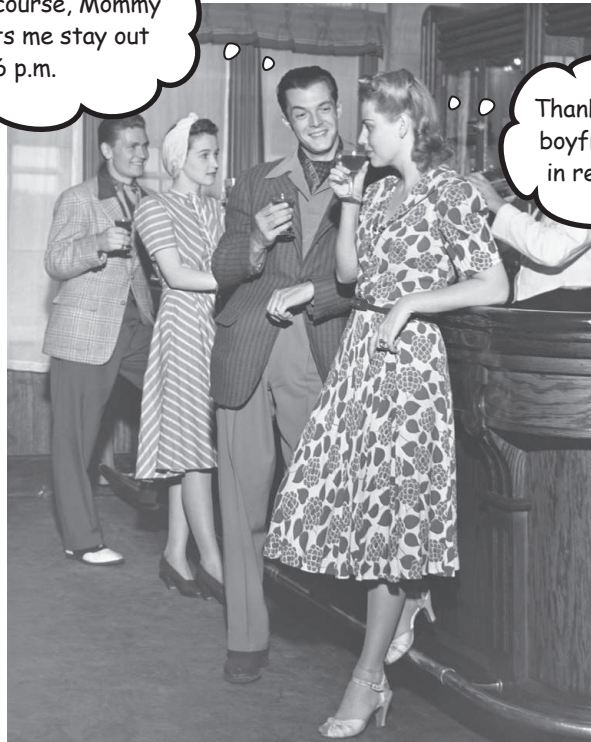# 2 memory and pointers

What are you pointing at?

> ...and of course, Mommy never lets me stay out after 6 p.m.

> Thank heavens my boyfriend variable isn't in read-only memory.

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 <u>pointers</u>

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.**

> I've got the answer you need; it's right here in the Encyclopedia Britannica.

This is a **copy** of the information you need.

> Or you could just look at page 241.

This is a pointer: the location of the information.

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

> You were supposed to sign the birthday card we left in the lunch room.

> But I prefer this one—it's got kittens!

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*.

Relax

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;
```

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

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

> 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:

> &x is the **address** of x.

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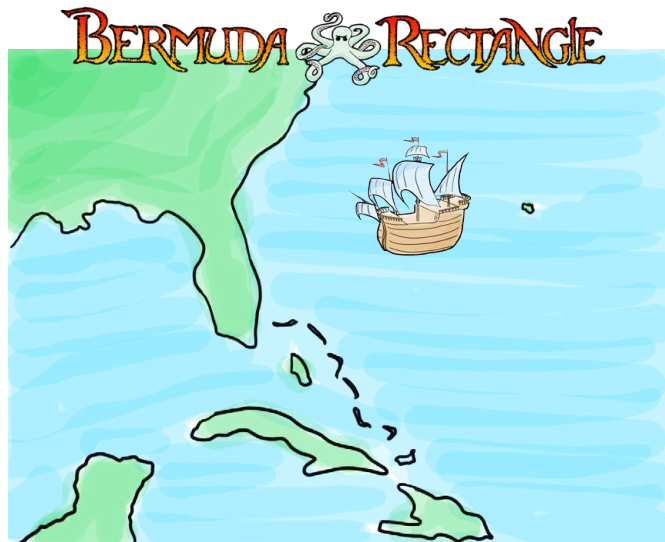

> x lives at location 4,100,000.

> y lives in globals.
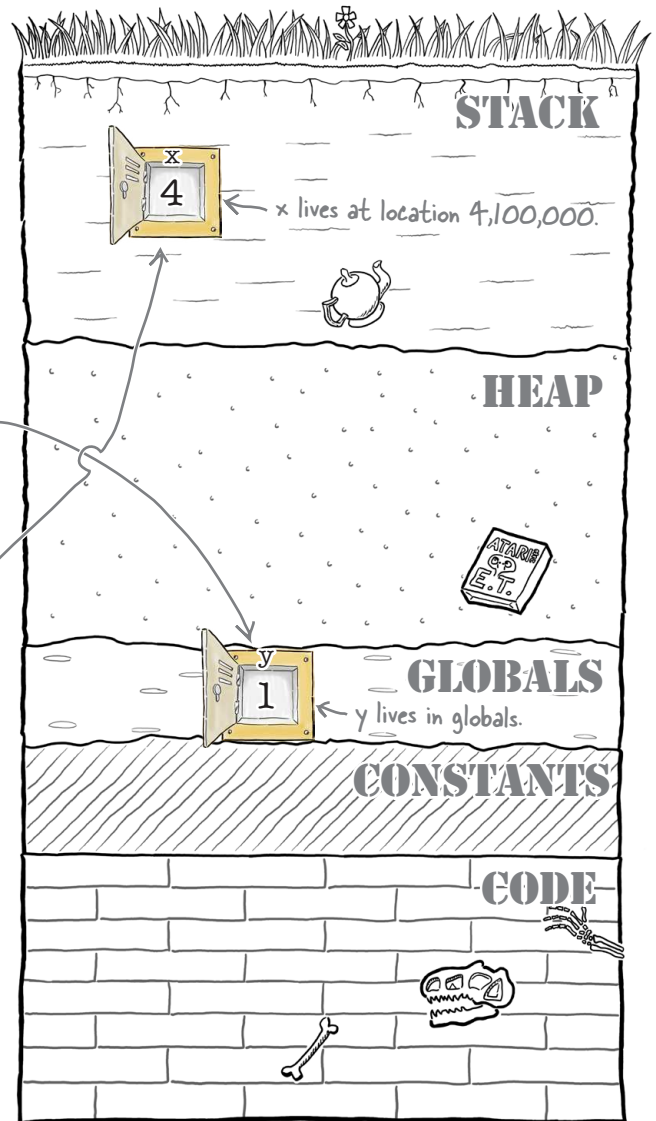
A variable declared inside a function is usually stored in the stack.
A variable declared outside a function is stored in globals.

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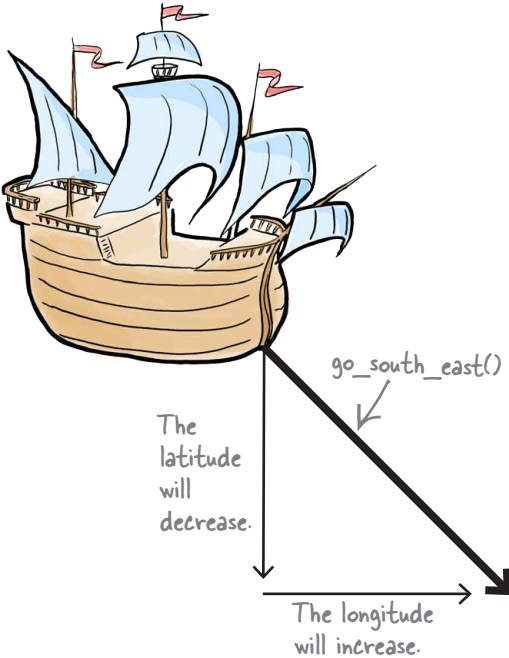
So you could write a go_south_east() function that takes arguments for the latitude and longitude, which it will then increase and decrease:

```
#include <stdio.h>                    Pass in the latitude
                                      and longitude.

void go_south_east(int lat, int lon)
{
   lat = lat - 1;      ← Decrease the
                          latitude.
   lon = lon + 1;
}
                   ↑
         Increase the longitude.

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

go_south_east()

The latitude will decrease.

The longitude will increase.

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…*

BRAIN POWER

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:

> WTF? The ship is still in the same place.

> Where's The Fightin'?

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

We be becalmed, cap'n!

Arr! We be writin' a bad Amazon review!

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.

This is a new variable containing a copy of the latitude value.

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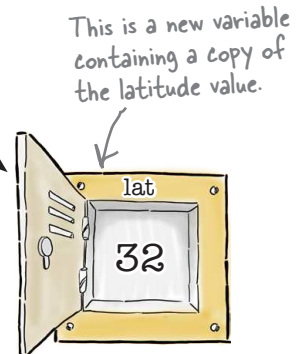
3 When the `go_south_east()` function changes the value of `lat`, the function is just changing its local copy. That means when the computer returns to the `main()` function, the `latitude` variable still has its original value of 32.
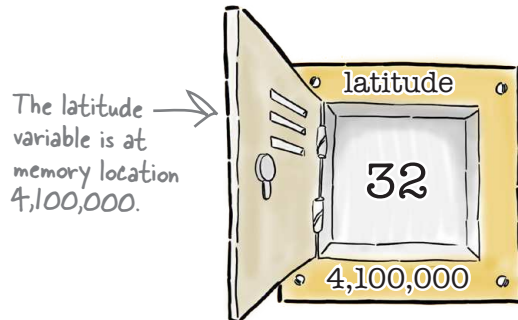
Only the local copy gets changed.

The original variable keeps its original value.
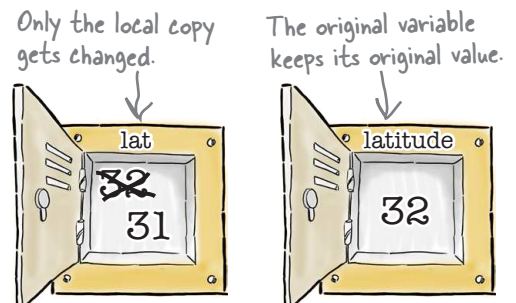
But if that's how C calls functions, how can you ever write a function that updates a variable?
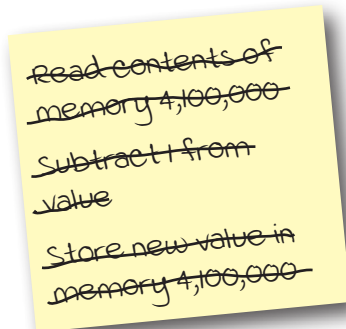
It's easy if you use pointers...

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?

The latitude variable is at memory location 4,100,000.

latitude

32

4,100,000

Please update locker 4,100,000

Instead of passing the value of the variable, pass its location.

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.

~~Read contents of memory 4,100,000~~

~~Subtract 1 from value~~

~~Store new value in memory 4,100,000~~

latitude

3̶2̶ 31

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);
```

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;
```

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;
```

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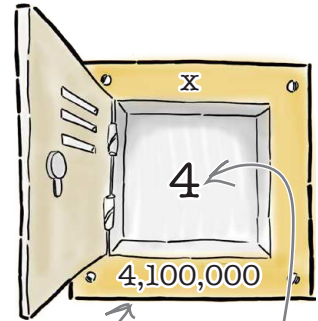
*& will find the
address of
the variable:
4,100,000.*

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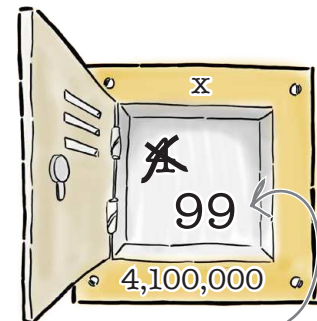
*This will change the contents of
the original x variable to 99.*

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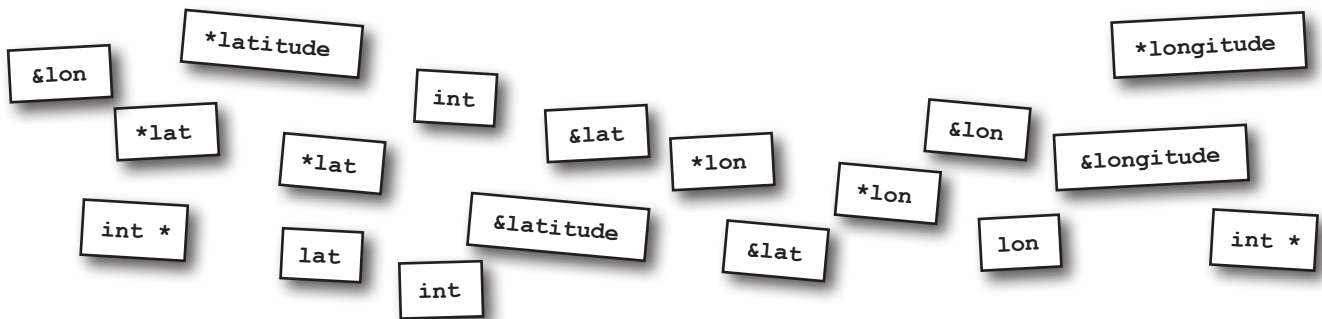
```
#include <stdio.h>
```

What kinds of arguments will store memory addresses for ints?

```
void go_south_east(..................... lat, ..................... lon)
{

   ..................... = ..................... - 1;

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

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

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

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

`&lon`   `*latitude`   `*longitude`

`int`

`*lat`   `&lat`   `&lon`

`*lat`   `*lon`   `&longitude`

`int *`   `&latitude`   `*lon`

`lat`   `&lat`   `lon`   `int *`

`int`

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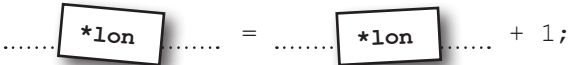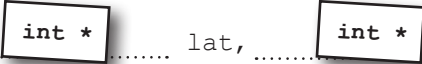
```
#include <stdio.h>
```

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

```
void go_south_east(    int *    lat,    int *    lon)
{

        *lat        =        *lat        - 1;

        *lon        =        *lon        + 1;
}
```

*lat can read the old value and set the new value.
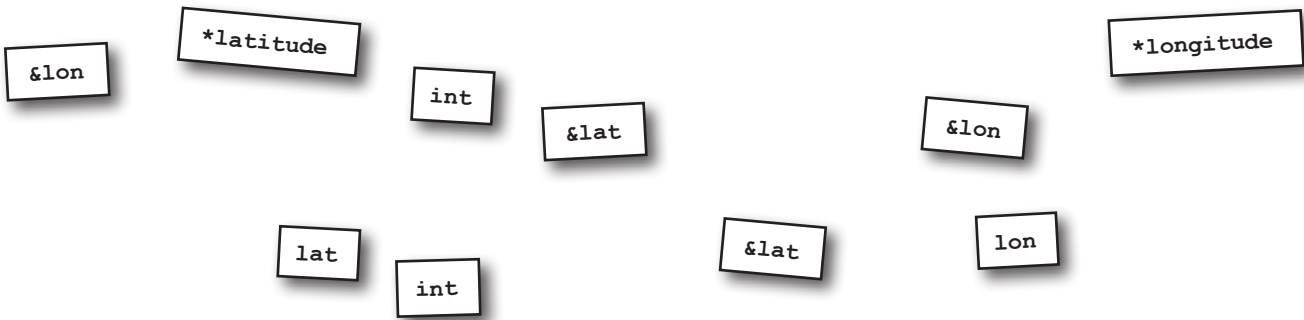
```
int main()
{
    int latitude = 32;
    int longitude = -64;

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

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

`&lon`

`*latitude`

`int`

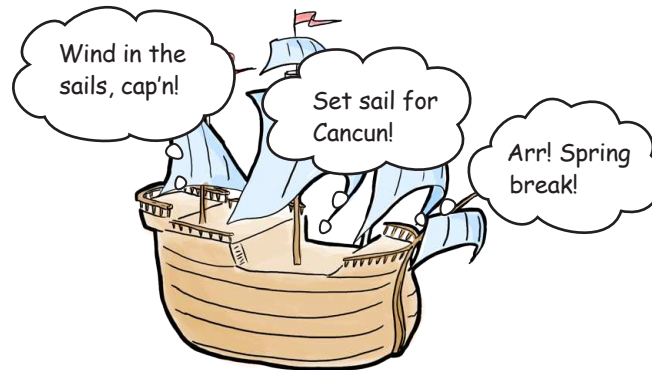`&lat`

`lat`

`int`

`&lat`

`&lon`

`lon`

`*longitude`

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]
>
```

Wind in the sails, cap'n!

Set sail for Cancun!

Arr! Spring break!

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?

> ## BRAIN POWER
>
> 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:

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

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

C o o k i e s ☐ ... \0

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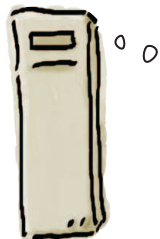
So quote's an array and I've got to pass the quote variable to fortune_cookie(). I'll set the msg argument to the address where the quote array starts in memory.

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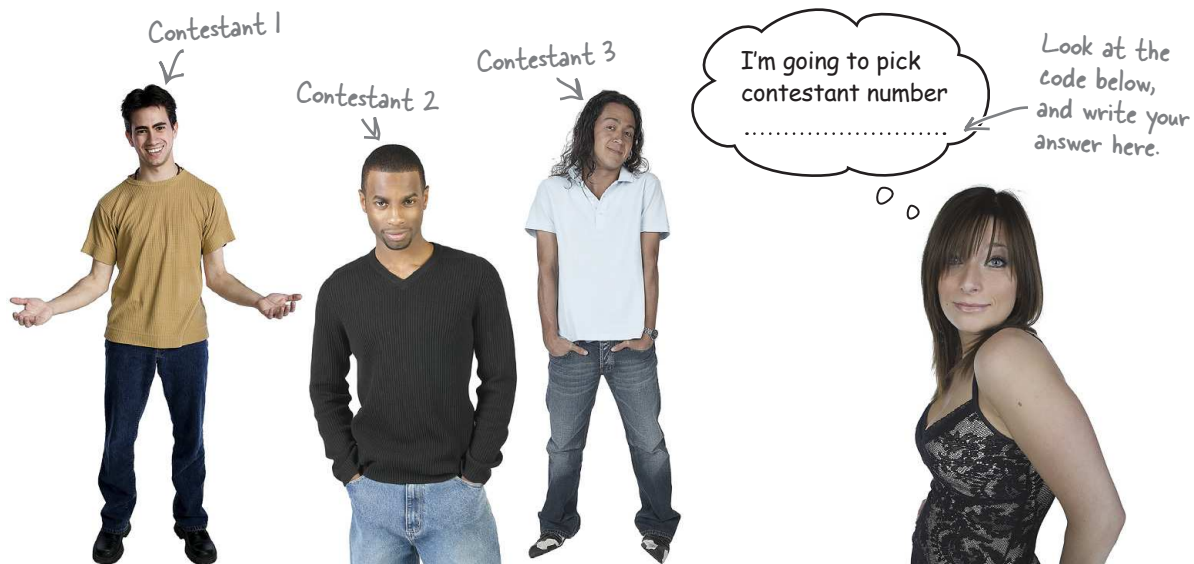
Q: **On *most* systems? So it's not guaranteed?**

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?
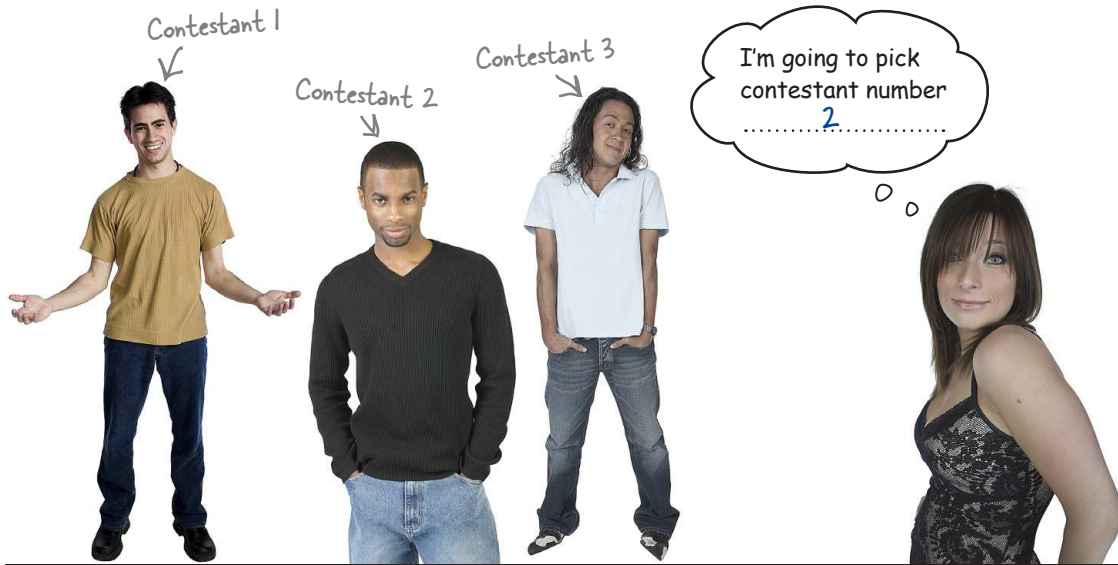
Contestant 1

Contestant 2

Contestant 3

I'm going to pick contestant number
....

Look at the code below, and write your answer here.

```c
#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?

Contestant 1

Contestant 2

Contestant 3

> I'm going to pick contestant number
> ....2...........

```c
#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;
}
```

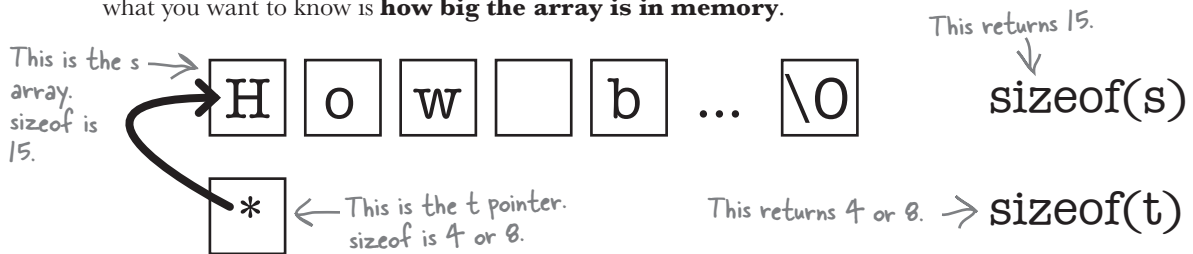"choice" is now the address of the "contestants" array.

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**.

This is the s array. sizeof is 15.

| H | o | w | | b | ... | \0 |

This returns 15.

$$sizeof(s)$$

This is the t pointer. sizeof is 4 or 8.

This returns 4 or 8. → $$sizeof(t)$$

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.

$$\&s == s \qquad \&t \mathrel{!=} t$$

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:

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

These lines of code are equivalent.

*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.

This is at location "drinks."   *This is at "drinks + 2."*

4 2 3

This is at "drinks + 1."
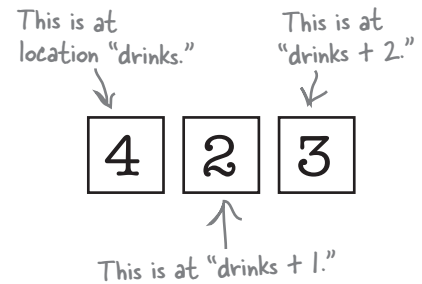
Sharpen your pencil

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.
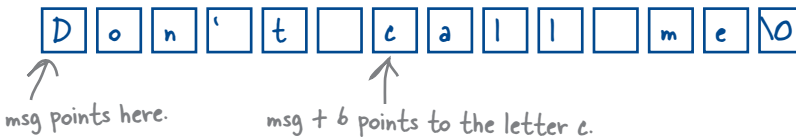
Sharpen your pencil
Solution

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.

| D | o | n | ' | t | | c | a | l | l | | m | e | \0 |

msg points here.

msg + 6 points to the letter c.

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? ints 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* | char* |
| long* | short* |

*Pointer variables have different types for each type of data.*

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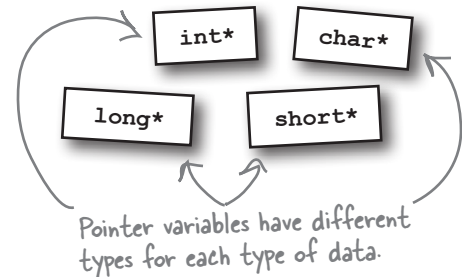
(nums + 1) is 4 bytes away from nums.

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.

Five-Minute Mystery Solved

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

```c
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:

```c
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…"

BULLET POINTS

- Array variables can be used as pointers…

- …but array variables are not quite the same.

- `sizeof` is different for array and pointer variables.

- Array variables can't point to anything else.

- Passing an array variable to a pointer decays it.

- Arrays start at zero because of pointer arithmetic.

- Pointer variables have types so they can adjust pointer arithmetic.

there are no Dumb Questions

Q: Do I really need to understand pointer arithmetic?

A: Some coders avoid using pointer arithmetic because it's easy to get it wrong. But it can be used to process arrays of data efficiently.

Q: Can I subtract numbers from pointers?

A: Yes. But be careful that you don't go back before the start of the allocated space in the array.

Q: When does C adjust the pointer arithmetic calculations?

A: It happens when the compiler is generating the executable. It looks at the type of the variable and then multiplies the pluses and minuses by the size of the underlying variable.

Q: Go on…

A: If the compiler sees that you are working with an `int` array and you are adding 2, the compiler will multiply that by 4 (the length of an `int`) and add 8.

Q: Does C use the `sizeof` operator when it is adjusting pointer arithmetic?

A: Effectively. The `sizeof` operator is also resolved at compile time, and both `sizeof` and the pointer arithmetic operations will use the same sizes for different data types.

Q: Can I multiply pointers?

A: No.

Using pointers for data entry

You already know how to get the user to enter a string from the keyboard. You can do it with the scanf() function:

You're going to store a name in this array. →
```
char name[40];
    printf("Enter your name: ");
    scanf("%39s", name);
```
← *scanf will read up to 39 characters plus the string terminator \0.*

How does scanf() work? It accepts a char pointer, and in this case you're passing it an array variable. By now, you might have an idea **why** it takes a pointer. It's because the scanf() function is going to *update* the contents of the array. Functions that need to update a variable don't want the value of the variable itself—they want its **address**.

Entering numbers with scanf()

So how do you enter data into a **numeric field**? You do it by passing a *pointer* to a number variable.

```
int age;
    printf("Enter your age: ");
    scanf("%i", &age);
```
%i means the user will enter an int value. ↘   ← *Use the & operator to get the address of the int.*

Because you pass the address of a number variable into the function, scanf() can update the contents of the variable. And to help you out, you can pass a format string that contains the same kind of format codes that you pass to the printf() function. You can even use scanf() to enter more than one piece of information at a time:

`%i` ← *Enter an integer.*

`%29s` ← *Enter up to 29 characters (+ '\0').*

`%f` ← *Enter a floating-point number.*

```
char first_name[20];
char last_name[20];
printf("Enter first and last name: ");
scanf("%19s %19s", first_name, last_name);
printf("First: %s Last:%s\n", first_name, last_name);
```
This reads a first name, then a space, then the second name. ↘
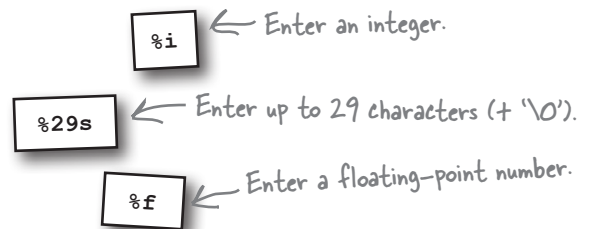
The first and last names are stored in separate arrays.

```
File Edit Window Help Meerkats
> ./name_test
Enter first and last name: Sanders Kleinfeld
First: Sanders Last: Kleinfeld
>
```

Be careful with scanf()

There's a little…problem with the scanf() function. So
far, all of the code you've written has very carefully put a limit
on the number of characters that scanf() will read into a
function:

```
scanf("%39s", name);


scanf("%2s", card_name);
```

SECURITY ALERT!
SECURITY ALERT!
SECURITY ALERT!!

Why is that? After all, scanf() uses the same kind of format
strings as printf(), but when we print a string with printf(),
you just use %s. Well, if you just use %s in scanf(), there can
be a problem if someone gets a little type-happy:

```
char food[5];
printf("Enter favorite food: ");
scanf("%s", food);
printf("Favorite food: %s\n", food);
```
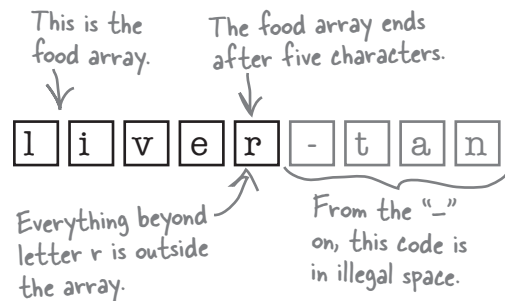
```
File  Edit  Window  Help  TakeAByte
> ./food
Enter favorite food: liver-tangerine-raccoon-toffee
Favorite food: liver-tangerine-raccoon-toffee
Segmentation fault: 11
>
```

The program crashes. The reason is because scanf() writes
data way beyond the end of the space allocated to the food array.

scanf() can cause buffer overflows

If you forget to limit the length of the string that you read with
scanf(), then any user can enter far more data than the
program has space to store. The extra data then gets written into
memory that has not been properly allocated by the computer.
Now, you might get lucky and the data will simply be stored and
not cause any problems.

But it's *very* likely that buffer overflows will cause bugs. It might
be called a *segmentation fault* or an *abort trap*, but whatever the
error message that appears, the result will be a ***crash***.

This is the
food array.

The food array ends
after five characters.

| l | i | v | e | r | - | t | a | n |

Everything beyond
letter r is outside
the array.

From the "–"
on, this code is
in illegal space.

fgets() is an alternative to scanf()

There's another function you can use to enter text data: **fgets()**. Just like the scanf() function, it takes a char pointer, but *unlike* the scanf() function, the fgets() function must be given a maximum length:

This is the same program as before.

```
char food[5];

printf("Enter favorite food: ");

fgets(food, sizeof(food), stdin);
```

First, it takes a pointer to a buffer.

Next, it takes a maximum size of the string ('\0' included).

stdin just means the data will be coming from the keyboard.

You'll find out more about stdin later.

That means that you can't accidentally forget to set a length when you call fgets(); it's right there in the function signature as a mandatory argument. Also, notice that the fgets() buffer size **includes** the final \0 character. So you don't need to subtract 1 from the length as you do with scanf().

OK, what else do you need to know about fgets()?

Using sizeof with fgets()

The code above sets the maximum length using the sizeof operator. Be careful with this. Remember: sizeof returns the amount of space occupied by a variable. In the code above, food is an array variable, so sizeof returns the size of the array. If food was just a simple pointer variable, the sizeof operator would have just returned the size of a pointer.

If you know that you are passing an array variable to fgets() function, then using sizeof is fine. If you're just passing a simple pointer, you should just enter the size you want:

If food was a simple pointer, you'd give an explicit length, rather than using sizeof.

```
printf("Enter favorite food: ");
fgets(food, 5, stdin);
```

Tales from the Crypt

The fgets() function actually comes from an older function called gets().

Even though fgets() *is seen as a safer-to-use function than* scanf(), *the truth is that the older* gets() *function is far more dangerous than either of them. The reason? The* gets() *function has* **no limits at all**:

```
char dangerous[10];
gets(dangerous);
```

Nooooooo!!!!! Seriously, don't use this.

gets() *is a function that's been around for a long time. But all you really need to know is that you* **really shouldn't use it**.

Title Fight

Roll up! Roll up! It's time for the title fight we've all been waiting for. In the red corner: nimble light, flexible but oh-so-slightly dangerous. It's the bad boy of data input: `scanf()`. And in the blue corner, he's simple, he's safe, he's the function you'd want to introduce to your mom: it's `fgets()`!

	scanf():	**fgets():**
Round 1: Limits		
Do you limit the number of characters that a user can enter?	`scanf()` can limit the data entered, so long as you remember to add the size to the format string.	`fgets()` has a mandatory limit. Nothing gets past him.

Result: fgets() takes this round on points.

Round 2: Multiple fields		
Can you be used to enter more than one field?	Yes! `scanf()` will not only allow you to enter more than one field, but it also allows you to enter **structured data** including the ability to specify what characters appear between fields.	Ouch! `fgets()` takes this one on the chin. `fgets()` allows you to enter just one string into a buffer. No other data types. Just strings. Just one buffer.

Result: scanf() clearly wins this round.

Round 3: Spaces in strings		
If someone enters a string, can it contain spaces?	Oof! `scanf()` gets hit badly by this one. When `scanf()` reads a string with the `%s`, it stops as soon as it hits a space. So if you want to enter more than one word, you either have to call it more than once, or use some fancy regular expression trick.	No problem with spaces at all. `fgets()` can read the whole string every time.

**Result:** A fightback! Round to fgets().

A good clean fight between these two feisty functions. Clearly, if you need to enter **structured data** with *several* fields, you'll want to use `scanf()`. If you're entering a **single unstructured string**, then `fgets()` is probably the way to go.

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:

```c
#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;
}
```
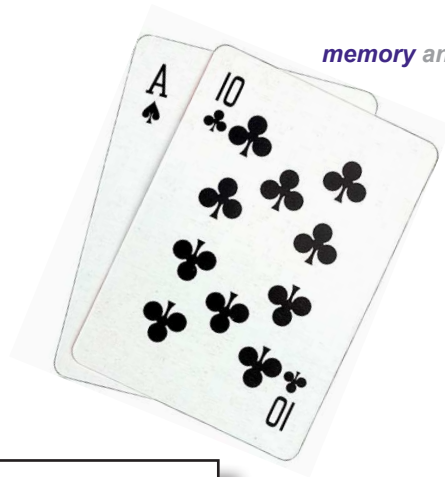
Find the Queen.

The code is designed to shuffle the letters in the three-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, and then the code is compiled and run.

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 monte.c -o monte && ./monte
bus error
```

Darn it. I knew that card shark couldn't be trusted...
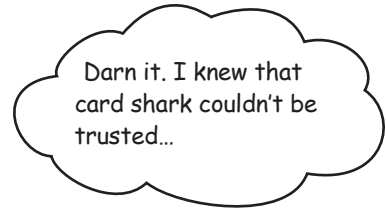
What's more, if the guys try the same code on different machines and operating systems, they get a whole bunch of different errors:

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

Whack!

SegPhault!

Kapow!

Bus Error!

Segmentation Error!

What's wrong with the code?

✱ WHAT'S YOUR HUNCH?

It's time to use your **intuition**. Don't overanalyze. 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 was time to use your **intuition**. You were to read through these possible answers and select *only* the one you think is correct.

What did **you** think the problem was?

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

String literals can never be updated

A variable that points to a string literal can't be used to change the contents of the string:
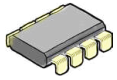
```
char *cards = "JQK";
```
← This variable can't modify this string.

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

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

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

In memory: char *cards="JQK";

To understand 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.

Highest address

1 **The computer loads the string literal.**
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 string literal "JQK**." String literals are usually stored in read-only memory to prevent anyone from changing them.

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.

STACK

2 `cards`

HEAP

3

GLOBALS

1

CONSTANTS

| J | Q | K | \0 |

CODE

Read-only memory

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

```
char *cards="JQK";
...
4 cards[2] = cards[1];
```

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

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

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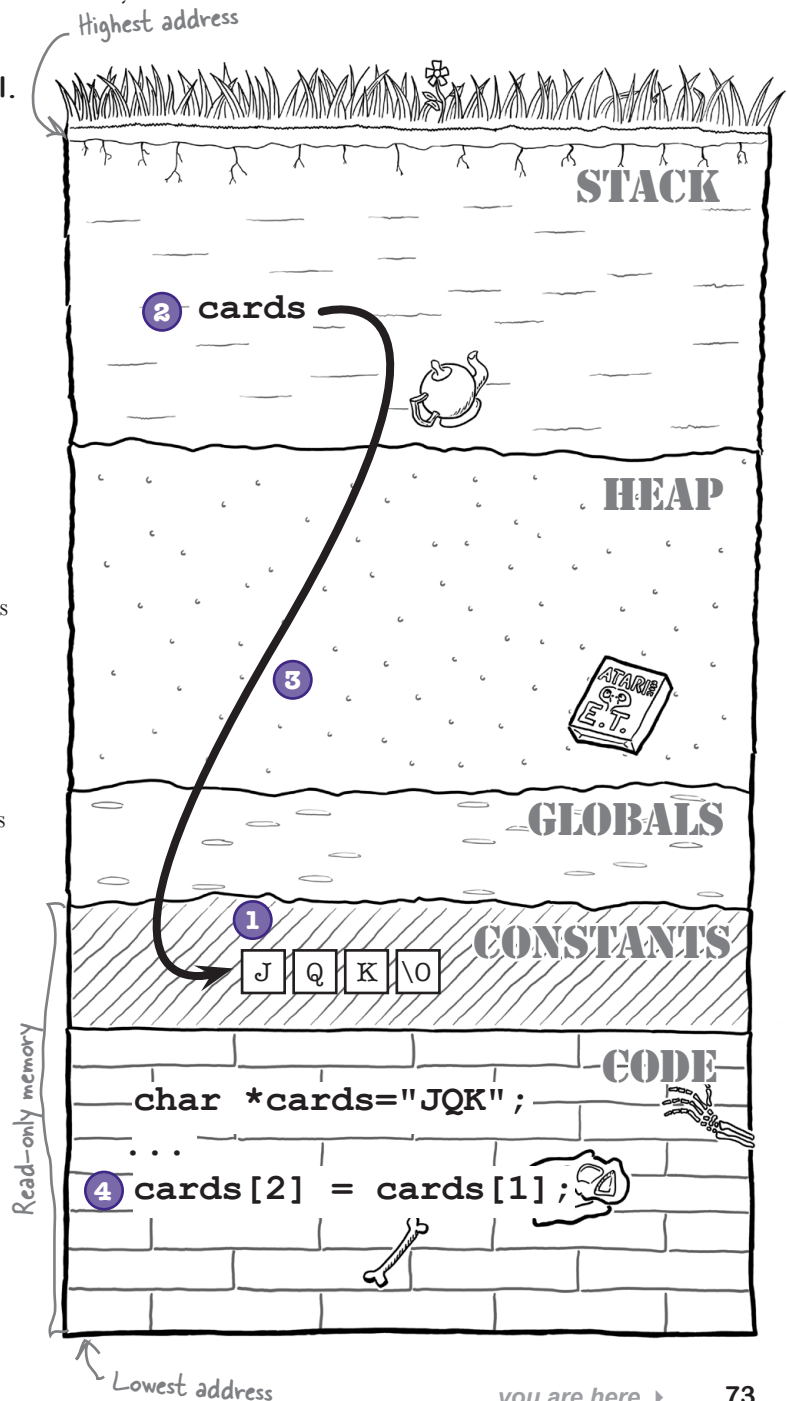
If you're going to change a string, make a copy

The truth is that if you want to change the contents of a string, you'll need to work on a ***copy***. If you create a copy of the string in an area of memory that's *not* read-only, there won't be a problem if you try to change the letters it contains.

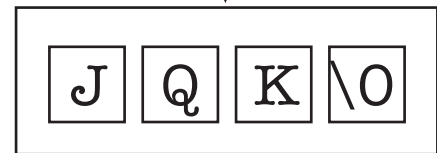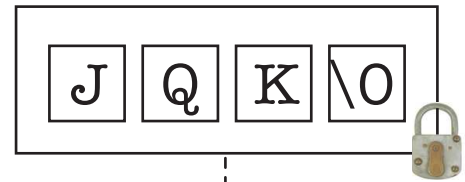But how do you make a copy? Well, just create the string as a *new array*.

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

cards is not just a pointer. cards is now an **array**.

This string is in read-only memory...

It's probably not too clear why this changes anything. *All* strings are arrays. But in the old code, cards was just a *pointer*. In the new code, it's an ***array***. If you declare an array called cards and then set it to a string literal, the cards array will be a completely new copy. The variable isn't just *pointing* at the string literal. It's a brand-new array that contains a fresh ***copy*** of the string literal.

To see how this works in practice, you'll need to look at what happens in memory.

...so make a copy of the string in a section of memory that **can** be amended.

Geek Bits

cards[] or cards*?

If you see a declaration like this, what does it *really* mean?

```
char cards[]
```

Well, it **depends on where you see it**. If it's a normal variable declaration, then it means that cards is an array, and you have to set it to a value immediately:

```
int my_function()
{
    char cards[] = "JQK";
    ...
}
```

cards is an array.

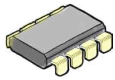There's no array size given, so you have to set it to something immediately.

But if cards is being declared as a *function argument*, it means that cards is a **pointer**:

```
void stack_deck(char cards[])
{
    ...
}
```

cards is a char pointer.

```
void stack_deck(char *cards)
{
    ...
}
```
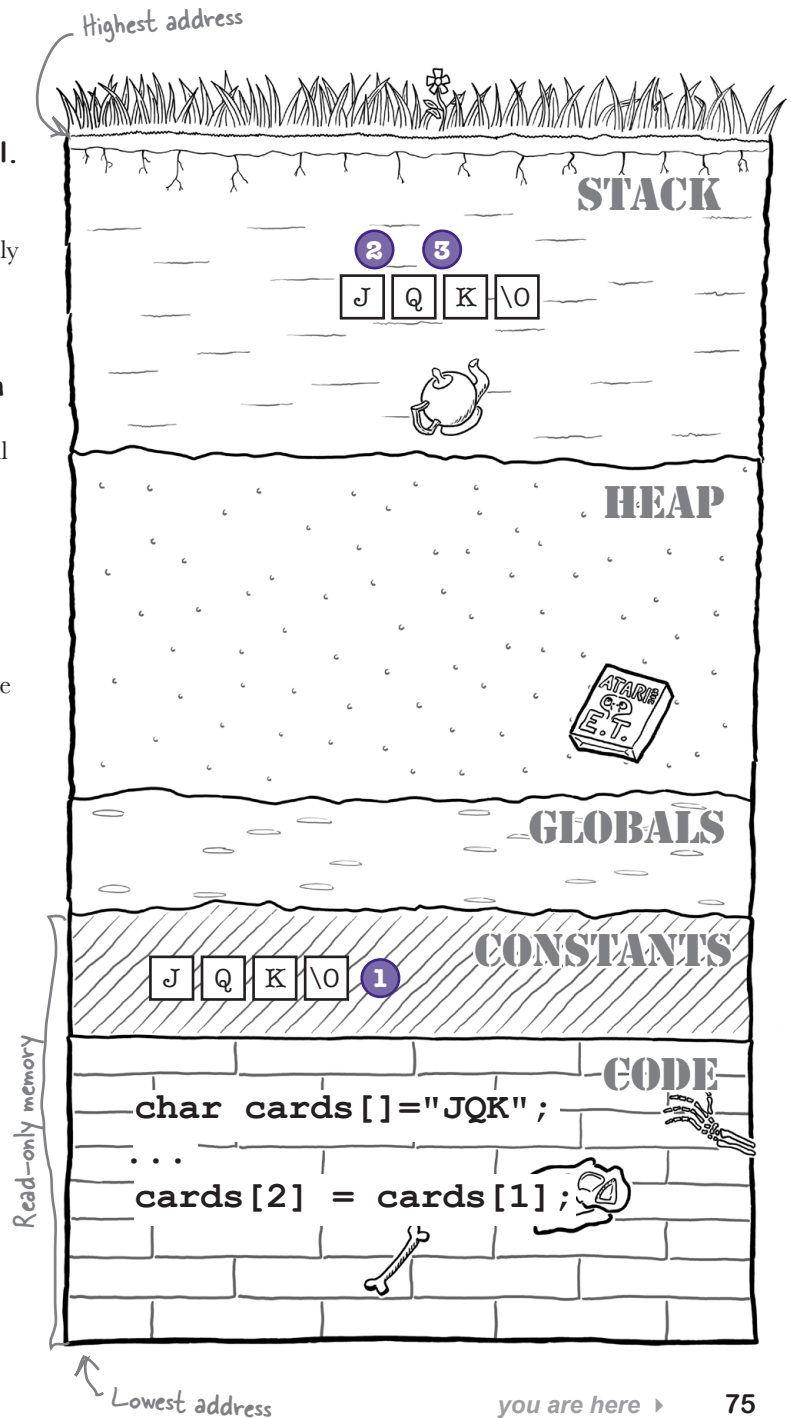
These two functions are equivalent.

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.

Highest address

1 The computer loads the string literal.
As before, when the computer loads the program into memory, it stores the constant values—like the string "JQK"—into read-only memory.

STACK

② **③**

| J | Q | K | \0 |

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—four characters' worth.

HEAP

3 The program initializes the array.
But as well as allocating the space, the program will also **copy the contents** of the string literal "JQK" into the stack memory.

So the difference is that the original code used a pointer to point to a read-only string literal. But if you initialize an array with a string literal, you then have a *copy* of the letters, and you can change them as much as you like.

GLOBALS

CONSTANTS

| J | Q | K | \0 | **①** |

Read-only memory

CODE

```
char cards[]="JQK";
...
cards[2] = cards[1];
```

Lowest address

TEST DRIVE

See what happens if you construct a **new array** in the code.

```
#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! Your `cards` variable now points to a string in an unprotected section of memory, so we are free to modify its 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 string literal 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 string literal. 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 tries to modify the string, it will give you a compile error:

```
s[0] = 'S';

monte.c:7: error: assignment of read-only location
```

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 Raider had met Jimmy in the morning for coffee and they'd both been alive:

```c
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 hadn't reckoned on the crowd of G-Men enjoying their weekly three-card monte session in the back room of the Head First Lounge. You get the picture. A rattle of gunfire, a scream, and moments later the villain was lying on the sidewalk, creating a public health hazard:

```c
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 to check in with her boyfriend at the coffee shop, she's told he's served his last orange mocha frappuccino:

```c
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?

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:

```c
#include <stdio.h>
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);
  return 0;
}
```

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 sip from a <u>Head First Brain Booster Fruit Beverage</u>. 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 oncoming 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 breath 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:

masked_raider → A l i v e \0
jimmy →
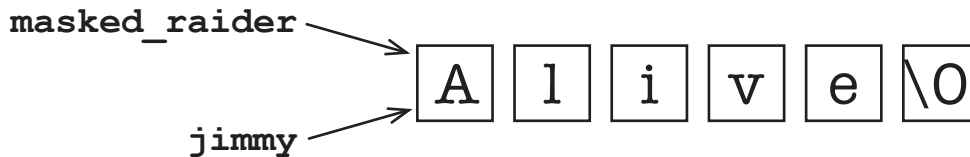
"`jimmy` and `masked_raider` are just aliases for the same memory address. They're pointing to the same place. When the `masked_raider` stopped the bullet, so did Jimmy. Add to that this invoice from the San Francisco elephant sanctuary and this order for 15 tons of packing material, and it's an open and shut case."

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 declare a `char` pointer as `const char *` to prevent the code from using it to modify a string.

---

there are no Dumb Questions

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

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

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" string literal.

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 string literal without complaining. But it is *always* wrong to do that.

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

A: String literals are read-only anyway. The `const` modifier means that 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.

Q: I still don't understand why an array variable isn't stored in memory. If it exists, surely it lives somewhere?

A: When the program is compiled, all the references to array variables are replaced with the addresses of the array. So the truth is that the array variable won't exist in the final executable. That's OK because the array variable will never be needed to point anywhere else.

Q: If I set a new array to a string literal, will the program really copy the contents each time?

A: It's down to the compiler. The final machine code will either copy the bytes of the string literal to the array, or else the program will simply set the values of each character every time it reaches the declaration.

Q: You keep saying "declaration." What does that mean?

A: A *declaration* is a piece of code that declares that something (a variable, a function) exists. A definition is a piece of code that says what something is. If you declare a variable and set it to a value (e.g., `int x = 4;`), then the code is both a declaration and a definition.

Q: Why is `scanf()` called `scanf()`?

A: `scanf()` means "scan formatted" because it's used to scan formatted input.
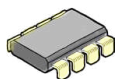
Memory memorizer

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 downward**.

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. You'll see later in the book how you'll use the heap.
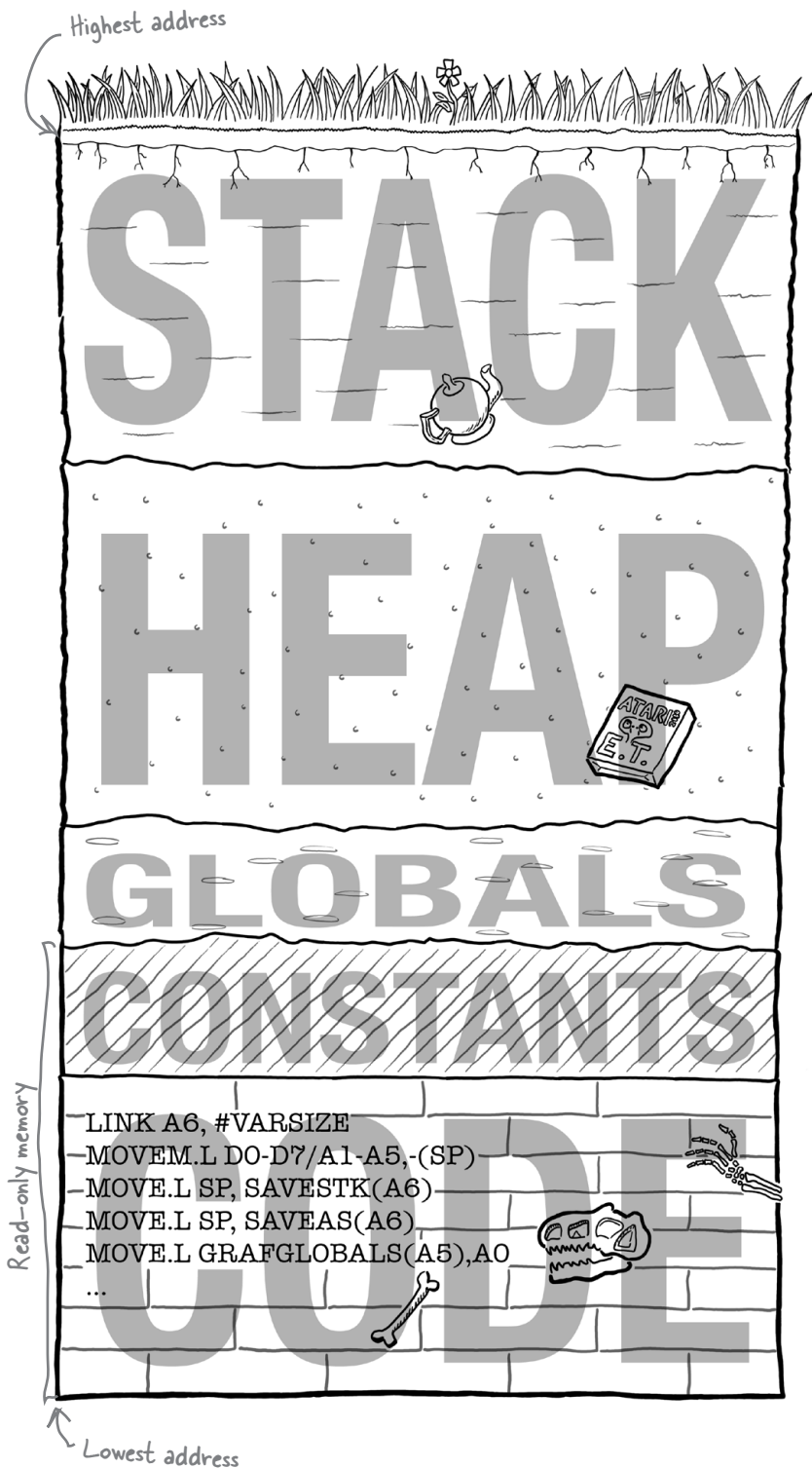
Globals

A global variable is a variable that lives outside 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 *string literals* 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.

Highest address

STACK

HEAP

GLOBALS

CONSTANTS

Read-only memory

```
LINK A6, #VARSIZE
MOVEM.L D0-D7/A1-A5,-(SP)
MOVE.L SP, SAVESTK(A6)
MOVE.L SP, SAVEAS(A6)
MOVE.L GRAFGLOBALS(A5),A0
…
```

CODE

Lowest address

Your C Toolbox

You've got Chapter 2 under your belt, and now you've added pointers and memory to your toolbox. For a complete list of tooltips in the book, see Appendix ii.

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.

A char pointer variable x is declared as char *x.

Local variables are stored on the stack.

String literals are stored in read-only memory.

Initialize a new array with a string, and it will copy it.

Array variables can be used as pointers.

Read the contents of an address a with *a.

fgets(buf, size, stdin) is a simpler way to enter text.