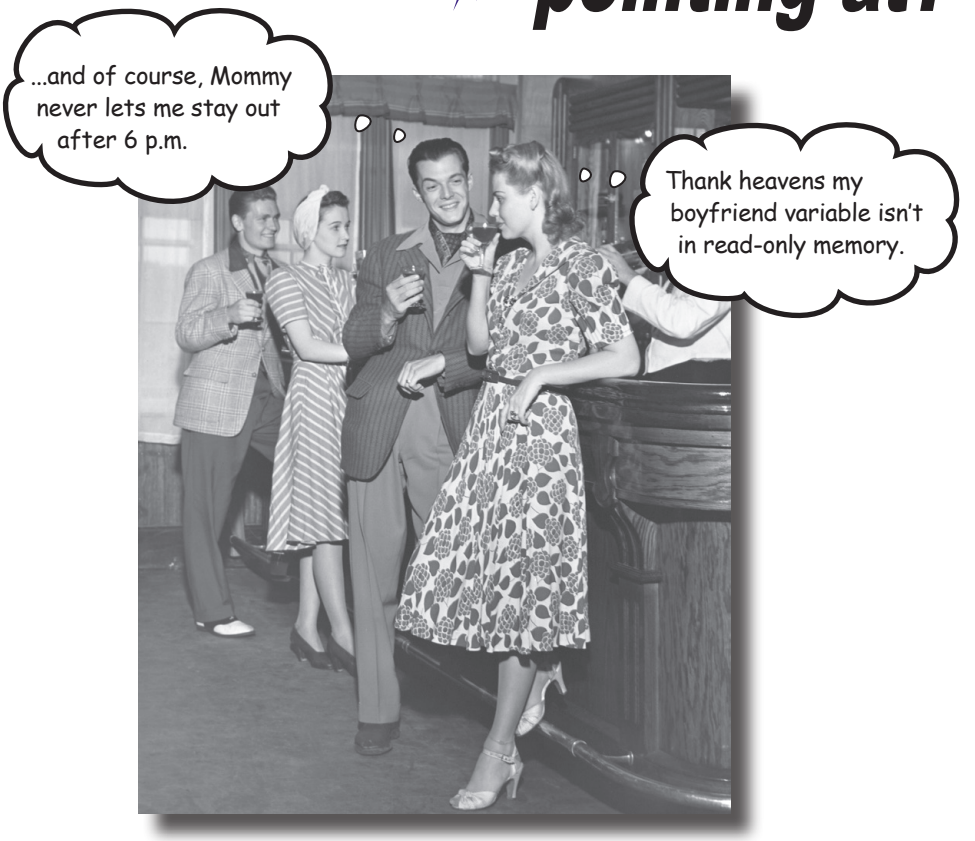
2 memory and pointers

What are you pointing at?



If you really 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.

C code includes pointers

Pointers are one of the most fundamental things to understand in the C programming language. So what's a pointer? A **pointer** is just the address of a piece of data in memory.

Pointers are used in C for a couple of reasons.

To best understand pointers, go slowly.

1. **Instead of passing around a whole copy of the data, you can just pass a pointer.**

2. **You might want two pieces of code to work on the same piece of data rather than a separate copy.**

Pointers help you do both these things: avoid copies and share data. But if pointers are just addresses, why do some people find them confusing? Because they're a **form of indirection**. If you're not careful, you can quickly get lost chasing pointers through memory. The trick to learning how to use C pointers is to *go slowly*.

Don't try to rush this chapter.

Pointers are a simple idea, but you need to take your time and understand everything. Take frequent breaks, drink plenty of water, and if you really get stuck, take a nice long bath.

Digging into memory

To understand what pointers are, you'll need to dig into the memory of the computer.

Every time you declare a variable, the computer creates space for it somewhere in memory. If you declare a variable *inside* a function like `main()`, the computer will store it in a section of memory called the **stack**. If a variable is declared *outside any function*, it will be stored in the **globals** section of memory.

```
int y = 1;

int main()
{
  int x = 4;
  return 0;
}
```

Variable y will live in the globals section. Memory address 1,000,000. Value 1.

Variable x will live in the stack. Memory address 4,100,000. Value 4.

The computer might allocate, say, memory location 4,100,000 in the stack for the x variable. If you assign the number 4 to the variable, the computer will store 4 at location 4,100,000.

If you want to find out the memory address of the variable, you can use the **&** operator:

```
printf("x is stored at %p\n", &x);
```

&x is the address of x.

%p is used to format addresses.

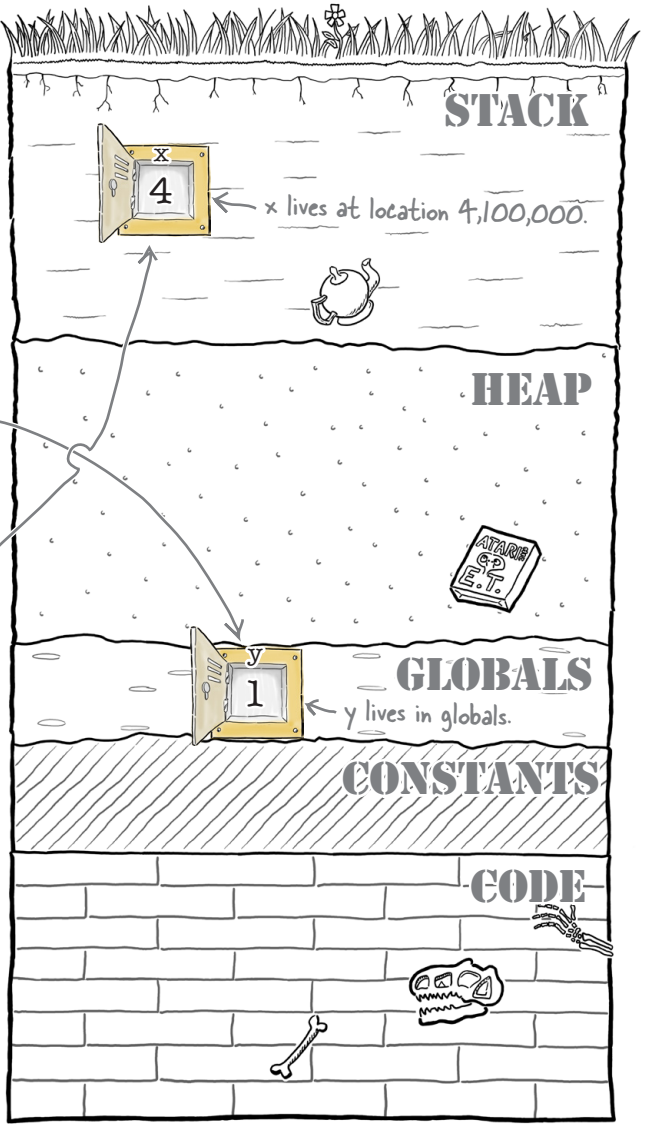
This is what the code will print.

```
x is stored at 0x3E8FA0
```

This is 4,100,000 in hex (base 16) format.

You'll probably get a different address on your machine.

The address of the variable tells you where to find the variable in memory. That's why an address is also called a ***pointer***, because it ***points*** to the variable in memory.

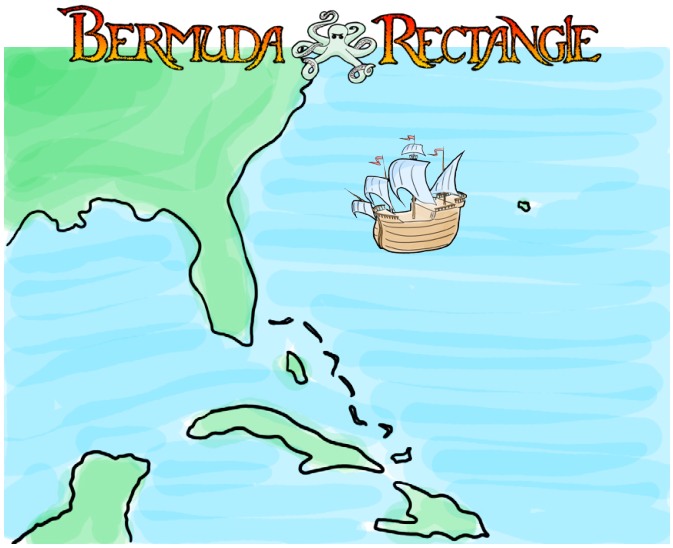


A variable declared inside a function is usually stored in the stack.

A variable declared outside a function is stored in globals.

Set sail with pointers

Imagine you're writing a game in which players have to navigate their way around the…



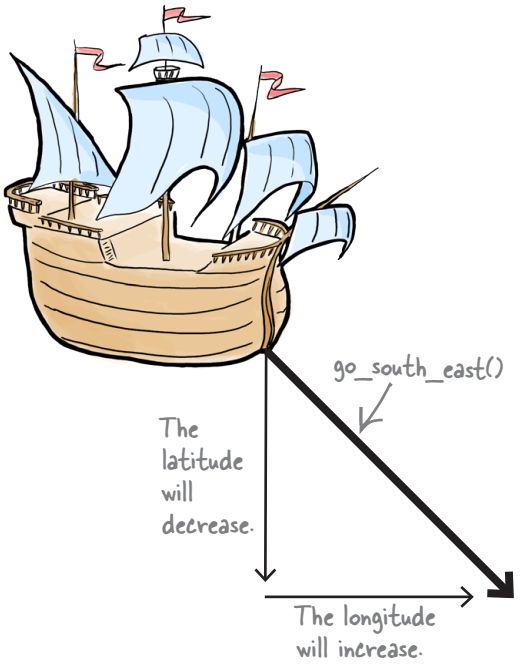
The game will need to keep control of lots of things, like scores and lives and the current location of the players. You won't want to write the game as one large piece of code; instead, you'll create lots of smaller functions that will each do something useful in the game:

```
go_south_east()
go_north_west()
speaks_in_present_tense()
go_south()
acquire_facial_hair()
eat_rat()
die_of_scurvy()
make_one_sequel_too_many()
```

What does any of this have to do with pointers? Let's begin coding without worrying about pointers at all. You'll just use variables as you always have. A major part of the game is going to be navigating your ship around the Bermuda Rectangle, so let's dive deeper into what the code will need to do in one of the navigation functions.

Set sail sou'east, Cap'n

The game will track the location of players using *latitudes* and *longitudes*. The latitude is how far north or south the player is, and the longitude is her position east or west. If a player wants to travel southeast, that means her latitude will go *down*, and her longitude will go *up*:



So you could write a `go_south_east()` function that takes arguments for the latitude and longitude, which it will then increase and decrease:

```
#include <stdio.h>

void go_south_east(int lat, int lon)
{
  lat = lat - 1;
  lon = lon + 1;
}

int main()
{
  int latitude = 32;
  int longitude = -64;
  go_south_east(latitude, longitude);
  printf("Avast! Now at: [%i, %i]\n", latitude, longitude);
  return 0;
}
```

Pass in the latitude and longitude.

Decrease the latitude.

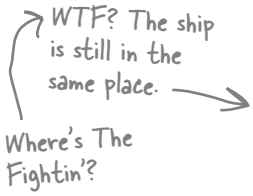
Increase the longitude.

The program starts a ship at location [32, –64], so if it heads southeast, the ship's new position will be [31, –63]. At least it will be *if the code works*…

Look at the code carefully. Do you think it will work? Why? Why not?

Test Drive

The code should move the ship southeast from [32, –64] to the new location at [31, –63]. But if you compile and run the program, this happens:



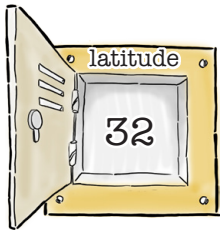

```
File Edit Window Help Savvy?
> gcc southeast.c -o southeast
> ./southeast
Avast! Now at: [32, -64]
>
```

The ship's location stays *exactly* the same as before.

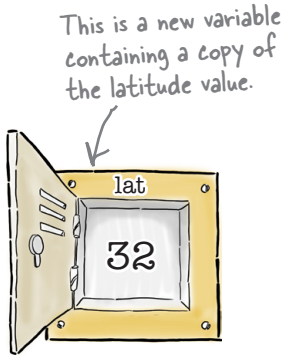
C sends arguments as values

The code broke because of the way that C calls functions.

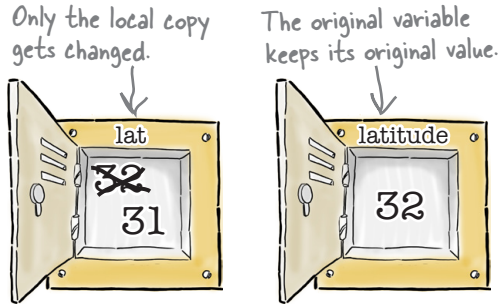
1. Initially, the `main()` function has a local variable called `latitude` that had value 32.



2. When the computer calls the `go_south_east()` function, it **copies the value** of the `latitude` variable to the `lat` argument. This is just an assignment from the `latitude` variable to the `lat` variable. When you call a function, you don't send the *variable* as an argument, just its *value*.



3. When the `go_south_east()` function changes the value of `lat`, the function is just changing its local copy. That means when the computer returns to the `main()` function, the `latitude` variable still has its original value of 32.

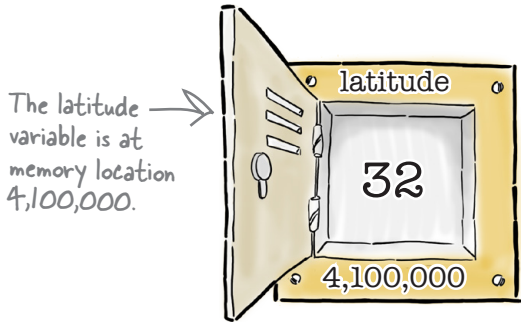


But if that's how C calls functions, how can you ever write a function that updates a variable?

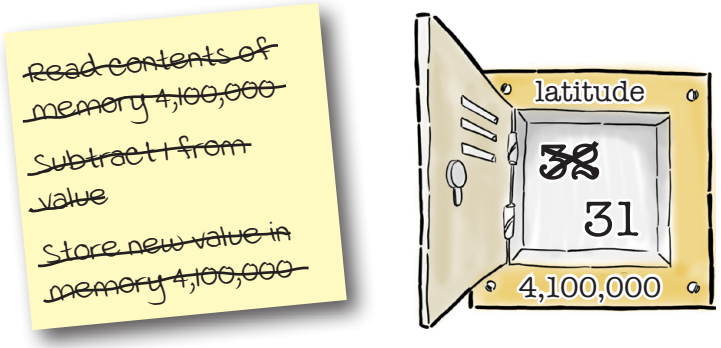
It's easy if you use pointers...

Try passing a pointer to the variable

Instead of passing the *value* of the `latitude` and `longitude` variables, what happens if you pass their *addresses*? If the `longitude` variable lives in the stack memory at location 4,100,000, what happens if you pass the location number 4,100,000 as a parameter to the `go_south_east()` function?



If the `go_south_east()` function is told that the `latitude` value lives at location 4,100,000, then it will not only be able to find the current `latitude` value, but it will also be able to change the contents of the original `latitude` variable. All the function needs to do is read and update the contents of memory location 4,100,000.



Because the `go_south_east()` function is updating the original `latitude` variable, the computer will be able to print out the updated location when it returns to the `main()` function.

Pointers make it easier to share memory

This is one of the main reasons for using pointers—to let functions *share* memory. The data created by one function can be modified by another function, so long as it knows where to find it in memory.

Now that you know the theory of using pointers to fix the `go_south_east()` function, it's time to look at the details of how you do it.

there are no Dumb Questions

Q: I printed the location of the variable on my machine and it wasn't 4,100,000. Did I do something wrong?

A: You did nothing wrong. The memory location your program uses for the variables will be different from machine to machine.

Q: Why are local variables stored in the stack and globals stored somewhere else?

A: Local and global variables are used differently. You will only ever get one copy of a global variable, but if you write a function that calls itself, you might get very many instances of the same local variable.

Q: What are the other areas of the memory used for?

A: You'll see what the other areas are for as you go through the rest of the book.

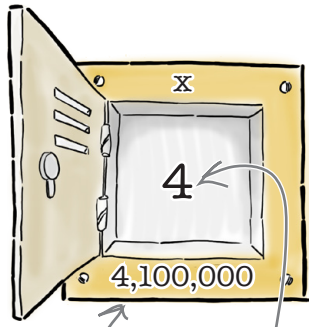
Using memory pointers

There are **three** things you need to know in order to use pointers to read and write data.

1 **Get the address of a variable.**
You've already seen that you can find where a variable is stored in memory using the **&** operator:

The %p format will print out the location in hex (base 16) format.

```
int x = 4;
printf("x lives at %p\n", &x);
```



& will find the address of the variable: 4,100,000.

But once you've got the address of a variable, you may want to store it somewhere. To do that, you will need a **pointer variable**. A pointer variable is just a variable that stores a memory address. When you declare a pointer variable, you need to say what kind of data is stored at the address it will point to:

This is a pointer variable for an address that stores an int.

```
int *address_of_x = &x;
```

2 **Read the contents of an address.**
When you have a memory address, you will want to read the data that's stored there. You do that with the ***** operator:

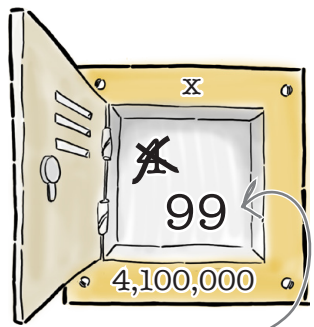
```
int value_stored = *address_of_x;
```

This will read the contents at the memory address given by address_of_x. This will be set to 4: the value originally stored in the x variable.

The * and & operators are opposites. The & operator takes a piece of data and tells you where it's stored. The * operator takes an address and tells you what's stored there. Because pointers are sometimes called *references*, the * operator is said to **dereference** a pointer.

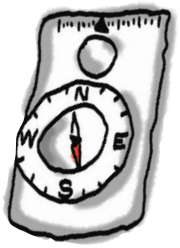
3 **Change the contents of an address.**
If you have a pointer variable and you want to change the data at the address where the variable's pointing, you can just use the * operator again. But this time you need to use it on the **left side** of an assignment:

```
*address_of_x = 99;
```



This will change the contents of the original x variable to 99.

OK, now that you know how to read and write the contents of a memory location, it's time for you to fix the go_south_east() function.



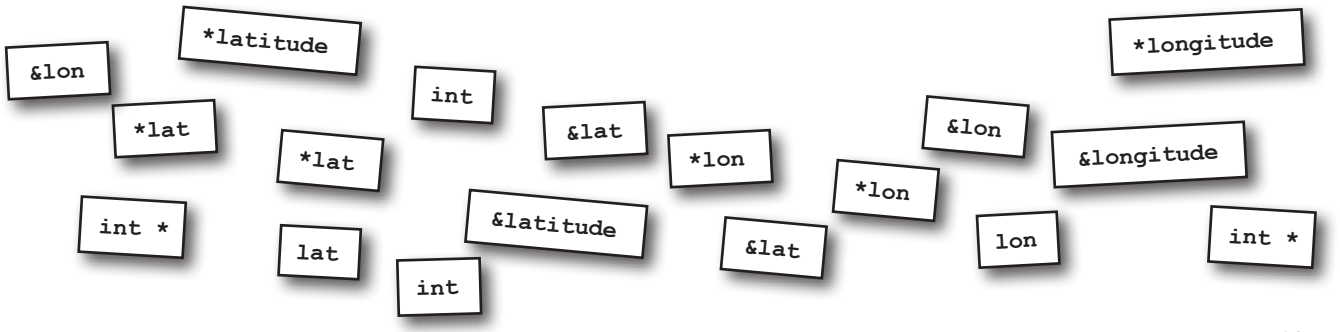
Compass Magnets

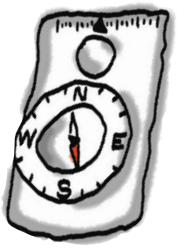
Now you need to fix the `go_south_east()` function so that it uses pointers to update the correct data. Think carefully about what type of data you want to pass to the function, and what operators you'll need to use to update the location of the ship.

```
#include <stdio.h>

void go_south_east(..... lat, ..... lon)
{
  ..... = ..... - 1;

  ..... = ..... + 1;
}

int main()
{
  int latitude = 32;
  int longitude = -64;

  go_south_east(....., .....);
  printf("Avast! Now at: [%i, %i]\n", latitude, longitude);
  return 0;
}
```

What kinds of arguments will store memory addresses for ints?

Remember: you're going to pass the **addresses** of variables.





Compass Magnets Solution

You needed to fix the `go_south_east()` function so that it uses pointers to update the correct data. You were to think carefully about what type of data you want to pass to the function, and what operators you'll need to use to update the location of the ship.

```
#include <stdio.h>

void go_south_east(int * lat, int * lon)
{
  *lat = *lat - 1;
  *lon = *lon + 1;
}

int main()
{
  int latitude = 32;
  int longitude = -64;

  go_south_east(&latitude, &longitude);
  printf("Avast! Now at: [%i, %i]\n", latitude, longitude);
  return 0;
}
```

The arguments will store pointers so they need to be int *.

*lat can read the old value and set the new value.

You need to find the address of the latitude and longitude variables with &.

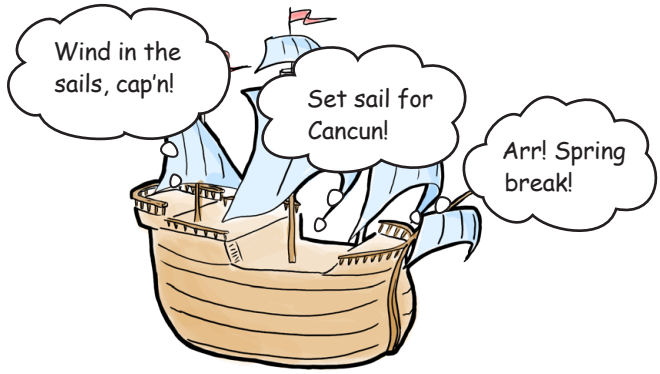
&lon *latitude int &lat *longitude &lon lat int &lat lon

Test Drive

Now if you compile and run the *new* version of the function, you get this:

This is southeast of the original location.

```
File  Edit  Window  Help  Savvy?
> gcc southeast.c -o southeast
> ./southeast
Avast! Now at: [31, -63]
>
```



The code works.

Because the function takes pointer arguments, it's able to update the original `latitude` and `longitude` variables. That means that you now know how to create functions that not only return values, but can also update any memory locations that are passed to them.

BULLET POINTS

- Variables are allocated storage in memory.
- Local variables live in the stack.
- Global variables live in the globals section.
- Pointers are just variables that store memory addresses.
- The `&` operator finds the address of a variable.
- The `*` operator can read the contents of a memory address.
- The `*` operator can also set the contents of a memory address.

there are no Dumb Questions

Q: Are pointers actual address locations? Or are they some other kind of reference?

A: They're actual numeric addresses in the process's memory.

Q: What does that mean?

A: Each process is given a simplified version of memory to make it look like a single long sequence of bytes.

Q: And memory's not like that?

A: It's more complicated in reality. But the details are hidden from the process so that the operating system can move the process around in memory, or unload it and reload it somewhere else.

Q: Is memory not just a long list of bytes?

A: The computer will probably structure its physical memory in a more complex way. The machine will typically group memory addresses into separate banks of memory chips.

Q: Do I need to understand this?

A: For most programs, you don't need to worry about the details of how the machine arranges its memory.

Q: Why do I have to print out pointers using the `%p` format string?

A: You don't have to use the `%p` string. On most modern machines, you can use `%li`—although the compiler may give you a warning if you do.

Q: Why does the `%p` format display the memory address in hex format?

A: It's the way engineers typically refer to memory addresses.

Q: If reading the contents of a memory location is called *dereferencing*, does that mean that pointers should be called *references*?

A: Sometimes coders will call pointers *references*, because they refer to a memory location. However, C++ programmers usually reserve the word *reference* for a slightly different concept in C++.

Q: Oh yeah, C++. Are we going to look at that?

A: No, this book looks at C only.

How do you pass a string to a function?

You know how to pass simple values as arguments to functions, but what if you want to send something more complex to a function, like a string? If you remember from the last chapter, strings in C are actually arrays of characters. That means if you want to pass a string to a function, you can do it like this:

```
void fortune_cookie(char msg[])
{
  printf("Message reads: %s\n", msg);
}

char quote[] = "Cookies make you fat";
fortune_cookie(quote);
```

The function will be passed a char array.

The `msg` argument is defined like an array, but because you won't know how long the string will be, the `msg` argument doesn't include a length. That *seems* straightforward, but there's something a little strange going on…

Honey, who shrank the string?

C has an operator called **`sizeof`** that can tell you how many bytes of space something takes in memory. You can either call it with a data type or with a piece of data:

On most machines, this will return the value 4.

```
sizeof(int)
sizeof("Turtles!")
```

This will return 9, which is 8 characters plus the \0 end character.

But a strange thing happens if you look at the length of the string you've passed in the function:

```
void fortune_cookie(char msg[])
{
  printf("Message reads: %s\n", msg);
  printf("msg occupies %i bytes\n", sizeof(msg));
}
```

8??? And on some machines, this might even say 4! What gives?

```
File Edit Window Help TakeAByte
> ./fortune_cookie
Message reads: Cookies make you fat
msg occupies 8 bytes
>
```

Instead of displaying the full length of the string, the code returns just 4 or 8 bytes. What's happened? Why does it think the string we passed in is shorter?

Why do you think `sizeof(msg)` is shorter than the length of the whole string? What is `msg`? Why would it return different sizes on different machines?

Array variables are like pointers...

When you create an array, the array variable can be used as a **pointer** to the start of the array in memory. When C sees a line of code in a function like this:

```
char quote[] = "Cookies make you fat";
```

The quote variable will represent the address of the first character in the string.

The computer will set aside space on the stack for each of the characters in the string, plus the `\0` end character. But it will also associate the **address of the first character** with the `quote` variable. Every time the `quote` variable is used in the code, the computer will substitute it with the address of the first character in the string. In fact, the array variable is *just like a pointer*:

You can use "quote" as a pointer variable, even though it's an array.

```
printf("The quote string is stored at: %p\n", quote);
```

If you write a test program to display the address, you will see something like this.

```
File Edit Window Help TakeAByte
> ./where_is_quote
The quote string is stored at: 0x7fff69d4bdd7
>
```

...so our function was passed a pointer

That's why that weird thing happened in the `fortune_cookie()` code. Even though it looked like you were passing a string to the `fortune_cookie()` function, you were actually just passing a pointer to it:

msg is actually a pointer variable.

msg points to the message.

```
void fortune_cookie(char msg[])
{
  printf("Message reads: %s\n", msg);
  printf("msg occupies %i bytes\n", sizeof(msg));
}
```

sizeof(msg) is just the size of a pointer.

And that's why the `sizeof` operator returned a weird result. It was just returning the size of a **pointer to a string**. On 32-bit operating systems, a pointer takes 4 bytes of memory and on 64-bit operating systems, a pointer takes 8 bytes.

What the computer thinks when it runs your code

1 **The computer sees the function.**

```
void fortune_cookie(char msg[])
{
  ...
}
```

Hmmm…looks like they intend to pass an array to this function. That means the function will receive the value of the array variable, which will be an address, so msg will be a pointer to a char.

2 **Then it sees the function contents.**

```
printf("Message reads: %s\n", msg);
printf("msg occupies %i bytes\n", sizeof(msg));
```

I can print the message because I know it starts at location msg. sizeof(msg). That's a pointer variable, so the answer is 8 bytes because that's how much memory it takes for me to store a pointer.

3 **The computer calls the function.**

```
char quote[] = "Cookies make you fat";
fortune_cookie(quote);
```

So quote's an array and I've got to pass the quote variable to fortune_cookie(). I'll set the msg argument to the address where the quote array starts in memory.

BULLET POINTS

- An array variable can be used as a pointer.
- The array variable points to the first element in the array.
- If you declare an array argument to a function, it will be treated as a pointer.
- The `sizeof` operator returns the space taken by a piece of data.
- You can also call `sizeof` for a data type, such as `sizeof(int)`.
- `sizeof(a pointer)` returns 4 on 32-bit operating systems and 8 on 64-bit.

there are no Dumb Questions

Q: Is `sizeof` a function?

A: No, it's an operator.

Q: What's the difference?

A: An operator is compiled to a sequence of instructions by the compiler. But if the code calls a function, it has to jump to a separate piece of code.

Q: So is `sizeof` calculated when the program is compiled?

A: Yes. The compiler can determine the size of the storage at compile time.

Q: Why are pointers different sizes on different machines?

A: On 32-bit operating systems, a memory address is stored as a 32-bit number. That's why it's called a 32-bit system. 32 bits == 4 bytes. That's why a 64-bit system uses 8 bytes to store an address.

Q: If I create a pointer variable, does the pointer variable live in memory?

A: Yes. A pointer variable is just a variable storing a number.

Q: So can I find the address of a pointer variable?

A: Yes—using the `&` operator.

Q: Can I convert a pointer to an ordinary number?

A: On most systems, yes. C compilers typically make the long data type the same size as a memory address. So if `p` is a pointer and you want to store it in a `long` variable `a`, you can type `a = (long)p`. We'll look at this in a later chapter.

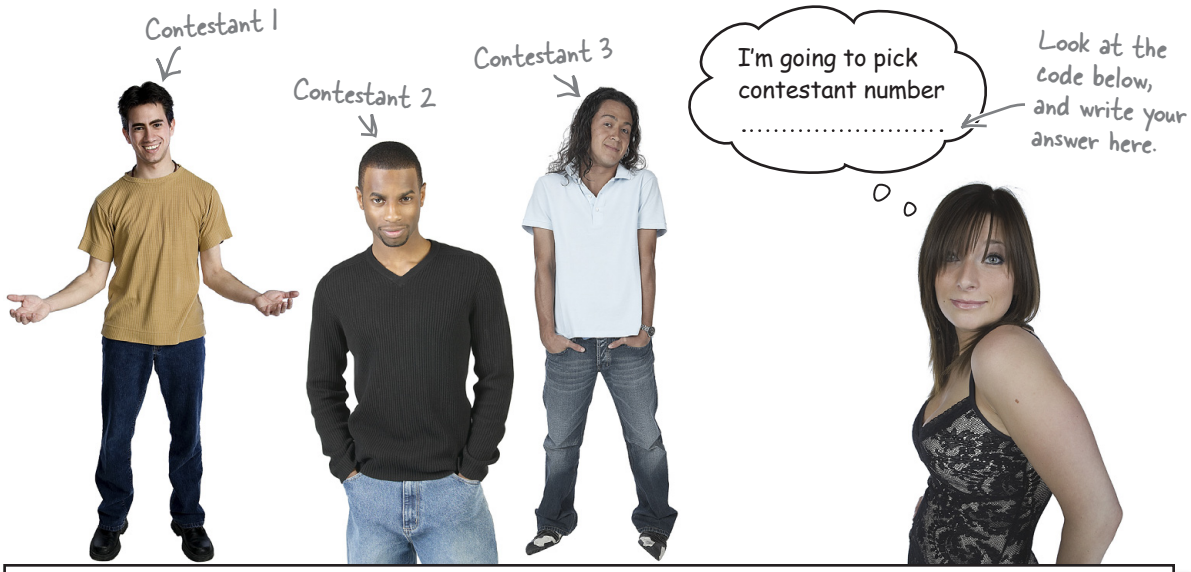
Q: On *most* systems? So it's not guaranteed?

A: It's not guaranteed.

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?

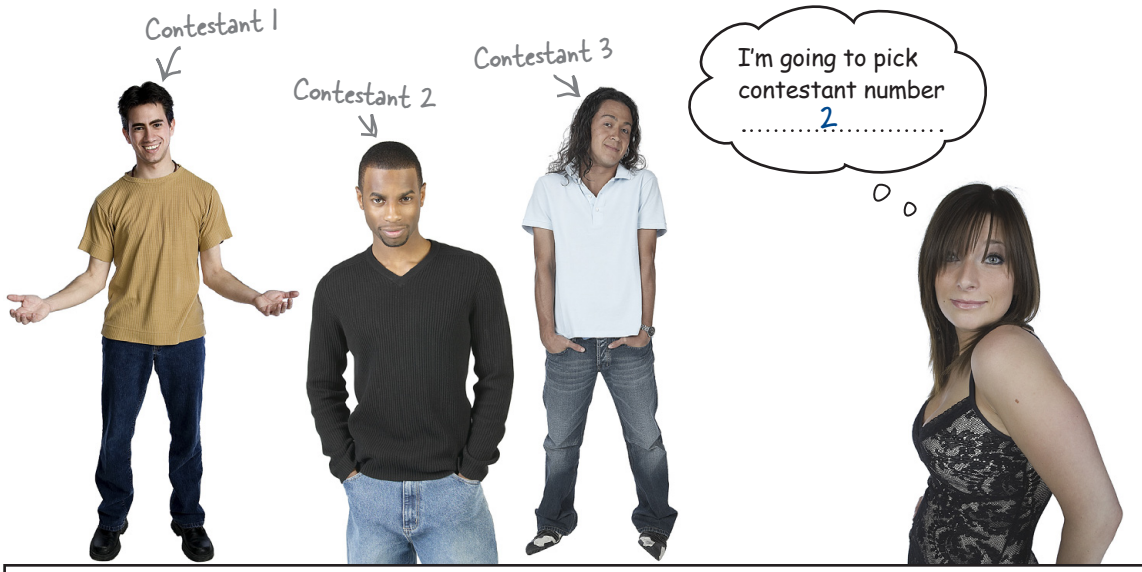



```
#include <stdio.h>

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

THE MATING GAME SOLUTION

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

Tonight's lucky lady picked one of these fine contestants. Who did she choose?




```
#include <stdio.h>

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

"choice" is now the address of the "contestants" array.

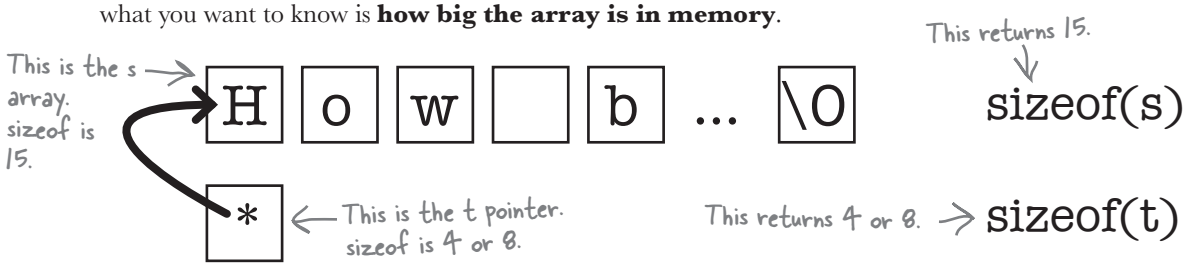
contestants[2]
== *choice
== contestants[0]
== 2

But array variables aren't quite pointers

Even though you can use an array variable as a pointer, there are still a few differences. To see the differences, think about this piece of code.

```
char s[] = "How big is it?";
char *t = s;
```

1 **`sizeof(an array)` is...the size of an array.**
You've seen that `sizeof(a pointer)` returns the value 4 or 8, because that's the size of pointers on 32- and 64-bit systems. But if you call `sizeof` on an array variable, C is smart enough to understand that what you want to know is **how big the array is in memory**.



2 **The address of the array...is the address of the array.**
A pointer variable is just a variable that stores a memory address. But what about an array variable? If you use the `&` operator on an array variable, the result equals the array variable itself.

```
&s == s        &t != t
```

If a coder writes `&s`, that means "What is the address of the `s` array?" The address of the `s` array is just...`s`. But if someone writes `&t`, that means "What is the address of the `t` variable?"

3 **An array variable can't point anywhere else.**
When you create a pointer variable, the machine will allocate 4 or 8 bytes of space to store it. But what if you create an array? The computer will allocate space to store the array, but it won't allocate *any* memory to store the array variable. The compiler simply plugs in the address of the start of the array.

But because array variables don't have allocated storage, it means you can't point them at anything else.

This will give a compile error. → `s = t;`

Pointer decay

Because array variables are slightly different from pointer variables, you need to be careful when you assign arrays to pointers. If you assign an array to a pointer variable, then the pointer variable will only contain the **address** of the array. The pointer doesn't know anything about the size of the array, so a little information has been lost. That loss of information is called **decay**.

Every time you pass an array to a function, you'll decay to a pointer, so it's unavoidable. But you need to keep track of where arrays decay in your code because it can cause very subtle bugs.

The Case of the Lethal List

The mansion had all the things he'd dreamed of: landscaped grounds, chandeliers, its own bathroom. The 94-year-old owner, Amory Mumford III, had been found dead in the garden, apparently of a heart attack. Natural causes? The doc thought it was an overdose of heart medication. Something stank here, and it wasn't just the dead guy in the gazebo. Walking past the cops in the hall, he approached Mumford's newly widowed 27-year-old wife, Bubbles.

"I don't understand. He was always so careful with his medication. Here's the list of doses." She showed him the code from the drug dispenser.

```
int doses[] = {1, 3, 2, 1000};
```

"The police say I reprogrammed the dispenser. But I'm no good with technology. They say I wrote this code, but I don't even think it'll compile. Will it?"

She slipped her manicured fingers into her purse and handed him a copy of the program the police had found lying by the millionaire's bed. It certainly didn't look like it would compile…

```
printf("Issue dose %i", 3[doses]);
```

What did the expression `3[doses]` mean? 3 wasn't an array. Bubbles blew her nose. "I could never write that. And anyway, a dose of 3 is not so bad, is it?"

A dose of size 3 wouldn't have killed the old guy. But maybe there was more to this code than met the eye…

Why arrays really start at 0

An array variable can be used as a pointer to the first element in an array. That means you can read the first element of the array either by using the brackets notation *or* using the * operator like this:

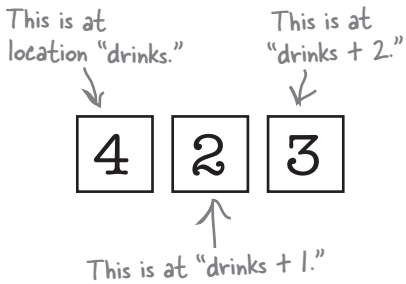
These lines of code are equivalent.

```
int drinks[] = {4, 2, 3};
printf("1st order: %i drinks\n", drinks[0]);
printf("1st order: %i drinks\n", *drinks);
```

drinks[0] == *drinks

But because an address is just a number, that means you can do **pointer arithmetic** and actually ***add*** values to a pointer value and find the next address. So you can either use brackets to read the element with index 2, or you can just add 2 to the address of the first element:

```
printf("3rd order: %i drinks\n", drinks[2]);
printf("3rd order: %i drinks\n", *(drinks + 2));
```



In general, the two expressions `drinks[i]` and `*(drinks + i)` are equivalent. That's why arrays begin with index 0. The index is just the number that's added to the pointer to find the location of the element.

Sharpen your pencil

Use the power of pointer arithmetic to mend a broken heart. This function will skip the first six characters of the text message.

```
void skip(char *msg)
{
  puts(.....);
}

char *msg_from_amy = "Don't call me";
skip(msg_from_amy);
```

What expression do you need here to print from the seventh character?

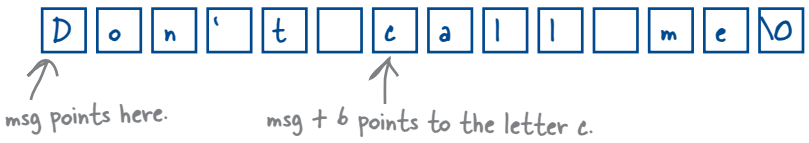
The function needs to print this message from the 'c' character on.

Sharpen your pencil Solution

You were to use the power of pointer arithmetic to mend a broken heart. This function skips the first six characters of the text message.

```
void skip(char *msg)
{
  puts(msg + 6);
}

char *msg_from_amy = "Don't call me";
skip(msg_from_amy);
```

If you add 6 to the msg pointer, you will print from character 7.



The code will display this.

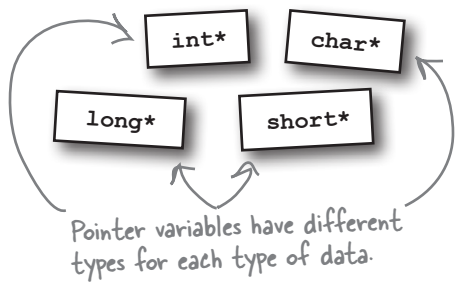
```
File Edit Window Help
> ./skip
call me
>
```

Why pointers have types

If pointers are just addresses, then why do pointer variables have types? Why can't you just store all pointers in some sort of general pointer variable?

The reason is that pointer arithmetic is *sneaky*. If you add **1** to a `char` pointer, the pointer will point to the very next memory address. But that's just because a `char` occupies **1 byte of memory**.

What if you have an `int` pointer? `int`s usually take 4 bytes of space, so if you add 1 to an `int` pointer, the compiled code will actually add 4 to the memory address.




```
int nums[] = {1, 2, 3};
printf("nums is at %p\n", nums);
printf("nums + 1 is at %p\n", nums + 1);
```

If you run this code, the two memory address will be *more* than one byte apart. So pointer types exist so that **the compiler knows how much to adjust the pointer arithmetic**.

(nums + 1) is 4 bytes away from nums.

```
File Edit Window Help
> ./print_nums
nums is at 0x7fff66ccedac
nums + 1 is at 0x7fff66ccedb0
```

Remember, these addresses are printed in hex format.

The Case of the Lethal List

Last time we left our hero interviewing Bubbles Mumford, whose husband had been given an overdose as a result of suspicious code. Was Bubbles the coding culprit or just a patsy? To find out, read on…

He put the code into his pocket. "It's been a pleasure, Mrs. Mumford. I don't think I need to bother you anymore." He shook her by the hand. "Thank you," she said, wiping the tears from her baby blue eyes, "You've been so kind."

"Not so fast, sister." Bubbles barely had time to gasp before he'd slapped the bracelets on her. "I can tell from your hacker manicure that you know more than you say about this crime." No one gets fingertip calluses like hers without logging plenty of time on the keyboard.

"Bubbles, you know a lot more about C than you let on. Take a look at this code again."

```
int doses[] = {1, 3, 2, 1000};
printf("Issue dose %i", 3[doses]);
```

"I knew something was wrong when I saw the expression `3[doses]`. You knew you could use an array variable like `doses` as a pointer. The fatal 1,000 dose could be written down like this…" He scribbled down a few coding options on his second-best Kleenex:

```
doses[3] == *(doses + 3) == *(3 + doses) == 3[doses]
```

"Your code was a dead giveaway, sister. It issued a dose of 1,000 to the old guy. And now you're going where you can never corruptly use C syntax again…"