O/S: Well - tools don't look that great. They work on the command line usually, so they don't have a lot of what you might call Eye Appeal.

Head First: Does that matter?

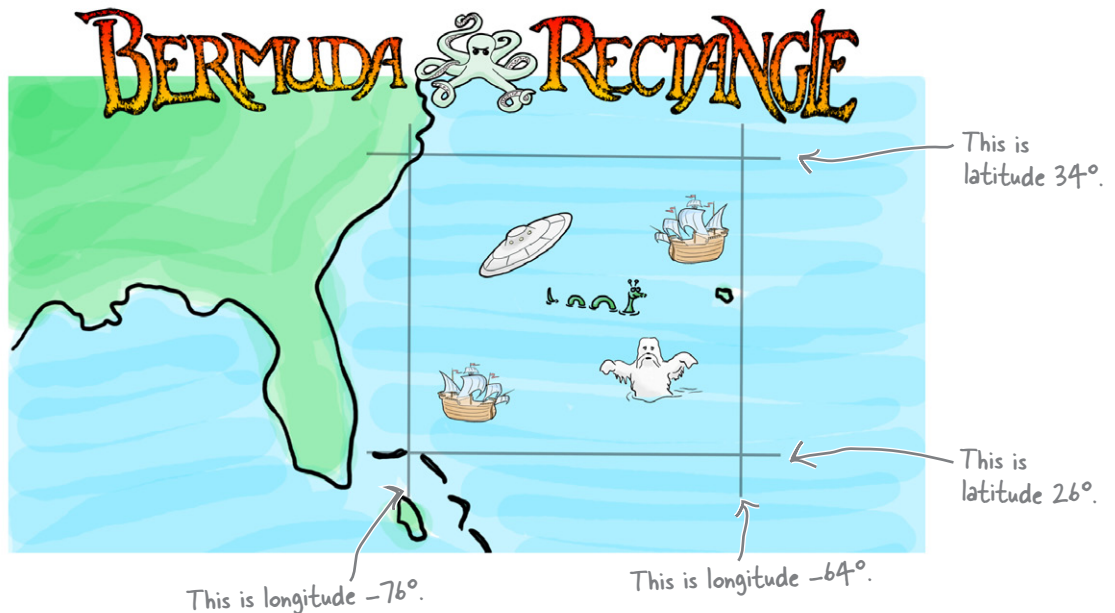
O/S: Not so much as you'd think. So long as you have a set of solid tools to do the important work, you can always connect them to a nice interface, whether it's a Desktop application or a web site. But - hey look at the time. Sorry I got to preempt you.

Head First: Oh well thank you O/S it's been a pleas... ZZZZZZ.....

Small tools are flexible

One of the great things about small tools is their flexibility. If you write a program that does one thing really well, chances are you will be able to use it in lots of contexts. If you create a program that can search for text inside a file, say, then chances are you going to find that program useful in more than one place.

For example - think about our `geo2json` tool. We created it to help display cycling data, right? But there's no reason we can't use it for some other purpose... like investigating... the



To see how flexible our tool is, let's use it for a completely different problem. Instead of just displaying data on a map, let's try to use it for something a little more complex. Let's say we want to read in a whole set of GPS data like before, but instead of just displaying everything, let's just display the information that falls inside the Bermuda Rectangle.

That means we will only display data that matches these conditions:

```
((latitude > 26) && (latitude < 34))
```

```
((longitude > -76) && (longitude < -64))
```

So where do we need to begin?


We don't want to change the geo2json tool

Our `geo2json` tool displays all of the data it's given. So what should we do? Should we *modify* `geo2json` so that it *exports* data and also *checks* the data?

Well, we *could* - but remember, a small tool:

Does one job and does it well

We don't really want to modify the `geo2json` tool because we want it to do just one task. If we make the program do something more complex, we'll cause problems for our users who expect the tool to keep working in exactly the same way.



I really don't want to filter data. I need to keep on displaying everything.

So if don't want to change the `geo2json` what should we do?



Tips for Designing Small Tools

Small tools like `geo2json` all follow these design principles:

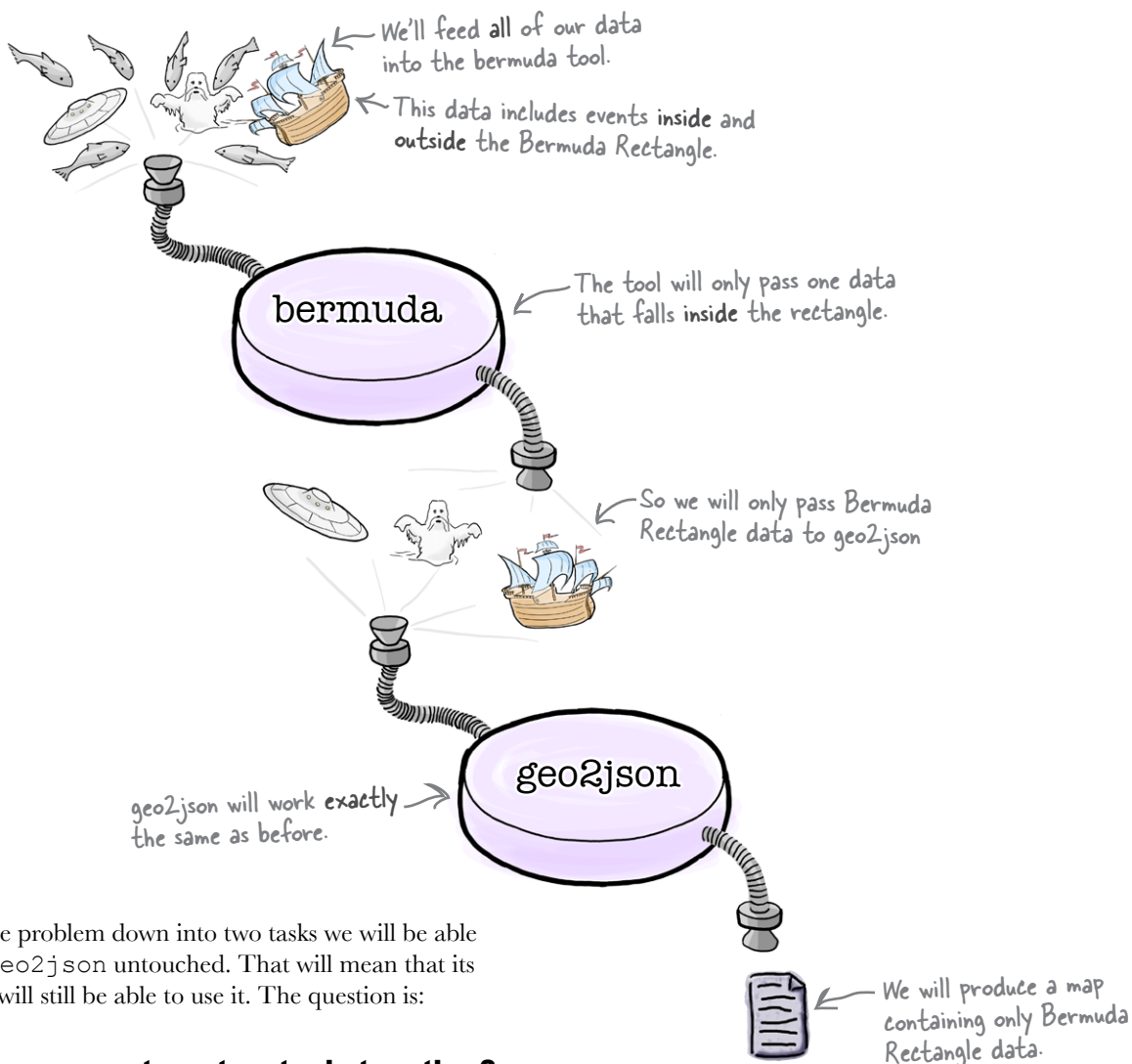
- * They can read data from the Standard Input stream
- * They can display data on the Standard Output stream
- * They deal with **text** data rather than obscure binary formats
- * They each perform **one simple task**

A different task needs a different tool

If we want to skip over the data that falls outside the Bermuda Rectangle, then we should build a separate tool that does just that.

So - we'll have **two** tools: a new **bermuda** tool that filters out data that is outside the Bermuda Rectangle, and then our original **geo2json** tool that will convert the remaining data for the map.

This is how we'll connect the program's together:



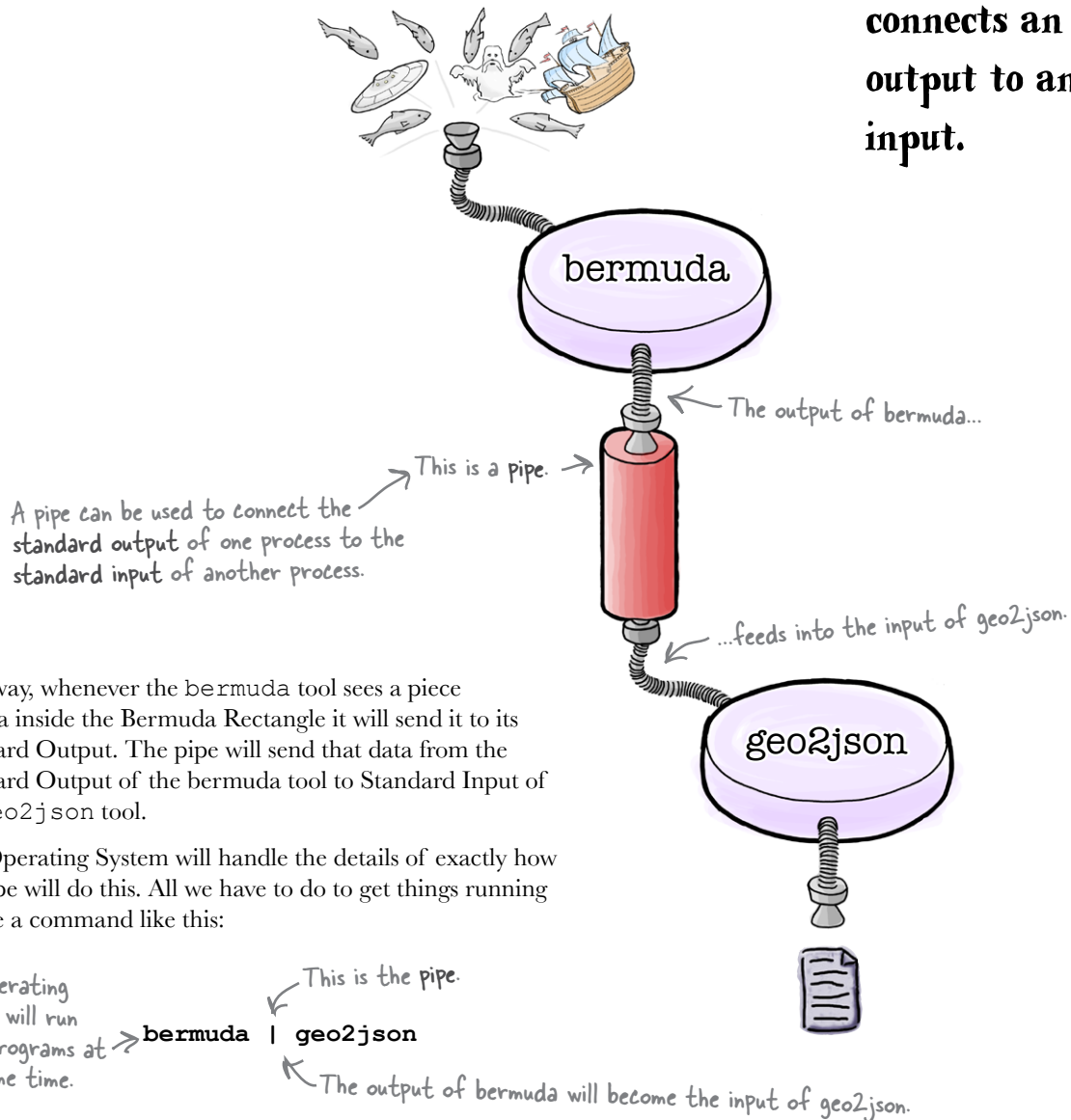
By splitting the problem down into two tasks we will be able to leave our **geo2json** untouched. That will mean that its current users will still be able to use it. The question is:

How will we connect our two tools together?

We'll connect our input and our output with a pipe

We've already seen how we can use redirection to connect the *Standard Input* and the *Standard Output* of a program files. But now we'll connect the **Standard Output** of the **bermuda** tool to the **Standard Input** of the **geo2json**, like this:

The **|** symbol is a pipe that connects an output to an input.



That way, whenever the bermuda tool sees a piece of data inside the Bermuda Rectangle it will send it to its Standard Output. The pipe will send that data from the Standard Output of the bermuda tool to Standard Input of the geo2json tool.

The Operating System will handle the details of exactly how the pipe will do this. All we have to do to get things running is issue a command like this:

The operating system will run both programs at the same time.

```
bermuda | geo2json
```

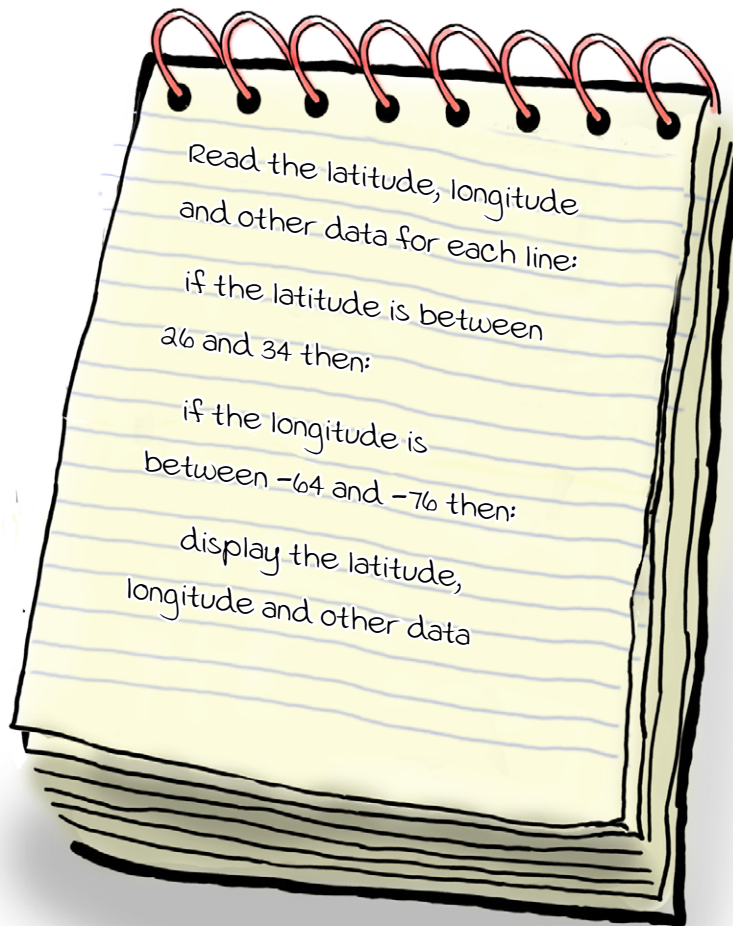
So now it's time we built the **bermuda** tool.

The bermuda tool

The bermuda tool will work in a very similar way to the geo2json tool - it will read through a set of GPS data, line by line, and then send data to the Standard Output.

But there will be two big differences. First, it won't send *every* piece of data to the Standard Output, just the lines that are inside the Bermuda Rectangle. The second difference is that the bermuda tool will always output data in the same CSV format used to store GPS data.

This is what the pseudo-code for the tool looks like:



Let's turn the pseudo-code into C.

Pool Puzzle



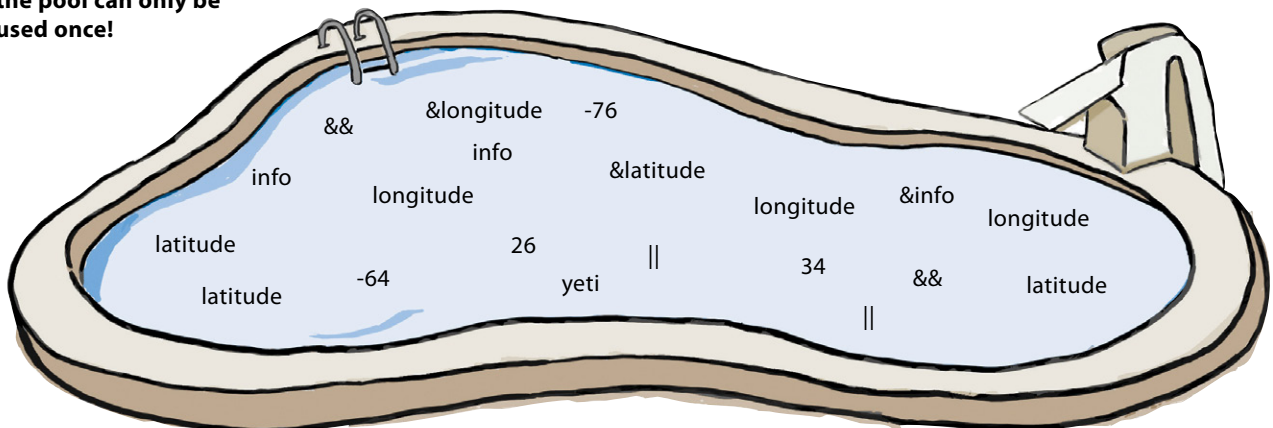
Your **goal** is to complete the code for the bermuda.c program. Take code snippets from the pool and place them into the blank lines below. You won't need to use all the snippets of code in the pool.

```
#include <stdio.h>

int main()
{
    float latitude;
    float longitude;
    char info[80];
    while (scanf("%f,%f,%79[^\n]", ..... , ..... , ..... ) == 3)
        if ((..... > ..... ) ..... (..... < ..... ))
            if ((..... > ..... ) ..... (..... < ..... ))
                printf("%f,%f,%s\n", ..... , ..... , ..... );

    return 0;
}
```

Note: each thing from the pool can only be used once!



Pool Puzzle Solution



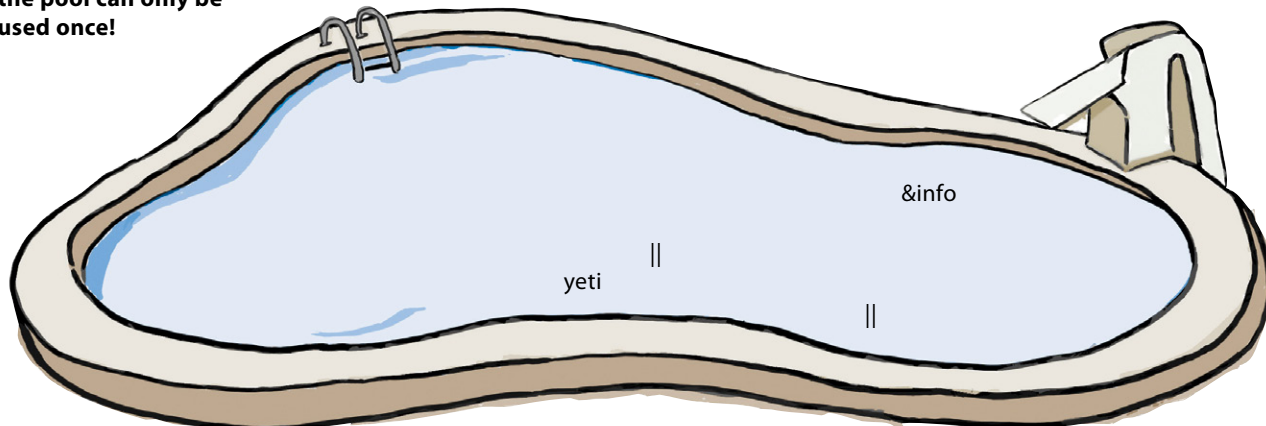
Your **goal** is to complete the code for the bermuda.c program. Take code snippets from the pool and place them into the blank lines below. You won't need to use all the snippets of code in the pool.

```
#include <stdio.h>

int main()
{
    float latitude;
    float longitude;
    char info[80];
    while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3)
        if ((latitude > 26) && (latitude < 34))
            if ((longitude > -76) && (longitude < -64))
                printf("%f,%f,%s\n", latitude, longitude, info);

    return 0;
}
```

Note: each thing from the pool can only be used once!



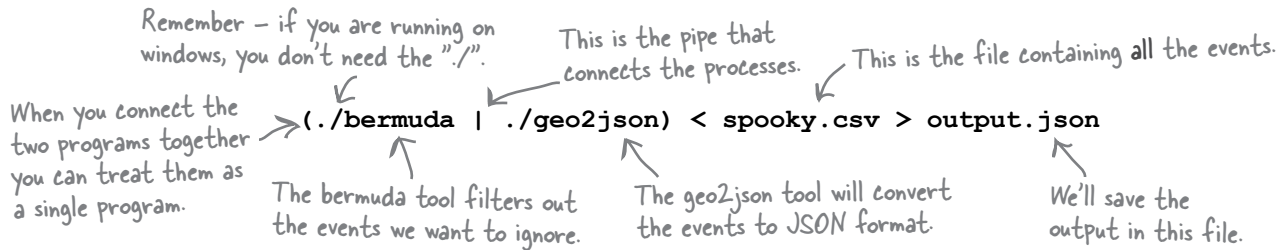


— TEST DRIVE

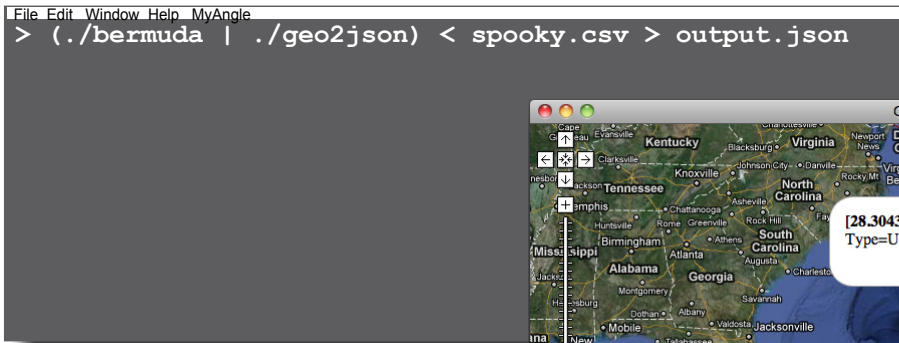
Now we've completed the bermuda tool, it's time to use it with the geo2json tool and see if we can map any weird occurrences inside the Bermuda Rectangle.

Once we've compiled both of the tools we can fire up a console and then run the two programs together like this:

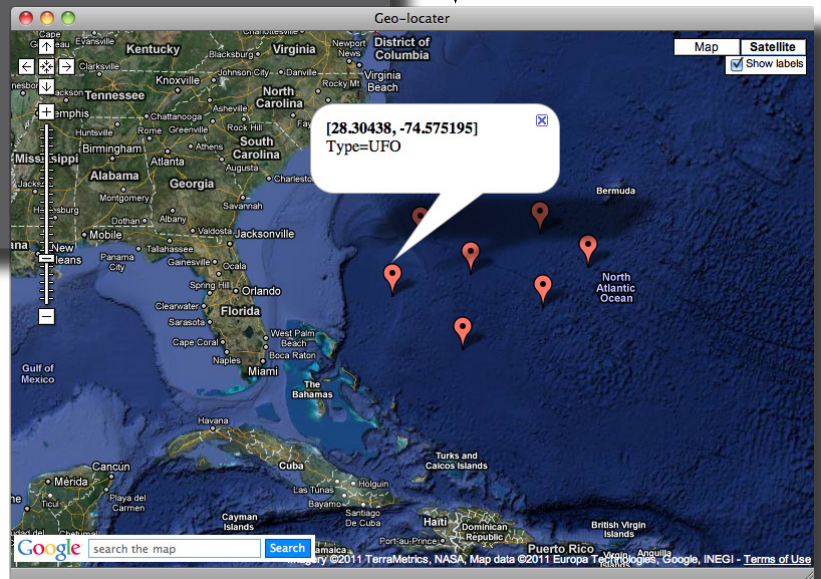
You can download the spooky.csv file at:
<http://oreillyhfc.appspot.com/spooky.csv>



By connecting the two programs together with a pipe, we can treat these two separate programs as if they were a single program - so we can redirect the standard input and standard output like we did before.



Excellent - the program works!



there are no
Dumb Questions

Q: Why is it important that small tools use the Standard Input and Standard Output streams?

A: Because by using the Standard Streams it makes it easier to connect tools together with pipes.

Q: Why does that matter?

A: Small tools usually don't solve an entire problem on their own - just a small technical problem, like converting data from one format to another. But if you can combine them together then you can solve large problems.

Q: What actually is a pipe?

A: The exact details are down to the operating system. Pipes might be made from sections of memory or temporary files. The important thing is that accept data in one end, and send the data out of the other in sequence.

Q: So if two programs are piped together, does the first program have to finish running before the second program can start?

A: No. Both of the programs will run at the same time and as output is produced by the first program it can be consumed by the second program.

Q: Why do small tools use text?

A: It's the most open format. If a small tool uses text it means that any other programmer can easily read and understand the output by just using a text editor. Binary formats are normally obscure and hard to understand.

Q: Can I connect several programs together with pipes?

A: Yes - just add more '|' between each program name. A series of connected processes is called a pipeline.

Q: If several processes are connected together with pipes and then I use '>' and '<' to redirect the Standard Input and Output, which processes will have their Input and Output redirected?

A: The '<' will send a file's contents to the first process in the pipeline. The '>' will capture the Standard Output from the last process in the pipeline.



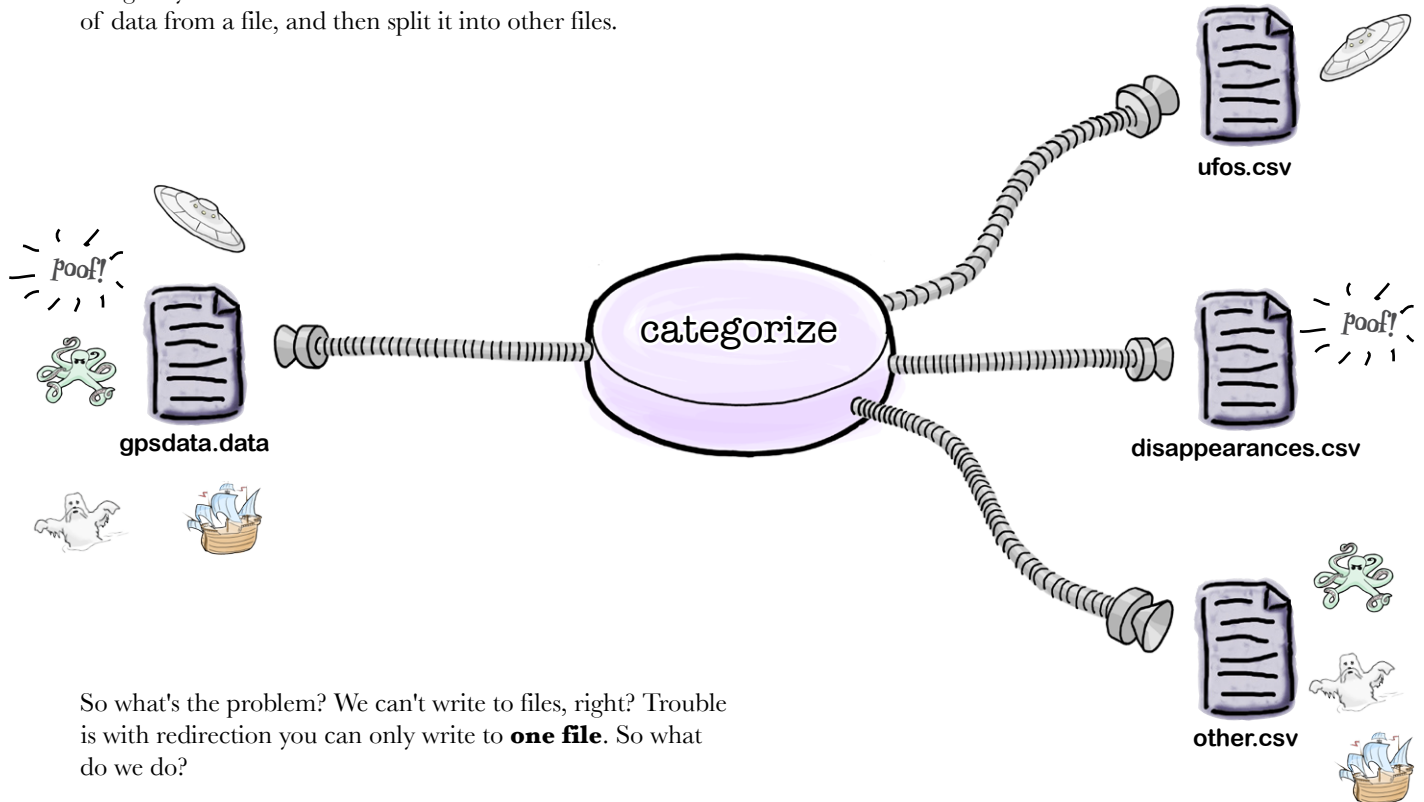
BULLET POINTS

- If you want to perform a different task, consider writing a separate small tool.
- Design tools to work with Standard Input and Standard Output.
- Small tools normally read and write text data.
- You can connect the Standard Output of one process to the Standard Input of another process using a **pipe**.

But what if we want to output to more than one file?

So we've looked at how to read data from one file and write to another file using redirection - but what if the program needs to do something a little more complex - like send data to **more than one file**?

Imagine you need to create another tool that will read a set of data from a file, and then split it into other files.




So what's the problem? We can't write to files, right? Trouble is with redirection you can only write to **one file**. So what do we do?

Roll your own data stream

When a program runs the operating system gives it three data streams: the Standard Input, the Standard Output and the Standard Error. But sometimes you need to create other data streams on the fly.

The good news is that the operating system doesn't limit you to the data streams you are dealt when the program starts. You can roll your own as the program runs.

Each data is represented by a pointer to a FILE and you can create a new data stream using the `fopen()` function:



```
FILE* in_file = fopen("input.txt", "r");
```

This will create a data stream to read from a file. →

This is the name of the file. ↓

This is the mode - "r" means "read". ←

```
FILE *out_file = fopen("output.txt", "w");
```

This will create a data stream to write to a file. →

This is the name of the file. ↓

This is the mode - "w" means "write". ←

The `fopen()` function takes **two** parameters - a *filename* and a *mode*. The mode can be **"w"** to write to a file, **"r"** to read from a file or **"a"** to append data to the *end* of a file.

Once you've created a data stream you can print to it using `fprintf` - just like before. But what if you need to read from a file? Well there's also a `fscanf()` function to help you do that to:

```
fprintf(out_file, "Don't wear %s with %s", "red", "green");
```

```
fscanf(in_file, "%79[^\n]\n", sentence);
```

Finally, when you're finished with a data stream, you need to *close* it. The truth is that all data streams are automatically closed when the program ends, but it's still a good idea to always close the data stream yourself:

```
fclose(in_file);
fclose(out_file);
```

Let's try this out now.

The data stream mode is:
"w" = write,
"r" = read or
"a" = append.



Sharpen your pencil

This is the code for a program to read all of the data from a GPS file and then write the data into one of three other files. See if you can fill in the missing blanks:

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

int main()
{
    char line[80];
    FILE* in = fopen("gpsdata.data", ..... );
    FILE* file1 = fopen("ufos.csv", ..... );
    FILE* file2 = fopen("disappearances.csv", ..... );
    FILE* file3 = fopen("others.csv", ..... );
    while ( ..... (in, "%79[^\n]\n", line) == 1) {
        if (strstr(line, "UFO"))
            ..... (file1, "%s\n", line);
        else if (strstr(line, "Disappearance"))
            ..... (file2, "%s\n", line);
        else
            ..... (file3, "%s\n", line);
    }
    ..... (file1);
    ..... (file2);
    ..... (file3);
    return 0;
}
```

Sharpen your pencil Solution



This is the code for a program to read all of the data from a GPS file and then write the data into one of three other files. See if you can fill in the missing blanks:

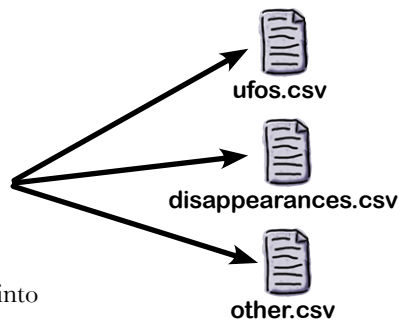
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char line[80];
    FILE* in = fopen("gpsdata.data", ..... );
    FILE* file1 = fopen("ufos.csv", ..... );
    FILE* file2 = fopen("disappearances.csv", ..... );
    FILE* file3 = fopen("others.csv", ..... );
    while ( ..... (in, "%79[^\n]\n", line) == 1) {
        if (strstr(line, "UFO"))
            ..... (file1, "%s\n", line);
        else if (strstr(line, "Disappearance"))
            ..... (file2, "%s\n", line);
        else
            ..... (file3, "%s\n", line);
    }
    ..... (file1);
    ..... (file2);
    ..... (file3);
    return 0;
}
```

The program runs, but...

If you compile and run the program with:

```
gcc categorize.c -o categorize && ./categorize
```



the program will read the `gpsdata.csv` and split out the data, line by line into three other files, `ufos.csv`, `disappearances.csv`, and `other.csv`.

That's great, but what if a user wanted to split the data up differently? What if they wanted to search for different words, or write to different files? Could they do that without needing to recompile the program each time?

There's more to main() than we said

The thing is, any program you write will need to give the user the ability to change the way it works. If it's a GUI program, you will probably need to give it preferences. And if it's a command line program, like our categorize tool, it will need to give the user the ability to pass it **command line arguments**:

This is the first word to filter for. All of the mermaid data will be stored in this file. This means we want to check for Elvis. Everything else goes into this file.

```
./categorize mermaid mermaid.csv Elvis elvises.csv the_rest.csv
```

All the Elvis sightings will be stored here.

But how do we read command line arguments from **within the program**? So far, every time we have created a `main()` function, we've written it without any arguments. But the truth is, there are actually *two* forms of the `main()` function we can use. This is the second version:

```
int main(int argc, char* args[])
{
    .... Do stuff....
}
```

The `main()` function can read the command line arguments as an **array of strings**. Actually, of course, because C doesn't really have strings built-in, it reads them as *an array of character pointers to strings*. Like this:

```
"/categorize" "mermaid" "mermaid.csv" "Elvis" "elvises.csv" "the_rest.csv"
```

This is args[0]. This is args[1]. This is args[2]. This is args[3]. This is args[4]. This is args[5].

The first argument is actually the name of the program being run.

Like any array in C, we need some way of knowing how long the array is. That's why the `main()` function has two parameters. The `argc` value is a count of the number of elements in the array.

Command line arguments really give your program a lot more flexibility and it's worth thinking about which things you want your users to *tweak* at run time. It will make your program a lot more valuable to them.

OK - let's see how we can add a little flexibility to the categorize program.



Watch it!

The first argument contains the name of the program as it was run by the user.

That means that the first proper command line argument is `args[1]`.



Code Magnets

This is a modified version of the `categorize` program that can read the keywords to search for and the files to use from the command line. See if you can fit the correct magnets into the correct slots.

The program is run using

```
./categorize mermaid mermaid.csv Elvis elvises.csv the_rest.csv
```

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

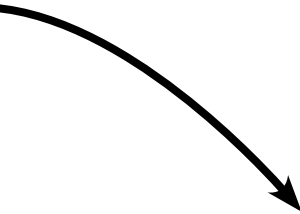
int main(int argc, char* args[])
{
    char line[80];

    if ( ..... != ..... ) {
        fprintf(stderr, "You need to give 5 arguments\n");
        return 1;
    }
    FILE* in = fopen("gpsdata.data", "r");

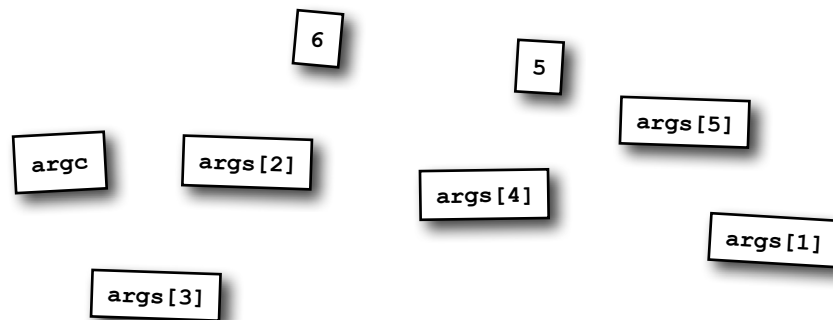
    FILE* file1 = fopen( ....., "w");

    FILE* file2 = fopen( ....., "w");

    FILE* file3 = fopen( ....., "w");
```



```
while (fscanf(in, "%79[^\n]\n", line) == 1) {  
  
    if (strstr(line, .....))  
        fprintf(file1, "%s\n", line);  
  
    else if (strstr(line, .....))  
        fprintf(file2, "%s\n", line);  
    else  
        fprintf(file3, "%s\n", line);  
}  
fclose(file1);  
fclose(file2);  
fclose(file3);  
return 0;  
}
```





Code Magnets Solution

This is a modified version of the `categorize` program that can read the keywords to search for and the files to use from the command line. See if you can fit the correct magnets into the correct slots.

The program is run using

```
./categorize mermaid mermaid.csv Elvis elvises.csv the_rest.csv
```

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

int main(int argc, char* args[])
{
    char line[80];

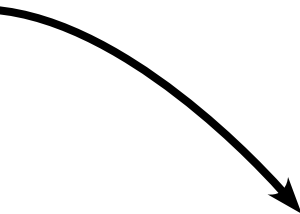
    if ( argc != 6 ) {
        fprintf(stderr, "You need to give 5 arguments\n");
        return 1;
    }

    FILE* in = fopen("gpsdata.data", "r");

    FILE* file1 = fopen( args[2] , "w");

    FILE* file2 = fopen( args[4] , "w");

    FILE* file3 = fopen( args[5] , "w");
```



```
while (fscanf(in, "%79[^\n]\n", line) == 1) {  
  
    if (strstr(line, args[1]))  
        fprintf(file1, "%s\n", line);  
  
    else if (strstr(line, args[3]))  
        fprintf(file2, "%s\n", line);  
    else  
        fprintf(file3, "%s\n", line);  
}  
fclose(file1);  
fclose(file2);  
fclose(file3);  
return 0;  
}
```

5



TEST DRIVE

OK - let's try out the new version of the code. We'll need a test data file called `gpsdata.csv`.

```
30.685163,-68.137207,Type=Yeti
28.304380,-74.575195,Type=UFO
29.132971,-71.136475,Type=Ship
28.343065,-62.753906,Type=Elvis
27.868217,-68.005371,Type=Goatsucker
30.496017,-73.333740,Type=Disappearance
26.224447,-71.477051,Type=UFO
29.401320,-66.027832,Type=Ship
37.879536,-69.477539,Type=Elvis
22.705256,-68.192139,Type=Elvis
27.166695,-87.484131,Type=Elvis
```



gpsdata.data

Now we'll need to run the `categorize` program with a few command line arguments saying what text to look for and what file names to use:

```
File Edit Window Help ThankYouVeryMuch
> categorize UFO aliens.csv Elvis elvises.csv the_rest.csv
```

When the program runs, the following files are produced:


```
28.304380,-74.575195,Type=UFO
26.224447,-71.477051,Type=UFO
```



aliens.csv

```
30.685163,-68.137207,Type=Yeti
29.132971,-71.136475,Type=Ship
27.868217,-68.005371,Type=Goatsucker
30.496017,-73.333740,Type=Disappearance
29.401320,-66.027832,Type=Ship
```



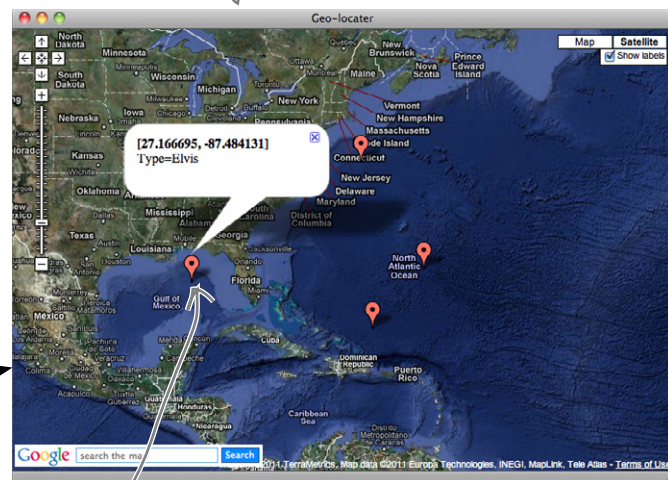
the_rest.csv

```
28.343065,-62.753906,Type=Elvis
37.879536,-69.477539,Type=Elvis
22.705256,-68.192139,Type=Elvis
27.166695,-87.484131,Type=Elvis
```



elvises.csv

If we run elvises.txt through geo2json we can display it on a map.



Elvis has left the building.



Safety Check

Although at Head First Labs we never make mistakes (cough) it's important in real-world programs to check for problems when you open a file for reading or writing. Fortunately if there's a problem opening a data stream, the `fopen()` function will return the value 0. That means if you want to check for errors, you should change code like:

```
FILE* in = fopen("i_dont_exist.txt", "r");
```

to this:

```
FILE* in;
if (!(in = fopen("dont_exist.txt", "r"))) {
    fprintf(stderr, "Can't open the file.\n");
    return 1;
}
```

Overheard at the Head First Pizzeria



Chances are any program you write is going to need options. If you create a chat program, it's going to need preferences. If you write a game, the user will want to change the shape of the blood spots. And if you're writing a command line tool, you are probably going to need to add **command-line options**.

Command-line options are the little switches you often see with command line tools:

`ps -ae` ← Display all the processes, including their environments.

`tail -f logfile.out` ← Display the end of the file, but wait for new data to be added to the end of the file.

Let the library do the work for you

Many programs use command line options - so there's a special library function you can use to make dealing with options a little easier. It's called `getopt()` and each time you call it, it returns the next option it finds on the command line.

Let's see how it works. Imagine we have a program that can take a set of different options:

Use 4 engines. → ← Awesomeness mode enabled.
rocket_to -e 4 -a Brasilia Tokyo London

This program needs one option that will take a value (-e = engines) and another that is simply *on* or *off* (-a = awesomeness). You can handle these options by calling `getopt()` in a loop like this:

```
#include <unistd.h>
...
while ((ch = getopt(argc, args, "ae:")) != EOF)
  switch(ch) {
    ...
    case 'e':
      engine_count = optarg;
    ...
  }
argc -= optind;
args += optind;
```

You will need to include this header. → `#include <unistd.h>`

... →

The code to handle each option goes here. → `while ((ch = getopt(argc, args, "ae:")) != EOF)`

We're reading the argument for the 'e' option here. → `case 'e':`

These final two lines make sure we skip past the options we read. → `argc -= optind;`
 → `args += optind;`

This means "The a option is valid, so is the e option." → `"ae:"`

The ":" means that the e option needs an argument. → `ae:`

`optind` stores the number of strings read from the command line to get past the options. → `optind`

Inside the loop we have a `switch` statement to handle each of the valid options. The string "ae:" tells the `getopt()` function that "a" and "e" are valid options. The "e" is followed by a ":" to tell `getopt()` that the -e needs to be followed by an extra argument. `getopt()` will point to that argument with the `optarg` variable.

When the loop finishes, we tweak the `args` and `argc` variables to skip past all of the options and get to the main command line arguments. That will make our `args` array look like this:

→ **Brasilia Tokyo London** ←
 This is `args[0]`. This is `args[1]`. This is `args[2]`.

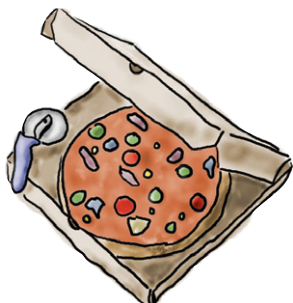


Watch it!

After processing the arguments, the 0th

argument will no longer be the program name.

args[0] will instead point to the first command line argument that follows the options.



Pizza Pieces

Looks like someone's been taking a bit out of the pizza code. See if you can replace the pizza slices and rebuild the `order_pizza` program.

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

int main(int argc, char* args[])
{
    char* delivery = "";
    int thick = 0;
    int count = 0;
    char ch;

    while ((ch = getopt(argc, args, "d .....")) != EOF)
        switch (ch) {
            case 'd':
                ..... = ..... ;
                break;
            case 't':
                ..... = ..... ;
                break;
            default:
                fprintf(stderr, "Unknown option: '%s'\n", optarg);

                return ..... ;
        }
}
```

```

argc -= optind;
args += optind;

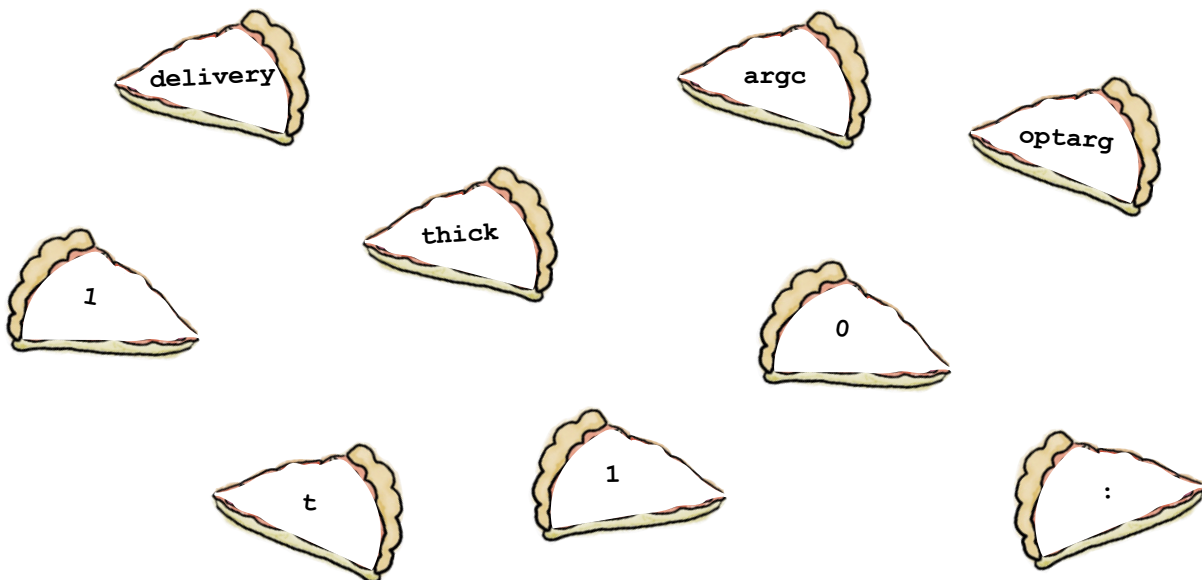
if (thick)
    puts("Thick crust.");

if (delivery[0])
    printf("To be delivered %s.\n", delivery);

puts("Ingredients:");

for (count = .....; count < .....; count++)
    puts(args[count]);
return 0;
}

```





Pizza Pieces Solution

Looks like someone's been taking a bit out of the pizza code. See if you can replace the pizza slices and rebuild the `order_pizza` program.

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

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

```
{
```

```
    char* delivery = "";
```

```
    int thick = 0;
```

```
    int count = 0;
```

```
    char ch;
```

The 'd' is followed by a colon because it takes an argument.

```
while ((ch = getopt(argc, args, "d...t...")) != EOF)
```

```
    switch (ch) {
```

```
        case 'd':
```



We'll point the delivery variable to the argument supplied with the 'd' option.

```
        break;
```

```
        case 't':
```



Remember - in C setting something to 1 is equivalent to setting it to true.

```
        break;
```

```
    default:
```

```
        fprintf(stderr, "Unknown option: '%s'\n", optarg);
```

```
    return ... 1 ... ;
}
```

```

argc -= optind;
args += optind;

if (thick)
    puts("Thick crust.");

if (delivery[0])
    printf("To be delivered %s.\n", delivery);

puts("Ingredients:");

```

```

for (count = ...; count < ...; count++)
    puts(args[count]);
return 0;
}

```

After processing the options, the first ingredient is args[0].

We'll keep looping while we're less than argc.



TEST DRIVE

Now we can try out the pizza order program:

We compile the program. →

We're not using any options the first couple of times we call it. →

Then we try out the 'd' option and give it an argument of 'now'. →

Then the 't' option. Remember - the 't' option doesn't take any arguments. →

Finally we'll try skipping the argument for 'd' - it creates an error. →

```

File Edit Window Help Anchovies?
> gcc order_pizza.c -o order_pizza
> ./order_pizza Anchovies
Ingredients:
Anchovies
> ./order_pizza Anchovies Pineapple
Ingredients:
Anchovies
Pineapple
> ./order_pizza -d now Anchovies Pineapple
To be delivered now.
Ingredients:
Anchovies
Pineapple
> ./order_pizza -d now -t Anchovies Pineapple
Thick crust.
To be delivered now.
Ingredients:
Anchovies
Pineapple
> ./order_pizza -d
order_pizza: option requires an argument -- d
Unknown option: '(null)'
>

```

It works!

Well - we've covered a lot in this chapter. We got deep into the Standard Input, Standard Output and Standard Error. Learned how to talk to files using redirection and our own custom data streams. Finally we learned how to deal with command line arguments and options.

A lot of C programmers spend their time creating small tools, and most of the small tools you see in operating systems like Linux are written in C. If you're careful in how you design them, and if you make sure that you design tools that **do one thing** and **do that one thing well**, you're well on course to becoming a kick-ass C coder.

there are no
Dumb Questions

Q: Can I combine options like "-td now" instead of "-d now -t"?

A: Yes you can. The `getopts()` function will handle all of that for you.

Q: What about changing the order of the options?

A: Yes - it won't matter if you type in "-d now -t" or "-t -d now" or "-td now".

Q: So if the program sees a value on the command line beginning with a "-" it will treat it as an option?

A: If it reads it before it gets to the main command line arguments it will, yes.

Q: But what if I want to pass negative numbers as command line arguments like "set_temperature -c -4"? Won't it think that the 4 is an option not an argument?

A: To avoid ambiguity, you can split your **main** arguments from the options using "--". So you would write "set_temperature -c -- -4". `getopts()` will stop reading options when it sees the "--", so the rest of the line will be read as simple arguments.



BULLET POINTS

- There are two versions of the `main()` function - one with, and one without command line arguments.
- Command line arguments are passed to `main()` as an argument count and an array of pointers to the argument strings.
- Command line options are command line arguments prefixed with '-'
- The `getopt()` function helps you deal with command line options.
- You define valid options by passing a string to `getopt()` like "**ae:**"
- A ':' following an option in the string means that the option takes an additional argument.
- `getopt()` will record the options argument using the **optarg** variable.
- After you have read all of the options you should skip past them using the **optind** variable.



Your C Toolbox

You've got Chapter 3 under your belt and now you've added small tools to your toolbox. For a complete list of tooltips in the book, see Appendix X.

C functions like `printf` and `scanf` use the Standard Output and Standard Input to communicate.

The Standard Output goes to the display by default.

The Standard Error is a separate output data stream intended for error messages.

The Standard Input reads from the keyboard by default.

You can change where the Standard Input and Output are connected to using Redirection.

You can print to the Standard Error using `fprintf(stderr,...)`

Command line arguments are passed to `main()` as an array of string pointers

You can create custom data streams with `fopen("filename", mode)`

The mode can be "w" to write, "r" to read or "a" to append.

The `getopts()` function makes it easier to read command line options.

4 using multiple source files

Break it down, Build it up



If you create a big program, you don't want a big source file.

Can you imagine how difficult and time-consuming a single source file for an enterprise level program would be to maintain? In this chapter, you'll learn how C allows you to break your source code into **small manageable chunks** and then rebuild them into **one huge program**. Along the way, you'll learn a bit more about **data-type subtleties**, and get to meet your new best friend: **make**.



The total number of components in the rocket

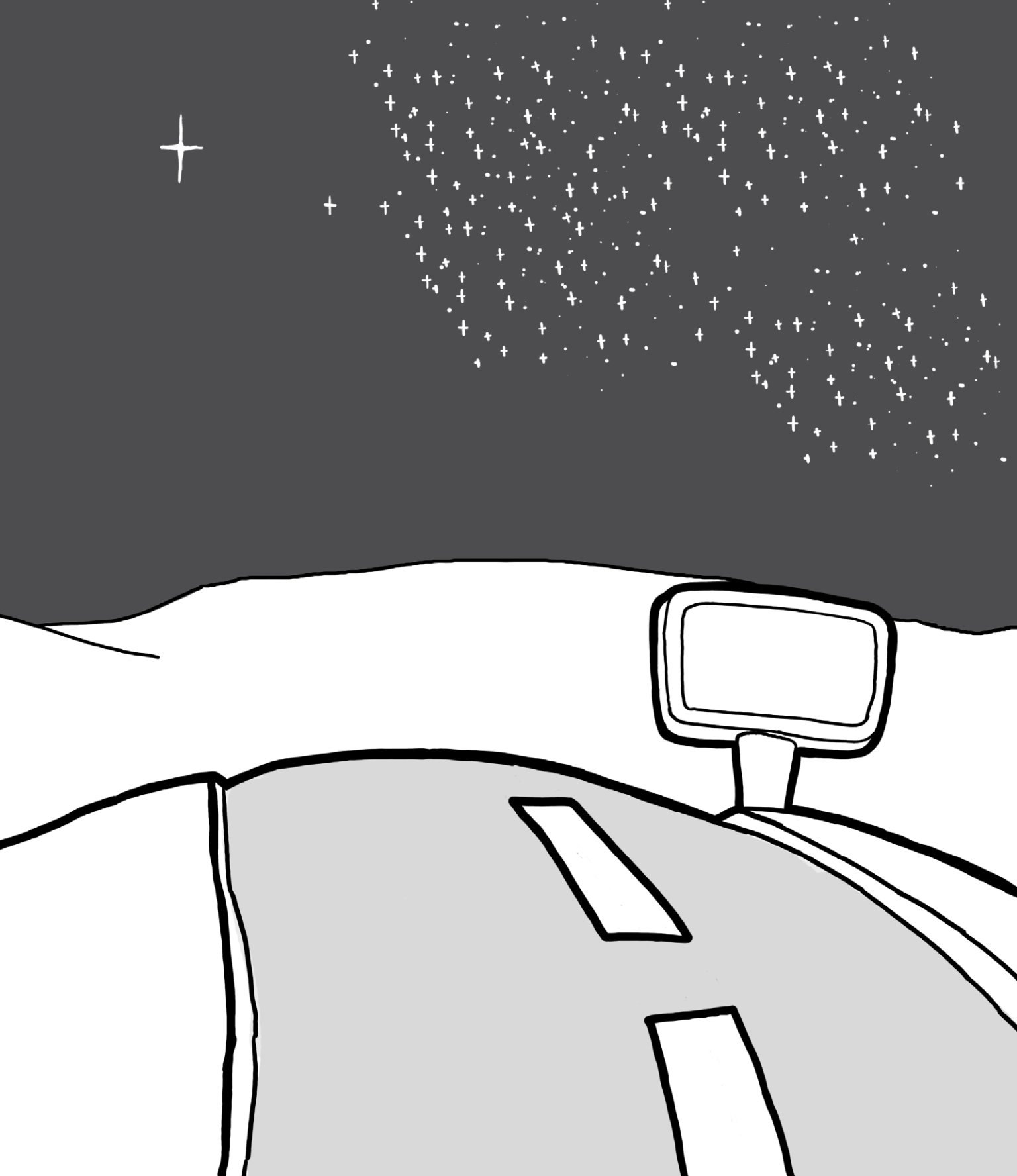
The amount of fuel the rocket will need (gallons)



Guess the Data Type

C can handle quite a few different types of data: characters and whole numbers, floating point values for every day values, and floating point numbers for really precise scientific calculations. You can see a few of these data-types listed on the opposite page. See if you can figure out which data-types were used in the each of the examples.

Remember: each example uses a different data-type.





The total number of components in the rocket

int

The amount of fuel the rocket will need (gallons)

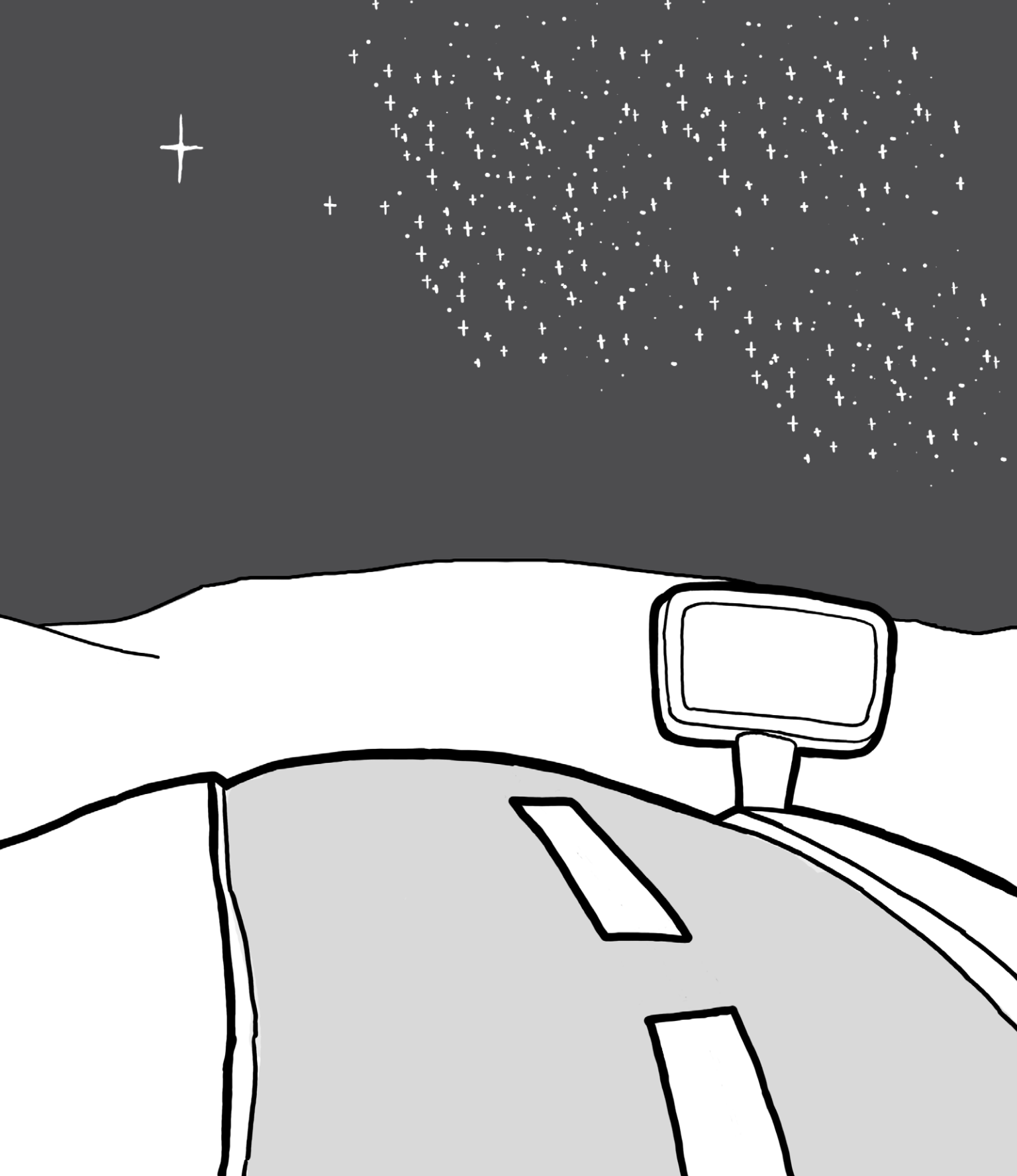
float



Guess the Data Type Solution

C can handle quite a few different types of data: characters and whole numbers, floating point values for every day values, and floating point numbers for really precise scientific calculations. You can see a few of these data-types listed on the opposite page. See if you can figure out which data-types were used in the each of the examples.

Remember: each example uses a different data-type.



Your quick guide to data types

char

Each character is stored in the computer's memory as a character code. And that's just a number. So when the computer sees 'A', to the computer it is just the same as seeing the literal number 65.

← 65 is the ASCII code for 'A'.

int

If you need to store a whole number, you can generally just use an `int`. On most machines an `int` can store numbers up to a few million.

short

But sometimes you want to save a little memory. Why use an `int` if you just want to store numbers up to few hundreds or thousands? That's what a `short` is for. A `short` number usually takes up about half the space of an `int`.

long

Yes - but what if you want to store a **really large count**? That's what the `long` data-type was invented for. On some machines the `long` data-types takes up *twice* the memory of an `int` and it can hold numbers up in the **billions**. But because most computers can deal with really large `ints`, on a lot of machines the `long` data-type is *exactly the same size* as an `int`.

float

The `float` data-type is the basic data-type for storing floating point numbers. For most everyday floating point numbers - like the amount of fluid in your orange frappe mochaccino - you can use a `float`.

double

Yes - but what if you want to get really **precise**? If you want to perform calculations that are accurate to a very large number of **decimal places** then you might want to use a `double`. A `double` takes up twice the memory of a `float`, and it uses that extra space to store numbers that are *larger and more precise*.

Don't put something big into something small

When you are passing around values you need to be careful that the type of the value matches the type of the variable you are going to store it in.

Different datatypes use different amounts of memory. So you need to be careful that you don't try to store a value that's too large for the amount of space allocated to a variable. `short` variables take up less memory than `ints` and `ints` take up less memory than `longs`.

Now there's no problem storing a `short` value inside an `int` or a `long` variable. There will plenty of space in memory and your code will work correctly:

```
short x = 15;
int y = x;
printf("The value of y = %i\n", y);
```

This will say that $y = 15$.

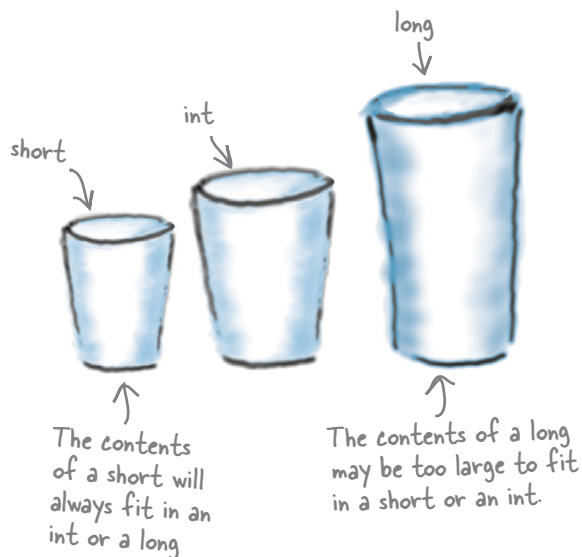
The problems start to happen if you go the other way round: if, say, you try to store an `int` value into a `short`.

```
int x = 100000;
short y = x;
print("The value of y = %hi\n", y);
```

'%hi' is the proper code to format a short value.

Sometimes the compiler will be able to spot that you are trying to store a really big value into a small variable, and then give you a warning. But a lot of the time the compiler won't be smart enough and it will compile the code without complaining. But then, when you come to run the code the computer won't be able to store a number 100,000 into a `short` variable. The computer will fit in as many 1s and 0s as it can fit, but the number that ends up stored inside the `y` variable will be *very different* from the one you sent it:

The value of `y` = -31072



Geek Bits

So why did putting a large number into a `short` go negative? Numbers are stored in binary. This is what 100,000 looks like in binary:

```
x <- 0001 1000 0110 1010 0000
```

But when the computer tried to store that value into a `short` the computer only allowed it a couple of bytes of storage. The program stored just the *left hand side* of the number:

```
y <- 1000 0110 1010 0000
```

Signed values in binary begin with a 1 in highest bit are treated as negative numbers. And this shortened value is equal to this in decimal:

```
-31072
```

Use casting to put floats into whole numbers

What do you think this piece of code will display?

```
int x = 7;
int y = 2;
float z = x / y;
printf("z = %f\n", z);
```

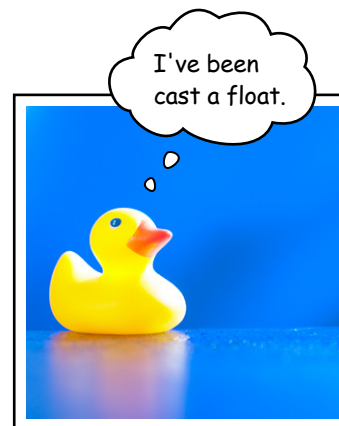
The answer? **3.0000**. Why is that? Well `x` and `y` are both integers and if you divide integers you always get a rounded off whole number - in this case **3**.

What do you do if you want to perform calculations on whole numbers and you want to get floating point results? You could store the whole numbers into float variables first, but that's a little wordy. Instead you can use a **cast** to convert the numbers on the fly:

```
int x = 7;
int y = 2;
float z = (float)x / (float)y;
printf("z = %f\n", z);
```

The **(float)** will *cast* an integer value into a `float` value. The calculation will then work just as if you were using floating point values the entire time. In fact, if the compiler sees you are adding, subtracting, multiplying or dividing a floating-point value with a whole number, it will automatically cast the numbers for you. That means you can cut down the number of explicit casts in your code:

```
float z = (float)x / y; ← The compiler will automatically cast y to a float.
```



You can put some other keywords before data-types to change the way that the numbers are interpreted:

unsigned

The number will always be positive. Because it doesn't need to worry about recording negative numbers, unsigned numbers can store larger numbers. So an unsigned int stores numbers from 0 to a maximum value that is about twice as large as the maximum number that can be stored inside an int. There's also a signed keyword, but you almost never see it, because all data-types are signed by default.

```
unsigned char c;
```

↑ This will probably store numbers from 0 to 255.

long

That's right - you can prefix a data-type with the word long and make it longer. So a long int is a longer version of an int - which means it can store a larger range of numbers. And a long long is longer than a long. You can also use long with floating point numbers. But a long double doesn't store a greater range of numbers than a double - it just records a more precise value.

```
long double d;
```

↑ A really REALLY precise number.



Exercise

There's a new program helping the waiters bus tables at the Head First Diner. The code automatically totals a bill and adds sales tax to each item. See if you can figure out what needs to go in each of the blanks.

Note: There are several data-types that could be used for this program, but what data-types would you use for the kind of figures you'd expect?

```
#include <stdio.h>

.....total = 0.0;
.....count = 0;
.....tax_percent = 6;

.....add_with_tax(float f);
{
    .....tax_rate = 1 + tax_percent / 100 ..... ;
    total = total + (f * tax_rate);
    count = count + 1;
    return total;
}

int main()
{
    .....val;
    printf("Price of item: ");
    while (scanf("%f", &val) == 1) {
        printf("Total so far: %.2f\n", add_with_tax(val));
        printf("Price of item: ");
    }
    printf("\nFinal total: %.2f\n", total);
    printf("Number of items: %i\n", count);
    return 0;
}
```



Exercise Solution

There's a new program helping the waiters bus tables at the Head First Diner. The code automatically totals a bill and adds sales tax to each item. See if you can figure out what needs to go in each of the blanks.

Note: There are several data-types that could be used for this program, but what data-types would you use for the kind of figures you'd expect?

We need

a small
floating-point
number to
total the cash.

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

```
.....float..... total = 0.0;
.....short..... count = 0;
.....short..... tax_percent = 6;
```

There won't be many items on an order so we'll choose a short.

A small number like 6 won't need an int.

```
.....float..... add_with_tax(float f);
```

We're returning a small cash value, so it'll be a float.

A float will
be OK for
this fraction.

```
.....float..... tax_rate = 1 + tax_percent / 100 ..... ;
total = total + (f * tax_rate);
count = count + 1;
return total;
}
```

By adding ".0" we make the calculation work as a float. If we left it as 100, it would have returned a whole number.

$1 + \text{tax_percent} / 100;$
would return the value 1 because $6/100 == 0$ in integer arithmetic.

```
int main()
```

```
{
.....float..... val;
printf("Price of item: ");
while (scanf("%f", &val) == 1) {
    printf("Total so far: %.2f\n", add_with_tax(val));
    printf("Price of item: ");
}
printf("\nFinal total: %.2f\n", total);
printf("Number of items: %i\n", count);
return 0;
}
```

Each price will easily fit in a float.

there are no Dumb Questions

Q: Why are data types different on different operating systems? Wouldn't it be less confusing to make them all the same?

A: C uses different data types of different operating systems and processors because it allows it to make the most out of the hardware.

Q: In what way?

A: When C was first created, most machines were 8 bit. Now most machines are 32 or 64 bit. Because C doesn't specify the exact size of its data-types, it's been able to adapt over time. And as newer machines are created C will be able to make the most of them as well.

Q: What does 8 bit? and 64 bit actually mean?

A: Technically the bit size of a computer can refer to several things - such as the size of its CPU instructions, or the amount of data the CPU can read from memory. The bit-size is really the favored size of numbers that the computer can deal with.

Q: So what does that have to do with the size of ints and doubles?

A: If a computer is optimized best to work with 32 bit numbers, it makes sense if the basic data-type - the int - is set at 32 bits.

Q: I understand how whole numbers like ints work, but how are floats and numbers stored? How does the computer represent a number with a decimal point?

A: It's complicated. Most computers used a standard published by the IEEE (<http://tinyurl.com/6defkv6>).

Q: Do I really need to understand how floating point numbers work?

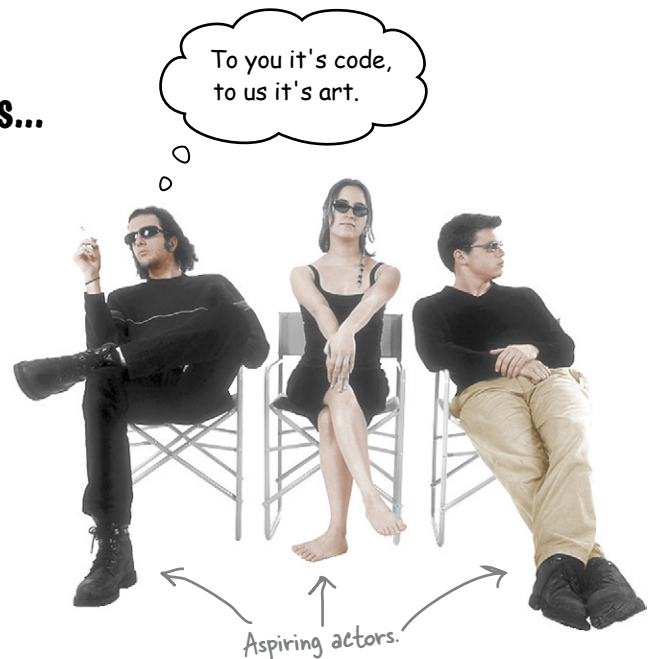
A: No. The vast majority of developers use floats and doubles without worrying about the details.

Oh no... it's the out of work actors...

Some people were never really cut out to be programmers. It seems that while some aspiring actors are filling in their time *between roles*, they're making a little extra cash cutting code, and they decided to spend some time freshening up the code in the bill-totalling program.

By the time they had rejigged the code the actors were much happier about the way everything looked... but there was just a tiny problem.

The code doesn't compile any more.



Let's see what's happened to the code

This is what the actors did to the code. You can see they really just did a couple of things:

```
#include <stdio.h>

float total = 0.0;
short count = 0;
/* This is 6%. Which is a lot less than my agent takes...*/
short tax_percent = 6;

int main()
{
    /* Hey - I was up for a movie with Val Kilmer */
    float val;
    printf("Price of item: ");
    while (scanf("%f", &val) == 1) {
        printf("Total so far: %.2f\n", add_with_tax(val));
        printf("Price of item: ");
    }
    printf("\nFinal total: %.2f\n", total);
    printf("Number of items: %i\n", count);
    return 0;
}

float add_with_tax(float f)
{
    float tax_rate = 1 + tax_percent / 100.0;
    /* And what about the tip? Voice lessons ain't free */
    total = total + (f * tax_rate);
    count = count + 1;
    return total;
}
```

The code has had some comments added and they also **changed the order of the functions**. There were no other changes made.

So really there shouldn't be a problem. The code should be good to go, right? Well - everything was great right up until the point they **compiled the code...**



TEST DRIVE

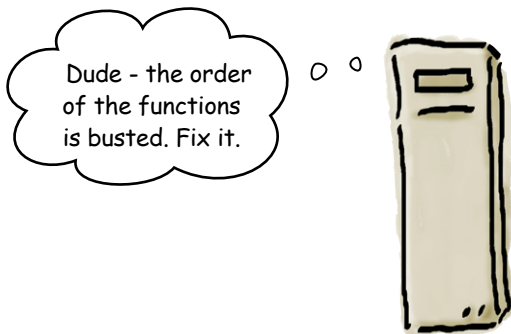
So if you open up the console and try to compile the program this happens:

```
File Edit Window Help StickToActing
> gcc totalling_broken.c -o totalling_broken && ./totalling_broken
totalling_broken.c: In function "main":
totalling_broken.c:14: warning: format "%.2f" expects type
"double", but argument 2 has type "int"
totalling_broken.c: At top level:
totalling_broken.c:23: error: conflicting types for "add with tax"
totalling_broken.c:14: error: previous implicit declaration of
"add_with_tax" was here
```

Bummer.

That's not good. What does that "error: conflicting types for 'add_with_tax'" mean? What is a previous implicit declaration? And why does it think the line that prints out the current total is now an int? Didn't we design that to be floating point?

The compiler will ignore the changes made to the comments, so that shouldn't make any difference. That means the problem must be caused by **changing the order of the functions**. But if the order is the problem why doesn't the compiler just return a message saying something like:



Seriously - why doesn't the compiler give us a little help here?

To understand exactly what's happening here we need to get inside the head of the compiler for a while and look at things from its point of view. You'll see that what's happening is that the compiler is actually trying to be a little too helpful.

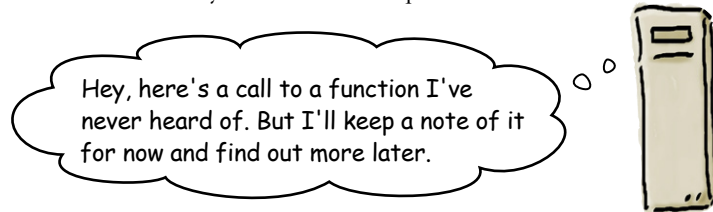
Compilers don't like surprises

So what happens when the compiler sees this line of code?

```
printf("Total so far: %.2f\n", add_with_tax(val));
```

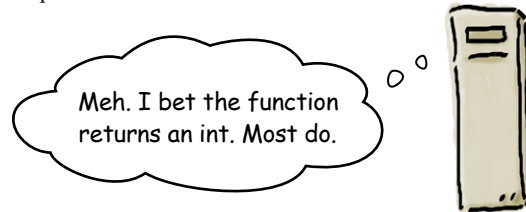
1 The compiler sees a call to a function it doesn't recognise.

Rather than complain about it, the compiler figures that it will find out more about the function later in the source file. The compiler simply remembers to look out for the function later on in the file. Unfortunately this is where the problem lies...



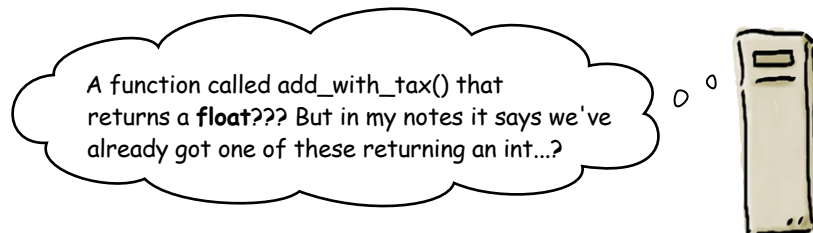
2 The compiler needs to know what data type the function will return.

Of course, the compiler can't know what the function will return just yet, so it makes an **assumption**. The compiler assumes it will return an `int`.



3 When it reaches the code for the actual function it returns a "conflicting types for 'add_with_tax'" error.

This is because the compiler thinks it has two functions with the same name. One function is the real one in the file. The other is the one that the compiler assumed would return an `int`.



The computer makes an assumption that the function returns an `int`, when in reality it returns a `float`. If you were designing the C language, how would you fix the problem?

Hello? I really don't care how the C language solves the problem. Just put the functions in the correct freaking order!



We could just put the functions back in the correct order and define the function before we call it in main.

Changing the order of the functions means that we can avoid the compiler ever making any dangerous assumptions about the return types of unknown functions. But if we force ourselves to always define functions in a specific order, there are a couple of consequences:

Fixing function order is a pain

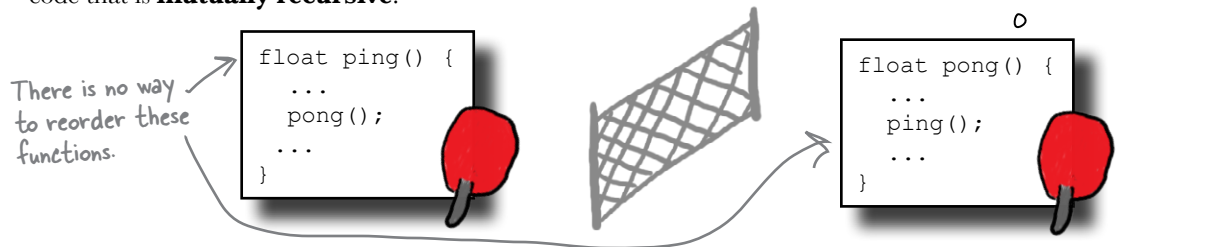
Say you've added a cool new function to your code that everyone thinks in fantastic:

```
int do_whatever(){...}
float do_something_fantastic(int awesome_level) {...}
int do_stuff() {
    do_something_fantastic(11);
}
```

What happens if you *then* decide that your program will be even *better* if you add a call to the `do_something_fantastic()` function in the existing `do_whatever()` code? You will have to **move the function** earlier in the file. Most coders want to spend their time improving what their code can do. It would be better if you didn't have to shuffle the order of the code just to keep the compiler happy.

In some situations there is no correct order

OK, so this situation is kind of rare, but occasionally you might write some code that is **mutually recursive**:



If you have two functions that call *each other* then **one of them will always be called in the file before it's defined.**

For both of those reasons it's really useful to be able to define functions in whatever order is easiest at the time. But how?

Split the declaration from the definition

Remember how the compiler made a note to itself about the function it was expecting to find later in the file? We can avoid the compiler making assumptions by **explicitly telling it what functions it should expect**. When we tell the compiler about a function it's called a **function declaration**:

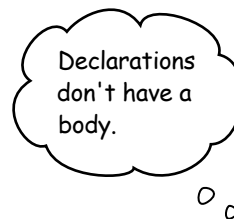
The declaration tells the compiler what return value to expect. → `float add_with_tax();` ← A declaration has no body code.
 ← It just ends with a ;

The declaration is just a function **signature** - it's a record of what the function will be called, what kind of parameters it will accept and **what type of data it will return**.

Once you've declared a function, the compiler won't need to make any assumptions so it won't matter if you define the the function after you call it.

So if you have a whole bunch of functions in your code and you don't want to worry about what order they appear in the file, you can put a list of function declarations at the start of your C program code:

```
float do_something_fantastic();
double awesomeness_2_dot_0();
int stinky_pete();
char make_maguerita(int count);
```



But even better than that, C allows you to take that whole set of declarations *out of your code* and put them in a **header file**. We've already used header files to include code from the C Standard Library:

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

← This line will include the contents of the header file called `stdio.h`.

Let's go see how we can create our own header files.



Creating your first header file

To create a header you just need to do **two things**:

- 1 **Create a new file with a .h extension.**
If you are writing a program called `totaller.c`, then create a file called `totaller.h` and write your declarations inside it:

```
float add_with_tax(float f);
```



totaller.h

You won't need to include the `main()` function in the header file because nothing else will need to call it.

- 2 **Include your header file in your main program.**
At the top of your program you should add an extra include line:

Add this include in with your other include lines.

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



totaller.c

When you write the name of the header file, make sure you surround it with double-quotes rather than angle brackets. Why the difference? When the compiler sees an include line with angle brackets it assumes it will find the header file somewhere off in the directories where the library code lives. But **our** header file is in the same directory as our `.c` file. By wrapping the header file name in quotes we are telling the compiler to look for a local file.

When the compiler reads the `#include` in the code it will read the contents of the header file, just as if they had been typed into the code.

Separating the declarations into a separate header file keeps your main code a little shorter, and it also has another *big advantage* that we'll find out about in a few pages.

For now, let's see if the header file fixed the mess.

← Local header files can also include directory names, but you will normally put them in the same directory as the C file.

#include is a precompiler instruction.



TEST DRIVE

Now when we compile the code, this happens:

```
File Edit Window Help UseHeaders
> gcc totaller.c -o totaller
```

No error messages this time.

The compiler reads the function declarations from the header file, which means it doesn't have to make any guesses about the return type of the function. It doesn't matter what order the functions are in.

Just to check that everything is OK, we can run the generated program to see if it works the same as before.

```
File Edit Window Help UseHeaders
> ./totalling_fixed
Price of item: 1.23
Total so far: 1.30
Price of item: 4.57
Total so far: 6.15
Price of item: 11.92
Total so far: 18.78
Price of item: ^D
Final total: 18.78
Number of items: 3
```

We'll press control-D here to stop the program asking for more prices.

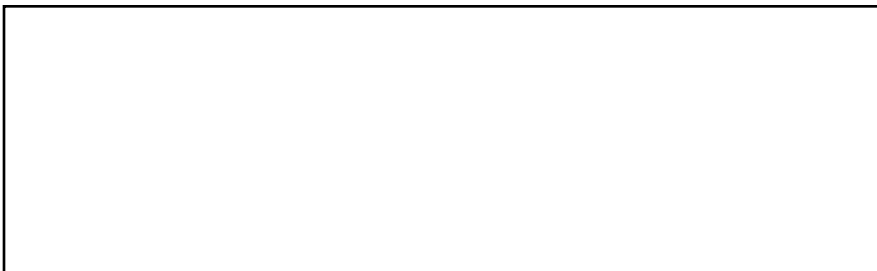


BE the Compiler

Look at the program below. Part of the program is missing. Your job is to play like you're the compiler and say what you would do if each of the candidate code fragments on the right were slotted into the missing space.

Candidate code goes here.

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



```
printf("A day on Mercury is %f hours\n", day);  
return 0;
```

```
}
```

```
float mercury_day_in_earth_days()
```

```
{
```

```
    return 58.65;
```

```
}
```


```
int hours_in_an_earth_day()
```

```
{
```

```
    return 24;
```

```
}
```

Here are the code fragments. 

 Tick the boxes that you think are correct.

```
float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
    float length_of_day = mercury_day_in_earth_days();
    int hours = hours_in_an_earth_day();
    float day = length_of_day * hours;
```

You can compile the code.

You should display a warning.

The program will work.

```
float mercury_day_in_earth_days();

int main()
{
    float length_of_day = mercury_day_in_earth_days();
    int hours = hours_in_an_earth_day();
    float day = length_of_day * hours;
```

You can compile the code.

You should display a warning.

The program will work.

```
int main()
{
    float length_of_day = mercury_day_in_earth_days();
    int hours = hours_in_an_earth_day();
    float day = length_of_day * hours;
```

You can compile the code.

You should display a warning.

The program will work.

```
float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
    int length_of_day = mercury_day_in_earth_days();
    int hours = hours_in_an_earth_day();
    float day = length_of_day * hours;
```

You can compile the code.

You should display a warning.

The program will work.



BE the Compiler Solution

Look at the program below. Part of the program is missing. Your job is to play like you're the compiler and say what you would do if each of the candidate code fragments on the right were slotted into the missing space.

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



```
printf("A day on Mercury is %f hours\n", day);  
return 0;
```

```
}
```

```
float mercury_day_in_earth_days()
```

```
{
```

```
    return 58.65;
```

```
}
```

```
int hours_in_an_earth_day()
```

```
{
```

```
    return 24;
```

```
}
```



```
float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
    float length_of_day = mercury_day_in_earth_days();
    int hours = hours_in_an_earth_day();
    float day = length_of_day * hours;
```



You can compile the code.



You should display a warning.



The program will work.

There will be a warning because we haven't declared the `hours_in_an_earth_day()` before calling it. The program will still work because it will guess the function returns an `int`.

```
float mercury_day_in_earth_days();

int main()
{
    float length_of_day = mercury_day_in_earth_days();
    int hours = hours_in_an_earth_day();
    float day = length_of_day * hours;
```



You can compile the code.



You should display a warning.



The program will work.

```
int main()
{
    float length_of_day = mercury_day_in_earth_days();
    int hours = hours_in_an_earth_day();
    float day = length_of_day * hours;
```

The program won't compile because we're calling a float function without declaring it first.



You can compile the code.



You should display a warning.



The program will work.

The program will compile without warnings, but the program won't work because there will be a rounding problem.

```
float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
    int length_of_day = mercury_day_in_earth_days();
    int hours = hours_in_an_earth_day();
    float day = length_of_day * hours;
```

The `length_of_day` variable should be a float.



You can compile the code.



You should display a warning.



The program will work.

there are no
Dumb Questions

Q: So I don't need to have declarations for int functions?

A: Not necessarily - unless you are sharing code. We'll see more about this soon.

Q: I'm confused. You talk about the compiler *precompiling*? Why does the compiler do that?

A: Strictly speaking the compiler just does the compilation step - it converts the C source code into assembly code. But in a looser sense, all of the stages that convert the C source code into the final executable are normally called *compilation*, and the gcc tool allows you to control those stages. The gcc tool does precompilation and compilation.

Q: What is the precompiler?

A: Precompilation is the first stage i covering the raw C source code into a working executable. Precompilation creates a modified version of the source just before the *proper* compilation begins. In our code the precompilation step read the contents of the header file into the main file.

Q: Does the precompiler create an actual file?

A: No - compilers normally just use pipes to sending the stuff through the phases of the compiler to make things more efficient.

Q: Why do some headers have quotes and other have angle brackets?

A: Quotes mean to simply look for a file using a relative path. So if you just include the name of a file, without including a directory name, the compiler will look in the current directory. If the thing uses angle brackets it will search for the file along a path of directories.

Q: What directories will the compiler search when it is looking for header files?

A: The gcc compiler knows where the standard headers are stored. On a Unix-style operating system, the header files are normally in places like `/usr/local/include`, `/usr/include` and a few others.

Q: So that's how it works for standard headers like `stdio.h`?

A: Yes. You can read through the `stdio.h` on a Unix-style machine in `/usr/include/stdio.h`. If you have the MingW compiler on Windows, it will probably be in `C:\MingW\include\stdio.h`.

Q: Can I create my own libraries?

A: Yes - we'll show you how to do that later in the book.



BULLET POINTS

- If the compiler finds a call to a function it's not heard of, it will assume the function returns an int.
- So if you try to call a function before you define it, there can be problems.
- Function declarations tell the compiler what your functions will look like before you define them.
- If function declarations appear at the top of your source code, the compiler won't get confused about return types.
- Function declarations are often put into header files.
- You can tell the compiler to read the contents of a header file using `#include`
- The compiler will treat included code the same as code that is typed into the source file.



This Table's Reserved...

C is a very small language. Here is the entire set of reserved words (in no useful order).

Every C program you ever see will break into just these words and a few symbols. If you use these for names, the compiler will be very, very upset.

auto	if	break
int	case	long
char	register	continue
return	default	short
do	sizeof	double
static	else	struct
entry	switch	extern
typedef	float	union
for	unsigned	goto
while	enum	void
const	signed	volatile

If you have common features...

Chances are when you begin to write several programs in C, you will find that there are some functions and features that you will want to reuse from other programs. For example, look at the specs of the two programs on the right.

XOR encryption is a very simple way of disguising a piece of text by XOR-ing each character with some value. It's not very secure, but it's very easy to do. And the same code that can encrypt text, can also be used to decrypt it. Here's the code to encrypt some text:

```
void encrypt(char* message)
{
    char c;
    while (*message) {
        *message = *message ^ 17;
        message++;
    }
}
```

void means we don't return anything.

We'll loop through the array and update each character with an encrypted version.

We pass a pointer to an array into the function.

This means we'll XOR each character with the number 17.

Doing math with a character? We can because char is a numeric datatype.

file_hider

Read the contents of a file and create an encrypted version using XOR encryption.

message_hider

Read a series of strings from the standard input and display and encrypted version on the standard output using XOR encryption.

...it's good to share code

Clearly both of those programs are going to need to use the same `encrypt()` function. So you could just copy the code from one program to the other, right? That's not seem so bad if there's just a small amount of code to copy, but what if there's a really large amount of code? Or what if the way the `encrypt()` function needs to change in the future? If there are two copies of the `encrypt` function, you will have to change it in more than one place.

In order for your code to scale properly, you really need to find some way to reuse common pieces of code. Some way of taking a set of functions and making them available in a bunch of different programs.

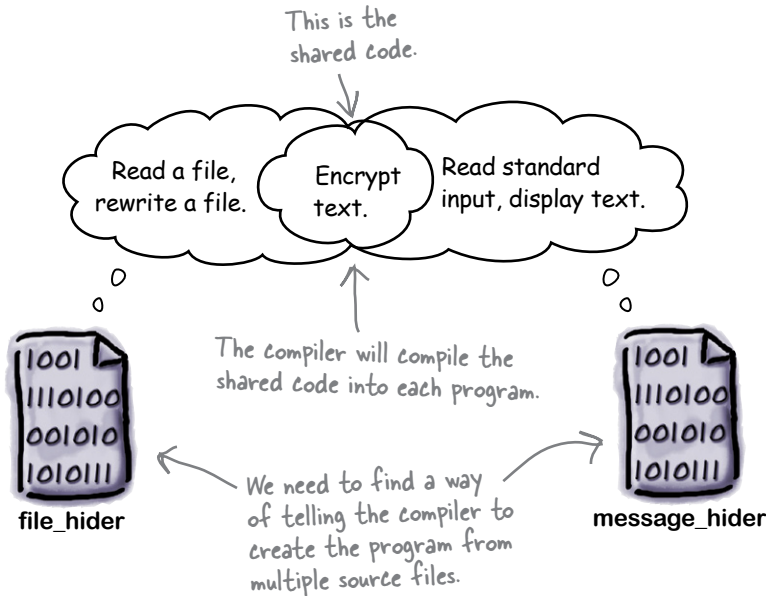
How would you do that?



Imagine you have a set of functions that you want to share between programs. If you had created the C programming language, how would you allow code to be shared?

You can split the code into separate files

If you have a set of code that you want to share amongst several files, it makes a lot of sense to put that shared code into a separate `.c` file. If the compiler can somehow include the shared code when it's compiling the program, you can use the same code in multiple applications at once. So if you ever need to change the shared code, you only have to do it in one place.



If we want to use a separate `.c` file for the shared code that gives us a *problem*. So far we have only ever created programs from single `.c` source files. So if we had a C program called `blitz_hack` we would have created it from a single source code file called `blitz_hack.c`.

But now, we want some way to give the compiler a **set of source code files** and say "Go make a program from those". How do we do that? What syntax do we use with the `gcc` compiler? And more importantly, what does it *mean* for a compiler to create a single executable program from several files? How would it work? How would it stitch them together?

In order to understand how the C compiler can create a single program from multiple files, let's take a look at how compilation works...

Compilation behind the scenes

In order to understand how a compiler can compile several source files into a single program, we'll need to pull back the curtain and see how compilation really works.

1 Precompilation: fix the source.

The first thing the compiler needs to do is fix the source. It needs to add in any extra header files it's been told about using the **#include directive**.

It might also need to expand or skip over some sections of the program. Once it's done, the source code will be ready for the actual compilation.

It can do this with command like `#define` and `#ifdef` - we'll see how to use them later in the book.



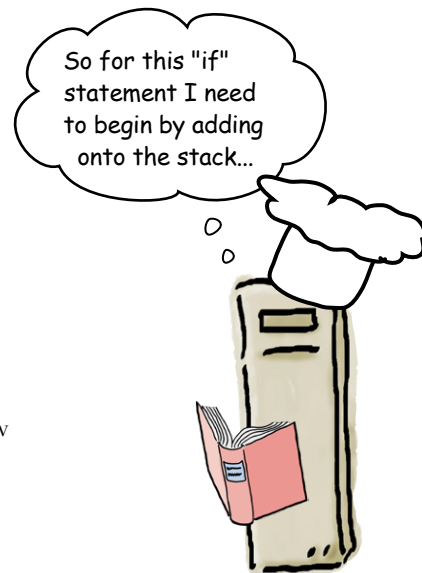
"directive" is just a fancy word for "command".

2 Compilation: translate into assembly.

The C programming language probably seems pretty low-level, but the truth is it's *not low-level enough* for the computer to understand. The computer only really understands very low-level **machine code** instructions, and the first step to generate machine code, is to convert the C source code into **assembly language symbols** like this:

```
movq -24(%rbp), %rax
movzbl(%rax), %eax
movl %eax, %edx
```

Looks pretty obscure? Assembly language describes the individual instructions the Central Processor will have to follow when running the program. The C compiler has a whole set of recipes for each of the different parts of the C language. These recipes will tell the compiler how to convert an if statement or a function call into a sequence of assembly language instructions. But even assembly isn't low-level enough for the computer. That's why it needs...



3 Assembly: generate the object code.

The compiler will need to *assemble* the symbol codes into *machine* or **object code**. This is the actual binary code that will be executed by the circuits inside the CPU.

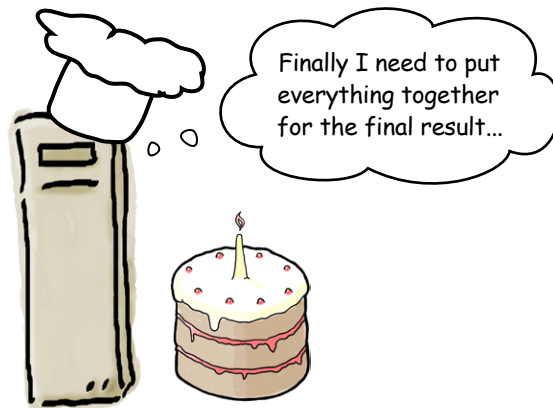
This is a really
dirty joke in
machine code.

10010101 00100101 11010101 01011100

So are we all done? After all, we've taken the original C source code and converted it into the 1s and 0s that the computer's circuits need. But no - there's still one more step. If we give the computer several files to compile for a program, the compiler will generate a piece of object code for each source file. But in order for these separate object files to form a single executable program, one more thing has to occur...

4 Linking: put it all together.

Once we have all of the separate pieces of object code, we need to piece them together like jigsaw pieces to form the **executable program**. The compiler will connect the code in one piece of object code that calls a function in another piece of object code. Linking will also make sure that the program is able to call library code properly. Finally, the program will be written out into the executable program file using a format that is supported by the operating system. The file format is important because it will allow the operating system to load the program into memory and make it run.

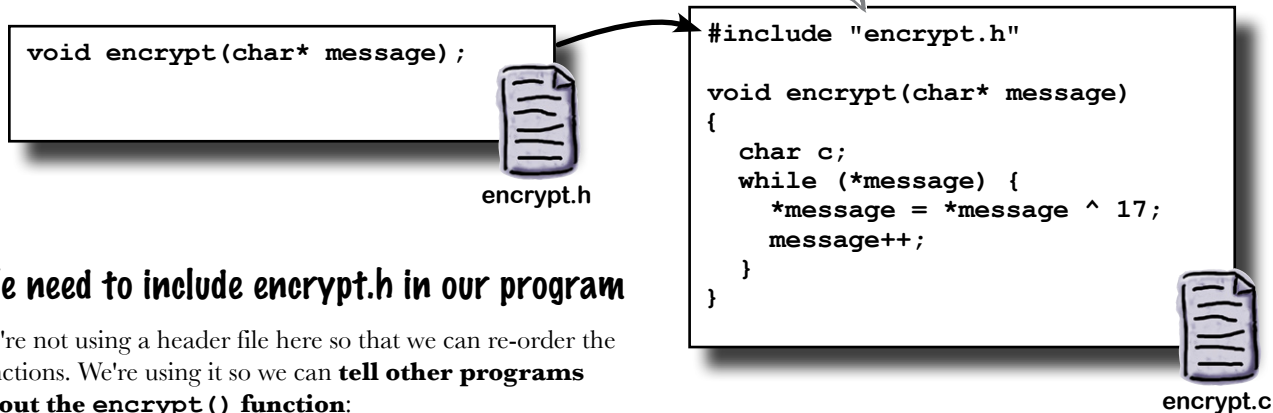


So how do we actually tell gcc that we want to make one executable program from several separate source files?

The shared code will need its own header file

If we are going to share the encrypt.c code between programs, we need some way to tell those programs about the encrypt code. We do that with a header file.

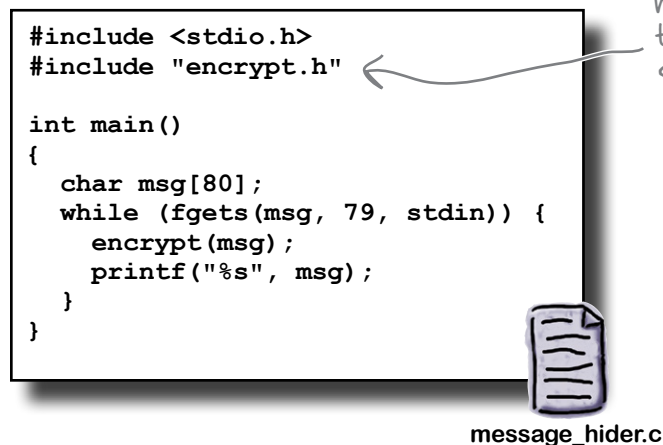
We'll include the header inside encrypt.c



We need to include encrypt.h in our program

We're not using a header file here so that we can re-order the functions. We're using it so we can **tell other programs about the encrypt () function**:

We'll include encrypt.h so that the program has the declaration of the encrypt() function.



Having encrypt.h inside the main program will mean the compiler will know enough about encrypt () function to compile the code. At the linking stage, the compiler will be able to connect the call to encrypt(msg) in message_hider.c to the actual encrypt () function in encrypt.h.

Finally, to compile everything together we just need to pass the source files to gcc:

```
gcc message_hider.c encrypt.c -o message_hider
```

Sharing Variables

We've shown you how to share functions between different files. But what if you want to share variables? Source code files normally contain their own separate variables to prevent a variable in one file affecting a variable in another file with the same name. But if you genuinely want to share variables, you should declare them in your header file and prefix them with the keyword **extern**:

```
extern int passcode;
```




TEST DRIVE

Let's see what happens when we compile our `message_hider` program:

When we run the program we can enter text and see the encrypted version.

We can even pass it the contents of the `encrypt.h` file to encrypt it.

The `message_hider` program is using the `encrypt()` function from `encrypt.c`.

```
File Edit Window Help Shhh...
> gcc message_hider.c encrypt.c -o message_hider
> ./message_hider
I am a secret message
X1p|1p1btrctel|tbbpvt
> ./message_hider < encrypt.h
g~xult□rchae9rypc;1|tbbpvt8*
```

We need to compile the code with both source files.

The program works. Now we have the `encrypt()` function in a separate file, we can use it in any program we like. If we ever change the `encrypt()` function to be something a little more secure, we will only need to amend the `encrypt.c` file.



BULLET POINTS

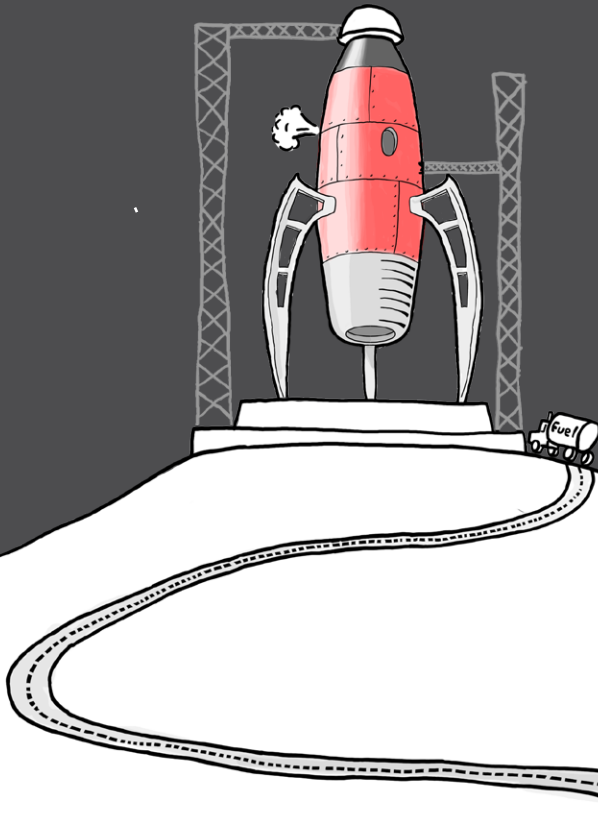
- You can share code by putting it into a separate C file.
- You need to put the function declarations in a separate `.h` header file.
- Include the header file in every C file that needs to use the shared code.
- List all of the C files needed in the compiler command.



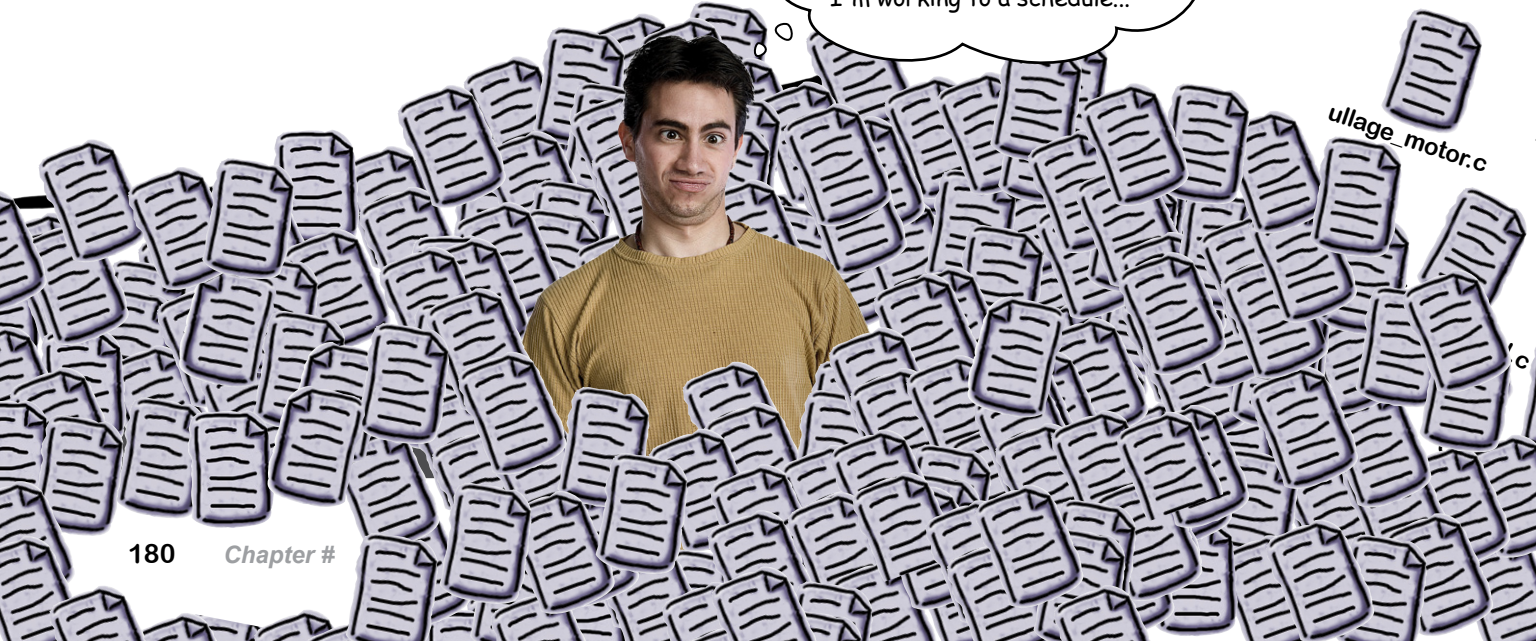
Go Off Piste

Write your own program using the `encrypt()` function. Remember - you can call the same function to decrypt text.





Man! Every time I make a simple change in **one** file it takes an age to recompile! And I'm working to a schedule...

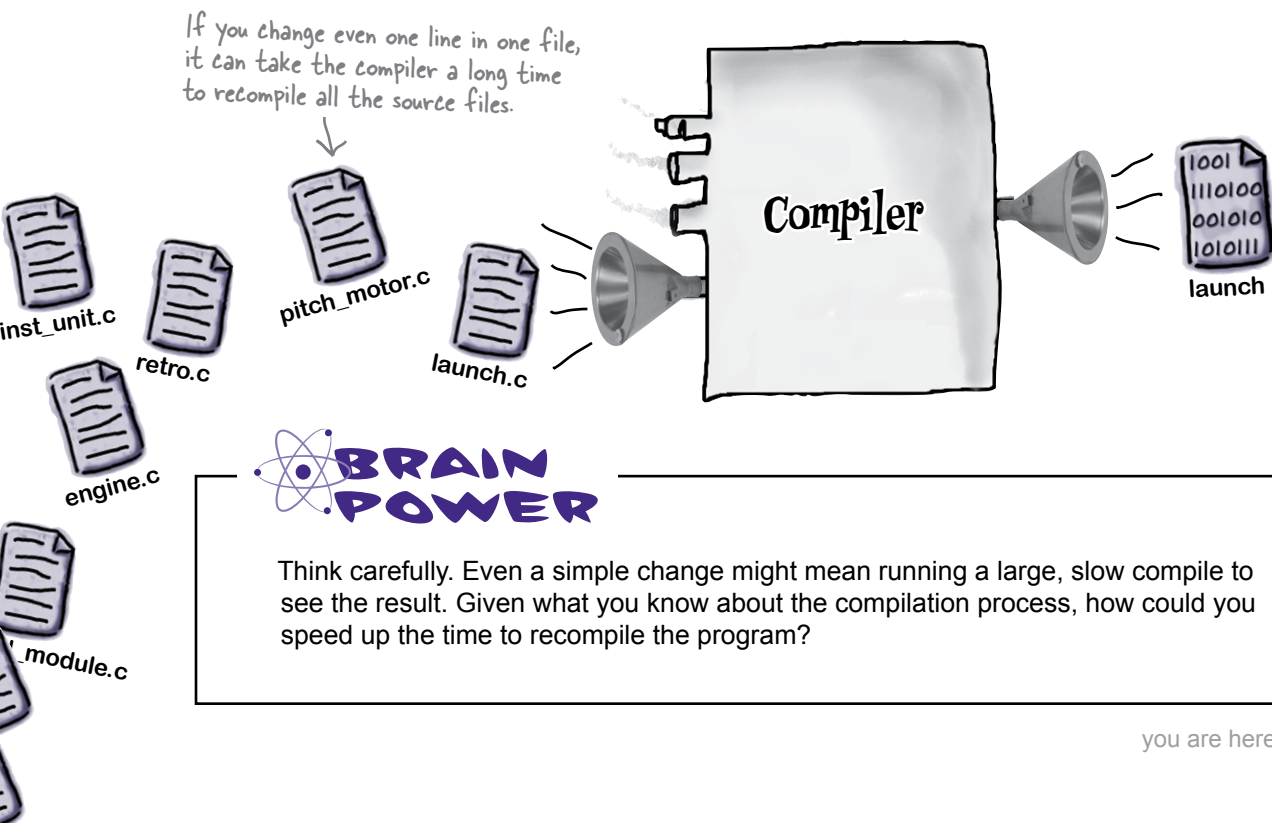


It's not Rocket Science... or is it?

Breaking your program out into separate source files not only means that you can *share code* between different programs - it also means you can start to create *really large* programs. Why? Well because you can start to break your program down into smaller **self-contained** pieces of code. Rather than being forced to have one *huge* source file, you can have lots of *simpler* files that are easier to understand, maintain and test.

So on the plus side, you can start to create really large programs. The downside? The downside is.... you can start to create really large programs. C compilers are really efficient pieces of software. They take your software through some very complex transformations. They can modify your source, link hundreds of files together without blowing your memory, and they can even optimize the code you wrote, along the way. And even though they do all that, they still manage to run quickly.

But if you create programs that use more than a few files, the time it takes to compile the code starts to become important. Let's say it take a minute to compile a large project. That might not sound like a lot of time, but it's more than long enough to break your train of thought. If you try out a change in a single line of code, you want to see the result of that change as quickly as possible. If you have to wait a full minute to see the result of every change, that will really start to slow you down.



Don't recompile every file

If you've just made a change to one or two of your source code files, it's a waste to recompile every source file for your program. Think what happens when you issue a command like this:

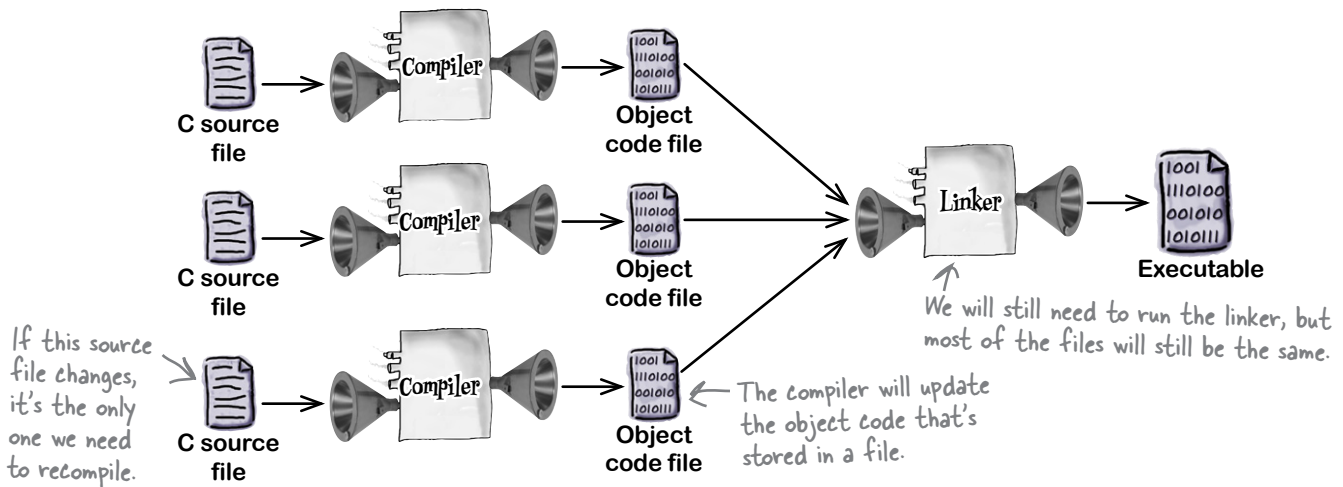
```
gcc reaction_control.c pitch_motor.c ... engine.c -o launch
```

We've skipped a few filenames here.

What will the compiler do? It will run the pre-compiler, compiler and assembler for *each source code file*. Even the ones that haven't change. And if the source code hasn't changed, the **object code** that's generated for that file won't change either. So if the compiler is generating the object code for every file, every time, what do we need to do?

You can save copies of the compiled code

If we tell the compiler to save the object code it generates into a file, then it shouldn't need to recreate it unless the source code changes. If a file does change, we can recreate the object code for that **one file** and then pass the whole set of object files to the compiler so they can be linked.



If we change a single file we will have to recreate the object code file from it, but we won't need to create the object code for any other file. Then we can pass all the object code files to linker and create a new version of the program.

So how do we tell gcc to save the object code in a file? And how do we then get the compiler to link the object files together?

First you compile the source into object files

We want object code for each of source files, and we can do that by typing this command:

This will create object code for every file. `gcc -c *.c` ← The operating system will replace *.c with all the C filenames.

The *.c will match every C file in the current directory and the -c will tell the compiler that you want to create an object file for each source file, but you don't want to link them together into a full executable program.

Then you link them together

Now we have a set of object files, we can link them together with a simple compile command. But instead of giving the compiler the names of the C source files, we tell it the names of the object files:

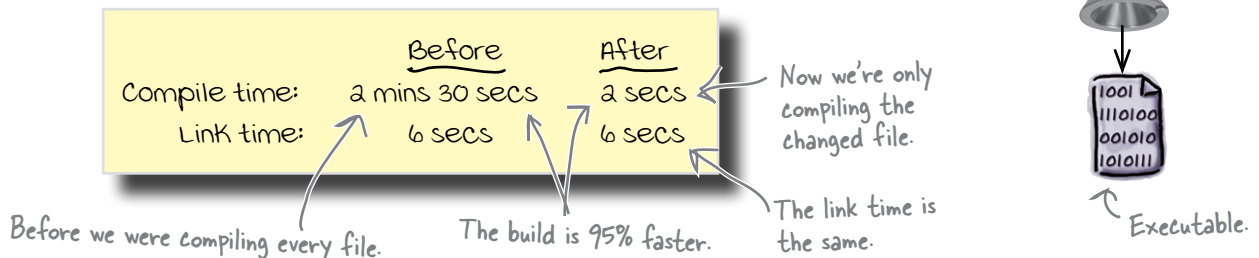
This is similar to the compile commands we've used before. `gcc *.o -o launch` ← Instead of C source files, we list the object files.
 This will match all the object files in the directory.

The compiler is smart enough to recognize the files as object files, rather than source files, so it will skip most of the compilation steps and just link them together into an executable program called launch.

OK - so now we have a compiled program just like before. But as well as that we have a set of object files that are already to be linked together if we need them again. So if we change just one of the files, we'll only need to recompile that single file and then relink the program:

This is a file that's changed. `gcc -c thruster.c` ← This will recreate the thruster.o file.
`gcc *.o -o launch` ← This will link everything together.

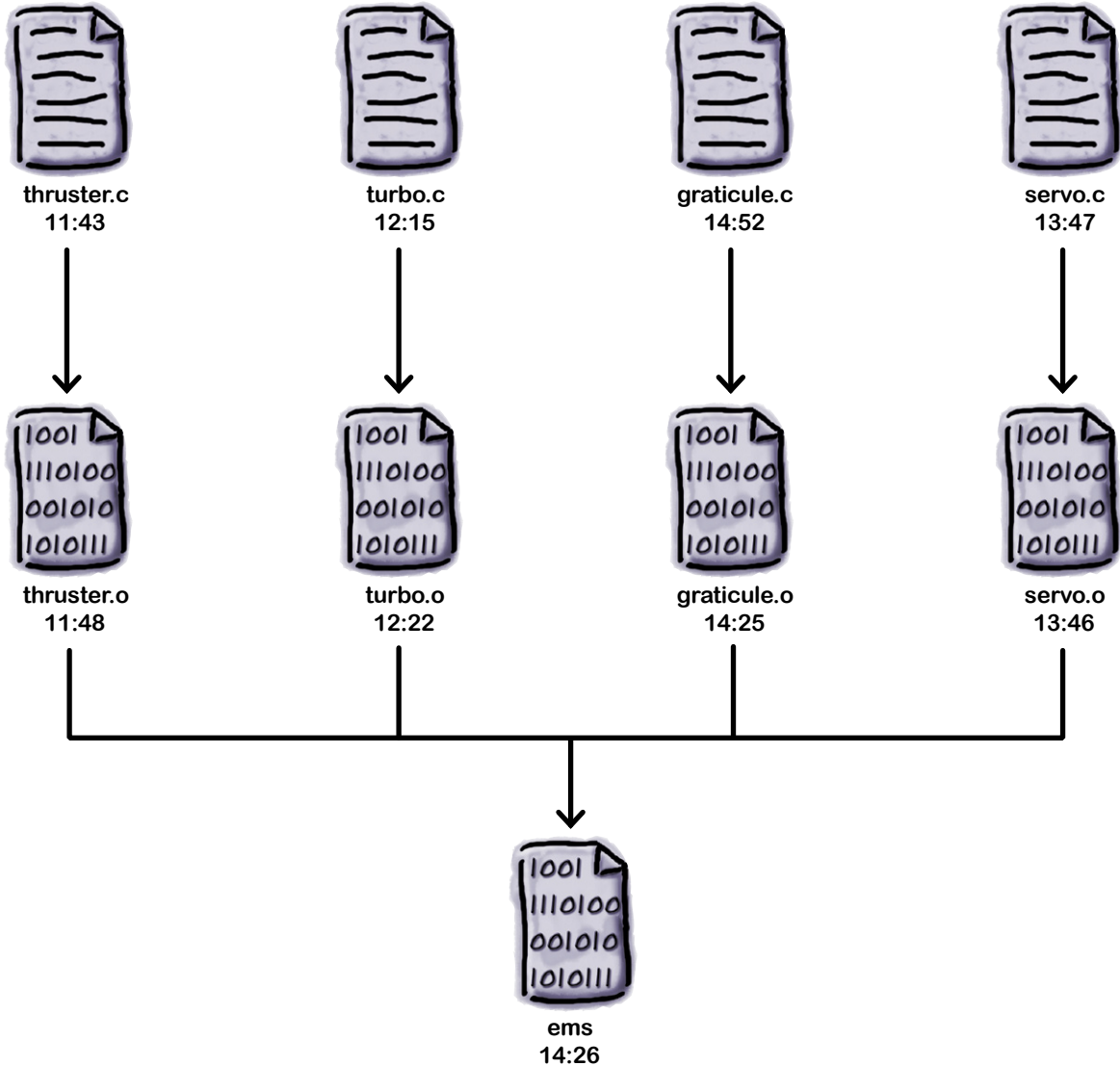
Even though we have to type two commands, we're saving a lot of time:



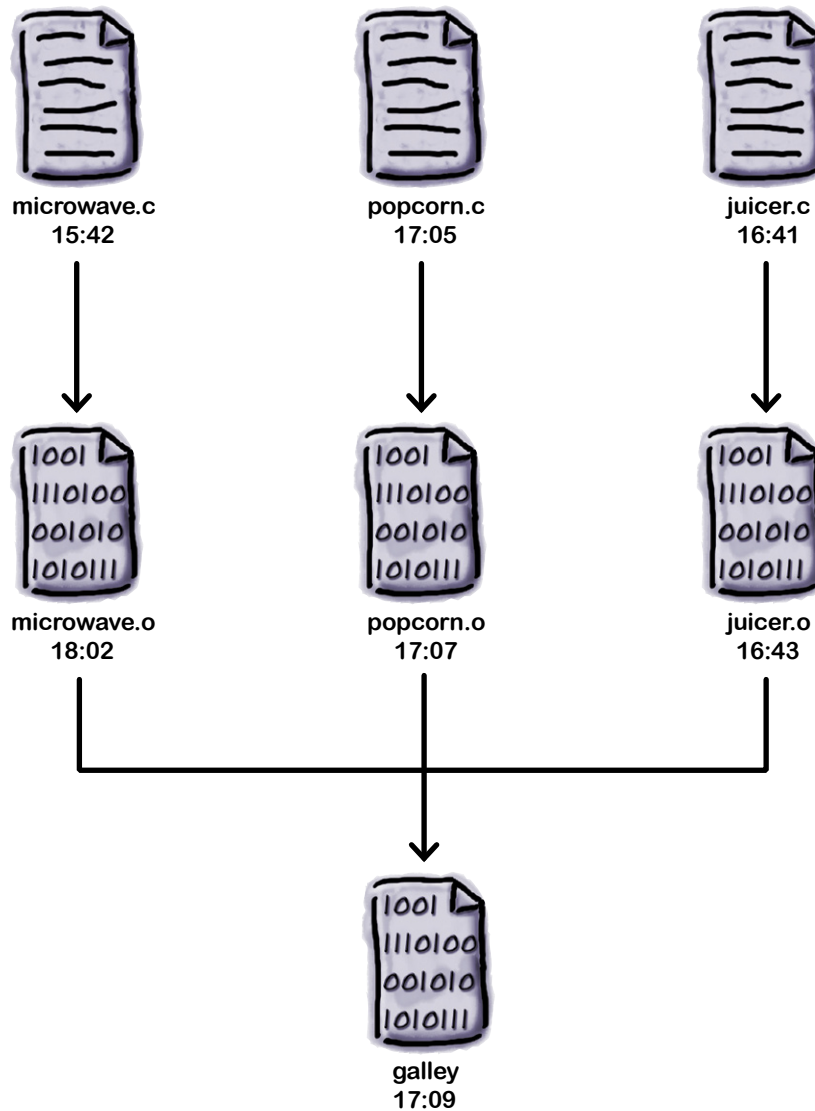


LONG Exercise

Here is some of the code that's used to control the engine management system on the craft. There's a time stamp on each file. Which files do you think need to be recreated to make the ems executable up to date? Circle the files you think need to be updated.



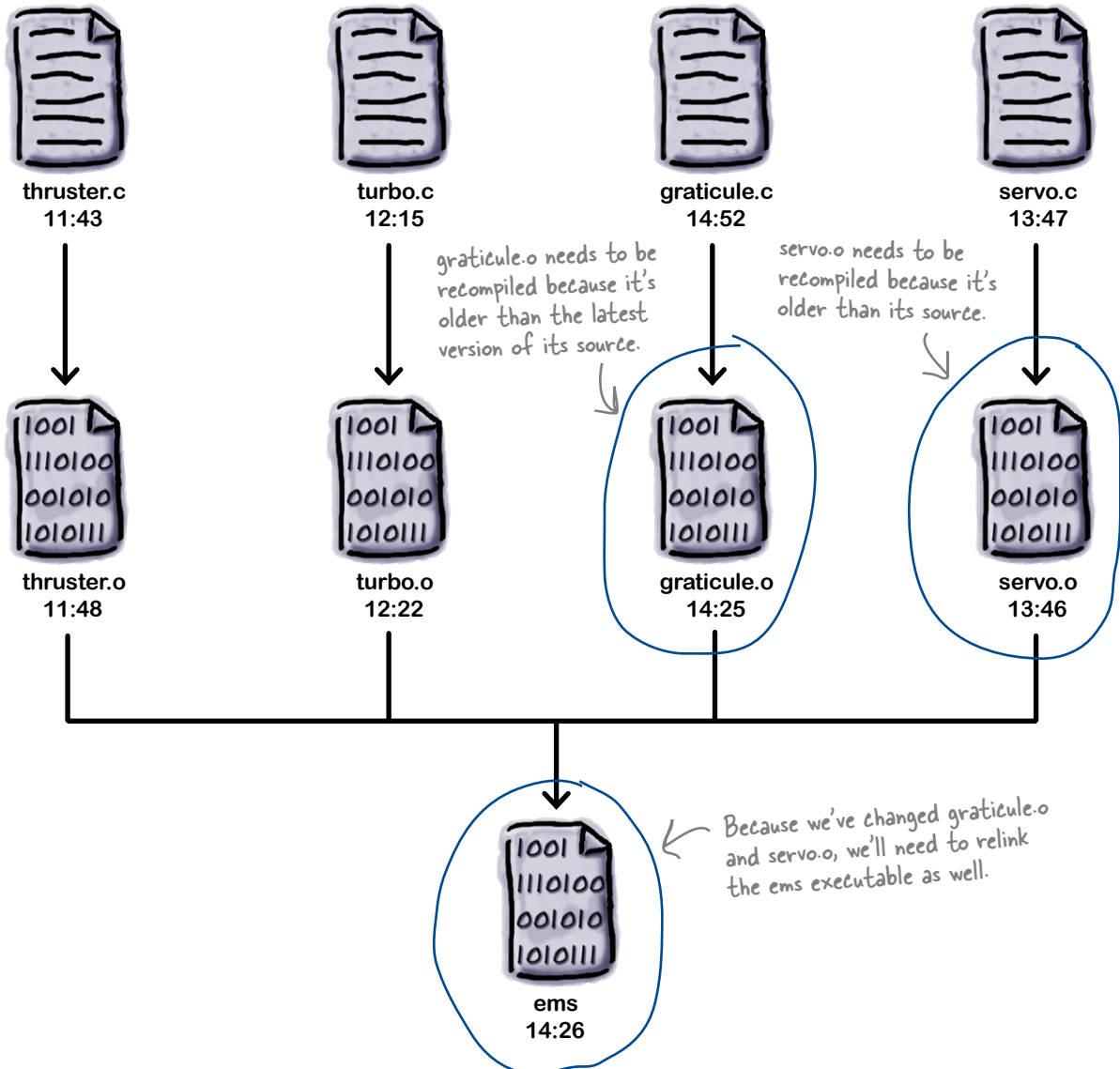
And in the galley, they need to check their code's up to date as well. Look at the times against the files. Which of these files need to be updated?



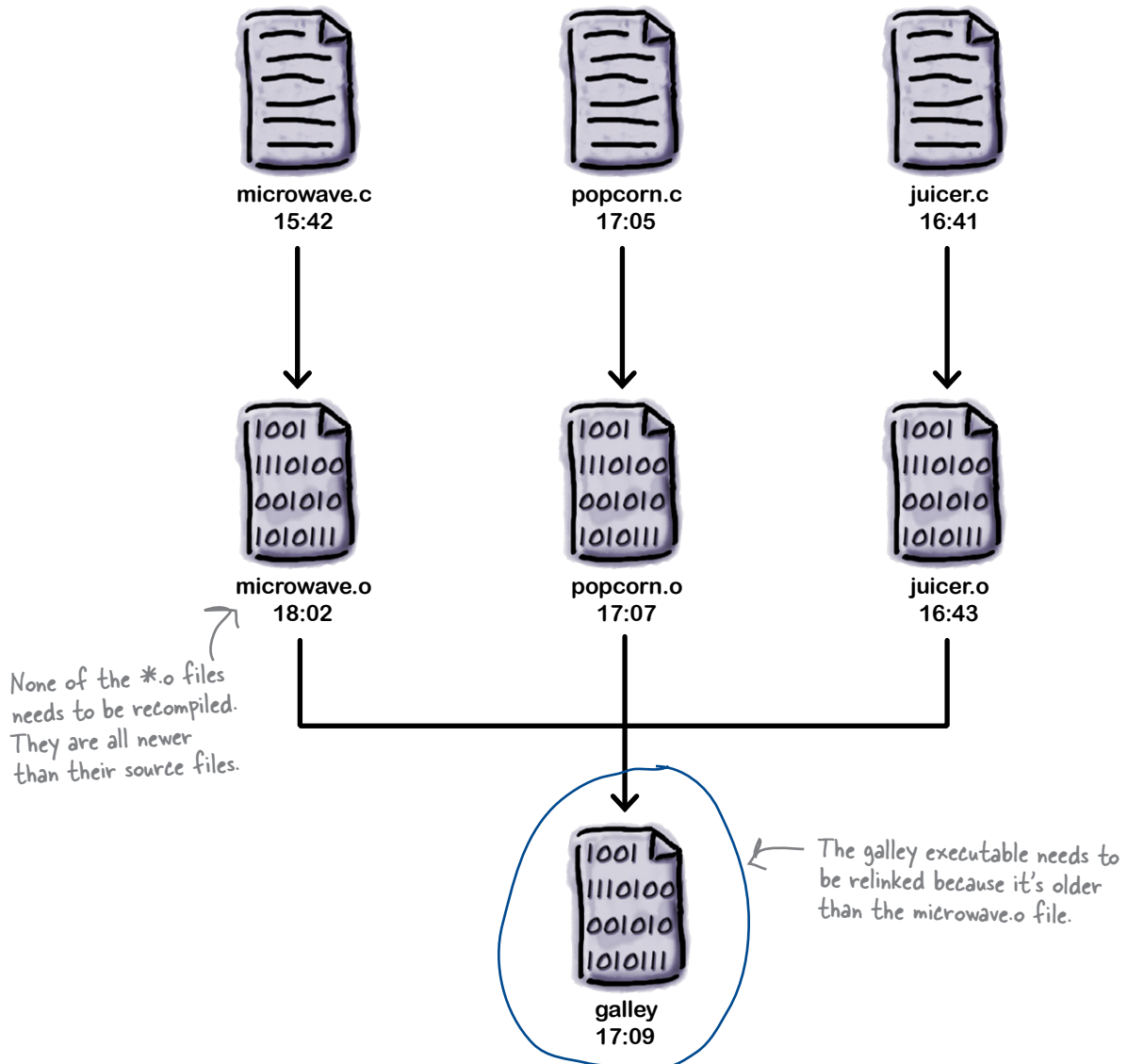
LONG EXERCISE SOLUTION



Here is some of the code that's used to control the engine management system on the craft. There's a time stamp on each file. Which files do you think need to be recreated to make the ems executable up to date? Circle the files you think need to be updated.



And in the galley, they need to check their code's up to date as well. Look at the times against the files. Which of these files need to be updated?



It's hard to keep track of the files



I **thought** the whole point of saving time was so I didn't have to get distracted. Now the compile is faster, but I have to think a **lot harder** about how to compile my code. Where's the sense in that?

It's true - partial compiles are faster, but you have to think more carefully to make sure you recompile everything you need.

If you are working on just one source file, things will be pretty simple. But if you've changed a few files, it's pretty easy to forget to recompile some of them. That means the newly compiled program won't pick up all the changes you made. Now of course, when you come to *ship* the final program, you can always make sure you can do a full recompile of *every* file, but you don't want to do that while you're still developing the code.

Even though it's a fairly **mechanical process** to look for files that need to be compiled, if you do it manually, it will be pretty easy to miss some changes.

Is there something we can use to **automate the process**?

Wouldn't it be dreamy if there was a tool that could automatically recompile just the source that's changed? But I know it's just a fantasy...



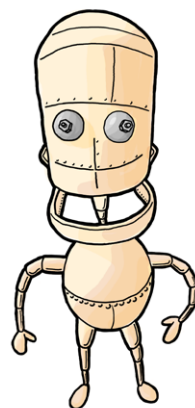
Automate your builds with the make tool

You can compile your applications really quickly in gcc, so long as you keep track of which files have changed. That's a tricky thing to do, but it's also pretty straightforward to automate. Imagine you have a file that is generated from some other file. Lets say it's an object file that is compiled from a source file:

If the `thruster.c` file is newer, you need to recompile.

`thruster.c` → `thruster.o`

If the `thruster.o` file is newer, you don't need to recompile.



How do you tell if the `thruster.o` needs to be recompiled? You just look at the timestamps of the two files. If the `thruster.o` file is older than the `thruster.c` file, then the `thruster.o` file needs to be recreated. Otherwise it's up to date.

That's a pretty simple rule. And if you have a simple rule for something, then don't think about it, **automate it...**

Make is a tool that can run the compile command for you. The make tool will check the timestamps of the source files and the generated files and then it will only recompile the files if things have gotten out of date.

But before can do all these things, we need to tell it about our source code. It needs to know the details of which files depend upon which files. And it also needs to be told exactly how to we want to build our code.

What does make need to know?

Every file that make compiles is called a **target**. Strictly speaking, make isn't limited to compiling files. A target is any file that is *generated* from some other files. So a target might be a zip archive that is generated from the set of files that need to be compressed.

For every target, make needs to be told *two things*:



The dependencies.

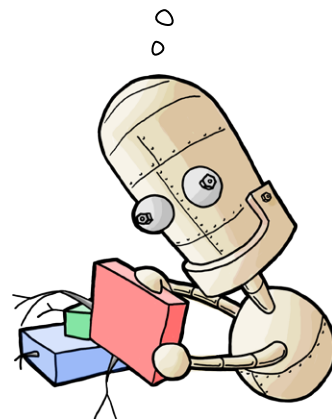
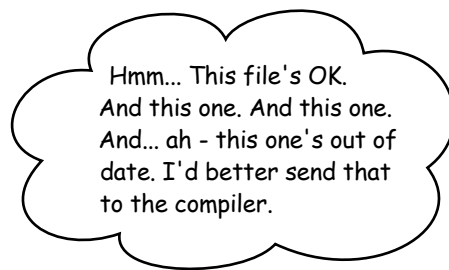
Which files the target is going to be generated from.



The recipe.

The set of instructions it needs to run to generate the file.

Together the dependencies and the recipe form a **rule**. A rule tells make all it need to know to create the generate target file.



Let's see how make works

Let's say you want to compile `thruster.c` into some object code in `thruster.o`. What are the dependencies and what's the recipe?

`thruster.c` \longrightarrow `thruster.o`

The `thruster.o` file is the target, because it's the file we want to generate. `thruster.c` is a dependency, because it's a file the compiler will need in order to create `thruster.o`. And what will the recipe be? That's the compile command to convert `thruster.c` into `thruster.o`.

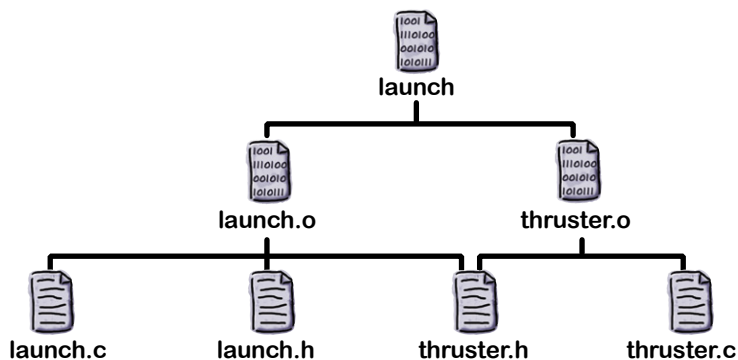
`gcc -c thruster.c` \longleftarrow This is the rule for creating `thruster.o`.

Make sense? If we tell the `make` tool about the dependencies and the recipe, we can leave it to decide when it needs to recompile `thruster.o`.

But we can go further than that. Once we have the `thruster.o` file we are going to use it to create the launch program. That means the launch file can also be set up as a target - because it's a file we want to generate. The dependency files for launch are all of the `.o` object files. The recipe is this command:

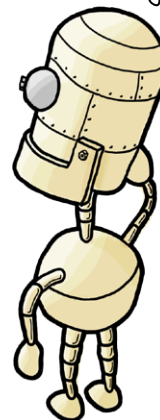
`gcc *.o -o launch`

Once `make` has been given the details of all of the dependencies and rules, all we have to do is tell it to create the launch file. `make` will work out the details.



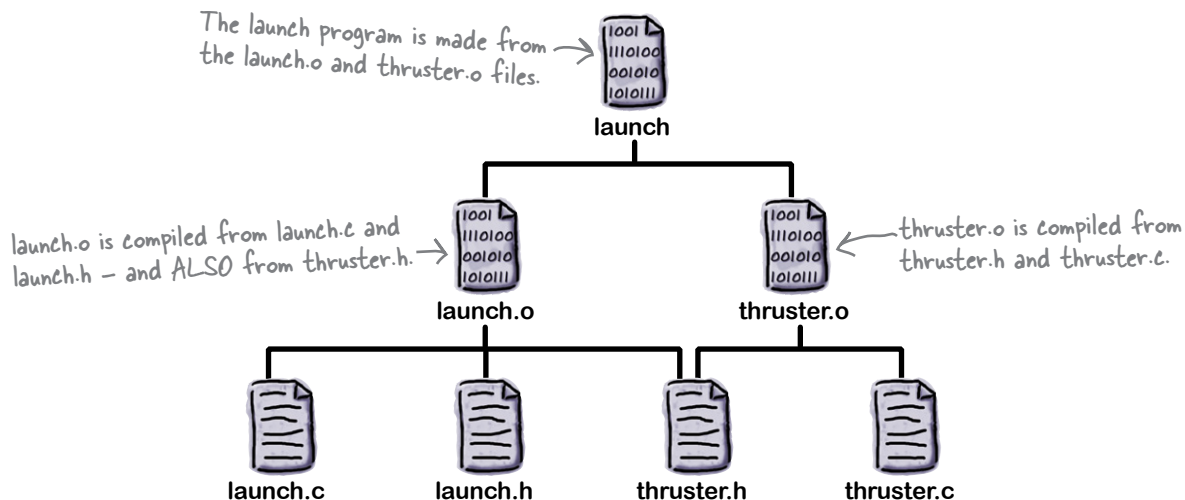
But how do we tell `make` about the dependencies and recipes? Let's find out.

So I've got to compile the launch program? Hmm... First I'll need to recompile `thruster.o` because it's out of date, then I just need to relink `launch`.



Tell make about your code with a makefile

All of the details about the targets, dependencies and recipes need to be stored in a file called either `makefile` or `Makefile`. To see how it works, imagine you have a pair of source files that together create the launch program:



The launch program is made by linking the `launch.o` and `thruster.o` files. Those files are compiled from their matching C and header files, but the `launch.o` file *also* depends on the `thruster.h` file because it contains code that will need to call a function in the thruster code.

This is how you'd describe that build in a makefile:

```

launch.o: launch.c launch.h thruster.h
gcc -c launch.c

thruster.o: thruster.h thruster.c
gcc -c thruster.c

launch: launch.o thruster.o
gcc launch.o thruster.o -o launch
    
```

Handwritten annotations for the makefile:

- "This is a target." points to `launch.o`.
- "A target is a file that is going to be generated." points to the target line `launch.o: launch.c launch.h thruster.h`.
- "There are 3 RULES." points to the three rule blocks.
- "launch.o depends on these 3 files." points to the dependencies `launch.c launch.h thruster.h`.
- "This is a recipe for creating thruster.o." points to the recipe `gcc -c thruster.c`.
- "The recipes MUST begin with a TAB character." points to the `gcc` command in the final rule.



Watch it!

All the recipe lines MUST begin

with a TAB character.

If you just try to indent the recipe lines with spaces, the build won't work.



TEST DRIVE

If you save this into a text file called Makefile then open up a console and type the following:

We are telling make to create the launch file.

make first needs to create a launch.o with this line.

make then needs to create thruster.o with this line.

Finally make links the object files to create the launch program.

```
File Edit Window Help MakelSo
> make launch
gcc -c launch.c
gcc -c thruster.c
gcc launch.o thruster.o -o launch
```

You can see that make was able to work out the sequence of commands required to create the launch program. But what happens if we make a change to the `thruster.c` file and then run make again?

make no longer needs to compile launch.e.

launch.o is already up to date.

```
File Edit Window Help MakelSo
> make launch
gcc -c thruster.c
gcc launch.o thruster.o -o launch
```

make is able to skip creating a new version of `launch.o`. Instead it just compiles `thruster.o` and then relinks the program.

there are no
Dumb Questions

Q: Is make just like ANT?

A: It's probably better to say that build tools like ANT and rake are like make. Make was one of the earliest tools used to automatically build programs from source code.

Q: This seems like a lot of work just to compile source code. Is it really that useful?

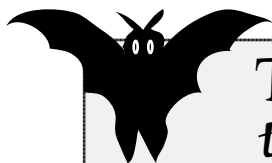
A: Yes - make is amazingly useful. For small projects make might not appear to save you that much time, but once you have more than a handful of files, compiling and linking code together can become very painful.

Q: If I write a makefile for a Windows machine, will it work on a Mac? Or a Linux machine?

A: Because make files calls commands in the underlying operating system, sometimes make files don't work on different operating systems.

Q: Can I use make for things other than compiling code?

A: Yes. Make is most commonly used to compile code. But it can also be used as a command line installer. Or a source control tool. In fact - you can use make for almost any task that you can perform on the command line.



Tales from the Crypt

Why indent with tabs?

It's easy to indent recipes with spaces instead of TABs. So why does make insist on using TABs? This is a quote from make's creator Stuart Feldman:

"Why the tab in column 1? ... It worked, it stayed. And then a few weeks later I had a user population of about a dozen, most of them friends, and I didn't want to screw up my embedded base. The rest, sadly, is history."



Geek Bits

make takes away a lot of the pain of compiling files. But if you find that even make is not automatic enough, take a look at a tool called **autoconf**:

<http://www.gnu.org/software/autoconf/>

Autoconf is used to generate makefiles. C programmers often create tools to automate the creation of software. And increasing number of them are available on the GNU web site.



Make Magnets

Hey baby, if you don't groove to the latest tunes, then you'll *love* the program the guys in the Head First Lounge just wrote! oggswing is a program that reads an Ogg Vorbis music file and creates a Swing version. Sweet! See if you can complete the makefile that compiles oggswing and then uses it to convert a .ogg file:

This converts
whitennerdy.ogg
to swing.ogg.



oggswing:

.....

swing.ogg:

.....

gcc oggswing.c -o oggswing

whitennerdy.ogg

oggswing whitennerdy.ogg swing.ogg

oggswing

[SPACES]

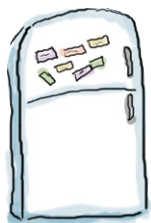
oggswing.h

oggswing.c

[TAB]

[SPACES]

[TAB]



Make Magnets Solution

Hey baby, if you don't groove to the latest tunes, then you'll *love* the program the guys in the Head First Lounge just wrote! oggswing is a program that reads an Ogg Vorbis music file and creates a Swing version. Sweet! See if you can complete the makefile that compiles oggswing and then uses it to convert a .ogg file:

```
oggswing: oggswing.c oggswing.h
[TAB] gcc oggswing.c -o oggswing

swing.ogg: whitennerdy.ogg oggswing
[TAB] oggswing whitennerdy.ogg swing.ogg
```

[SPACES]

[SPACES]



Geek Bits

The make tool can do far, far more than we have space to discuss here. To find out more about make and what it can do for you, visit the *GNU Make Manual* at:

<http://tinyurl.com/yczmjx>

Liftoff!

If you have a really slow build, then make will really speed things up. Most developers are so used to building their code with make that they even use it for small programs. make is like having a really careful developer sitting alongside you. If you have a large amount of code, make will always take care to build just the code you need at just the time you need it.

And sometimes getting things done in time is important...



BULLET POINTS

- It can take a long time to compile a large number of files.
- You can speed up compilation time by storing object code in *.o files.
- The gcc can compile programs from object files as well as source files.
- The make tool can be used to automate your builds.
- make knows about the dependencies between files so it can compile just the files that change.
- make needs to be told about your build with a Makefile.
- Be careful formatting your Makefile - don't forget to indent lines with TABs instead of spaces.





Your C Toolbox

You've got Chapter 4 under your belt and now you've added data types and header files to your tool box. For a complete list of tooltips in the book, see Appendix X.

chars are numbers

Use shorts for small whole numbers

Use ints for most whole numbers

Use longs for really big whole numbers

Use floats for most floating points

Use doubles for really precise floating points

Split function declarations from definitions

Put declarations in a header file

Save object code into files to speed up your builds

#include <> for library headers

#include "" for local headers

Use make to manage your builds

5 structs, unions, and bitfields

Roll your own structures

```
struct tea =  
{ "tealeaves", "milk",  
  "sugar", "water", "gin"};
```



Most things in life are more complex than a simple number.

So far we've looked at the basic data-types of the C language, but what if you want to go beyond numbers and pieces of text, and **model things in the real world**? **Structs** allow you to model **real-world complexities** by writing your own structures. We'll show you how to **combine the basic data-types** into structs, and even **handle life's uncertainties** with **unions**. And if you're after a simple yes or no, **bitfields** may be just what you need.



Sometimes you need to hand around a lot data

You've seen that C can handle a lot of different types of data: small numbers and large numbers, floating point numbers, characters and text. But quite often when you are recording data about something in the real world, you'll find that you need to use more than one piece of data. Take a look at this example. Here we have two functions that *both* need the same set of data because they are both dealing with the same real world *thing*:

"const char*" just means we're going to pass literal strings.

Both of these functions take the same set of parameters.

```
/* Print out the catalog entry */
void catalog(const char* name, const char* species, int teeth, int age)
{
    printf("%s is a %s with %i teeth. He is %i\n",
           name, species, teeth, age);
}

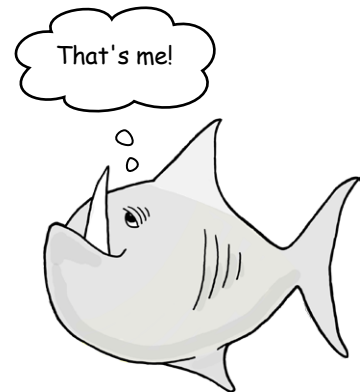
/* Print the label for the tank */
void label(const char* name, const char* species, int teeth, int age)
{
    printf("Name:%s\nSpecies:%s\n%i years old, %i teeth\n",
           name, species, teeth, age);
}
```

Now that's not really so bad is it? But even though you're just passing 4 pieces of data, the code's starting to look a little messy:

```
int main()
{
    catalog("Snappy", "Piranha", 69, 4);
    label("Snappy", "Piranha", 69, 4);
    return 0;
}
```

We are passing the same 4 pieces of data twice.

There's only one fish, but we're passing four pieces of data.



So how do you get around this problem? What can you do to avoid passing around lots and lots of data if you're really only using it to describe a single thing?

Cubicle conversation

I don't really see the problem. It's only **four** pieces of data.

Joe: Sure - it's four pieces of data *now*, but what if we change the system to record another piece of data for the fish?

Frank: That's only *one more parameter*.

Jill: Yes - it's only one piece of data, but we'll have to add that to *every function* that needs data about a fish.

Joe: Yeah - for a big system, that might be *hundreds* of functions. And all because we add *one more piece of data*.

Frank: That's a good point. But how do we get round it?

Joe: Easy - we just group the data into a *single thing*. Something like an array.

Jill: I'm not sure that would work. Arrays normally store a list of data of the *same type*.

Joe: Good point.

Frank: I see. We're recording strings and ints. Yeah - we can't put those into the same array.

Jill: I don't think we can.

Joe: But come on - there must be some way of doing this in C. Let's think about what we need.

Frank: OK - we want something that let's use refer to a whole set of data of different types all at once, like it was a single piece of data.

Jill: I don't think we've seen anything like that yet have we?



What we need is something that will let us record a set of different pieces of data into one single piece of data.

Create your own structured data types with a struct

If you had a set of data that you need to bundle together into a *single thing*, then you can use a **struct**. The word `struct` is short for **Structured Data Type**. A structured data type will let us take all of those different pieces of data into the code and wrap them up into one large new data-type, like this:

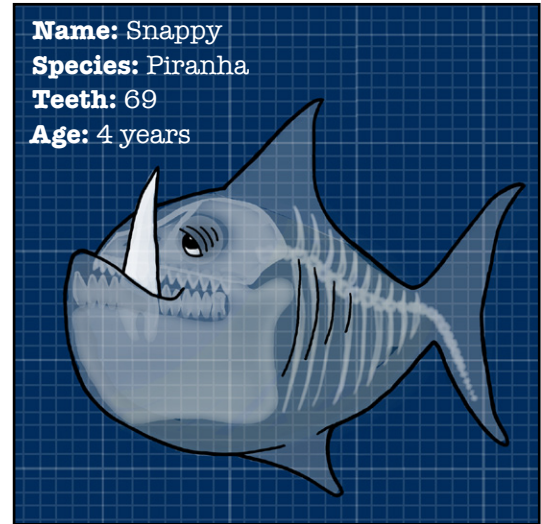
```
struct fish {
    const char* name;
    const char* species;
    int teeth;
    int age;
};
```

This will create a new custom data-type: it's a data type that is made up of a collection of other pieces of data. In fact, it's a little bit like an array, except:

- ★ **It's fixed length.**
- ★ **The pieces of data inside the struct are given names**

But once you've defined what your new struct looks like, how do you create pieces of data that use it? Well it's very similar to creating a new array. You just need to make sure the individual pieces of data are in the order that they are defined in the `struct`:

`struct fish` is the data-type. → `struct fish snappy = {"Snappy", "Piranha", 69, 4};`
 "snappy" is the variable name. → `snappy`
 This is the name. → `"Snappy"`
 This is the species. → `"Piranha"`
 This is the number of teeth. → `69`
 This is Snappy's age. → `4`



there are no Dumb Questions

Q: Hey - wait a minute. What's that `const char` thing again?

A: `const char*` is used for strings that you don't want to change. That means it's often used to record literal strings.

Q: OK. So does this struct store the string?

A: In this case - no. The struct here just stores a pointer to a string. That means it's just recording an address and the string lives somewhere else in memory.

Q: But you can store the whole string in there if you want?

A: Yes - if you define a char array in the string, like `char name[20];`.

Just give them the fish

Now instead of having to pass around a whole collection of individual pieces of data to the functions, you can just pass our new custom piece of data:

```
/* Print out the catalog entry */
void catalog(struct fish f)
{
    ...
}

/* Print the label for the tank */
void label(struct fish f)
{
    ...
}
```

Looks a lot simpler doesn't it? Not only does it mean the functions now only need a *single piece of data*, but the code that calls them is easier to read:

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
catalog(snappy);
label(snappy);
```

So that's how you can define your custom data type, but how do you *use* it? How will our functions be able to read the individual pieces of data stored inside the struct?

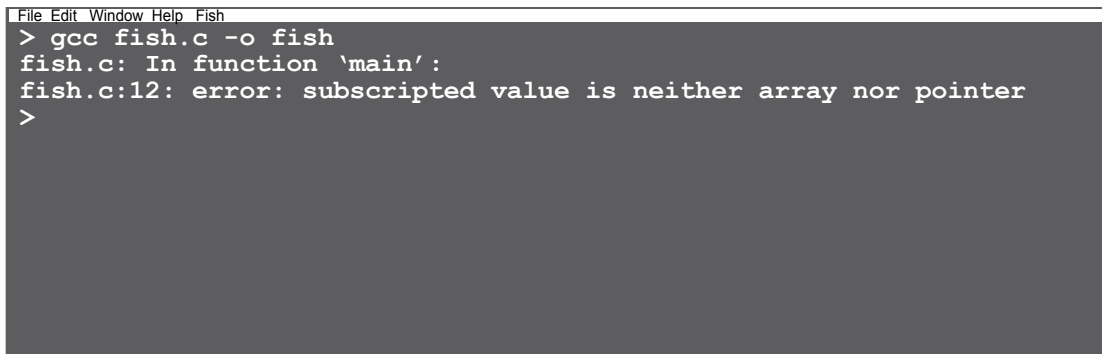
You can read a struct's fields with the "." operator

Because a struct's a little like an array, you might think you can read its fields like an array:

```
struct fish snappy = {"Snappy", "piranha", 69, 4};  
printf("Name = %s\n", snappy[0]);
```

← If snappy was a pointer to an array, you would access the first field like this.

You get an error if you try to read struct fields like it's an array.

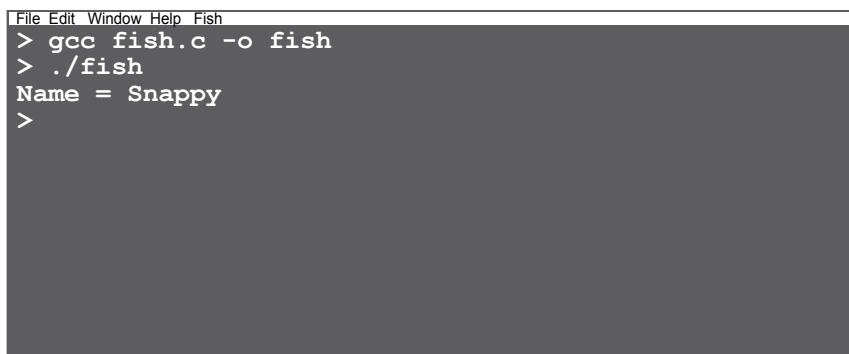


```
File Edit Window Help Fish  
> gcc fish.c -o fish  
fish.c: In function 'main':  
fish.c:12: error: subscripted value is neither array nor pointer  
>
```

But you can't. Even though a struct stores fields like an array, the only way to access them is **by name**. You can do this using the "." operator. If you've used another language like JavaScript or Ruby, this will look familiar:

```
struct fish snappy = {"Snappy", "piranha", 69, 4};  
printf("Name = %s\n", snappy.name);
```

← This is the name attribute in snappy.



```
File Edit Window Help Fish  
> gcc fish.c -o fish  
> ./fish  
Name = Snappy  
>
```

↑ This will return the string "Snappy".

OK - now we know a few things about using structs, let's see if we can go back and update that code...

Piranha ~~Pool~~ Puzzle

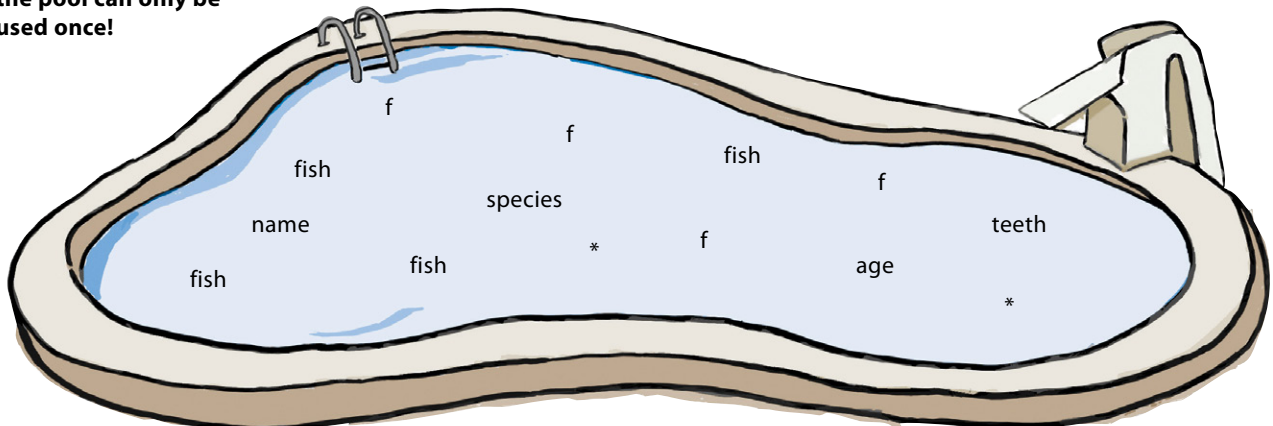


Your job is to write a new version of the catalog function using the fish struct. Take fragments of code from the pool and place them in the blank lines below. You may not use the same fragment more than once, and you won't need to use all the fragments.

```
void catalog(struct fish f)
{
    printf("%s is a %s with %i teeth. He is %i\n",
           ..... ' ..... ' ..... ' ..... ' ..... );
}

int main()
{
    struct fish snappy = {"Snappy", "Piranha", 69, 4};
    catalog(snappy);
    /* We're skipping calling label for now */
    return 0;
}
```

Note: each thing from the pool can only be used once!



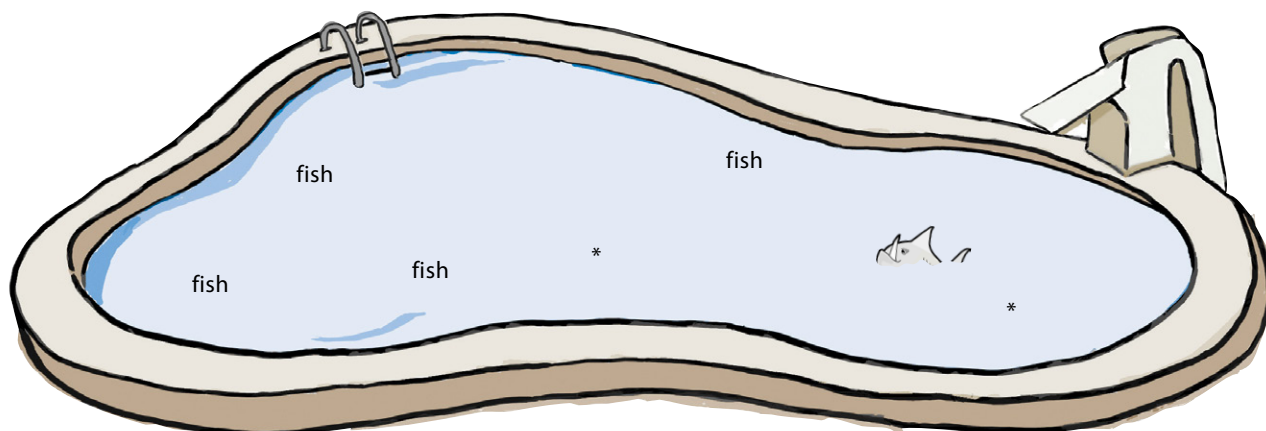
Piranha ~~Pool~~ Puzzle Solution



Your job is to write a new version of the catalog function using the fish struct. Take fragments of code from the pool and place them in the blank lines below. You may not use the same fragment more than once, and you won't need to use all the fragments.

```
void catalog(struct fish f)
{
    printf("%s is a %s with %i teeth. He is %i\n",
        ...f... .name... / ...f... .species... / ...f... .teeth... / ...f... .age.... );
}

int main()
{
    struct fish snappy = {"Snappy", "Piranha", 69, 4};
    catalog(snappy);
    /* We're skipping calling label for now */
    return 0;
}
```





TEST DRIVE

You've re-written the `catalog()` function, so it's pretty easy to re-write the `label()` function as well. Once you've done that you can compile the program and check that it still works:

They look - someone's using `make`... →

This line is printed out by the `catalog()` function. →

These lines are printed by the `label()` function. →

```
File Edit Window Help FishAreFriendsNotFood
> make pool_puzzle && ./pool_puzzle
gcc pool_puzzle.c -o pool_puzzle
Snappy is a Piranha with 69 teeth. He is 4
Name:Snappy
Species:Piranha
4 years old, 69 teeth
>
```

That's great. The code works the same as it did before, but now we have really simple lines of code that call the two functions:

```
catalog(snappy);
```

```
label(snappy);
```

But not only is the code more readable, but if we ever decide to record some extra data in the struct, we won't have to change anything in the functions that use it.

there are no Dumb Questions

Q: So is a struct just an array?

A: No - but like an array it groups a number of pieces of data together.

Q: An array variable is just a pointer to the array. Is a struct variable a pointer to a struct?

A: No. A struct variable is a name for the struct itself.

Q: I know I don't have to, but could I use `[0]`, `[1]`,... to access the fields of a struct?

A: No - you can only access fields by name.

Q: Are structs like classes in other languages?

A: They are similar, but it's not so easy to add methods to structs.



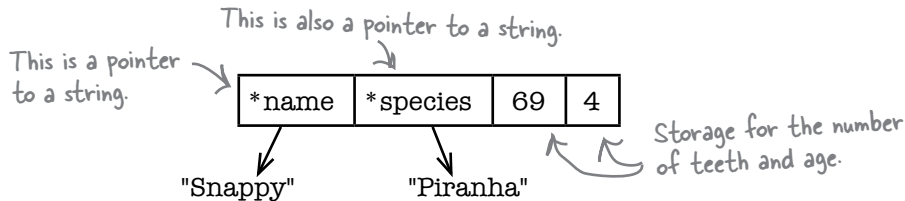
Structs In Memory Up Close

When you define a **struct**, you're not telling the computer to create anything in memory. You're just giving it a **template** for how you want a new type of data to look.

```
struct fish {
    const char* name;
    const char* species;
    int teeth;
    int age;
};
```

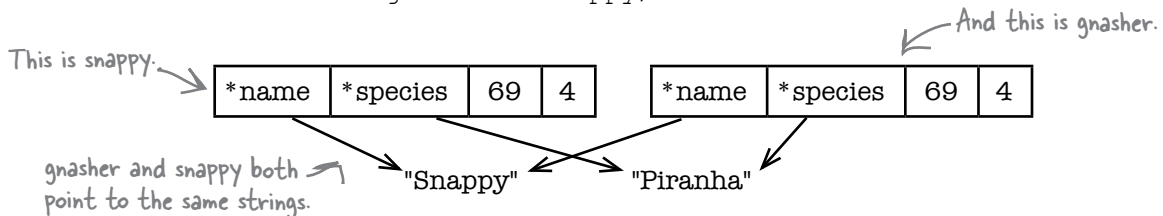
But when you define a new variable, the computer will need to create some space in memory for an **instance** of the struct. That space in memory will need to be big enough to contain all of the fields within the struct:

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
```



So what do you think happens when you assign a struct to another variable? Well the computer will create a **brand new copy of the struct**. That means it will need to allocate another piece of memory of the same size, and then copy over each of the fields.

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
struct fish gnasher = snappy;
```



Remember: when you're assigning struct variables, you are telling the computer to copy data.

But what if you need to connect structs together?

Remember that when you define a struct we are actually creating a *new data-type*. C gives us lots of built-in data-types like ints and shorts, but a struct lets us combine existing types together so that you can describe more complex objects to the computer.

But if a struct creates a data-type from existing data-types, that means you can also **create structs from other structs**. To see how this works, let's look at an example.

```

struct preferences {
    const char* food;
    float exercise_hours;
};

struct fish {
    const char* name;
    const char* species;
    int teeth;
    int age;
    struct preferences care;
};

```

← These are things our fish likes.

← This is a struct inside a struct.

→ This is a new field.

↑ Our new field is called "care" but it will contain fields defined by the "preferences" struct.

Why Combine Structs?

Why would you want to do this? So we can cope with **complexity**. structs give us bigger *building blocks* of data. By combining structs together we can create larger and larger data structures. We might have to begin with just ints and shorts, but with structs we can describe hugely complex things, like **network streams**, or **video images**.

This code tells the computer one struct will contain another struct. You can then create variables using the same array-like code as before, but now you can include the data for one struct inside another:

```
struct fish snappy = {"Snappy", "Piranha", 69, 4, {"Meat", 7.5}};
```

Once you've combined structs together, we can access the fields using a chain of "." operators:

```
printf("Snappy likes to eat %s", snappy.care.food);
printf("Snappy likes to exercise for %f hours", snappy.care.exercise_hours);
```

↓ This is the struct data for the "care" field.

↑ This is the value for care.food.

↑ This is the value for care.exercise_hours.

OK - let's try out your new struct skillz...



The guys at the Head First Aquarium are starting to record lots of data about each of their fish guests. Here are their structs:

```
struct exercise {
    const char* description;
    float duration;
};

struct meal {
    const char* ingredients;
    float weight;
};

struct preferences {
    struct meal food;
    struct exercise exercise;
};

struct fish {
    const char* name;
    const char* species;
    int teeth;
    int age;
    struct preferences care;
};
```


This is the data that will be recorded for one of the fish:

```
Name: Snappy
Species: Piranha
Food ingredients: meat
Food weight: 0.2 lbs
Exercise description: swim in the jacuzzi
Exercise duration 7.5 hours
```

Question 0: How would you write this data in C?

```
struct fish snappy = .....
```

Question 1: Complete the code of the `label()` function so it produces output like this:

```
Name:Snappy
Species:Piranha
4 years old, 69 teeth
Feed with 0.20 lbs of meat and allow to swim in the jacuzzi for 7.50 hours
```

```
void label(struct fish a)
{
    printf("Name:%s\nSpecies:%s\n%i years old, %i teeth\n",
          a.name, a.species, a.teeth, a.age);
    printf("Feed with %2.2f lbs of %s and allow to %s for %2.2f hours\n",
          ..... ' ..... '
          ..... ' ..... ');
}
```



The guys at the Head First Aquarium are starting to record lots of data about each of their fish guests. Here are their structs:

```
struct exercise {
    const char* description;
    float duration;
};

struct meal {
    const char* ingredients;
    float weight;
};

struct preferences {
    struct meal food;
    struct exercise exercise;
};

struct fish {
    const char* name;
    const char* species;
    int teeth;
    int age;
    struct preferences care;
};
```

This is the data that will be recorded for one of the fish:

```
Name: Snappy
Species: Piranha
Food ingredients: meat
Food weight: 0.2 lbs
Exercise description: swim in the jacuzzi
Exercise duration 7.5 hours
```

Question 0: How would you write this data in C?

```
struct fish snappy = {"Snappy", "Piranha", 69, 4, {"meat", 0.2}, {"swim in the jacuzzi", 7.5}};
```

Question 1: Complete the code of the `label()` function so it produces output like this:

```
Name:Snappy
Species:Piranha
4 years old, 69 teeth
Feed with 0.20 lbs of meat and allow to swim in the jacuzzi for 7.50 hours
```

```
void label(struct fish a)
{
    printf("Name:%s\nSpecies:%s\n%i years old, %i teeth\n",
          a.name, a.species, a.teeth, a.age);
    printf("Feed with %2.2f lbs of %s and allow to %s for %2.2f hours\n",
          ..... a.care.food.weight ..... , ..... a.care.food.ingredients ..... ,
          ..... a.care.exercise.description ..... , ..... a.care.exercise.duration ..... );
}
```

Hmm... all these struct commands seem kind of wordy. I have to use the struct keyword when I define a struct, then I have to use it again when I define a variable. I wonder if there's some way of simplifying this?



You can give your struct a proper name using typedef

When you create variables for built-in data-types, you can use simple short-names like `int` or `double`, but so far every time we've created a variable containing a struct we've had to include the `struct` keyword.

```
struct cell_phone {
    int cell_no;
    const char * wallpaper;
    float minutes_of_charge;
};
...
struct cell_phone p = {5557879, "sinatra.png", 1.35};
```

But C allows us to create an **alias** for any struct that you create. If you add the word **typedef** before the `struct` keyword, and a **type name** after the closing brace, you can call the new type whatever you like:

```
typedef
means we are going to give the struct type a new name.
→ typedef struct cell_phone {
    int cell_no;
    const char * wallpaper;
    float minutes_of_charge;
} phone; ← phone will become an alias for "struct cell_phone".
```

```
...
phone p {5557879, "sinatra.png", 1.35};
```

Now when the compiler sees "phone" it will treat it like "struct cell_phone".

typedefs can shorten your code and make it easier to read. Let's see what our code will look like if you start to add typedefs to it...

What should I call my new type?

If you use `typedef` to create an alias for a struct you will need to decide what your **alias** will be. The alias is just the name of your type. That means there are *two names* to think about: the name of the struct (`struct cell_phone`) and the name of the **type** (`phone`). Why have two names? You usually don't need both. The compiler is quite happy for you to skip the struct name - like this:

```
typedef struct {
    int cell_no;
    const char * wallpaper;
    float minutes_of_charge;
} phone;
phone p = {5557879, "s.png", 1.35};
```

This is the alias. →



Exercise

It's time for the scuba diver to make his daily round of the tanks and he needs a new label on his suit. Trouble is, it looks like some of the code has gone missing. Can you work out what the missing words are?

```
#include <stdio.h>

.....struct {
    float tank_capacity;
    int tank_psi;
    const char* suit_material;
} .....;

.....struct scuba {
    const char* name;
    equipment kit;
} diver;

void badge(..... d)
{
    printf("Name: %s Tank: %2.2f(%i) Suit: %s\n",
        d.name, d.kit.tank_capacity, d.kit.tank_psi, d.kit.suit_material);
}

int main()
{
    ..... randy = {"Randy", {5.5, 3500, "Neoprene"}};
    badge(randy);
    return 0;
}
```



Exercise Solution

It's time for the scuba diver to make his daily round of the tanks and he needs a new label on his suit. Trouble is, it looks like some of the code has gone missing. Can you work out what the missing words are?

```
#include <stdio.h>

typedef.....struct {
    float tank_capacity;
    int tank_psi;
    const char* suit_material;
} ..equipment.....;

typedef.....struct scuba {
    const char* name;
    equipment kit;
} diver;

void badge(..diver..... d)
{
    printf("Name: %s Tank: %2.2f(%i) Suit: %s\n",
        d.name, d.kit.tank_capacity, d.kit.tank_psi, d.kit.suit_material);
}

int main()
{
    ..diver..... randy = {"Randy", {5.5, 3500, "Neoprene"}};
    badge(randy);
    return 0;
}
```

The coder decided to give the struct the name "scuba" here. But we'll just use the diver type name.



BULLET POINTS

- A `struct` is a data type made from a sequence of other data types.
- `structs` are fixed length.
- `struct fields` are accessed by name, using *dot-notation*.
- `struct fields` are stored in memory in the same order they appear in the code.
- You can connect `structs` together.
- `typedef` creates an *alias* for a data type.
- If you use `typedef` with a `struct`, then you can skip giving the `struct` a name.

there are no Dumb Questions

Q: Do `struct fields` placed next to each other in memory?

A: Sometimes there are small gaps between the fields.

Q: Why's that?

A: The computer likes data to fit inside word boundaries. So if a computer uses 32-bit words, it won't want a short, say, to be split over a 32-bit boundary.

Q: So it would leave a gap and start the short in the next 32-bit word?

A: Yes.

Q: Does that mean each field takes up a whole word?

A: No. The computer only leaves gaps to prevent fields splitting across word boundaries. If it can fit several fields into a single word, it will.

Q: Why does the computer care about word boundaries?

A: It will read complete words from the memory. If a field is split across more than 1 word, the CPU would have to read several locations and somehow stitch the value together.

Q: And that'd be slow?

A: That'd be slow.

Q: In languages like Java, if I assign an object to a variable, it doesn't copy the object, it just copies a reference. Why is it different in C?

A: In C all assignments copy data. If you want to copy a reference to a piece of data, you should assign a pointer.

Q: I'm really confused about `struct` names. What's the `struct` name and what's the alias?

A: The `struct` name is the word that follows the `struct` keyword. If you write `struct fred { ... }`, then the name is "fred" and when you create variables you would say `struct fred x`.

Q: And the alias?

A: Sometimes you don't want to keep using the `struct` keyword when you declare variables, so `typedef` allows you to create a single word alias. In `typedef struct fred { ... } james;`, then the word "james" is the alias.

Q: So what's an anonymous `struct`?

A: One without a name. So `typedef struct { ... } james;` has an alias of "james", but no alias. Most of the time, if you create an alias, you don't need a name.

So how do you update a struct?

A struct is really just a bundle of variables, grouped together and treated like a single piece of data. You've already seen how to create a struct object, and how to access its values using the dot-notation.

But how do you *change* the value of a struct that already exists? Well you can change the fields just like any other variable:

This creates a struct. → `fish snappy = {"Snappy", "piranha", 69, 4};`
This SETs the value of the teeth field. → `printf("Hello %s\n", snappy.name);` ← This reads the value of the name field.
→ `snappy.teeth = 68;` ← Ouch! Looks like Snappy bit something hard.

That means if you look at this piece of code, you should be able to work out what it does, right?

```
#include <stdio.h>

typedef struct {
    const char* name;
    const char* species;
    int age;
} turtle;

void happy_birthday(turtle t)
{
    t.age = t.age + 1;
    printf("Happy Birthday %s! You are now %i years old!\n",
        t.name, t.age);
}

int main()
{
    turtle myrtle = {"Myrtle", "Leatherback sea turtle", 99};
    happy_birthday(myrtle);
    printf("%s's age is now %i\n", myrtle.name, myrtle.age);
    return 0;
}
```



Myrtle the turtle.

But there's something odd about this code...



TEST DRIVE

This is what happens when you compile and run the code:

```

File Edit Window Help ILikeTurtles
> gcc turtle.c -o turtle && ./turtle
Happy Birthday Myrtle! You are now 100 years old!
Myrtle's age is now 99
>
  
```

WTF???? →

Something weird has happened.

The code creates a new struct and then passes it to a function that was *supposed* to increase the value of one of the fields by 1. And *that's exactly what the code did....* at least - for a while.

Inside the `happy_birthday()` function, the `age` field was updated - and we know that it worked because the `printf()` function displayed the new increased age value. But that's when the weird thing happened. Even though the age was updated by the function, when the code returned to the `main()` function, the age seemed to reset itself.



This code is doing something weird. But you've already been given enough information to tell you exactly **what** happened. Can you work out what it is?

The code is cloning the turtle

Let's take a closer look at the code that called the `happy_birthday()` function:

```
void happy_birthday(turtle t)
{
    ...
}
```

This is the turtle that we are passing to the function.

```
...
happy_birthday(myrtle);
```

The myrtle struct will be copied to this parameter.

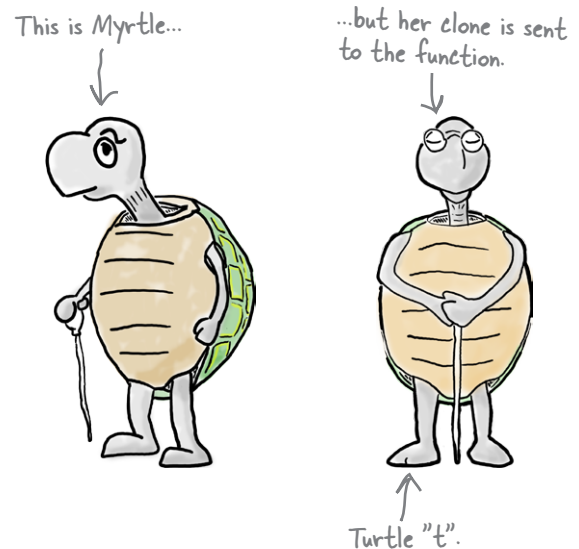
In C, parameters are passed to functions **by value** - that means when you call a function, the values you pass into it are *assigned* to the parameters. So in this code, it's almost as if we had written something like this:

```
turtle t = myrtle;
```

But *remember* - when we assign structs in C, the values are copied. When you call the function, the parameter `t` will contain a *copy* of the `myrtle` struct. It's as if the function **has a clone of the original turtle**. So the code inside the function *does* update the age of the turtle - but it's a different turtle.

What happens when the function returns? The `t` parameter disappears, and the rest of the code in `main()` uses the `myrtle` struct. But the value of `myrtle` was never changed by the code. It was always a completely separate piece of code.

So what do you do if you want pass a struct to a function that needs to update it?



You need a pointer to the struct

When we passed a variable to the `scanf()` function, we couldn't pass the variable itself to the `scanf`, we had to pass a **pointer**:

```
scanf("%f", &length_of_run);
```

Why did we do that? Because if we tell the `scanf()` function where the variable lives in memory, then the function will be able to update the data stored at that place in memory, which means it can update the variable.

And you can do just the same with `structs`. If you want a function to update a `struct` variable, you can't just pass the `struct` as a parameter because that will simply send a *copy* of the data to the function. Instead, you can pass the address of the `struct`:

```
void happy_birthday(turtle* t)
{
    ...
}
```

This means "Someone is going to give me a pointer to a struct".

Remember: an address == a pointer.

```
...
happy_birthday(&myrtle);
```

This means we will pass the address of the myrtle variable to the function.

Sharpen your pencil

See if you can figure out what *expression* needs to fit into each of the gaps in this new version of the `happy_birthday()` function.

Be careful. Don't forget that `t` is now a **pointer variable**.

```
void happy_birthday(turtle* t)
{
    .....age = .....age + 1;
    printf("Happy Birthday %s! You are now %i years old!\n",
           .....name, .....age);
}
```



Sharpen your pencil Solution

See if you can figure out what *expression* needs to fit into each of the gaps in this new version of the `happy_birthday()` function.

Be careful. Don't forget that `t` is now a **pointer variable**.

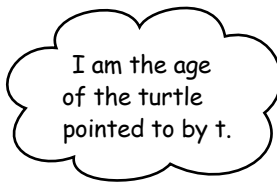
```
void happy_birthday(turtle* t)
{
    .....(*).....age = .....(*).....age + 1;
    printf("Happy Birthday %s! You are now %i years old!\n",
           .....(*).....name, .....(*).....age);
}
```

We need to put a "*" before the variable name because we want the value it points to.

The parentheses are really important. The code will break without them.

(*t).age vs. *t.age

So why did you need to make sure that `*t` was wrapped in parentheses? It's because the two expressions `(*t).age` and `*t.age` are very different:



(*t).age \neq ***t.age**

So the expression `*t.age` is really the same as `*(t.age)`. Think about that expression for the moment. It means "The contents of the memory location given by `t.age`". But `t.age` isn't a memory location.

So be careful with your parentheses when using structs - parentheses really matter.



TEST DRIVE

Let's check if we got around the bug:

```
File Edit Window Help ILikeTurtles
> gcc happy_birthday_turtle_works.c -o happy_birthday_turtle_works
Happy Birthday Myrtle! You are now 100 years old!
Myrtle's age is now 100
>
```

That's great. The function now works.

By passing a pointer to the `struct` we allowed the function to update the *original data* rather than taking a local copy.

`t->age`
means
`(*t).age`

I can see how the new code works. But the stuff about parentheses and *-notation don't make the code all that readable. I wonder if there something that would help with that...?

Yes - there is another struct pointer notation that is more readable.

Because you need to be careful to use parentheses in the right way when you're dealing with pointers, the inventors of the C language came up with a simpler and easier to read piece of syntax. These two expressions mean the same thing:

`(*t).age`
`t->age`
These two mean the same.



So `t->age` means "The age field in the struct that `t` points to". That means we can also write the function like this:

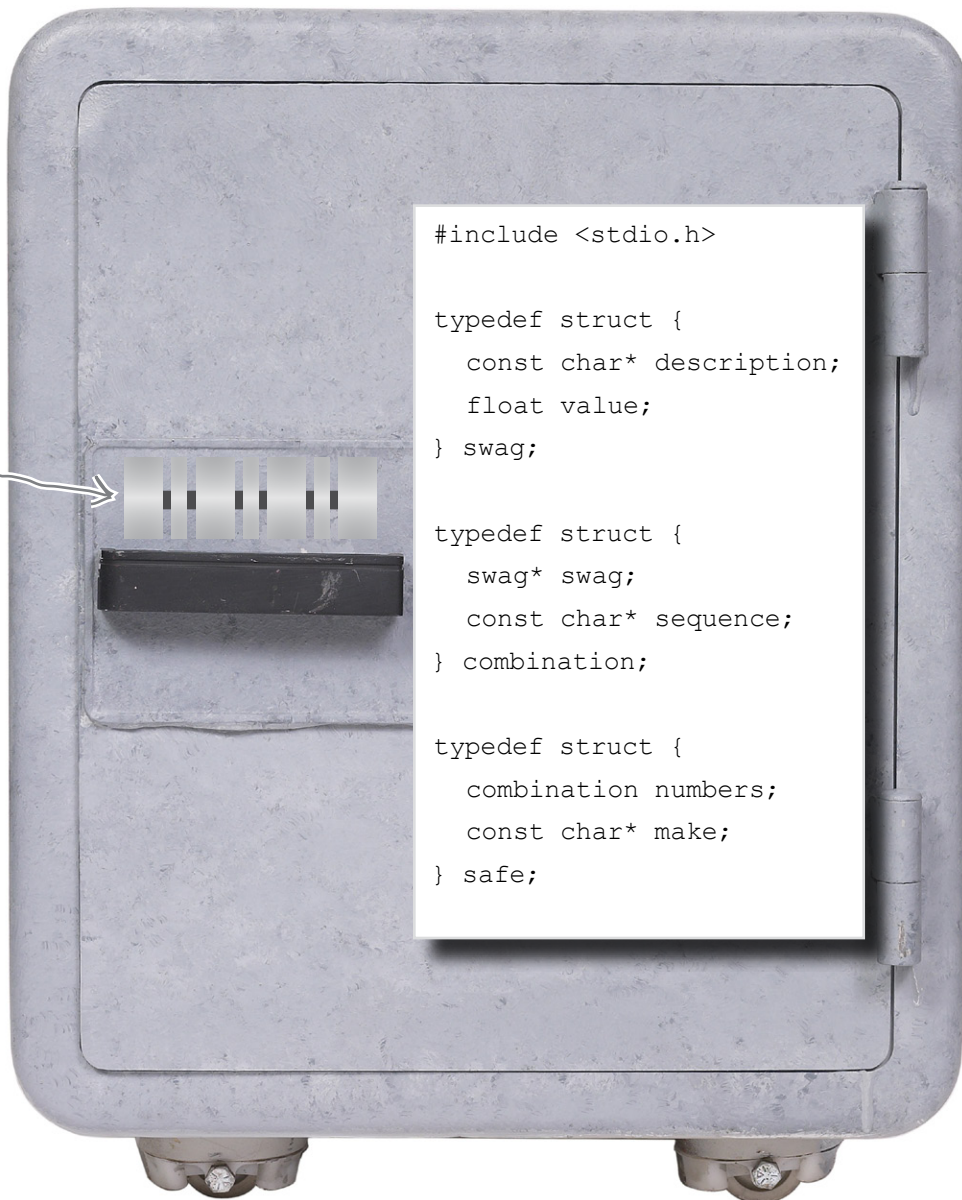
```
void happy_birthday(turtle *a)
{
    a->age = a->age + 1;
    printf("Happy Birthday %s! You are now %i years old!\n",
          a->name, a->age);
}
```



Safe Cracker

Shhh... it's late at night in the bank vault. Can you spin the correct combination to crack the safe? Study these pieces of code, then see if you can find the correct combination that will allow you to get to the gold.

You need to crack this combination.



```
#include <stdio.h>

typedef struct {
    const char* description;
    float value;
} swag;

typedef struct {
    swag* swag;
    const char* sequence;
} combination;

typedef struct {
    combination numbers;
    const char* make;
} safe;
```

The bank created their safe like this:

```
swag gold = {"GOLD!", 1000000.0};
combination numbers = {&gold, "6502"};
safe s = {numbers, "RAMACON250"};
```

What combination will get you to the string "GOLD!". Select one symbol or word from each column to assemble the expression.

con	.	s	+	swag	.	value
<u>s</u>	->	numbers	.	description	-	swag
numbers	:	swag	->	value	->	description
swap	-	gold	-	sequence	+	gold

there are no
Dumb Questions

Q: Why are values copied to parameter variables?

A: The computer will assign the value to each parameter, just as if you'd typed parameter=value and assignments always copy values.

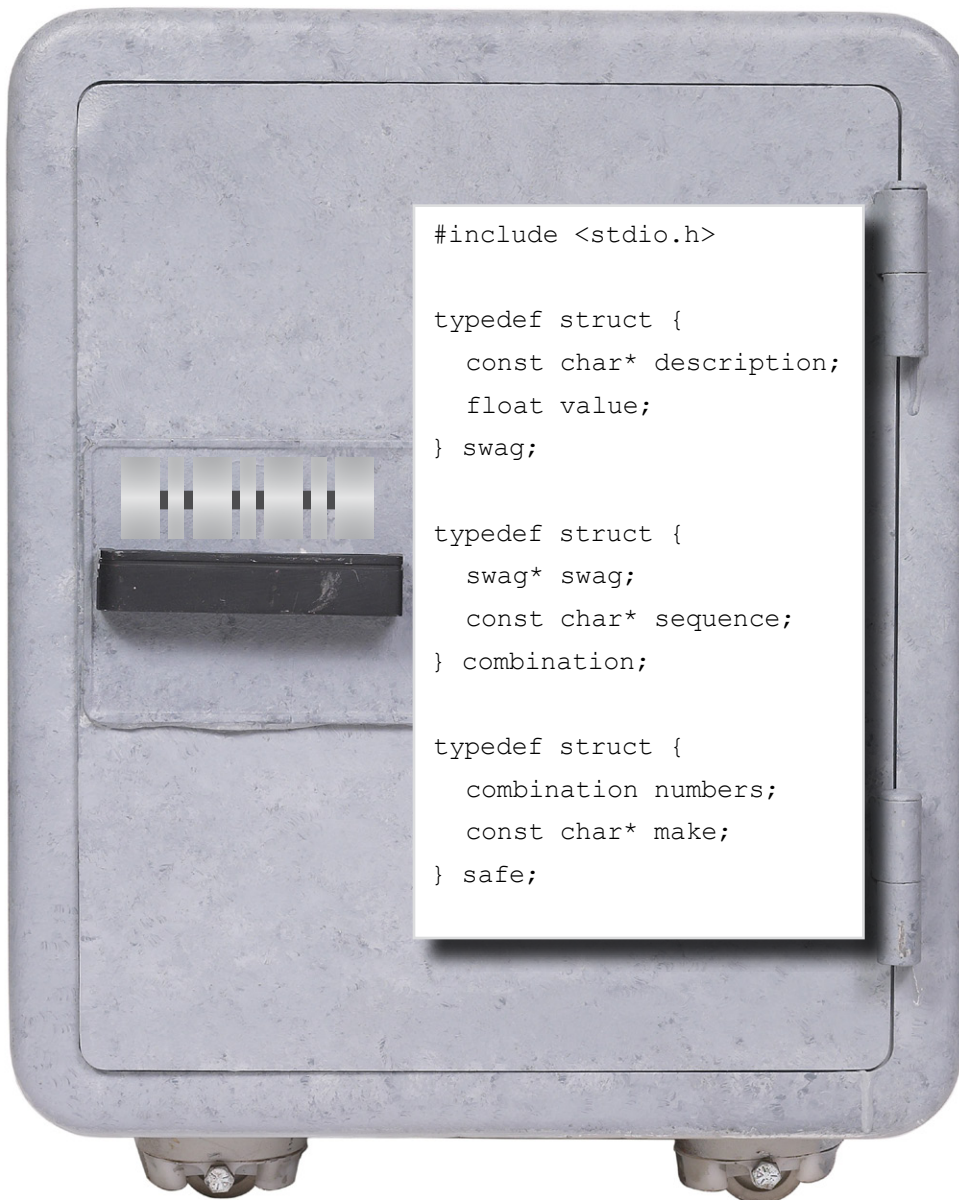
Q: Why isn't *t.age just read as (*t).age?

A: Because the computer evaluates the dot-operator before it evaluates the *.



Safe Cracker Solution

Shhh... it's late at night in the bank vault. Can you spin the correct combination to crack the safe? Study these pieces of code, then see if you can find the correct combination that will allow you to get to the gold.



The bank created their safe like this:

```
swag gold = {"GOLD!", 1000000.0};
combination numbers = {&gold, "6502"};
safe s = {numbers, "RAMACON250"};
```

What combination will get you to the string "GOLD!". Select one symbol or word from each column to assemble the expression.

con	.	s	+	swag	.	value
s	->	numbers	.	description	-	swag
numbers	:	swag	->	value	->	description
swap	-	gold	-	sequence	+	gold

So you can display the gold in the safe with:

```
printf("Contents = %s\n", s.numbers.swag->description);
```



BULLET POINTS

- When you call a function, the values are **copied** to the parameter variables.
- You can create pointers to `structs`, just like any other type.
- `pointer->field`** is the same as **`(*pointer)->field`**.
- The `->` notation cuts down on parentheses and makes the code more readable.

Sometimes the same type of thing needs different types of data

structs gives you the ability to model more complex things from the real world. But sometimes there are pieces of data that don't have a single data-type:



So if you want to record, say, a *quantity* of something, and that quantity might be a **count**, a **weight** or a **volume**, how would you do that? Well - you *could* create several fields with a struct like this:

```
typedef struct {  
    ...  
    short count;  
    float weight;  
    float volume;  
    ...  
} fruit;
```

But there are a few reasons why this is not a good idea:

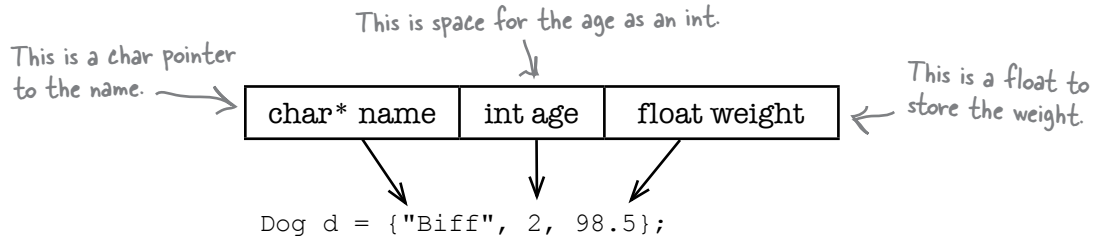
- ★ **It will take up more space in memory.**
- ★ **Someone might set more than one value.**
- ★ **There's nothing called "quantity".**

It would be *really useful* if you could specify something called *quantity* in a data-type and then decide for each particular piece of data whether you are going to record a count, a weight or a volume against it.

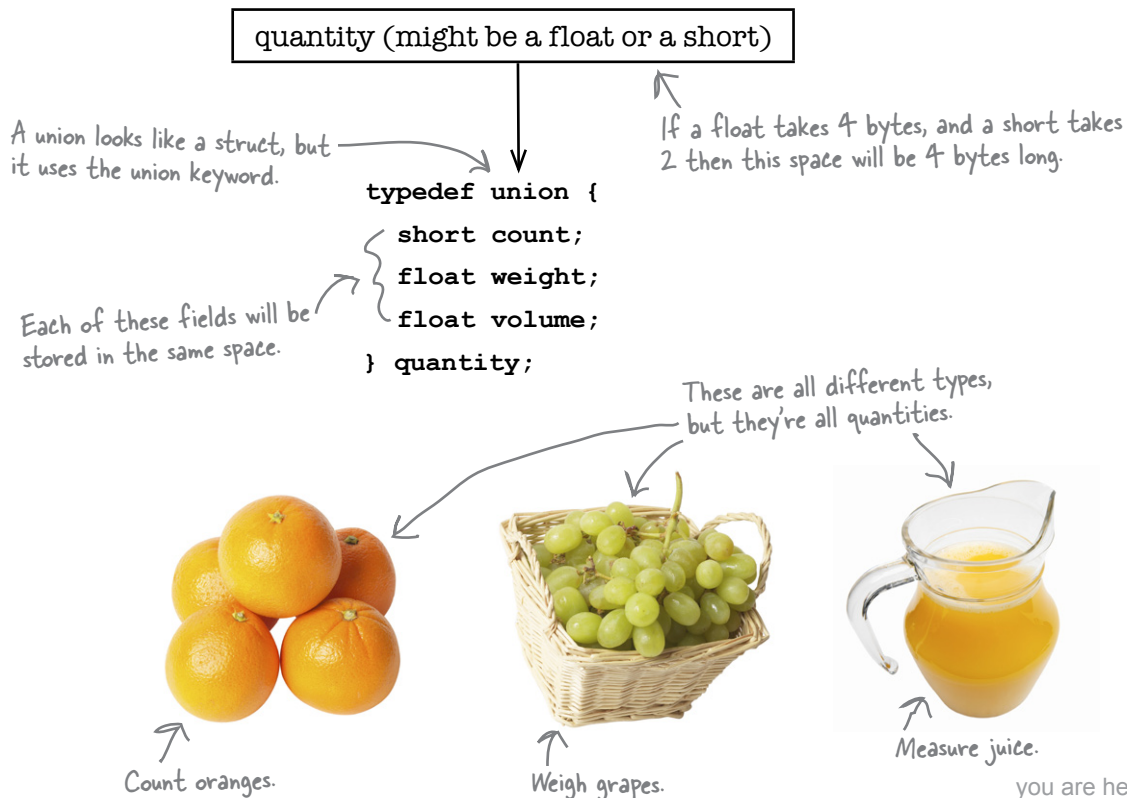
In C you can do just that - by using a union.

A union let's you reuse memory space

Every time you create an instance of a `struct`, the computer will lay out the fields in memory, one after the other:



A **union** is different. A union will use the space for just one of the fields in its definition. So, if you have a union called `quantity`, with fields called `count`, `weight` and `volume`, the computer will give the union enough space for its largest field, and then leave it up to you which value you will store in there. Whether you set the `count`, `weight` or `volume` field, the data will go into the same space in memory:



So how do you use a union?

When you declare a union variable, there are a few ways of setting its value.

C89 style for the first field

If the union is going to store a value for the **first field**, then you can use C89 notation. To give the union a value for its first field, just wrap the value in braces:

```
quantity q = {4};
```

← This means the quantity is a count of 4.

Designated initializers set other values

A **designated initializer** sets a union field value by **name** - like this:

```
quantity q = {.weight=1.5};
```

← This will set the union for a floating point weight value.

Set the value with dot-notation

The third way of setting a union value is by creating the variable on one line, and setting an field value on another line:

```
quantity q;  
q.float = 3.7;
```

Remember: whichever way you set the union's value, there will only ever be **one piece of data stored**. The union just gives you a way of creating a variable that supports **several different data-types**.

there are no Dumb Questions

Q: Why is a union always set to the size of the *largest* field?

A: The computer needs to make sure that a union is always the same size. The only way it can do that is by making sure it is large enough to contain any of the fields.

Q: Why does the C89 notation only set the first field? Why not set it to the first float if I pass it a float value?

A: To avoid ambiguity. If you had, say, a float and a double field, should the computer store {2.1} as a float or a double. By always storing the value in the first field, you know exactly how the data will be



The Polite Guide to Standards

Designated initializers allow you to set struct and union fields by name and are part of the C99 C-standard. They are supported by most modern compilers, but be careful if you are using some *variant* of the C language. For example, *Objective C* supports designated initializers, but **C++ does not**.

Those designated initializers look like they could be useful for structs as well. I wonder if I can use them there...?

Yes - designated initializers can be used to set the initial values of fields in structs as well.

They can be very useful if you have a `struct` that contains a large number of fields and you initially just want to set a few of them. It's also a good way of making your code more readable:

```
typedef struct {
    const char * color;
    int gears;
    int height;
} bike;
bike b = {.height=17, .gears=21};
```

This will set the gears and the height fields, but won't set the color field.



unions are often used with structs

Once you've created a union, you've created a *new data type*. That means you can use its values anywhere you would use another data-type like an `int` or a `struct`. That means you can combine them with structs:

```
typedef struct {
    const char* name;
    const char* country;
    quantity amount;
} fruit_order;
```

And you can access the values in the struct/union combination using the dot or "`->`" notation you used before:

It's `.amount` because that's the name of the struct quantity variable.

```
fruit_order apples = {"apples", "England", .amount.weight=4.2};
printf("This order contains %2.2f lbs of %s\n", apples.amount.weight, apples.name);
```

Here we're using a double designated identifier. `.amount` for the struct and `.weight` for the `.amount`.

This will print "This order contains 4.20 lbs of apples".



Mixed UP Mixers

It's Margarita Night at the Head First Lounge, but after one too many samples, it looks like the guys have mixed up their recipes. See if you can find the matching code fragments for the different Margarita mixes.

Here are the basic ingredients:

```
typedef union {
    float lemon;
    int lime_pieces;
} lemon_lime;

typedef struct {
    float tequila;
    float cointreau;
    lemon_lime citrus;
} margarita;
```

Here are the different margaritas:

```
margarita m = {2.0, 1.0, {0.5}};
```

```
margarita m = {2.0, 1.0, .citrus.lemon=2};
```

```
margarita m = {2.0, 1.0, 0.5};
```

```
margarita m = {2.0, 1.0, {.lime_pieces=1}};
```

```
margarita m = {2.0, 1.0, {1}};
```

```
margarita m = {2.0, 1.0, {2}};
```

And finally here are the different mixes and the drinks recipes they produce. Which of the margaritas need to be added to these pieces of code to generate the correct recipes:

```
.....
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%2.1f
measures of juice\n", m.tequila, m.cointreau, m.citrus.lemon);
```

```
2.0 measures of tequila
1.0 measures of cointreau
2.0 measures of juice
```

```
.....
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%2.1f
measures of juice\n", m.tequila, m.cointreau, m.citrus.lemon);
```

```
2.0 measures of tequila
1.0 measures of cointreau
0.5 measures of juice
```

```
.....
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%i pieces
of lime\n", m.tequila, m.cointreau, m.citrus.lime_pieces);
```

```
2.0 measures of tequila
1.0 measures of cointreau
1 pieces of lime
```

BE the Compiler

One of these pieces of code compiles, the other doesn't. Your job is to play like you're the compiler and say which one compiles, and why the other one doesn't.



```
margarita m = {2.0, 1.0, {0.5}};
```

```
margarita m;
m = {2.0, 1.0, {0.5}};
```



Mixed UP Mixers Solution

It's Margarita Night at the Head First Lounge, but after one too many samples, it looks like the guys have mixed up their recipes. See if you can find the matching code fragments for the different Margarita mixes.

Here are the basic ingredients:

```
typedef union {
    float lemon;
    int lime_pieces;
} lemon_lime;

typedef struct {
    float tequila;
    float cointreau;
    lemon_lime citrus;
} margarita;
```

Here are the different margaritas:

```
margarita m = {2.0, 1.0, .citrus.lemon=2};
```

```
margarita m = {2.0, 1.0, 0.5};
```

```
margarita m = {2.0, 1.0, {1}};
```

None of these
lines were used.

And finally here are the different mixes and the drinks recipes they produce. Which of the margaritas need to be added to these pieces of code to generate the correct recipes:

```
margarita m = {2.0, 1.0, {2}};
.....
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%2.1f
measures of juice\n", m.tequila, m.cointreau, m.citrus.lemon);

2.0 measures of tequila
1.0 measures of cointreau
2.0 measures of juice
```

```
margarita m = {2.0, 1.0, {0.5}};
.....
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%2.1f
measures of juice\n", m.tequila, m.cointreau, m.citrus.lemon);

2.0 measures of tequila
1.0 measures of cointreau
0.5 measures of juice
```

```
margarita m = {2.0, 1.0, {.lime_pieces=1}};
.....
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%i pieces
of lime\n", m.tequila, m.cointreau, m.citrus.lime_pieces);

2.0 measures of tequila
1.0 measures of cointreau
1 pieces of lime
```

BE the Compiler Solution

One of these pieces of code compiles, the other doesn't. Your job is to play like you're the compiler and say which one compiles, and why the other one doesn't.



```
margarita m = {2.0, 1.0, {0.5}};
```

← This one compiles perfectly. It's actually just one of the drinks above!

```
margarita m;
```

```
m = {2.0, 1.0, {0.5}};
```

→ This one doesn't compile because the compiler will only know that {2.0, 1.0, {0.5}} represents a struct if it's used on the same line that a struct is declared. When it's on a separate line the compiler thinks it's an array.

Hey - wait a minute... You're setting all these different values with all these different types and you're storing them in **the same place in memory**.... How do I know if I stored a float in there once I've stored it? What's to stop me reading it as a short or something...??? Hello...?



That's a really good point - you can store lots of possible values in a union, but you have *no way of knowing* what type it was once it's stored.

The compiler won't be able to keep track of the fields that are set and read in a union so there's nothing to stop us setting one field and reading another. Is that a problem? Sometimes it can be a **BIG PROBLEM**...

```
typedef union {
    float weight;
    int count;
} cupcake;

int main()
{
    cupcake order = {2};
    printf("Cupcakes quantity: %i\n", order.count);
    return 0;
}
```

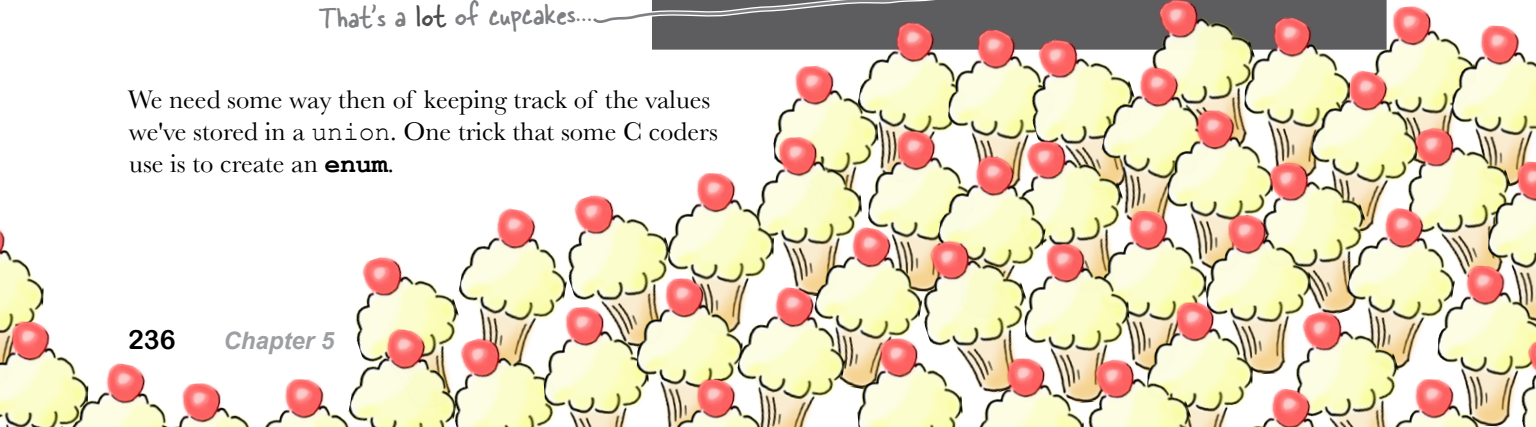
By mistake the programmer has set the weight not the count.

They set the weight - but they're reading the count.

This is what the program did.

```
File Edit Window Help
> gcc badunion.c -o badunion && ./badunion
Cupcakes quantity: 1073741824
```

That's a lot of cupcakes...



We need some way then of keeping track of the values we've stored in a union. One trick that some C coders use is to create an **enum**.

An enum variable stores a symbol

Sometimes you don't want to store a number or a piece of text, you want to store something from a list of **symbols**. If you want to record a day of the week, you only want to store MONDAY, TUESDAY, WEDNESDAY,.... You don't need to store the text, because there are only ever going to be 7 different values to choose from.

That's why enums were invented.

enum let you create a list of symbols, like this:

Possible colors in our enum. → `enum colors {RED, GREEN, PUCE};`

← The values are separated by commas.

← We could have given our type a proper name with typedef.

Any variable that is defined with a type of `enum colors` can then only be set to one of the keywords in the list. So you might define an `enum colors` variable like this:

```
enum colors favorite = PUCE;
```

Under the covers the computer will just assign numbers to each of the symbols in your list, and the enum will just store a number. But you don't need to worry about what the numbers are - your C code can just refer to the symbols. That'll make your code easier to read and it will prevent storing values like "REB" or "PUSE":



Watch it!

structs and unions separate items with semi-colons (;) but enums use commas.

The computer will spot this is not a legal value so it won't compile.

Nope I'm not compiling that - it's not on my list;

```
enum colors favorite = PUCE;
```



So that's how enums work, but how do they help us keep track of unions? Let's look at an example...



Code Magnets

Because you can create new data-types with enums, you can store them inside structs and unions. In this program an enum is being used to track the kinds of quantities being stored. Do you think you can work out where the missing pieces of code go?

```
#include <stdio.h>

typedef enum {
    COUNT, POUNDS, PINTS
} unit_of_measure;

typedef union {
    short count;
    float weight;
    float volume;
} quantity;

typedef struct {
    const char* name;
    const char* country;
    quantity amount;
    unit_of_measure units;
} fruit_order;

void display(fruit_order order)
{
    printf("This order contains ");

    if (..... == PINTS)

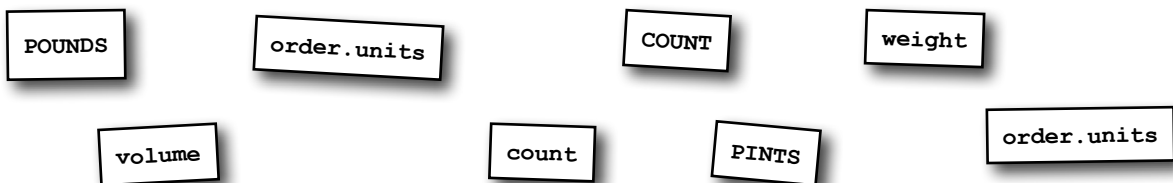
        printf("%2.2f pints of %s\n", order.amount....., order.name);
}
```

```

else if ( ..... == ..... )
    printf("%.2f lbs of %s\n", order.amount.weight, order.name);
else
    printf("%i %s\n", order.amount. .... , order.name);
}

int main()
{
    fruit_order apples = {"apples", "England", .amount.count=144, .....};
    fruit_order strawberries = {"strawberries", "Spain", .amount.....=17.6, POUNDS};
    fruit_order oj = {"orange juice", "U.S.A.", .amount.volume=10.5, .....};
    display(apples);
    display(strawberries);
    display(oj);
    return 0;
}

```





Code Magnets Solution

Because you can create new data-types with enums, you can store them inside structs and unions. In this program an enum is being used to track the kinds of quantities being stored. Do you think you can work out where the missing pieces of code go?

```
#include <stdio.h>

typedef enum {
    COUNT, POUNDS, PINTS
} unit_of_measure;

typedef union {
    short count;
    float weight;
    float volume;
} quantity;

typedef struct {
    const char* name;
    const char* country;
    quantity amount;
    unit_of_measure units;
} fruit_order;

void display(fruit_order order)
{
    printf("This order contains ");

    if (order.units == PINTS)
        printf("%.2f pints of %s\n", order.amount.volume, order.name);
```

```

else if (order.units == POUNDS)
    printf("%.2f lbs of %s\n", order.amount.weight, order.name);
else
    printf("%i %s\n", order.amount.count, order.name);
}

int main()
{
    fruit_order apples = {"apples", "England", .amount.count=144, COUNT};
    fruit_order strawberries = {"strawberries", "Spain", .amount.weight=17.6, POUNDS};
    fruit_order oj = {"orange juice", "U.S.A.", .amount.volume=10.5, PINTS};
    display(apples);
    display(strawberries);
    display(oj);
    return 0;
}

```

When you run the program you get this:

```

File Edit Window Help
> gcc enumtest.c -o enumtest
This order contains 144 apples
This order contains 17.60 lbs of strawberries
This order contains 10.50 pints of orange juice

```



union: ...so I said to the code - Hey look. I don't care if you gave me a float or not. You asked for an int. You got an int

struct: Dude, that was totally uncalled for.

union: That's what I said. It's totally uncalled for.

struct: Everyone knows you only have one storage location.

union: Exactly. Everything is one. I'm, like, zen that way...

enum: What happened, dude?

struct: Shut up Enum. I mean the guy was crossing the line.

union: I mean if he had just left a record. You know said I stored this as an int. It just needed an enum or something.

enum: You want me to do what?

struct: Shut up Enum.

union: I mean if he'd wanted to store several things at once, he should have called you - am I right?

struct: Order. That's what these people don't grasp.

enum: Ordering what?

struct: Separation and sequencing. I keep several things alongside each other. All at the same time dude.

union: That's just my point.

struct: All. At. The. Same. Time.

enum: (Pause)So has there been a problem?

union: Please, Enum? I mean these people just need to

make a decision. Wanna store several things - use you. But store just one thing with different possible types? Dude's your man.

struct: I'm calling him.

union: Hey, wait...

enum: Who's he calling, dude?

struct/union: Shut up Enum.

union: Look - let's not cause any more problems here.

struct: Hello? Could I speak to the Bluetooth service, please?

union: Hey - let's just think about this.

struct: What do you mean, he'll give me a callback?

union: I'm just. This doesn't seem like a good idea.

struct: No - let me leave you a message, my friend.

union: Please - just put the phone down.

enum: Who's on the phone, dude?

struct: Be quiet Enum. Can't you see I'm on the phone here? Listen you just tell him that if he wants to store a float and an int, he needs to come see me. Or I'm going to come see him. Understand me? Hello? Hello?

union: Easy man. Just try and keep calm.

struct: On hold? They put me on ^*^ing hold!

union: They what? Pass me the phone... Oh.... that... man. The Eagles! I hate the Eagles....

enum: So if you pack your fields, is that why you're so fat?

struct: You are entering a world of pain, my friend.

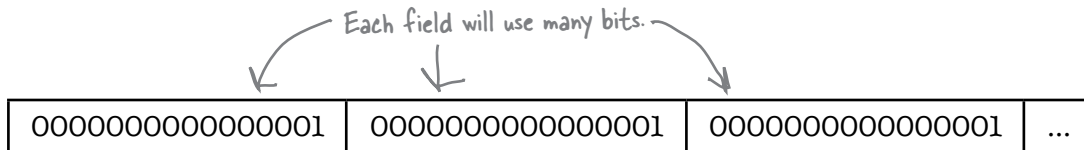
Sometimes you want control at the bit level

Let's say you need a struct that will contain a lot of yes/no values. You *could* create the struct with a series of shorts or ints:

```
typedef struct {
    short low_pass_vcf;
    short filter_coupler;
    short reverb;
    short sequential;
    ...
} synth;
```

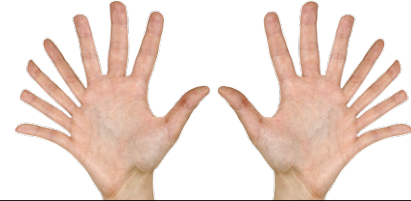
Each of these fields will contain 1 for true or 0 for false.

There are a lot more fields that follow this.



And that would work. The problem? The short fields will take up a lot more space than the *single bit* that we need for **true/false** value. It's wasteful. It would be much better if we could create a struct that could hold a sequence of single bits for the values.

That's why **bitfields** were created.



Geek Binary Digits

When you're dealing with binary value, it would be great if you had some way of specifying the 1s and 0s in a literal, like:

```
int x = 01010100;
```

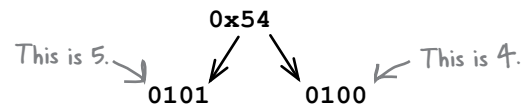
Unfortunately, C doesn't support **binary literals** but it *does* support **hexadecimal literals**. Every time C sees a number beginning with 0x, it treats the number as **base 16**:

```
int x = 0x54;
```

← This is not decimal 54.

But how do you convert back and forth between hexadecimal and binary? And is it any easier than

converting binary and **decimal**? The good news is that you can convert hex to binary **one digit at a time**:



Each hexadecimal digit matches a binary digit of length 4. All you need to learn are the binary patterns for the numbers 0 - 15, and you will soon be able to convert binary to hex and back again in your head within seconds.

Bitfields store a custom number of bits

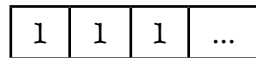
A **bitfield** lets you specify *how many bits* an individual field will store. For example, we could write our struct like this:

```
typedef struct {
    unsigned int low_pass_vcf:1;
    unsigned int filter_coupler:1;
    unsigned int reverb:1;
    unsigned int sequential:1;
    ...
} synth;
```

Each field must be an unsigned int.

This means the field will only use 1 bit of storage.

By using bitfields we can make sure each field takes up only one bit.



If you have a sequence of bitfields, the computer can **squash them together** to save space. So if you have 8 single-bit bitfields, the computer can store them in a single byte.

Let's see how good you are at using bitfields.



Watch it!

Bitfields can save space if they are collected together in a struct.

But if the compiler finds a single bitfield on its own, it might still have to pad it out to the size of a word. That's why bitfields are usually grouped together.

How many bits do I need?

Bitfields can be used to store a sequence of true/false values, but they're also useful for other short-range value, like months of the year. If you want to store a month number in a struct you know it will have a value of, say, 0 - 11. You can store those values in **4 bits**. Why? Because 4 bits let you store 0-15, but 3 bits only store 0-7.

```
...
    unsigned int month_no:4;
...
```



Exercise

Back at the Head First Aquarium they're creating a customer satisfaction survey. Let's see if you can use bitfields to create a matching struct.



Aquarium Questionnaire

Is this your first visit?	
Will you come again?	
Number of fingers lost in the piranha tank:	
Did you lose a child in the shark exhibit?	
How many days a week would you visit if you could?	

```
typedef struct {
    unsigned int first_visit: .....;
    unsigned int come_again: .....;
    unsigned int fingers_lost: .....;
    unsigned int shark_attack: .....;
    unsigned int days_a_week: .....;
} survey;
```

↙ You need to decide
how many bits to use.



**Exercise
Solution**

Back at the Head First Aquarium they're creating a customer satisfaction survey. Let's see if you can use bitfields to create a matching struct.



Aquarium Questionnaire

Is this your first visit?	
Will you come again?	
Number of fingers lost in the piranha tank:	
Did you lose a child in the shark exhibit?	
How many days a week would you visit if you could?	

```
typedef struct {
    unsigned int first_visit: 1 ;
    unsigned int come_again: 1 ;
    unsigned int fingers_lost: 4 ;
    unsigned int shark_attack: 1 ;
    unsigned int days_a_week: 3 ;
} survey;
```

← 1 bit can store 2 values: true/false.
 ← 4 bits are needed to store up to 10.
 ← 3 bits can store numbers up to 7.

there are no
Dumb Questions

Q: Why doesn't C support binary literals?

A: Because they take up a lot of space and it's usually more efficient to write hex values.

Q: Why do I need 4 bits to store a value up to 10?

A: 4 bits can store values from 0 to binary 1111 == 15. But 3 bits can only store values up to binary 111 == 7.

Q: What happens if you try to put a number into a bitfield that is too large?

A: The computer will transfer just the bits it needs.

Q: So what if I try to put the value 9 into a 3 bit field?

A: The computer will store a value of 1 in it, because 9 == 1001 in binary, so the computer transfers 001.

Q: Are bitfields really just used to save space?

A: No. They are important if you need to read low-level binary information.

Q: Such as?

A: If you are reading or writing some sort of custom binary file.



BULLET POINTS

- A `union` allows you to store different data-types in the same memory location.
- A designated initializer sets a field value by name.
- Designated initializers are part of the C99 standard. They are not supported in C++.
- If you declare a `union` with a value in `{braces}`, it will be stored with the type of the first field.
- The compiler will let you store one field in an union and read a complete different field.
- `enums` store symbols.
- Bit-fields allow you to store a field with a custom number of bits.
- Bit-fields are always declared as `unsigned ints`.



Your C Toolbox

You've got Chapter 5 under your belt and now you've added structs, unions and bitfields to your tool box. For a complete list of tooltips in the book, see Appendix X.

typedef lets you create an alias for a data-type. .

A struct combines data types together.

You can read struct fields with dot notation.

You can initialize structs with {array, like, notation}.

-> notation lets you easily update fields using a struct pointer.

Designated initializers lets you set struct and union fields by name.

unions can different data types in one location.

enums let you create a set of symbols.

Bitfields give you control over the exact bits stored in a struct.