C is a language for small, fast programs

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

The way C works

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

1

Source

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

2

Compile

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

3

Output

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

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

There are three C standards that you may stumble across. ANSI C is from the late 1980s and is used for the oldest code. A lot of things were fixed up in the C99 standard from 1999. And some cool new language features were added in the current standard, C11, released in 2011. The differences between the different versions aren't huge, and we'll point them out along the way.

Sharpen your pencil

Try to guess what each of these code fragments does.

Describe what you think the code does.

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

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

```
/* Assume name shorter than 20 chars. */
char ex[20];
puts("Enter boyfriend's name: ");
scanf("%19s", ex);
printf("Dear %s.\n\n\tYou're history.\n", ex);
```

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

Sharpen your pencil Solution

Don't worry if you don't understand all of this yet. Everything is explained in greater detail later in the book.

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

An integer is a whole number.

This displays a string on the command prompt or terminal.

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

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

The braces define a block statement.

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

```
/* Assume name shorter than 20 chars. */
char ex[20];
puts("Enter boyfriend's name: ");
scanf("%19s", ex);
printf("Dear %s.\n\n\tYou're history.\n", ex);
```

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

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

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

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

A switch statement checks a single variable for different values.

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

But what does a complete C program look like?

To create a full program, you need to enter your code into a *C source file*. C source files can be created by any text editor, and their filenames usually end with *.c*.

This is just a convention, but you should follow it.

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

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

The comment starts with /*.

These *s are optional. They're only there to make it look pretty.

The comment ends with */.

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

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

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

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

3 The last thing you find in a source file are the functions.
All C code runs inside functions. The most important function you will find in any C program is called the **`main()` function**. The `main()` function is the starting point for all of the code in your program.

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

The main() Function Up Close

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

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

This is the return type. It should always be int for the main() function.

Because the function is called "main," the program will start here.

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

The body of the function is always surrounded by braces.

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

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

Geek Bits

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

The first parameter will be inserted here as a string.

First parameter

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

The second parameter will be inserted here as an integer.

Second parameter

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

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

```
echo %ErrorLevel%
```

in Windows, or:

```
echo $?
```

in Linux or on the Mac.

Code Magnets

The College Blackjack Team was working on some code on the dorm fridge, but someone mixed up the magnets! Can you reassemble the code from the magnets?

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

.....

.....

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

      .....

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

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

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

   .....0;
}
```

Enter two characters for the card name.

This converts the text into a number.

<stdlib.h>

;

;

val = 11

int

'J'

#include

'A'

return

else

#include

if

val = 10

<stdio.h>

Code Magnets Solution

The College Blackjack Team was working on some code on the dorm fridge, but someone mixed up the magnets! You were to reassemble the code from the magnets.

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

there are no Dumb Questions

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

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

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

A: If your compiler supports the C99 standard, then you can begin a comment with `//`. The compiler treats the rest of that line as a comment.

Q: How do I know which standard my compiler supports?

A: Check the documentation for your compiler. `gcc` supports all three standards: ANSI C, C99, and C11.

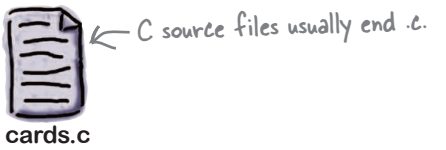
But how do you run the program?

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

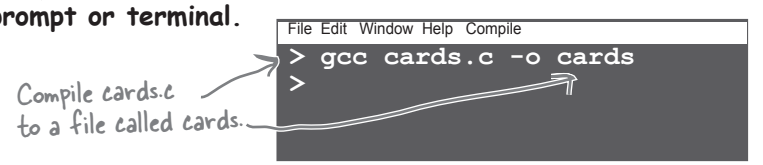
To compile the code, you need a program called a **compiler**. One of the most popular C compilers is the *GNU Compiler Collection* or **gcc**. `gcc` is available on a lot of operating systems, and it can compile lots of languages other than C. Best of all, it's completely free.

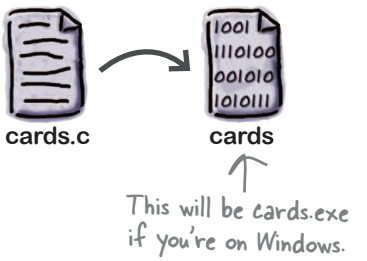
Here's how you can compile and run the program using `gcc`.

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

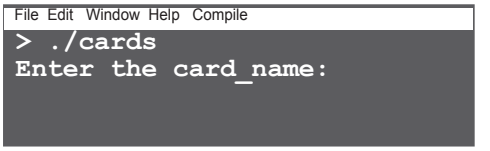


2. **Compile with `gcc cards.c -o cards` at a command prompt or terminal.**





3. **Run by typing `cards` on Windows, or `./cards` on Mac, Linux, and Cygwin.**



Geek Bits

You can compile and run your code on most machines using this trick:

`&&` here means "and then if it's successful, do this..."

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

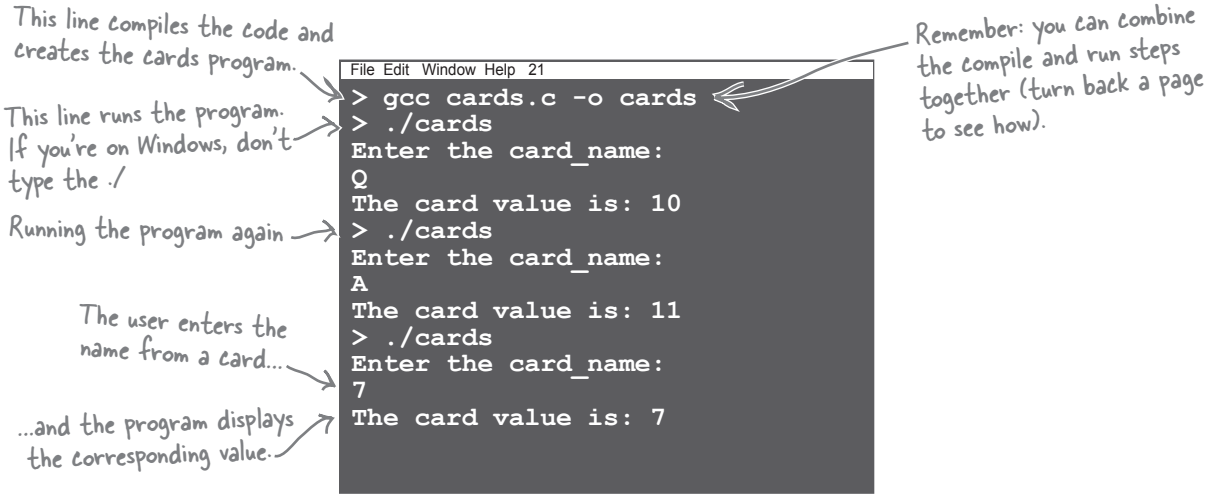
You should put "zork" instead of "./zork" on a Windows machine.

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

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

Test Drive

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



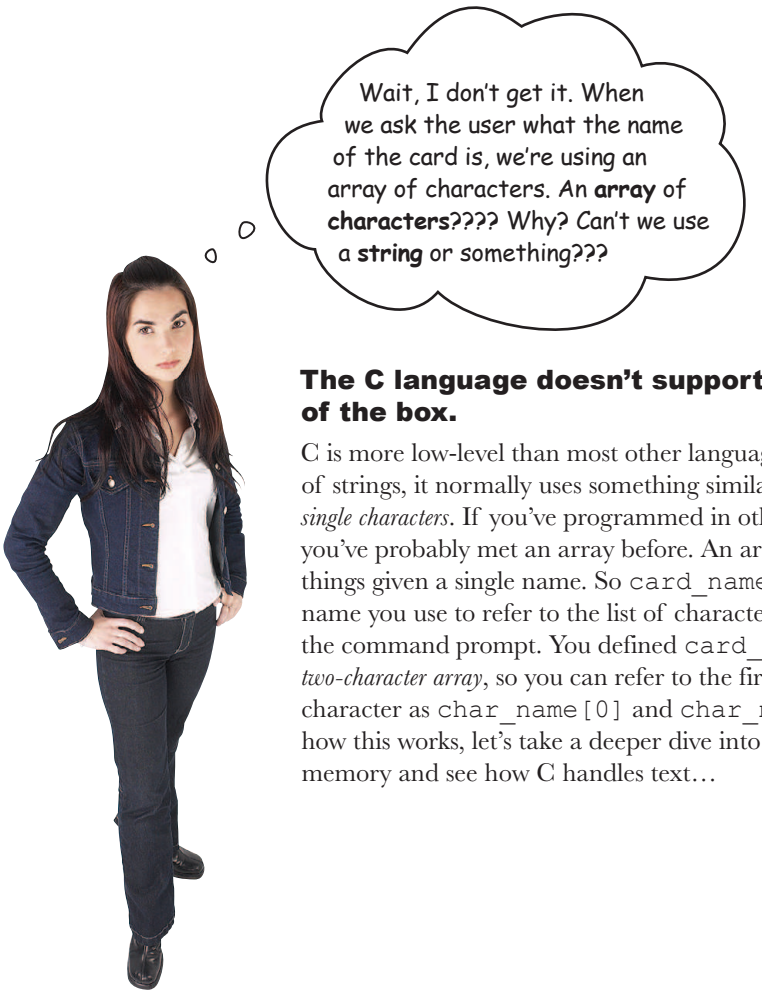
The program works!

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

there are no Dumb Questions

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

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



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

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

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

Strings Way Up Close

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

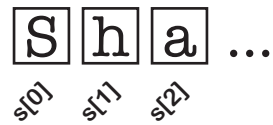
```
s = "Shatner"
```

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

This is how you define an array in C.

```
s = {'S', 'h', 'a', 't', 'n', 'e', 'r'}
```

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



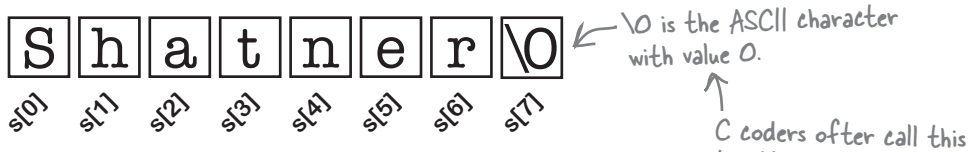
Don't fall off the end of the string

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

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

```
s = "Shatner"
```

it actually stores it in memory like this:



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

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

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

there are no Dumb Questions

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

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

Q: Why?

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

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

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

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

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

Q: Does it matter if I use single quotes or double quotes?

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

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

A: Usually you will define strings using quotes. They are called **string literals,** and they are easier to type.

Q: Are there any differences between string literals and character arrays?

A: Only one: string literals are constant.

Q: What does that mean?

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

Q: What will happen if I try?

A: It depends on the compiler, but `gcc` will usually display a bus error.

Q: A bus error? What the heck's a bus error?

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

Painless Operations

Not all equals signs are equal.

In C, the equals sign (=) is used for **assignment**. But a double equals sign (==) is used for **testing equality**.

Set teeth to the value 4.

```
teeth = 4;
```

```
teeth == 4;
```

Test if teeth has the value 4.

If you want to increase or decrease a variable, then you can save space with the += and -= assignments.

Adds 2 to teeth.

```
teeth += 2;
```

```
teeth -= 2;
```

Takes away 2 teeth.

Finally, if you want to increase or decrease a variable by 1, use ++ and --.

```
teeth++;
```

Increase by 1.

```
teeth--;
```

Decrease by 1.

Two types of command

So far, every command you've seen has fallen into one of the following two categories.

Do something

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

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

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

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

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

Do something only if something is true

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

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

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

```
if (dealer_card == 6) {
    double_down();
    hit();
}
```
BOTH of these commands will run if the condition is true. The commands are grouped inside a single block statement.

Do you need braces?

Block statements allow you to treat a *whole set of statements* as if they were a *single statement*. In C, the `if` condition works like this:

```
if (countdown == 0)
  do_this_thing();
```

The `if` condition runs a **single statement**. So what if you want to run several statements in an `if`? If you wrap a list of statements in braces, C will treat them as though they were just one statement:

```
if (x == 2) {
  call_whitehouse();
  sell_oil();
  x = 0;
}
```

C coders like to keep their code short and snappy, so most will omit braces on `if` conditions and `while` loops. So instead of writing:

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

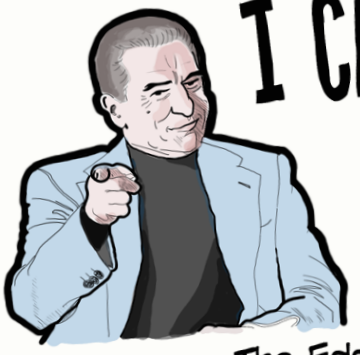
most C programmers write:

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

Here's the code so far

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

Advertising Feature



I CAN MAKE YOU RICH JUST LIKE ME!

The Eddie Rich blackjack correspondence school

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

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

Card counting helps you keep track of the number of high-value cards left. Say you start with a count of 0. Then the dealer leads with a Queen—that's a high card. That's one less available in the deck, so you reduce the count by one:

It's a queen → count – 1

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

It's a four → count + 1

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

You keep doing this for every low card and every high card until the count gets real high, then you lay on cash in your next bet and ba-da-bing! Soon you'll have more money than my third wife!

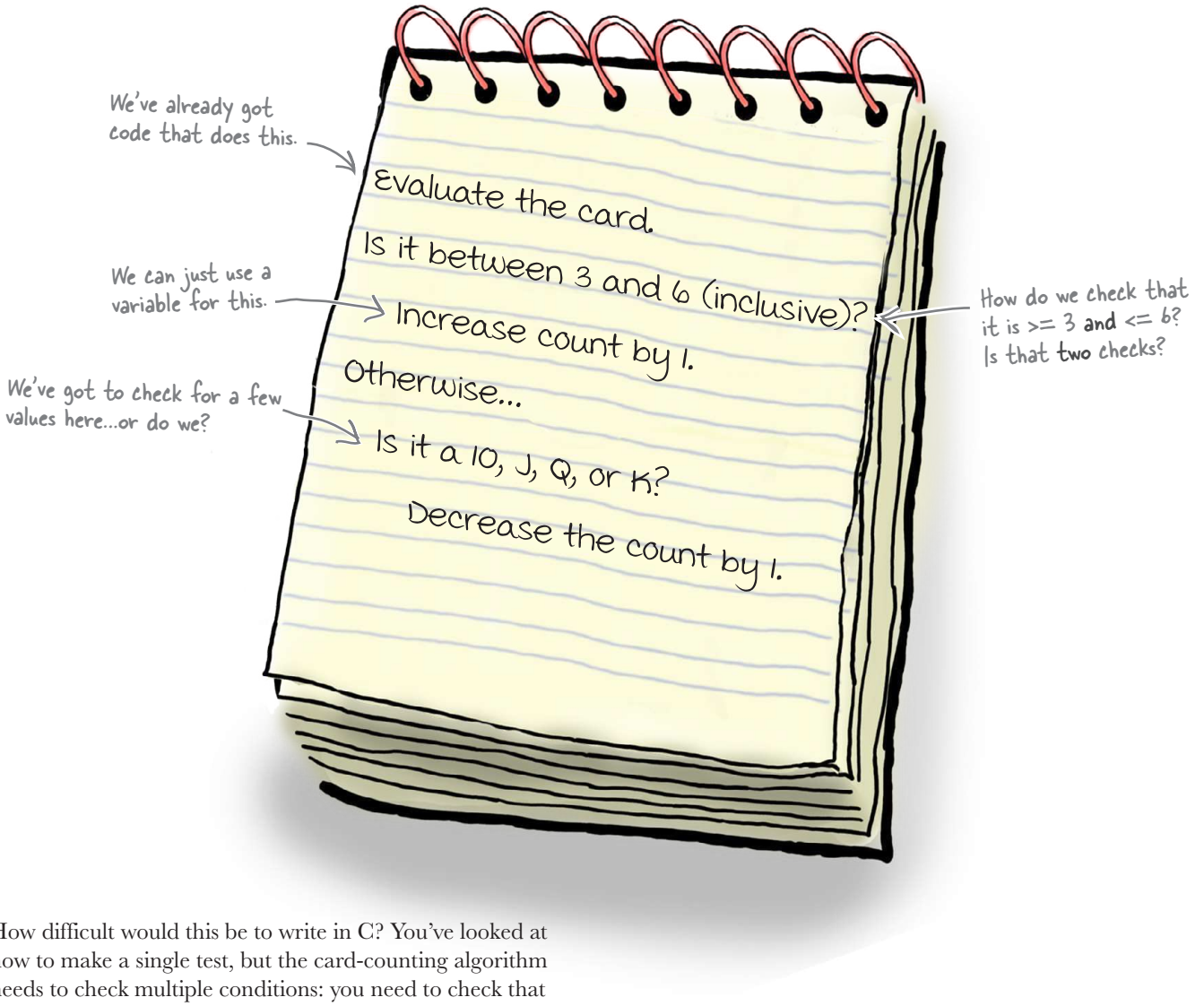
If you'd like to learn more, then enroll today in my Blackjack Correspondence School. Learn more about card counting as well as:

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

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

Card counting? In C?

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



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

You need a set of operations that will allow you to combine conditions together.

There's more to booleans than equals...

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

&& checks if two conditions are true

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

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

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

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

|| checks if one of two conditions is true

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

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

Either can be true.

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

! flips the value of a condition

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

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

! means "not"

Geek Bits

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

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

In fact, C programs often use this as a shorthand way of checking if something is not 0.

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

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

The Polite Guide to Standards

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

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

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

There are a few ways of writing this condition.

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

there are no Dumb Questions

Q: Why not just | and &?

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

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

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

Q: Huh? What do you mean?

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

Test Drive

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

This line compiles and runs the code.

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

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

> ./cards
Enter the card_name:
8

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

>
```

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

The Compiler Exposed

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

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

gcc: That's not a problem, my friend. A pleasure to help.

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

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

Head First: Really?

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

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

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

Head First: How have you achieved such incredible flexibility?

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

Head First: Written in a language such as C?

gcc: Exactly. My frontend can convert that language into an intermediate code. All of my language frontends produce the same sort of code.

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

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

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

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

Head First: Such as?

gcc: Well, I can check obvious things such as misspelled variable names. But I also look for subtler things, such as the redefinition of variables. Or I can warn the programmer if he chooses to name variables after existing functions and so on.

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

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

Head First: You do rather a lot!

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

Head First: `gcc`, thank you.

BE the Compiler

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

A

```
#include <stdio.h>

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

B

```
#include <stdio.h>

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

C

```
#include <stdio.h>

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

    return 0;
}
```

D

```
#include <stdio.h>

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

    return 0;
}
```

BE the Compiler Solution

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

A

```
#include <stdio.h>

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

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

B

```
#include <stdio.h>

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

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

C

```
#include <stdio.h>

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

    return 0;
}
```

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

D

```
#include <stdio.h>

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

    return 0;
}
```

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

What's the code like now?

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

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

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

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

C can perform logical tests with the switch statement.

Pulling the ol' switcheroo

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

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

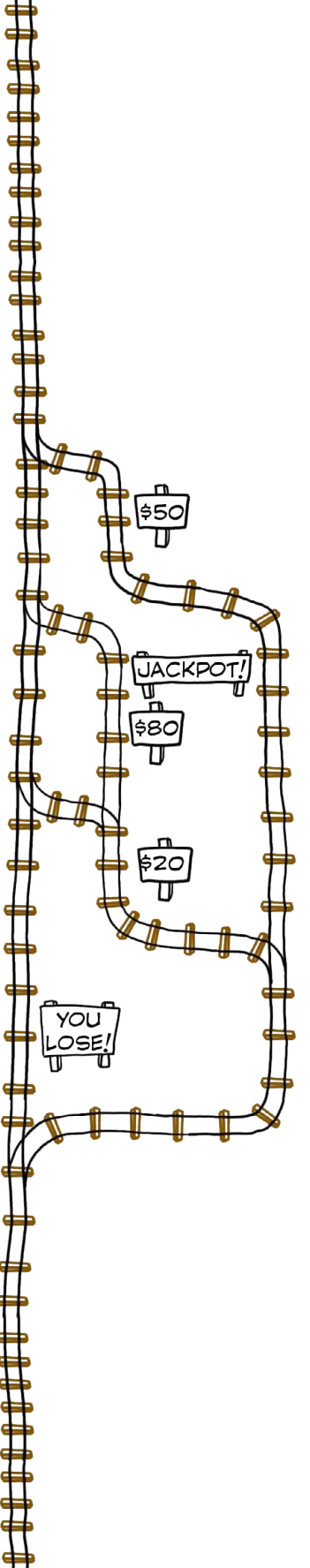
```
switch(train) {
case 37:
  winnings = winnings + 50;
  break;
case 65:
  puts("Jackpot!");
  winnings = winnings + 80;
case 12:
  winnings = winnings + 20;
  break;
default:
  winnings = 0;
}
```

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

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

If the train == 12, just add 20 to the winnings.

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



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

Watch it!

Missing breaks can make your code buggy.

Most C programs have a `break` at the end of each `case` section to make the code easier to understand, even at the cost of some efficiency.

Sharpen your pencil

Let's look at that section of your `cards` program again:

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

Do you think you can rewrite this code using a `switch` statement? Write your answer below:

Sharpen your pencil Solution

You were to rewrite the code using a `switch` statement.

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

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

BULLET POINTS

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

there are no Dumb Questions

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

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

Q: What are the advantages of using a `switch` statement?

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

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

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

Q: Can I check strings in a `switch` statement?

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

Sometimes once is not enough...

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

Using while loops in C

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

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

This checks the condition before running the body.

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

The body is between the braces.

If you have only one line in the body, you don't need the braces.

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

```
while (more_balls)
    keep_juggling();
```

Do you do while?

There's another form of the `while` loop that checks the loop condition *after* the loop body is run. That means the loop always executes **at least once**. It's called the **`do...while` loop**:

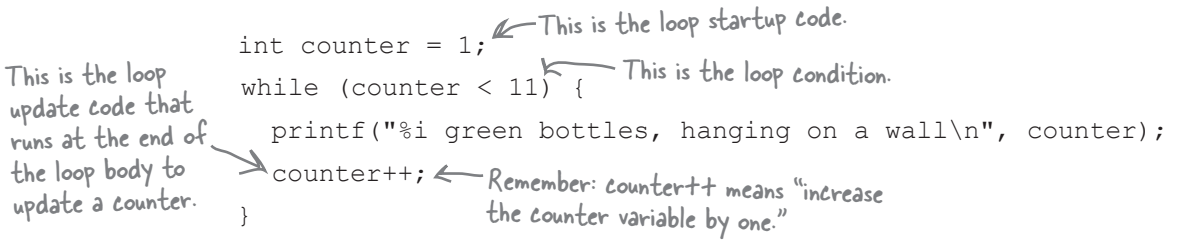
```
do {
  /* Buy lottery ticket */
} while(have_not_won);
```

Loops often follow the same structure...

You can use the `while` loop anytime you need to repeat a piece of code, but a lot of the time your loops will have the same kind of structure:

- Do something simple before the loop, like set a counter.
- Have a simple test condition on the loop.
- Do something at the end of a loop, like update a counter.

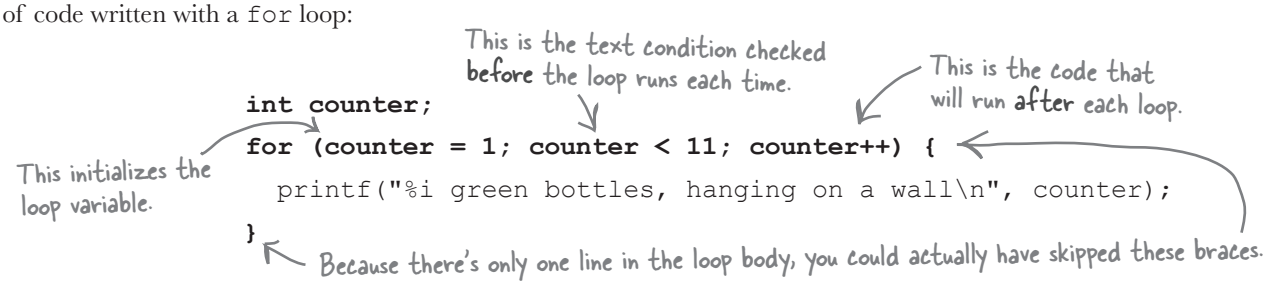
For example, this is a `while` loop that counts from 1 to 10:



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

...and the for loop makes this easy

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



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

Every for loop needs to have something in the body.

You use break to break out...

You can create loops that check a condition at the beginning or end of the loop body. But what if you want to escape from the loop from somewhere in the middle? You could always restructure your code, but sometimes it's just simpler skip out of the loop immediately using the **break** statement:

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

"break" skips out of the loop immediately.

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

Watch it!

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

Make sure that you know what you're `break`*ing out of when you* `break`.

...and continue to continue

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

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

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

Tales from the Crypt

breaks don't break if statements.

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

Writing Functions Up Close

Before you try out your new loop mojo, let's go on a detour and take a quick look at functions.

So far, you've had to create one function in every program you've written, the `main()` function:

```
int main()
{
  puts("Too young to die; too beautiful to live");
  return 0;
}
```

This is the name of the function.

Nothing between these parentheses.

This function returns an int value.

The body of the function is surrounded by braces.

The body of the function—the part that does stuff.

When you're done, you return a value.

Pretty much all functions in C follow the same format. For example, this is a program with a custom function that gets called by `main()`:

```
#include <stdio.h>

int larger(int a, int b)
{
  if (a > b)
    return a;
  return b;
}

int main()
{
  int greatest = larger(100, 1000);
  printf("%i is the greatest!\n", greatest);
  return 0;
}
```

Returns an int value

This function takes two arguments: a and b. Both arguments are ints.

Calling the function here

The `larger()` function is slightly different from `main()` because it takes ***arguments*** or ***parameters***. An **argument** is just a local variable that gets its value from the code that calls the function. The `larger()` function takes two arguments—`a` and `b`—and then it returns the value of whichever one is larger.

The Polite Guide to Standards

The `main()` function has an `int` return type, so you should include a `return` statement when you get to the end. But if you leave the `return` statement out, the code will still compile—though you may get a warning from the compiler. A **C99** compiler will insert a `return` statement for you if you forget. Use **`-std=c99`** to compile to the C99 standard.

Void Functions Up Close

Most functions in C have a return value, but sometimes you might want to create a function that has nothing useful to return. It might just *do* stuff rather than *calculate* stuff. Normally, functions always have to contain a `return` statement, but not if you give your function the return type **`void`**:

The void return type means the function won't return anything.

```
void complain()
{
  puts("I'm really not happy");
}
```

There's no need for a return statement because it's a void function.

In C, the keyword `void` means *it doesn't matter*. As soon as you tell the C compiler that you don't care about returning a value from the function, you don't need to have a `return` statement in your function.

there are no Dumb Questions

Q: If I create a `void` function, does that mean it can't contain a `return` statement?

A: You can still include a `return` statement, but the compiler will most likely generate a warning. Also, there's no point to including a `return` statement in a `void` function.

Q: Really? Why not?

A: Because if you try to read the value of your `void` function, the compiler will refuse to compile your code.

Chaining Assignments

Almost everything in C has a return value, and not just function calls. In fact, even things like assignments have return values. For example, if you look at this statement:

```
x = 4;
```

It assigns the number 4 to a variable. The interesting thing is that the expression "`x = 4`" *itself* has the value that was assigned: 4. So why does that matter? Because it means you can do cool tricks, like chaining assignments together:

The assignment "x = 4" has the value 4.

So now y is also set to 4.

```
y = (x = 4);
```

That line of code will set both `x` **and** `y` to the value 4. In fact, you can shorten the code slightly by removing the parentheses:

```
y = x = 4;
```

You'll often see chained assignments in code that needs to set several variables to the same value.

Mixed Messages

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

```
#include <stdio.h>

int main()
{
    int x = 0;
    int y = 0;
    while (x < 5) {
        [                    ]   <- Candidate code goes here.
        printf("%i%i ", x, y);
        x = x + 1;
    }
    return 0;
}
```

Candidates:

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

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

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

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

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

Match each candidate with one of the possible outputs.

Possible output:

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

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

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

What will you do here?

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

Add 1 to count.

Subtract 1 from count.

You need to stop if she enters X.

Mixed Messages Solution

A short C program is listed below. One block of the program is missing. Your challenge was to **match the candidate block of code** (on the left) **with the output** that you'd see if the block were inserted. Not all of the lines of output were used. You were to draw lines connecting the candidate blocks of code with their matching command-line output.

```
#include <stdio.h>

int main()
{
    int x = 0;
    int y = 0;
    while (x < 5) {
        [                    ]   <-- Candidate code goes here.
        printf("%i%i ", x, y);
        x = x + 1;
    }
    return 0;
}
```

Candidates:

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

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

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

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

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

Possible output:

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

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

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

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

This is just one way of writing this condition.

You need another continue here because you want to keep looping.

You need to check if the first character was an X.

Test Drive

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

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

This will compile and run the program.

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

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

The count is increasing!

The card counting program works!

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

Great job!

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

there are no Dumb Questions

Q: Why do I need to compile C? Other languages like JavaScript aren't compiled, are they?

A: C is compiled to make the code fast. Even though there are languages that aren't compiled, some of those—like JavaScript and Python—often use some sort of hidden compilation to improve their speed.

Q: Is C++ just another version of C?

A: No. C++ was originally designed as an extension of C, but now it's a little more than that. C++ and Objective-C were both created to use object orientation with C.

Q: What's object orientation? Will we learn it in this book?

A: Object orientation is a technique to deal with complexity. We won't specifically look at it in this book.

Q: C looks a lot like JavaScript, Java, C#, etc.

A: C has a very compact syntax and it's influenced many other languages.

Q: What does `gcc` stand for?

A: The Gnu Compiler Collection.

Q: Why "collection"? Is there more than one?

A: The Gnu Compiler Collection can be used to compile many languages, though C is probably still the language with which it's used most frequently.

Q: Can I create a loop that runs forever?

A: Yes. If the condition on a loop is the value 1, then the loop will run forever.

Q: Is it a good idea to create a loop that runs forever?

A: Sometimes. An infinite loop (a loop that runs forever) is often used in programs like network servers that perform one thing repeatedly until they are stopped. But most coders design loops so that they will stop sometime.

BULLET POINTS

- A `while` loop runs code as long as its condition is true.
- A `do-while` loop is similar, but runs the code at least once.
- The `for` loop is a more compact way of writing certain kinds of loops.
- You can exit a loop at any time with `break`.
- You can skip to the loop condition at any time with `continue`.
- The `return` statement returns a value from a function.
- `void` functions don't need `return` statements.
- Most expressions in C have values.
- Assignments have values so you can chain them together (`x = y = 0`).

Your C Toolbox

You've got Chapter 1 under your belt, and now you've added C basics to your toolbox. For a complete list of tooltips in the book, see Appendix ii.

Simple statements are commands.

Block statements are surrounded by { and } (braces).

switch statements efficiently check for multiple values of a variable.

Every program needs a main() function.

You need to compile your C program before you run it.

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

#include includes external code for things like input and output.

if statements run code if something is true.

-o specifies the output file.

You can use && and || to combine conditions together.

while repeats code as long as a condition is true.

gcc is the most popular C compiler.

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

do-while loops run code at least once.

count++ means add 1 to count.

count-- means subtract 1 from count.

for loops are a more compact way of writing loops.