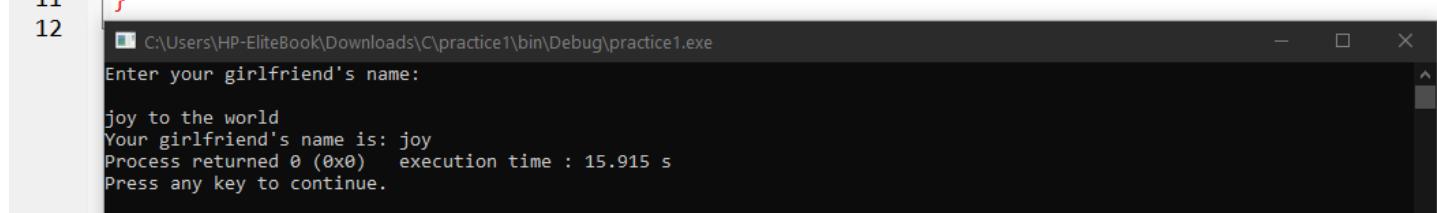


HeadFirst C Chapter 1

Scnf stores a value. **scnf** takes the address of the variable where it should store the input value. This is because scnf needs to modify the variable itself, and not just a copy of it.

EXAMPLE 1:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char ex[20];//array of 19 characters
7     puts("Enter your girlfriend's name: \n");//display on the screen
8     scanf("%19s", &ex); //pass the address of the character array, so it modifies the original string
9     printf("Your girlfriend's name is: %s", ex); //pass the modified array
10    return 0;//exit status success
11 }
12
```



```
C:\Users\HP-EliteBook\Downloads\C\practice1\bin\Debug\practice1.exe
Enter your girlfriend's name:
joy to the world
Your girlfriend's name is: joy
Process returned 0 (0x0)   execution time : 15.915 s
Press any key to continue.
```

Why did this code not take the whole girlfriend name and print it??

The reason your program did not take the whole name "Joy to the world" is because of the format specifier used in the **scanf** statement.

The **%s** format specifier reads a string of characters from standard input, but it stops reading as soon as it **encounters a whitespace** character (space, tab, newline, etc.).

To fix this, you can use the **%[^\\n]** format specifier instead, which reads a string of characters until a newline character is encountered. Here's the corrected program:

With this change, the **scanf** statement will read the entire line of input until a newline character is encountered, and store it in the **ex** array.

Note that I also removed the **&** operator in the **scanf** statement since the name of an array is already a pointer to the first element of the array.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char ex[20];//array of 19 characters
7     puts("Enter your girlfriend's name: \n");//display on the screen
8     scanf("%19[^\\n]", ex);//remove &, because name of array is a pointer to the first element.
9     printf("Your girlfriend's name is: %s", ex); //pass the modified array
10    return 0;//exit status success
11 }
12
```



```
C:\Users\HP-EliteBook\Downloads\C\practice1\bin\Debug\practice1.exe
Enter your girlfriend's name:
joy to the nick
Your girlfriend's name is: joy to the nick
Process returned 0 (0x0)  execution time : 24.729 s
Press any key to continue.
```

EXAMPLE 2:

In C, when a variable is passed to a function, it is passed by value, which means that a copy of the variable is made and passed to the function.

If `scanf` were to receive only the value of the variable, it would modify the copy of the variable and not the original variable.

By passing the address of the variable (using the address-of operator `&`), `scanf` can modify the original variable itself.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int num;
7     puts("Enter number: \n"); //enter a number
8     scanf("%d",&num); //store it in that address
9     printf("The number is: %d\n", num); //print the value
10    return 0;
11 }
12

```

C:\Users\HP-EliteBook\Downloads\C\practice1\bin\Debug\practice1.exe

Enter number:

4

The number is: 4

Switch checks a single variable for different values.

```

1 /* 
2  this program performs checks on the value of
3  letter variable and displays different messages
4  according to the value
5 */
6
7 #include <stdio.h>
8 #include <stdlib.h>
9
10 int main(){
11     //store character H in letter variable
12     char letter = 'H';
13     //perform checks for the value of the variable
14     switch(letter){
15         //is it C,D,H display that message.Default message is also set.
16         //If its value is found, skip past the other checks.
17         case 'C':
18             puts("Clubs, not this one!");
19             break;
20
21         case 'D':
22             puts("Donkey, not this one!");
23             break;
24         case 'H':
25             puts("Found H as the character needed!");
26             break;
27         default:
28             puts("Spade, not this one!");
29     }
30

```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char letter = 'H';
7     switch(letter)
8     {
9         case 'A':
10            puts("Gotcha! its apple ios");
11            break;
12         case 'H':
13            puts("Gotcha! its android os version H");
14            break;
15     }
16 }
17
```

```
C:\Users\HP-EliteBook\Downloads\C\practice1\bin\Debug\practice1.exe
Gotcha! its android os version H

Process returned 0 (0x0)  execution time : 0.058 s
Press any key to continue.
```

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

int main()
{
    int choice;
    puts("Enter a number between 1 and 5");
    scanf("%d", &choice);
    switch(choice)
    {
        case 1:
            puts("Your input is one...");
            break;
        case 2:
            puts("Your input is two...");
            break;
        case 3:
            puts("Your input is three...");
            break;
        case 4:
            puts("Your input is four...");
            break;
    }
}
```

```

case 5:
    puts("Your input is five...");
    break;
default:
    puts("Invalid input!");
}
}

```

If you are checking for multiple values in an integer, you pass the address-of.

If you are checking for multiple values in a character, you pass the name of the character array, since its already a pointer.

Let's log the output to a file and exit the program:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int choice;
    puts("Enter a number between 1 and 5");
    scanf("%d", &choice);
    switch(choice)
    {
        case 1:
            puts("Your input is one...");
            break;
        case 2:
            puts("Your input is two...");
            break;
        case 3:
            puts("Your input is three...");
            break;
        case 4:
            puts("Your input is four...");
            break;
        case 5:
            puts("Your input is five...");
            break;
        default:
            freopen("output.log", "a" ,stdout);
            puts("Invalid input!");
            fclose(stdout);
            exit(1); //exit with a non-zero
            break;
    }
}

```

In this modified program, if the user enters an invalid input, the `freopen` function is used to redirect the standard output (`stdout`) to a file named "output.log" in append mode ("a"). This means that any subsequent output sent to `stdout` will be written to the file instead of the console.

The program then outputs an error message to `stdout`, which will also be written to the file. After the error message is written, the `fclose` function is called to close the file and restore `stdout` to its original state.

Finally, the program exits with an error code (1) using the `exit` function. This is because an invalid input is considered an error condition, and it's often a good practice to **exit a program with a non-zero status code** when an error occurs.

Note that you may need to add `#include <unistd.h>` to your program to use the `exit` function. Also, make sure that you have write permissions for the directory where the output file will be created.

You include the **relevant header files** according to what you want to do.

stdio allows you to read and write data to and from the terminal.

Its the most used.

All C code runs inside functions.

Function is a block of code that runs when called.

Main() function is the starting point of your program.

Main function has is of type integer, that's why it returns an integer.

If it returns 0, the program was successful in running.

Return any other value, the program run was unsuccessful.

Brackets of main can hold **arguments/parameters**.

Printf is used to print out a message on the screen.

Scanf is used to store a value.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 //main function, start of your program.
5 int main()
6 {
7     //display message 'hello world' and print a newline.
8     printf("Hello world!\n");
9     //placeholders for ben and 20.
10    printf("%s is %d years old", "Ben", 20);
11    //return type is integer as our main function type is an integer.
12    return 0;
13 }
```

You can use as many placeholders as you want but make sure you have a matching type for each one.

For example %s is a string, so the first placeholder must be replaced by a string, "ben".

%d is an integer, so the second placeholder must be replaced by an integer, 20, no quotations on integers.

Compiler converts High level language code to machine language code.

Assembler converts assembly language code to machine code.

Compiling C language in other operating systems that don't have a compiler like codeblocks or clion.

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

meaning compile zork and run it.

Compile C program && if compilation is successful run it.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 //main function
5 int main(){
6     //create a character holding two values called card_name.
7     char card_name[3];
8     //display this for user to fill the card_name.
9     puts("Enter card_name: ");
10    //store the entered value.
11    scanf("%2s", card_name);
12    //initialise an integer called val.
13    int val = 0;
14    //check for the different values for the first value of the array and compare.
15    if (card_name[0] == 'K'){
16        val = 10;
17    }
18    else if(card_name[0] == 'Q'){
19        val = 10;
20    }

21    else if(card_name[0] == 'J'){
22        val = 10;
23    }
24    else if(card_name[0] == 'A'){
25        val = 11;
26    }
27    //convert the value to an integer.
28    else{
29        val = atoi(card_name);
30    }
31    //display card name.
32    printf("The card name is %i\n",val);
33    return 0;
34 }
35

```

SENTINEL CHARACTER.

Why do we use an array of characters instead of just using a string to ask for the data??

C language doesn't support strings out of the box, coz its low level than other languages, but it has some libraries that give you strings.

An array is a list of items/objects, given a single name.

First character of an array always starts at index[0].

Let's see how C programming handles text.

A 2-character-array, you write array[3].

A 3-character-array, you write arrray[4].

A 19-character-array, you write arrray[20].

```
39 //how you see a string
40 s = shatner;
41 //how C compiler sees it.
42 s = {'s','h','a','t','n','e','r'};
```

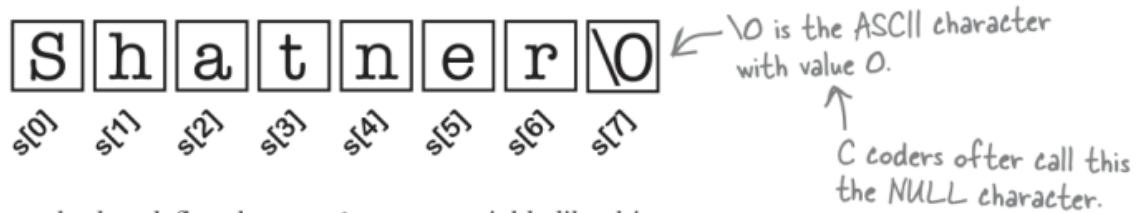
Each of the characters in the string is just an element in the array.

That means you can refer to each element with an index starting from index[0].

C has to know when it has reached the end of the array, so by this, it uses a sentinel character.(\0) - backslash zero.

```
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.

That's why we have to allow for an extra character in the array.

WHY ARE INDEXES COUNTED FROM 0 NOT 1?

It will be easier to calculate the location of a character eg if 'a' is at memory location 1,000,000 then 'z' will be at memory location 1,000,000 + 26.

Coz an index is an offset, the measure of how far a character is from another.

That is why 0,1,2,3 is confusing.

We are storing characters in this array.

The space between them is the offset - How far.

0 to 1 - Stores character 1. Offset of 1, so only holds 1 character.

1 to 2 - Stores character 2.

2 to 3 - Stores character 3.(sentinel character)

The offset is 1, that is from 0 to 1, 1 to 2, 2 to 3, but the index of the character stored is 0, 1,2.

WHY DOES IT NEED SENTINEL??

C doesn't keep track of how long an array is.

And a string is an array.

C relies on you to keep track of your arrays.

DOUBLE AND SINGLE QUOTES??

Double quotes is for strings, single quotes is for characters.

You usually don't write a string as an array `s = {.....}`, but just as string literal eg. "ben".

STRINGS/STRING LITERALS VS ARRAYS

Strings are sequence of characters represented as a single data type, while arrays are a sequential collection of datatype char.

Strings are immutable, you can't change its individual characters once created, while character arrays are mutable.

String literals are constants, but arrays aren't.

If you try change the values of string literals, gcc will display a bus error, meaning, you can't update that piece of memory.

EQUALS VS ASSIGNMENT

= is for assignment.

== is for equality.

```
37 //set the value of teeth to 4.
38 teeth = 4;
39 //test if the value stored in teeth is actually 4.
40 teeth == 4;
41 //take away two teeth.
42 teeth -= 2;
43 //add two teeth.
44 teeth += 2;
45 //increment teeth by 1;
46 teeth++;
47 //decrement teeth by 1;
48 teeth--;
```

BLOCK STATEMENTS.

Simple statements are actions.

They do things and tell us things.

Block statements are groups of commands surrounded by curly braces.

```
37 {  
38     eat_food();  
39     drink_water();  
40     int s = 40;  
41     int derk;  
42 }  
43
```

They allow you to treat a whole set of statements as a single statement.

```
36 if (countdown==0){  
37     do_this_thing();  
38 }
```

The if above runs a single statement. What if you want to run a whole set of commands using the same if statement??

If you wrap a list of statements in braces, C treats them as though they were a single command.

```
36 if (countdown==0){  
37     stop_clock();  
38     call_whitehouse();  
39     sell_oil();  
40     go_jogging();  
41     int x = 20;  
42 }
```

C programmers like to simplify their code, so they will omit braces from while loops and if statements.

DO SOMETHING ONLY IF ANOTHER IS TRUE

```
36 //run the one that is true.  
37 if (fight <= 500)  
38     hit();  
39  
40 else(milk <= 500)  
41     stand();  
42
```

Or if you don't want to omit braces.

```

36  if (countdown==100){
37      do_this();
38  }
39  else if(countdown==50){
40      prepare();
41  }
42  else if(countdown==20){
43      stand_up();
44  }
45  else(countdown==10){
46      start_jogging();
47  }

```

If statements usually do more than one thing when a condition is true, so we usually use them with block statements eg.

```

36  if(card_name == 40){
37      drink_milk();
38      double_down();
39      run();
40  }

```

Commands are grouped inside one block statement.

ATOI AND PLACEHOLDERS.

Atoi() is a function that converts a string to integer.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  //main function
4  int main(){
5      //create 19 character array.
6      char string1[20] = "19841982";
7      //convert the given string to an integer.
8      int val = atoi(string1);
9      //print out both initial string and new integer.
10     printf("String value = %s\n",string1);
11     printf("Integer value = %d\n",val);
12     return 0;
13 }
14

```

String to integer conversion.

It receives a constant and returns a converted integer value.

```
27 //convert the value to an integer.  
28 else{  
29     val = atoi(card_name);  
30 }  
31 //display card name.  
32 printf("The card name is %i\n",val);  
33 return 0;  
34 }
```

The placeholder above is standing for a variable val, that holds an integer.

NOT IMPORTANT FOR NOW.

%i means an unsigned integer(can only be positive).

%lu means an unsigned integer(can be negative or positive).

%u means unsigned integer.

%d means a decimal.

%s means a string.

%p means pointer.

%l or %ld or %li means a long.

%lf means a double.

%Lf means a long double.

%lli or %lld means long long.

%llu means unsigned long long.

%g or %G or %e or %E means scientific notations for floats.

%f means a float.

%o means octal representation.

%n prints nothing.

%% prints percent character.

%x means hexadecimal representation.

CARD COUNTING.

Could we use the program above to know if a card is in a particular range??

Card counting is a way of improving your odds when you play blackjack.

If there are plenty of high value cards left in the shoe, then the odds are slanted in favour of the player, that's you.

Card counting assists you keep track of the 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 bada- 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!

Summary: Cards btwn 3 to 6, ($>=3$ and $<=6$), increase count by 1. Cards 10, Q, J, K, decrease count by 1.

&& CHECKS IF TWO CONDITIONS ARE TRUE.(AND)

`&&` evaluates to true if both conditions are true.

If one condition is false, it doesn't bother evaluate.

```
10  //&& evaluates to true if both conditions are true.  
11  if (dealer_card == 6) && (my_card == 11)  
12      double_down();
```

|| CHECKS IF ONE OF TWO CONDITIONS IS TRUE.(OR)

Evaluates to true if one of the statements is true.

Computer won't bother evaluate the second statement if the first one is true.

It assumes the whole is true, if the first one is true.

```
11  //&& evaluates to true if both conditions are true.  
12  if (cupcake_in_fridge) || (chips_in_table)  
13      sit_and_eat();
```

! FLIPS THE VALUE OF A CONDITION.

Reverses the value of a condition.

```
11 // !flips value of a condition.  
12 if (!brad_on_phone)  
13     answer_phone();
```

BOOLEANS

0(zero) represents false.

Any other value is treated as true.

So, no problem in writing your code like this:

```
11 //set the value of buying glasses to true.  
12 // if you set it to zero, means its false.  
13 int peopleBuyingGlasses = 34;  
14  
15 //if its true people are buying glasses, buy too.  
16 if (peopleBuyingGlasses)  
17     takeOutYourMoney();  
18  
19 //if they aren't buying glasses, save your money.  
20 else(!peopleBuyingGlasses)  
21     saveYourMoney();
```

You can use any value to represent true, as above 34, but only 0 represents false.

THE CARD COUNTER

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      char card_name[3];
6      puts("Enter the card_name: ");
7      scanf("%2s", card_name);
8      int val = 0;
9
10     if (card_name[0]=='K'){
11         val = 10;
12     }
13     else if (card_name[0] == 'Q'){
14         val = 10;
15     }
16     else if(card_name[0] == 'A'){
17         val = 11;
18     }
19     else{
20         val = atoi(card_name);
21     }
22
23     /*
24
25     check if the value is 3 to 6
26     3 to 6 means greater than 2 and less than 7
27
28     */
29
30     if ((val>2) && (val<7))
31         puts("Count has gone up.");
32
33     //otherwise if its 10,J,K,Q.
34     else if(val == 10)
35         puts("Count has gone down.");
36
37     return 0;
38 }
```

&& and || (and and or) perform logical operations.

6 & 4 is 4 coz both have 100's in their binary(110 and 100).

SWITCH.

The code above is repetitious, checking for the same value.

We are going to replace it with another code.

Switch replaces lots of if statements.

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.

Missing breaks can make your code buggy.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      //checks on train variable.
6      switch(train){
7
8          //if 37 or 65 or 12, perform something.
9          case 37:
10             winnings = winnings + 50;
11             break;
12          case 65:
13             puts("Jackpot!");
14             winnings = winnings + 80;
15          case 12:
16             winnings = winnings + 12;
17             break;
18
19          //for any other value of train, reset win to zero.
20          default:
21             winnings = 0;
22
23     }
24 }
```

Code changes.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     char card_name[3];
6     puts("Enter the card name: ");
7     scanf("%2s", card_name);
8     int val = 0;
9
10    switch(card_name[0]){
11        case "K":
12        case "Q":
13        case "J":
14            val = 10;
15            break;
16        case "A":
17            val = 11;
18            break;
19        default:
20            val = atoi(card_name);
```

Switch statements can replace a sequence of if statements.

They perform multiple checks on the same variable.

Switch statements check a single value.

It can check values, compare them if they are equal.

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.

The computer will start to run the code at the first matching case statement.

It will continue to run until it reaches a break or gets to the end of the switch statement.

Check that you've included breaks in the right places; otherwise, your switches will be buggy.

LOOPING.

A **control statement** decides if a section of code will be run.

A **loop statement** decides how many times a piece of code will be run.



WHILE LOOPS

A **while loop** runs code over and over and over as long as some condition remains true.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5
6      int nixon = 21;
7      //as long as condition is true, keep looping.
8      //if you have one line of code in the body, you don't need braces.
9      while(nixon == 21){
10          printf("%i years old\n",nixon);
11      }
12
13      return 0;
14  }
```

```
17 int main(){
18     int i = 1;
19     while(i <= 5){
20         printf("%d\n", i);
21         i++;
22     }
23     return 0;
24 }
```

```
49 int counter = 1;
50 while (counter<10){
51     counter--;
52     printf("The current value is: %d \n", counter);
53 }
```

DO...WHILE LOOPS

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.

```
31 int main(){
32     double number, sum = 0;
33
34     //body of loop is executed at least once.
35     do{
36         printf("Enter the number: ");
37         scanf("%lf", &number);
38         sum += number;
39     }
40
41     while(number != 0.0);
42
43     printf("Sum =%.2lf", sum);
44
45     return 0;
46 }
```

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

```
56 do{  
57     printf("Enter a number: ");  
58     scanf("%lf", &number);  
59     sum += number;  
60 }  
61  
62 while(number!= 0.0);  
--
```

```
% cat fib.c  
#include <stdio.h>  
  
int main(void) {  
    int x, y, z;  
  
    while (1) {  
        x = 0;  
        y = 1;  
        do {  
            printf("%d\n", x);  
  
            z = x + y;  
            x = y;  
            y = z;  
        } while (x < 255);  
    }  
}  
  
% ./fib  
0  
1  
1  
2  
3  
5  
8  
13  
21  
34  
55  
89  
144  
233  
0  
1  
1  
2  
3
```

%lf means double.

```
64 int main(){
65     //local variable definition.
66     int a = 10;
67
68     //do loop execution.
69     do{
70         printf("value of a: %d\n", a);
71         a = a+1;
72     }
73     while(a<20);
74
75     return 0;
76 }
```

FOR LOOPS.

For loop to make it a little more concise.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     //initialize loop variable, check condition ie. if its less than 10.
6     //increment counter.
7     //skip braces, only one line of code in the body.
8     int counter;
9     for(counter=0; counter<10; counter++)
10        printf("%i bottles hanging on the wall\n", counter);
11
12 }
13
```

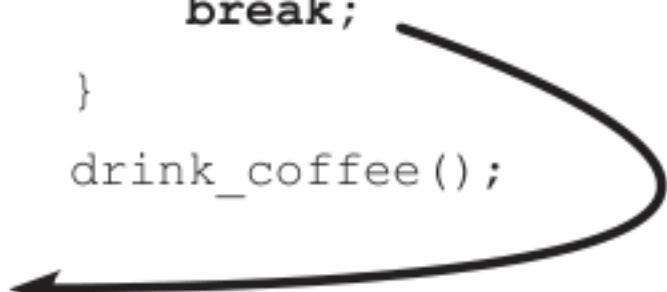
Not only do they make the code slightly shorter, but they're also easier for other C programmers to read, because all of the code that controls the loop—the stuff that controls the value of the counter variable—is now contained in the for statement and is taken out of the loop body.

ctrl + d - copies the line of code, you don't have to repeat the same lines.

SKIP OUT OF THE LOOP USING BREAKS.

```
17  /*SKIPPING OUT OF THE LOOP USING BREAKS*/
18
19  int main(){
20
21      while(feeling_hungry){
22
23          eat_cake();
24
25          if(feeling_quesy){
26              do_this();
27              break; //break out of the while loop.(external loop is cancelled)
28          }
29
30          drink_coffee();
31      }
32  }
```

```
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.

Breaks 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.

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

Make sure that you know what you're breaking out of when you break.

SKIPPING TO THE START OF A LOOP USING 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:

```
34 int main(){
35
36     while(feeling_hungry) {
37
38         if(not_yet_lunch)
39             /*Go back to the loop condition*/
40             continue;
41     }
42
43     eat_cake;
44
45 }
46 }
```

```
→ while(feeling_hungry) {
    if (not_lunch_yet) {
        /* Go back to the loop condition */
        continue; "continue" takes you back
    }                                to the start of the loop.
    eat_cake();
}
```

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 **breaks** don't break out of ifs.

Instead, the program skipped an entire section of code and introduced a bug that interrupted 70 million phone calls over nine hours.

FUNCTIONS.

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.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /*
5   * functions in C have the same format, and they can call each other
6   * function takes two arguments, both are integers.
7  */
8
9  int larger(int a , int b){
10
11     if(a>b)
12         return a;
13     return b;
14
15 }
16
17 int main(){
18
19     int greatest = larger(100,1000); //call the function.
20     printf("%i is the greatest number!!!\n", greatest);
21     return 0;
22
23 }
24
```

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=99` to compile to the C99 standard.

VOID FUNCTIONS.

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`:

```
25 //function returns nothing, so needs no return type.
26 //void just does something.
27 void empty_function(){
28     puts("I am a void function!");
29 }
30 }
```

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.

Add a return statement and compiler throws an error.

Why not? 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;
```

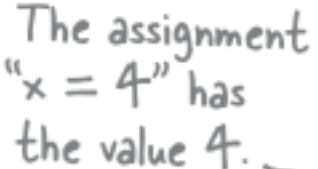
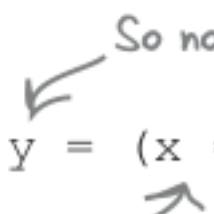
It assigns the number 4 to a variable.

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:

That line of code will set both x and y to the value 4.

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

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     char card_name[3];
6     int count = 0;
7     while(card_name[0]!='X'){
8         puts("Enter the card name: ");
9         scanf("%2s", card_name);
10        int val = 0;
11        switch(card_name[0]){
12            case 'K': //Don't mess and use double quotes.
13            case 'Q':
14            case 'J':
15                val = 10;
16                break;
17            case 'A':
18                val = 11;
19                break;
20            //break wouldn't break us out of the loop, we are inside a switch.
21            //use continue to go back and check the loop condition again.
22            case 'X':
23                continue;
24
25                default:
26                    val = atoi(card_name);
27
28                    if((val<1) || (val>10)){
29                        puts("I don't understand that value!");
30                        continue; //keep looping.
31                    }
32
33                    if((val<2) && (val>7)){
34                        count++;
35                    }
36                    else if (val == 10){
37                        count--;
38                    }
39                    printf("Current count : %i\n",count);
40
41                }
42
43
44    }
45

```

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 ($x = y = 0$).

Simple statements are commands.

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

If statements run code if something is true.

switch statements efficiently check for multiple values of a variable.

Every program needs a main() function.

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

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

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

gcc is the most popular C compiler.

Your source files should have a name ending in .c. -o specifies the output file.

count++ means add 1 to count.

count-- means subtract 1 from count.

CHAPTER 1 while repeats code as long as a condition is true.

do-while loops run code at least once.

for loops are a more compact way of writing loops.

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

Headfirst C Pointers Chapter 2

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

A **pointer** is just the address of a piece of data in memory.

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



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

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.

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.

MEMORY

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.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  /*
4   An address is a pointer coz it points to a variable in memory
5   */
6
7  //create variable x and its stored in globals section of memory.
8  int x = 1;
9
10 int main()
11 {
12     //create variable y and its stored in stack section of memory.
13     int y = 10;
14     //print location of y in stack section
15     printf("y is found at memory location %p\n", &y);
16     //print location of x in globals section
17     printf("x is found at memory location %p\n", &x);
18
19     return 0;
20 }
21
22
```

The computer assigns a memory location to a variable that is created.

Then you assign a value to the variable.

Now you will have a pointer(the address to that variable).

You access a memory location/address using the **& operator**.

GO_SOUTH_EAST FUNCTION

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 //function that adds and subtracts longitudes and latitudes
5 void go_south_east(int lat, int lon){
6     lat = lat - 1;
7     lon = lon + 1;
8
9 }
10
11 //give integers lat and long values, then call the function, print it out
12 int main(){
13     int latitude = 32;
14     int longitude = -64;
15     go_south_east(latitude, longitude);
16     printf("Avast!Now at: [%i,%i]\n", latitude, longitude);
17     return 0;
18 }
19
```

C SENDS ARGUMENTS AS VALUES

The code breaks.

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

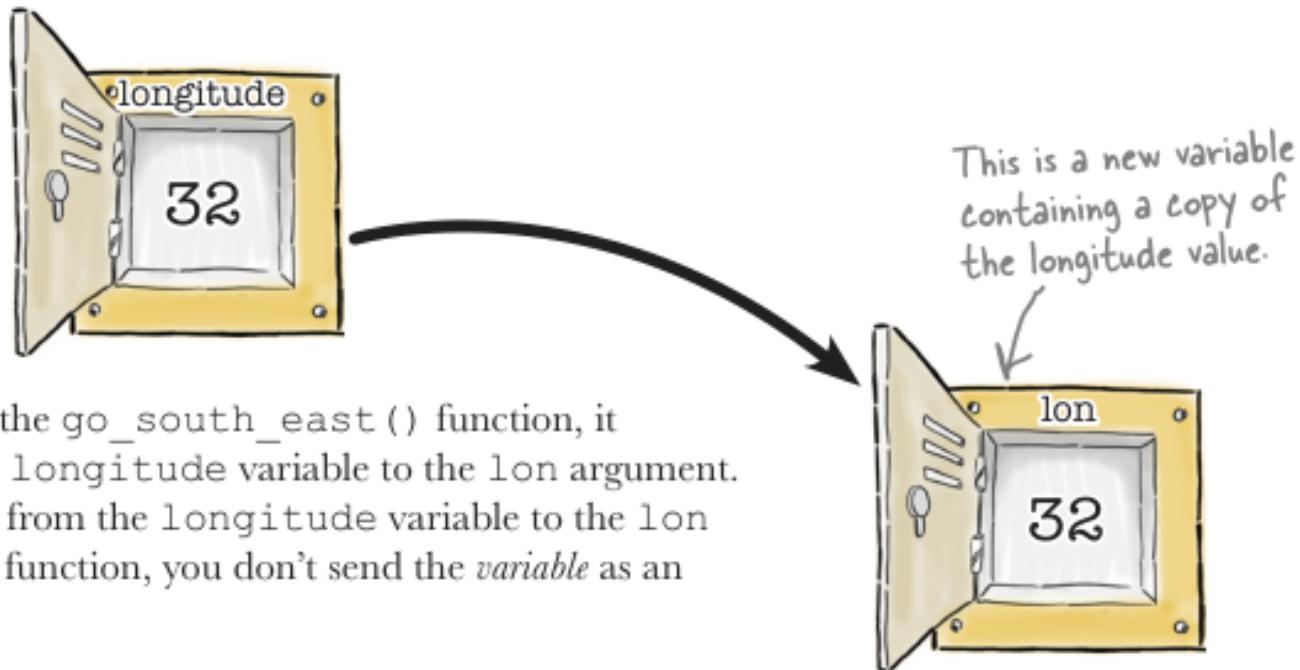
When the computer calls the `go_south_east()` function, it copies the value of the `longitude` variable to the `lon` argument.

This is **just an assignment** from the `longitude` variable to the `lon` argument.

When you call a function, you don't send the variable as an argument, just its value.

When the `go_south_east()` function changes the value of `lon`, the function is just changing its local copy.

That means when the computer returns to the `main()` function, the `longitude` variable still has its original value of 32.



Lon variable changes, longitude variable keeps its 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?

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.

USING POINTERS

Find where a variable is stored in memory using the & operator:

Store it somewhere, in 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:

Read the contents of an address.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 /*
6 the & operator takes A PIECE OF DATA and tells us WHERE ITS STORED.
7 the * operator takes AN ADDRESS and tells us WHAT IS STORED THERE.
8 */
9 int main()
10 {
11     //find the memory location of x
12     int x = 4;
13     printf("x is found at memory location %p\n", &x);
14     //store the address in a pointer variable
15     //what kind of data will be stored
16     int *address_of_x = &x;
17     int value_stored = *address_of_x;
18     //Changing the contents of the address
19     *address_of_x = 99;
20 }
21

```

POINTER OPERATORS

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.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int val = 3; //create an integer, assign it a value
7     printf("val is found at memory location %p\n", &val); //print out its memory location
8     int *addr_of_val = &val; //pointer variable will store the address as an integer
9     *addr_of_val = 10; //change the value of the pointer variable
10
11 }

```

CHANGING 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:

```
//Changing the contents of the address
*address_of_x = 99;
```

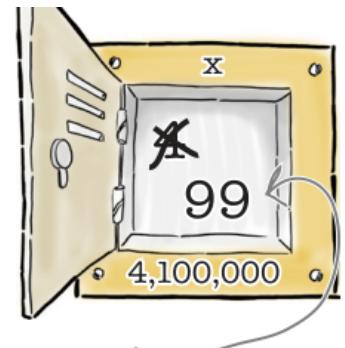
3

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.



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.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void go_south_east(int *lat, int *lon) //show us what's in the memory address
5  {
6      *lat = *lat-1;
7      *lon = *lon+1;
8  }
9
10 int main()
11 {
12     int latitude = 32;
13     int longitude = -64;
14     go_south_east(&latitude, &longitude); //show us the memory address
15     printf("The ship is now at: [%i : %i]\n", latitude, longitude);
16
17 }
```

"C:\Users\ Nixon\Downloads\C programming\Fake projects\Pointers3\bin\Debug\Pointers3.exe"

The ship is now at: [31 : -63]

Process returned 0 (0x0) execution time : 1.898 s

Press any key to continue.

Debugger X Doxy

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.

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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /*
5   show us what's in the memory address
6   arguments will store pointers, so they need to be int*
7   *lat, *lon will read the old value and set the new value
8   You need to find the address of the latitude and longitude variables with &
9
10 */
11
12
13 void go_south_east(int *lat, int *lon)
14 {
15     *lat = *lat-1;
16     *lon = *lon+1;
17 }
18
19 int main()
20 {
21     int latitude = 32;
22     int longitude = -64;
23     go_south_east(&latitude, &longitude); //show us the memory address
24     printf("The ship is now at: [%i : %i]\n", latitude, longitude);
25
26 }
```

SUMMARY

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. 003

The `*` operator can read the contents of a memory address. 003

The `*` operator can also set the contents of a memory address.

QUESTIONS

Are pointers actual address locations? Or are they some other kind of reference? They're actual numeric addresses in the process's memory.

What does that mean? Each process is given a simplified version of memory to make it look like a single long sequence of bytes.

And memory's not like that? 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.

Is memory not just a long list of bytes? 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.

Do I need to understand this? For most programs, you don't need to worry about the details of how the machine arranges its memory.

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

Why does the `%p` format display the memory address in hex format? It's the way engineers typically refer to memory addresses.

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

Oh yeah, C++. Are we going to look at that? No, this book looks at C only.

HOW DO I PASS A STRING TO A FUNCTION

1E6 - 29:37 mins

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void fortune_cookie(char msg[]) //the function will be passed a char array
5  {
6      print("Message reads: %s\n", msg);
7  }
8
9  char quote[] = "Cookies make you fat!";
10 fortune_cookie(quote);
11

```

You know how to pass simple values as arguments to functions(`int lat, int lon`) , 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:

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.

C has an operator called `sizeof` that can tell you how many bytes of space something takes in memory.

On most machines, this → `sizeof(int)` This will return 4, which is 8
will return the value 4. `sizeof("Turtles!")` ← characters plus the \0 end character.

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

/*
to run it ./fortune_cookie
*/
```

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
>
```

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?

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?

When you create an array, the array variable can be used as a pointer to the start of the array in memory.



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 ,(C), 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:

```
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
>
```

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

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:

Phyllis Wangari

Joel

Anthony ndegwa

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

y the `sizeof` operator returned a weird result. It

msg is actually a pointer variable.
msg points to the message.
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.

How C sees my code.

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`.

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.

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.

NOTE:

Size of msg is 8 bytes in 64 bit machine, coz it points to the first character in the array.(a pointer is either 4 bytes or 8 bytes).

Sizeof(pointer). 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.

Msg contains the address of the first character in the array.

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.

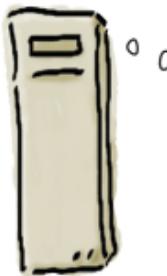
So quote's an array and I've got to pass the quote variable to fortune_cookie().

I'll set the msg argument to the address where the quote array starts in memory.

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.

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.

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.

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

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

So can I find the address of a pointer variable? Yes—using the & operator.

Can I convert a pointer to an ordinary number?

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.

CONTESTANTS MATING GAME

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int contestants[] = {1,2,3};
6     int *choice = contestants;
7     contestants[0] = 2;
8     contestants[1] = contestants[2];
9     contestants[2] = *choice;
10    printf("I'm going to pick contestant number %i\n",contests[2]);
11    return 0;
12 }
13
```

```
#include <stdio.h>
int main()
{
    int contestants[] = {1,2,3};
    int *choice = contestants; //Choice carries or becomes the address of the contestants array
    contestants[0] = 2;
    contestants[1] = contestants[2];
    contestants[2] = *choice;
    printf("I'm going to pick contestant number %i\n",contests[2]);
    return 0;
}
```

contestants[2]

== *choice

== contestants[0]

== 2

when a man loves a woman song.

ARRAY VARIABLE ARE NOT ACTUALLY 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.

`sizeof(apointer)` - returns 4 bytes in 32bit, and 8 bytes in 64bit systems.

`sizeof(array)` - returns the whole length.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char s[] = "How are you doing!"; //is there a character array, and an integer array?? like contestants??
7
8      char *t = s; //pointer variable of s, shows address and what's inside it.
9
10     printf("address of s array is %p\n", &s); //show address of s array
11
12     sizeof(t); //returns 4 or 8 because that's the size of a pointer in 32 or 64 bit systems.
13
14     sizeof(s); //returns the whole length of the array
15
16 }
```

Pointer variable stores the address, while array variable stores the array itself.

If you use the `&` operator on an array variable, the result equals the array variable

itself.

If a coder writes `&s`, that means "What is the address of the `s` array?"

But if someone writes `&t`, that means "What is the address of the `t` variable?"

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?

NOTE:

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.

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.

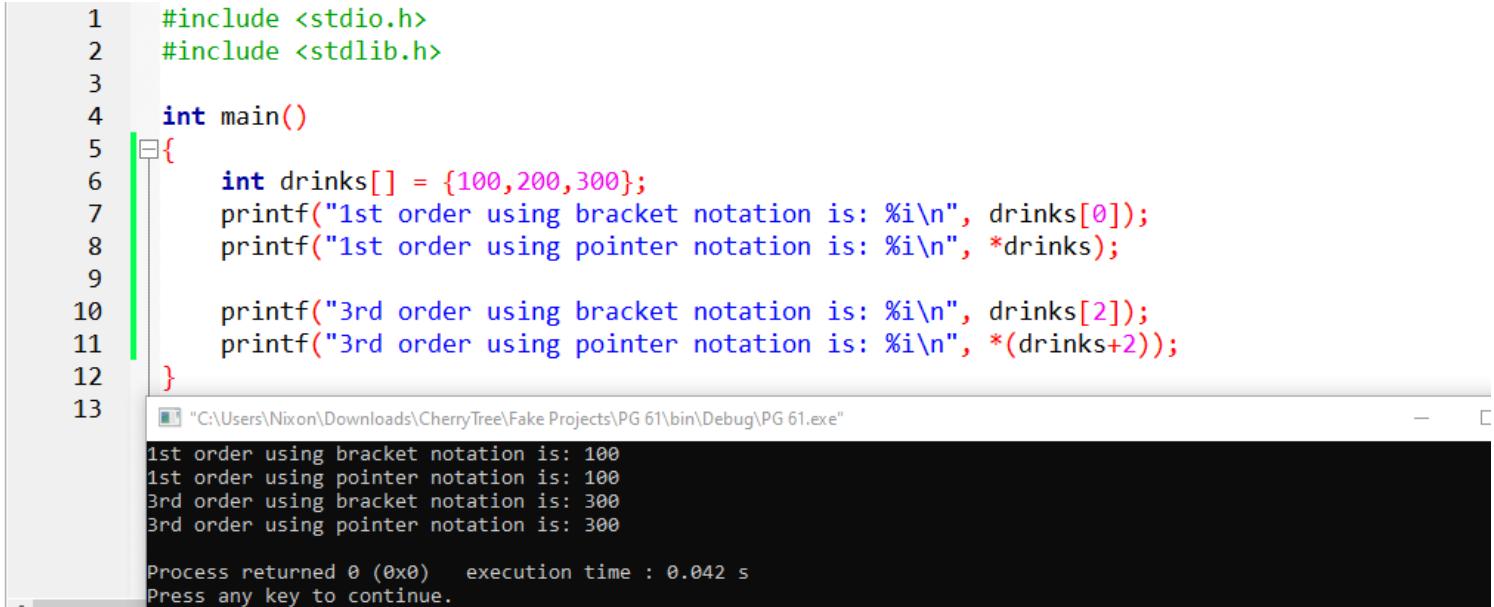
The pointer doesn't know anything about the size of the array, so a little information has been lost.

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.

WHY ARRAYS START AT 0(OFFSET AND POINTER ARITHMETIC)



```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int drinks[] = {100,200,300};
7      printf("1st order using bracket notation is: %i\n", drinks[0]);
8      printf("1st order using pointer notation is: %i\n", *drinks);
9
10     printf("3rd order using bracket notation is: %i\n", drinks[2]);
11     printf("3rd order using pointer notation is: %i\n", *(drinks+2));
12 }
13
```

"C:\Users\Nixon\Downloads\CherryTree\Fake Projects\PG 61\bin\Debug\PG 61.exe"

```
1st order using bracket notation is: 100
1st order using pointer notation is: 100
3rd order using bracket notation is: 300
3rd order using pointer notation is: 300

Process returned 0 (0x0)  execution time : 0.042 s
Press any key to continue.
```

An array variable can be used as a pointer to the first element in the array(IN AN INTEGER ARRAY),and as a pointer to the first character(IN A CHARACTER ARRAY).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int nums[] = {100, 200, 300};
7     printf("The first number using bracket notation method: %i\n", nums[0]);
8     printf("The first number using pointer notation method : %i\n", *nums);
9
10    printf("The third number using bracket notation method: %i\n", nums[2]);
11    printf("The third number using pointer notation method : %i\n", *(nums+2));
12
13
14 }
15
```

```
"C:\Users\ Nixon\Downloads\Code\ C Code\23 IntegerArrays\bin\Debug\23 IntegerArrays.exe"
The first number using bracket notation method: 100
The first number using pointer notation method : 100
The third number using bracket notation method: 300
The third number using pointer notation method : 300

Process returned 0 (0x0)  execution time : 0.022 s
Press any key to continue.
```

That means you can read the first element of the array either by using the brackets notation or using the `*` operator like the examples above.

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:

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.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int nums[] = {1,2,3};
7     //nums will point to the first element of array
8     printf("The nums is at address %p\n", nums);
9     printf("The nums+1 is at address %p\n", nums+1);
10    return 0;
11 }
12
```

```
C:\Users\Nixon\Downloads\C Code\25 PointersHaveTypes\bin\Debug\25 PointersHaveTypes.exe"
The nums is at address 000000000061FE14
The nums+1 is at address 000000000061FE18

Process returned 0 (0x0)  execution time : 2.136 s
Press any key to continue.
```

Case of a lethal dose:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int doses[] = {1,3,2,1000};
7     printf("Issue dose %i\n", 3[doses]);
8     printf("Issue dose %i\n", *(doses +3));
9     printf("Issue dose %i\n", *(3 + doses));
10    printf("Issue dose %i\n", doses[3]);
11 }
12
```

```
C:\Users\Nixon\Downloads\C Code\26 Doses+3\bin\Debug\26 Doses+3.exe"
Issue dose 1000
Issue dose 1000
Issue dose 1000
Issue dose 1000

Process returned 0 (0x0)  execution time : 4.095 s
Press any key to continue.
```

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.

QUESTIONS

- Do I really need to understand pointer arithmetic? 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.
- Can I subtract numbers from pointers? Yes. But be careful that you don't go back before the start of the allocated space in the array.
- When does C adjust the pointer arithmetic calculations? 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. 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.
- Does C use the sizeof operator when it is adjusting pointer arithmetic? 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.
- Can I multiply pointers? No.

USING POINTERS FOR DATA ENTRY

How to ask a user for string input in C using scanf()...

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char name[40];
7     printf("Enter your name: \n");
8     scanf("%39s", name);
9 }
10
```

```
"C:\Users\Nixon\Downloads\C Code\27 Pointers for data entry\bin\Debug\27 Pointers for data entry.exe"
Enter your name:
Nixon Eliakim Ekirapa

Process returned 0 (0x0)  execution time : 16.303 s
Press any key to continue.
```

You can't bypass the number of characters:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char name[40];
7     printf("Enter your name: \n");
8     scanf("%39s", name);
9     printf("My name is: %39s\n", name);
10 }
11
```

```
"C:\Users\Nixon\Downloads\C Code\27 Pointers for data entry\bin\Debug\27 Pointers for data entry.exe"
Enter your name:
nixoneliakimekirapachrispinewangariandthecoadvocatesofruirutownandpolicechaseactivity
My name is: nixoneliakimekirapachrispinewangariandt

Process returned 0 (0x0)  execution time : 30.100 s
Press any key to continue.
```

scanf() reads and stores 39 characters and adds the sentinel character(\0).

Name above is a array variable which is a pointer to the first element in the array, that's why we don't have an & symbol when writing scanf().

Let's see another example:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int age;
7     printf("Enter your age: \n");
8     scanf("%i", &age); //enter integer value, get its address...
9     printf("Your age is: %i", age);
10 }
11
```

```
"C:\Users\Nixon\Downloads\C Code\28 PointerForDataEntry2\bin\Debug\28 PointerForDataEntry2.exe"
Enter your age:
4430
Your age is: 4430
Process returned 0 (0x0)  execution time : 6.890 s
Press any key to continue.
```

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.

`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.

`%i` means the user will enter an `int` value.

Use the & operator to get the address of the int.

Enter an integer.

Enter up to 19 characters (+ '\0').

Enter a floating-point number USING %f.

This reads a first name, then a space, then the second name.

The first and last names are stored in separate arrays.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int age;
7      char first_name[20];
8      char last_name[20];
9      printf("Enter your age: \n");
10     scanf("%i", &age); //enter integer value, get its address...
11     printf("Your age is: %i\n", age);
12     printf("Enter your first and last name: \n");
13     scanf("%19s %19s", first_name, last_name); //array variables are like pointers, no need for &
14     printf("Your first name is: %19s \n Your second name is: %19s", first_name, last_name);
15 }
16
17 Enter your age:
21
Your age is: 21
Enter your first and last name:
nixon
ekirapa
others Your first name is: nixon
Code::Blocks Your second name is: ekirapa
Process returned 0 (0x0)  execution time : 11.821 s
non FilesPress any key to continue.
gram File
```

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:

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:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char food[5];
7     printf("Enter your best food: \n");
8     scanf("%s", food); //food is array variable, also considered a pointer to first character.
9 }
10
```

"C:\Users\Nixon\Downloads\C Code\30 BE CAREFUL WITH SCANF\bin\Debug\30 BE CAREFUL WITH SCANF.exe"

Enter your best food:
aldfkjadlfkjadjfalkdsfjadlkfajdsfaskdfja

Process returned -1073741819 (0xC0000005) execution time : 9.628 s

Press any key to continue.

-1 shows me its unsuccessful

SCANF() CAN CAUSE BUFFER OVERFLOWS

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char food[5];
7     printf("Enter your best food: \n");
8     scanf("%s", food); //food is array variable, also considered a pointer to first character.
9 }
10
```

"C:\Users\Nixon\Downloads\C Code\30 BE CAREFUL WITH SCANF\bin\Debug\30 BE CAREFUL WITH SCANF.exe"

Enter your best food:
liver-meat-and-chapati

Process returned -1073741819 (0xC0000005) execution time : 10.276 s

Press any key to continue.

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.

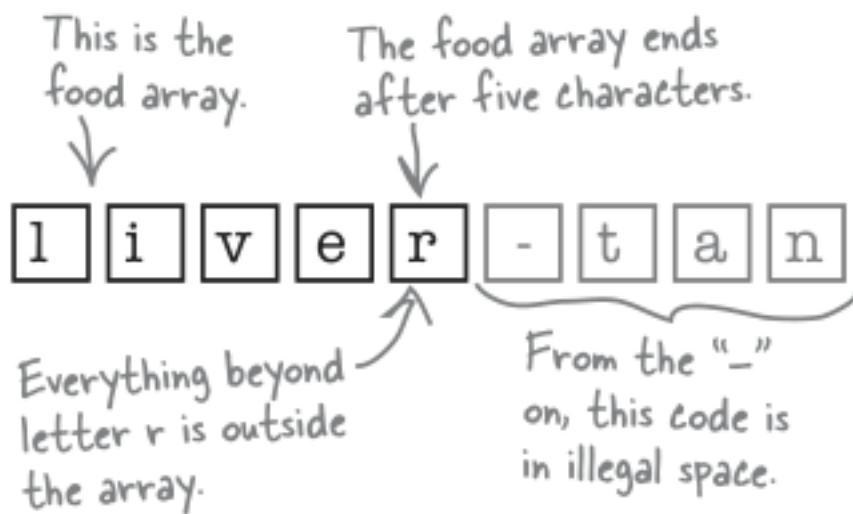
The extra data then gets written into memory that has not been properly allocated

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.



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.

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().

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char food[5];
7     printf("Enter your favorite food: \n");
8     fgets(food, sizeof(food), stdin);
9     //take a pointer to a buffer/array.
10    //take the maximum size of the string including \0.
11    //stdin means data comes from keyboard(more later).
12 }
13
```

```
"C:\Users\Nixon\Downloads\C Code\31 FGETS replaces SCNF\bin\Debug\31 FGETS replaces SCNF.exe"
Enter your favorite food:
liver-meat-and-chapati

Process returned 0 (0x0)  execution time : 14.416 s
Press any key to continue.
```

ONLY USE sizeof WITH fgets(), WHEN DEALING WITH ARRAY VARIABLES NOT NORMAL POINTERS

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);
```

*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.*

Noooooooo!!!!

Seriously,
don't use
this.

pod: ");

--- TITLE FIGHT : SCANF VS 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.

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.

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 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.

THREE CARD MOUNT

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:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char *cards = "JQK";
6      char a_card = cards[2];
7      cards[2] = cards[1];
8      cards[1] = cards[0];
9      cards[0] = cards[2];
10     cards[2] = cards[1];
11     cards[1] = a_card;
12     puts(cards);
13     return 0;
14 }
15
```

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.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char *cards = "JQK";
6     char a_card = cards[2];
7     cards[2] = cards[1];
8     cards[1] = cards[0];
9     cards[0] = cards[2];
10    cards[2] = cards[1];
11    cards[1] = a_card;
12    puts(cards);
13    return 0;
14 }
15
```

"C:\Users\Nixon\Downloads\C Code\32 ThreeCardMount\bin\Debug\32 ThreeCardMount.exe"

Process returned -1073741819 (0xC0000005) execution time : 0.789 s
Press any key to continue.
-1(error)

memory problems

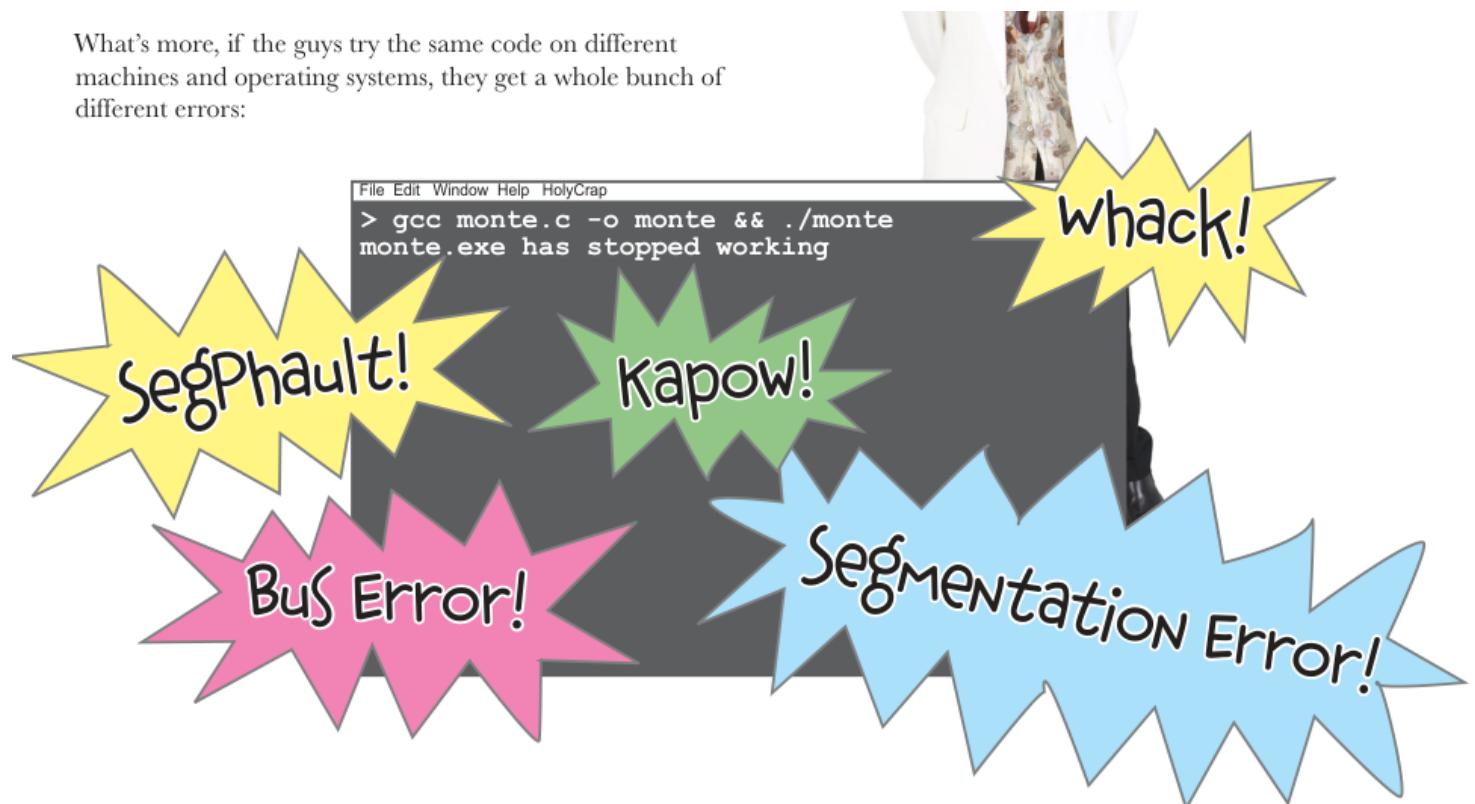
Oops...there's a memory problem...

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
```



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



What's wrong with the code?

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.

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";
```

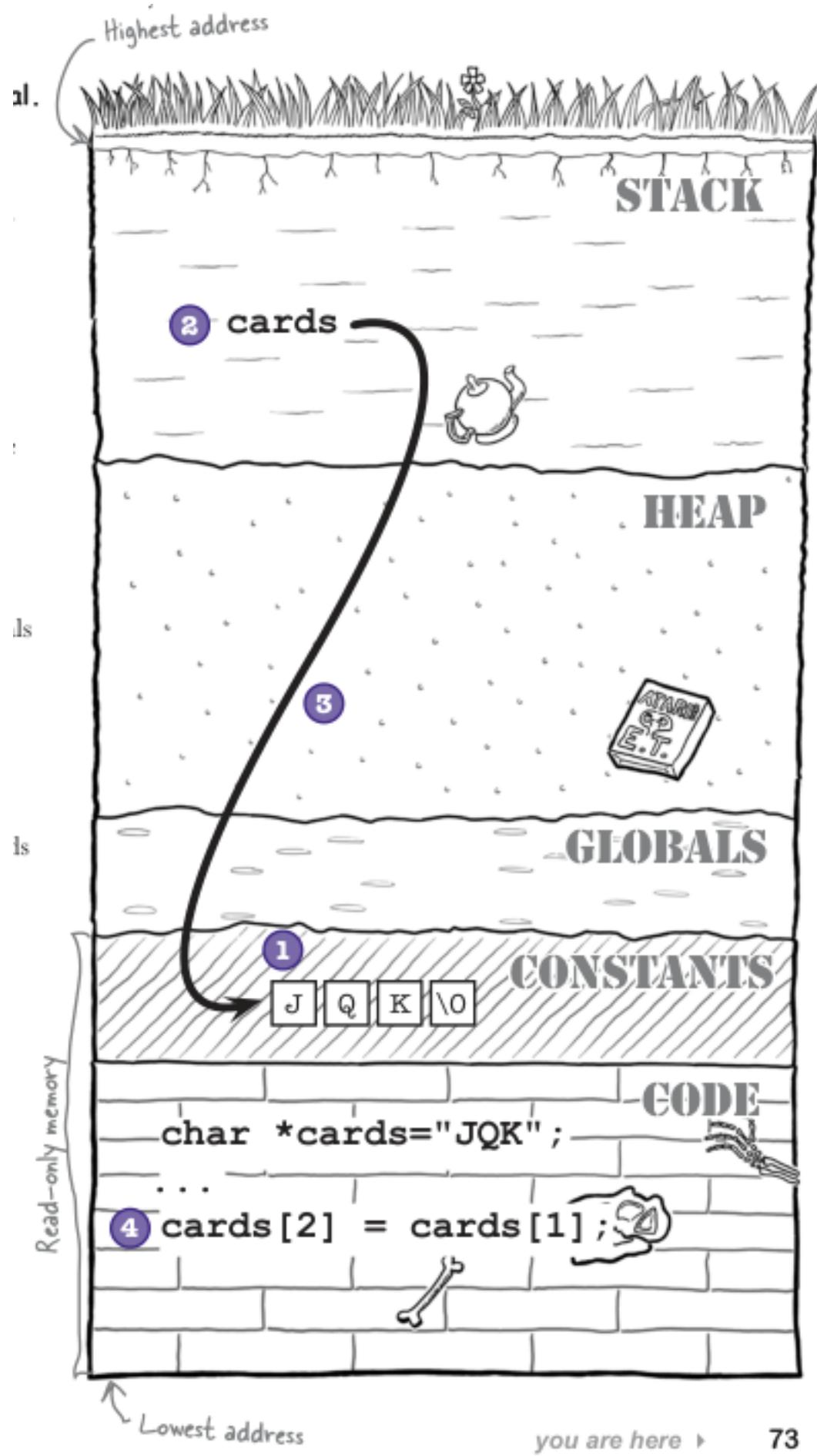
It all comes down to how C uses memory.

HOW C SEES THE POINTER VARIABLE ABOVE AND JQK

char *cards is the pointer variable to hold 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**.



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.

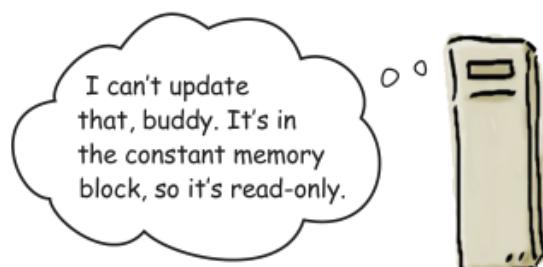
The `cards` variable is set to the address of "JQK."

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.

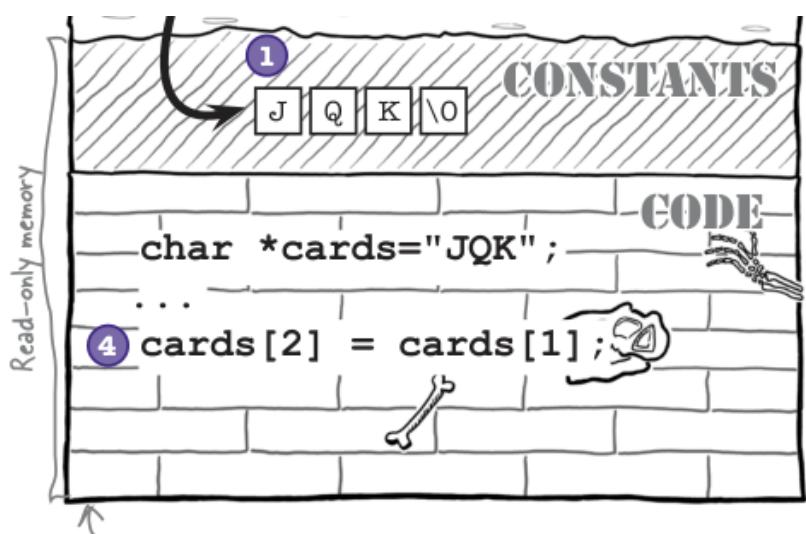
The computer tries to change the string.

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.



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?



IF YOU ARE 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.

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.

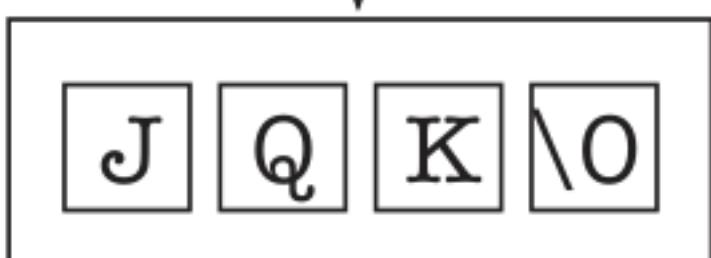
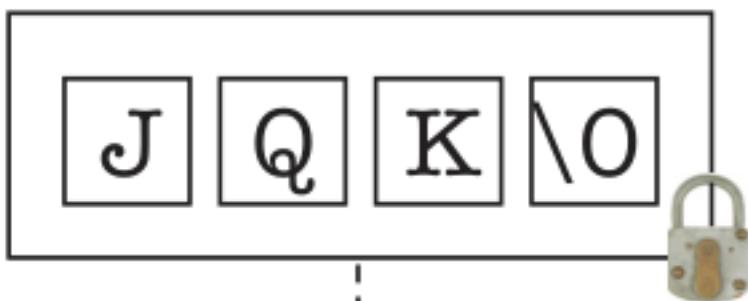
If you declare an array called cards and then set it to a string literal, the cards array will be a completely new copy.

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.

This string is in read-only 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[])
{
    ...
}

void stack_deck(char *cards)
{
    ...
}
```

These two functions are equivalent.

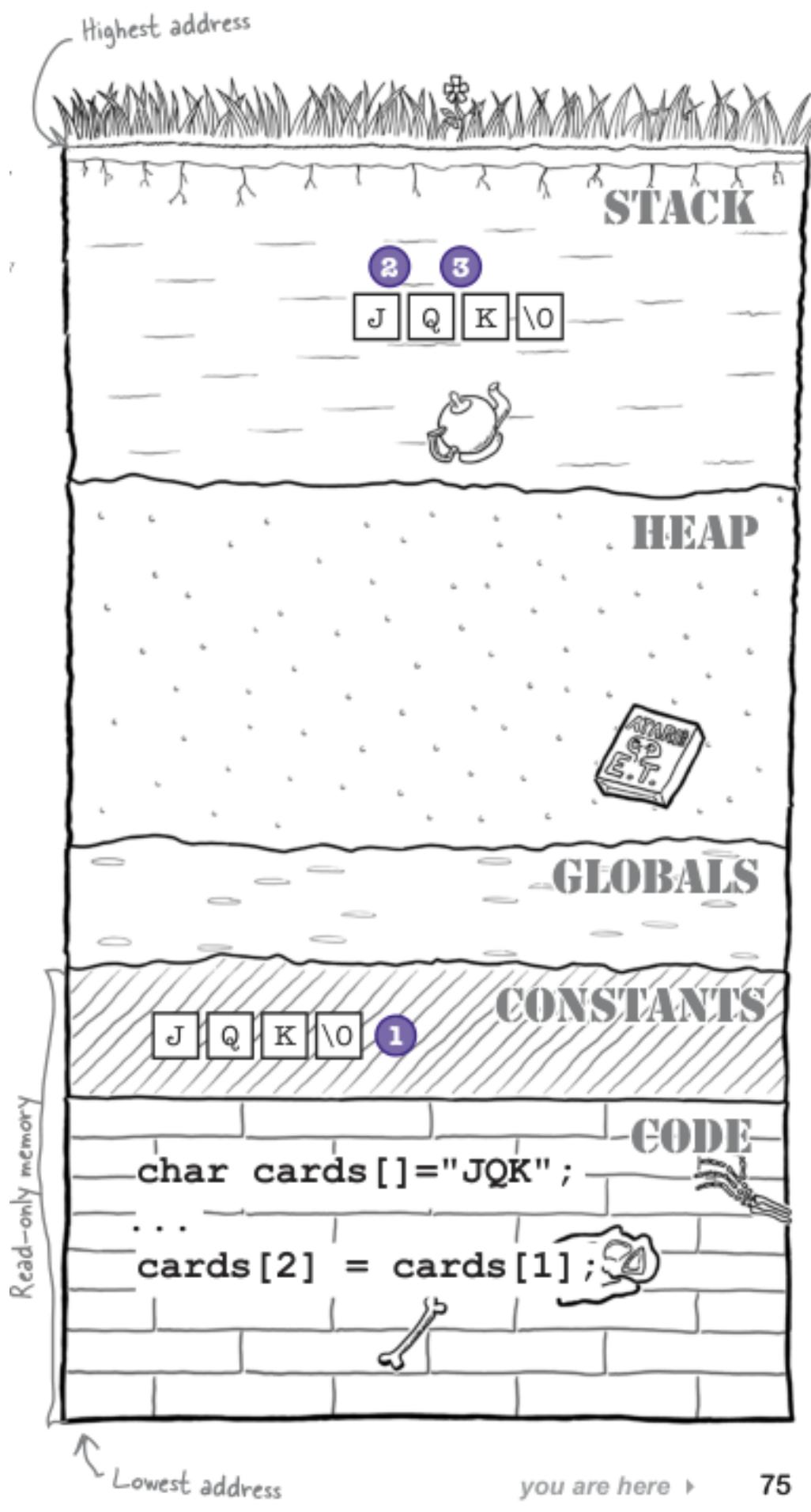
cards is a char pointer.

IN MEMORY OF `char cards[] = "JQK";`

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

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.



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.

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.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     //cards variable is set to address of JQK.
6     //string literal stored in constant, it does not change.
7     char cards[] = "JQK";
8     char a_card = cards[2];
9     cards[2] = cards[1]; //local variable stored in stack.
10    cards[1] = cards[0];
11    cards[0] = cards[2];
12    cards[2] = cards[1];
13    cards[1] = a_card;
14    puts(cards);
15    return 0;
16 }
17
```

"C:\Users\Nixon\Downloads\C Code\33 JQKCorrectCode\bin\Debug\33 JQKCorrectCode.exe"

QKJ

Process returned 0 (0x0) execution time : 3.952 s

Press any key to continue.

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.



; in an
tents.

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 = "This will bug your code";
```

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
```

JIMMY AND MASKED RAIDER

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6
7      char masked_raider[] = "Alive";
8      char *jimmy = masked_raider;
9      printf("Masked raider is %s , Jimmy is %s",masked_raider,jimmy);
10
11
12 }
13
```

"C:\Users\Nixon\Downloads\C Code\34 JimmyMaskedRaider\bin\Debug\34 JimmyMaskedRaider.exe"

```
Masked raider is Alive , Jimmy is Alive
Process returned 0 (0x0)  execution time : 3.950 s
Press any key to continue.
```

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6
7     char masked_raider[] = "Alive";
8     char *jimmy = masked_raider;
9     printf("Masked raider is %s , Jimmy is %s\n", masked_raider, jimmy);
10    masked_raider[0] = 'D';
11    masked_raider[1] = 'E';
12    masked_raider[2] = 'A';
13    masked_raider[3] = 'D';
14    masked_raider[4] = '!';
15    printf("Masked raider is %s, Jimmy is %s\n", masked_raider, jimmy);
16
17
18 }
19

```

```

"C:\Users\Nixon\Downloads\C Code\34 JimmyMaskedRaider\bin\Debug\34 JimmyMaskedRaider.exe"
Masked raider is Alive , Jimmy is Alive
Masked raider is DEAD!, Jimmy is DEAD!

Process returned 0 (0x0)  execution time : 3.917 s
Press any key to continue.

```

"I'm afraid I got some bad news for you. Jimmy and the Masked Raider...were one and the same man!"



"`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."

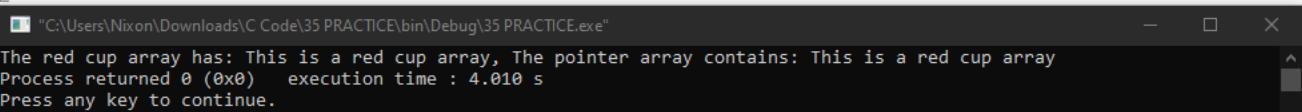
If you see a `*` in a variable declaration, it means the variable will be a pointer.

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.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char cups[] = "This is a red cup array";
7     char *red_cup = cups; //no [] after the cups
8     printf("The red cup array has: %s, The pointer array contains: %s", red_cup, cups); //no * in red cup
9 }
10
```



QUESTIONS FOR CHAPTER 2

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

Why are string literals stored in read-only memory? 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.

Do all operating systems enforce the read-only rule? 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.

What does const actually mean? Does it make the string read- only? 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.

Do the different memory segments always appear in the same order in memory? 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.

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

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

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

Why is `scanf()` called `scanf()`? `scanf()` means "scan formatted" because it's used to scan formatted input.

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.



This is where `string.h` comes in. `string.h` is part of the C Standard Library that's dedicated to string manipulation.

If you want to concatenate strings together, copy one string to another, or compare two strings, the functions in `string.h` are there to help.

In this chapter, you'll see how to create an array of strings, and then take a close look at how to search within strings using the `strstr()` function.

ARRAY OF ARRAYS

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

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

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

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

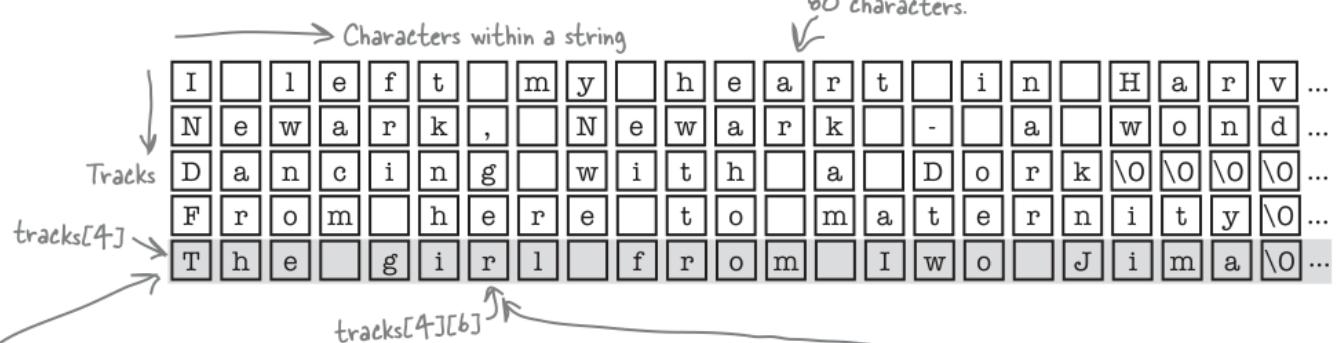
You know that track names will never get longer than 79 characters, so set the value to 80.

This first set of brackets is for the array of all strings.
The compiler can tell that you have five strings, so you don't need a number between these brackets.
Each string is an array, so this is an array of arrays.

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

The second set of brackets is used for each individual string.
You know that track names will never get longer than 79 characters, so set the value to 80.

The array of arrays looks something like this in memory:



That means that you'll be able to find an individual track name like this:

`tracks[4]` → This has this value. → "The girl from Iwo Jima" → This is the fifth string. ← Remember: arrays begin at zero.

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

`tracks[4][6]` → 'r' → This is the seventh character in the fifth string.

Bevon makori - character array 600/=

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char tracks[][][80] = {
7         "I left my heart in Havard Med School",
8         "Newark, Newark - a wonderful town",
9         "Dancing with a Dork",
10        "From here to maternity",
11        "The girl from Iwo Jima",
12
13    };
14
15    printf("The array of arrays is: %s\n", tracks);
16    printf("Track 4 is: %s\n", tracks[4]);
17
18 }
```

```
C:\Users\ Nixon\Downloads\ C Code\36ArrayOfArrays\bin\Debug\36ArrayOfArrays.exe
```

```
The array of arrays is: I left my heart in Havard Med School
Track 4 is: The girl from Iwo Jima
```

```
Process returned 0 (0x0)  execution time : 3.926 s
Press any key to continue.
```

Ask the user for the text she's looking for. Loop through all of the track names.

If a track name contains the search text, display the track name.

Find strings containing the search text.

Well, you know how to record the tracks.

You also know how to read the value of an individual track name, so it shouldn't be too difficult to loop through each of them.

You even know how to ask the user for a piece of text to search for.

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

STRING.H

The C Standard Library is a bunch of useful code that you get for free when you install a C compiler.

The library code does useful stuff like opening files, or doing math, or managing memory.

Now, chances are, you won't want to use the whole of the Standard Library at once, so the library is broken up into several sections, and each one has a header file.

The header file lists all of the functions that live in a particular section of the library.

So far, you have only really used the stdio.h header file.

stdio.h lets you use the standard input/output functions like printf and scanf.

But the Standard Library also contains code to process strings.

String processing is required by a lot of the programs, and the string code in the Standard Library is tested, stable, and fast.



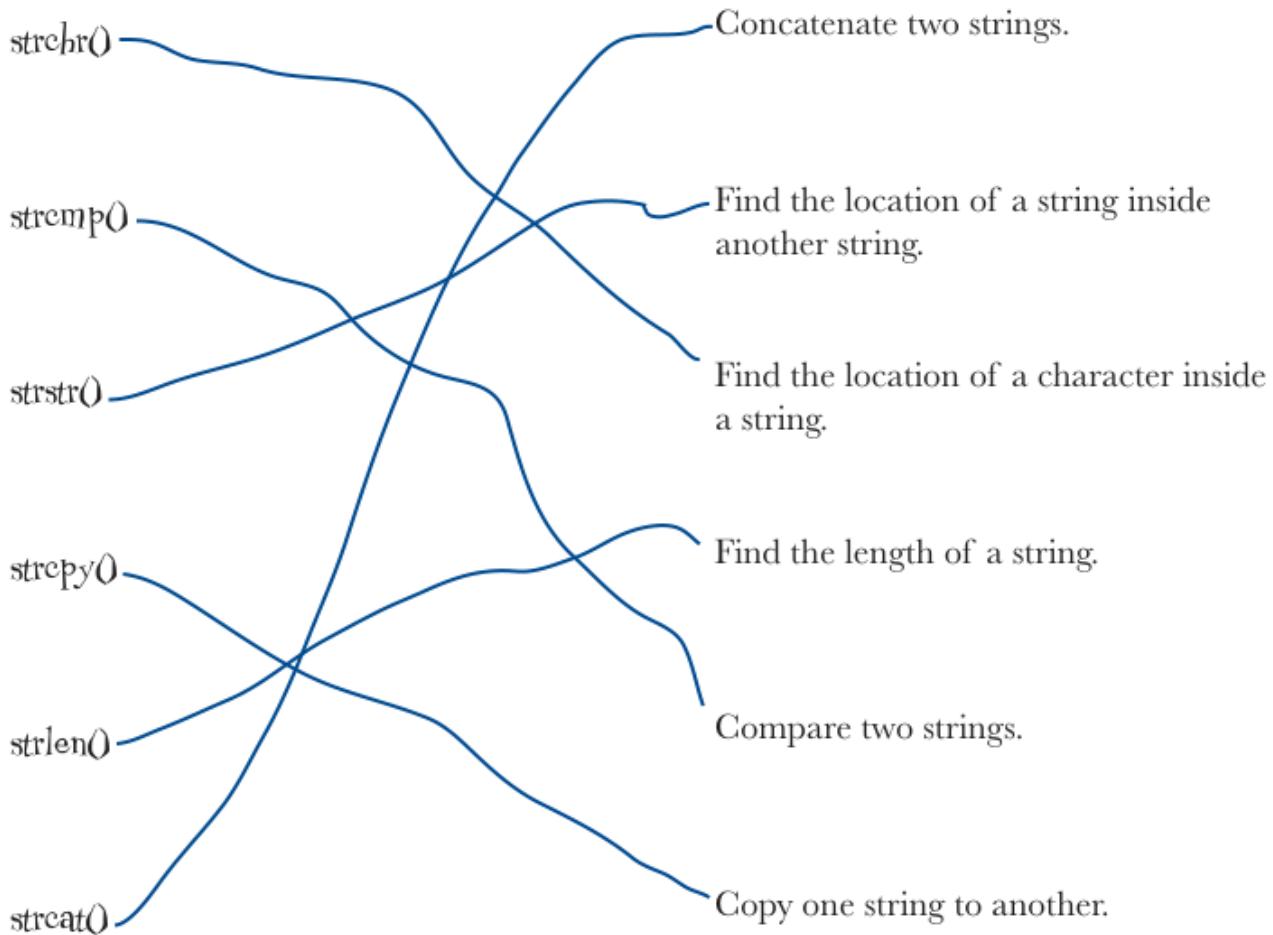
You include the string code into your program using the `string.h` header file.

You add it at the top of your program, just like you include `stdio.h`.

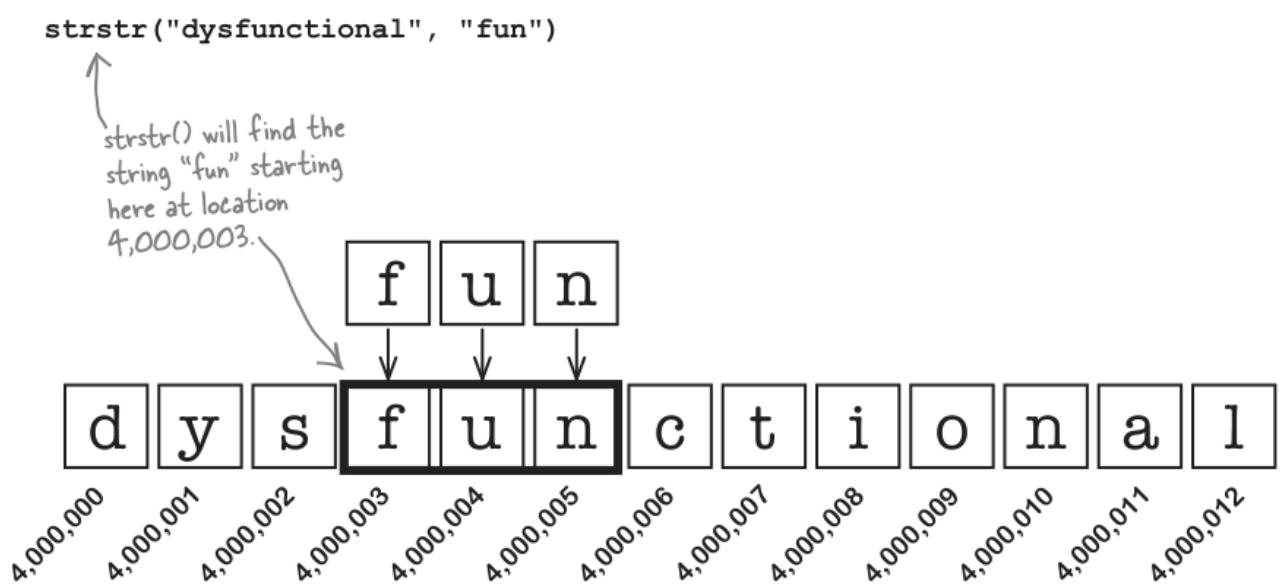
```
#include <stdio.h>
#include <string.h>
```

You'll use both `stdio.h` and `string.h` in your jukebox program.

You were to match up each *string.h* function with the description of what it does.



JukeBox program needs you to find a string inside another string, so **strstr()**.



The **strstr()** function will search for the second string in the first string.

If it finds the string, it will return the address of the located string in memory.

In the example here, the function would find that the fun substring begins at memory location 4,000,003.

But what if the strstr() can't find the substring? What then?

In that case, strstr() returns the value 0.

Can you think why that is?

Well, if you remember, C treats zero as false.

That means you can use strstr() to check for the existence of one string inside another, like this:

Look at this code, then see how i never used strstr, yet the result came out.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main()
6  {
7      char string1 = "dysfunctional";
8      char string2 = "fun";
9      if(string1, string2)
10         puts("I've found fun in dysfunctional");
11
12     return 0;
13
14
```

"C:\Users\Nixon\Downloads\C Code\37 StrStr JukeBox\bin\Debug\37 StrStr JukeBox.exe"

I've found fun in dysfunctional

Process returned 0 (0x0) execution time : 3.913 s

Press any key to continue.

Then followed by this other one, whose code is obviously not doing anything, but gives a result. Watch out when using string functions:

```
int main()
{
    char string1 = "JukeBox";
    char string2 = "window"; //random letter
    if(string1, string2)
        puts("String 2 was found inside string 1");
}
```

```
"C:\Users\Nixon\Downloads\C Code\37 StrStr JukeBox\bin\Debug\37 StrStr JukeBox.exe"
String 2 was found inside string 1

Process returned 0 (0x0)    execution time : 3.913 s
Press any key to continue.
```

this is wrong code, it doesn't know what its doing....

Then this code looks correct, until you notice there are no square brackets.

See the mistakes you can make?

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main()
5 {
6     char string1 = "dysfunctional";
7     char string2 = "fun";
8     if(strstr(string1, string2))
9         puts("I've found fun in dysfunctional");
10    else
11        puts("The program crashed!");
12
13    return 0;
14 }
```

```
"C:\Users\Nixon\Downloads\C Code\37 StrStr JukeBox\bin\Debug\37 StrStr JukeBox.exe"
```

```
Process returned -1073741819 (0xC0000005)    execution time : 4.583 s
Press any key to continue.
```

Failed because I never used the brackets to indicate string 1 and 2 are arrays.... []

This is the correct program.

Don't forget to import the header file, when working with strings.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main()
5 {
6     char string1[] = "dysfunctional";
7     char string2[] = "fun";
8     if(strstr(string1, string2))
9         puts("I've found fun in dysfunctional");
10    else
11        puts("The program crashed!");
12
13    return 0;
14 }
15
```

```
["C:\Users\ Nixon\Downloads\ C Code\37 StrStr JukeBox\bin\Debug\37 StrStr JukeBox.exe"]
```

```
I've found fun in dysfunctional
```

```
Process returned 0 (0x0) execution time : 3.951 s
```

```
Press any key to continue. It run successfully this time.... Don't forget brackets!!
```

This one is also correct:

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main()
5 {
6     char name1[] = "Nixon Ekirapa";
7     char name2[] = "Nixon";
8
9     if(strstr(name1, name2))
10        puts("That is nixon's name ");
11    else
12        puts("That's was not nixon's name");
13 }
14
```

```
["C:\Users\ Nixon\Downloads\ C Code\38 StrStr 2\bin\Debug\38 StrStr 2.exe"]
```

```
That is nixon's name
```

```
Process returned 0 (0x0) execution time : 3.944 s
```

```
Press any key to continue.
```

If you'd have misspelt the name Nixon(capital letter), the result would be:

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main()
5 {
6     char name1[] = "Nixon Ekirapa";
7     char name2[] = "nixon";//misspelt
8
9     if(strstr(name1, name2))
10        puts("That is nixon's name ");
11    else
12        puts("That's was not nixon's name");
13 }
14
```

```
"C:\Users\Nixon\Downloads\C Code\38 StrStr 2\bin\Debug\38 StrStr 2.exe"
That's was not nixon's name

Process returned 0 (0x0)  execution time : 3.912 s
Press any key to continue.
```

JUEBOX PROGRAM

```
1 #include <stdio.h>
2 #include <string.h>
3
4 //GLOBAL ARRAY OF ARRAY
5 char tracks[][][80] = {
6     "I left my heart in Harvard Med School",
7     "Newark, Newark - a wonderful town",
8     "Dancing with a Dork",
9     "From here to maternity",
10    "The girl from Two Jima",
11};
12
13 //WRITE FIND_TRACK FUNCTION FIRST
14 void find_track(char search_for[])
15 {
16     int i;
17     for(i=0; i<5; i++)
18     {
19         if strstr(tracks[i], search_for))
20             printf("Tracks %i: '%s' \n", i , tracks[i]);
21     }
22 }
23
24 //CALL FIND_TRACK FROM MAIN
25 int main()
26 {
27     char search_for[80];
28     printf("Search for: ");
29     fgets(search_for, 80, stdin); //use full length
30     find_track(search_for); //pass the char array in find track when calling it
31
32 }
```

It's time for a code review

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

```
You still need to stdio.h for the
printf() and scanf() functions. → #include <stdio.h>
You'll set the tracks array
outside of the main() and
find_track() functions; that
way, the tracks will be usable
everywhere in the program. → #include <string.h> ← You will also need the string.h
header, so you can search
with the strstr() function.

This is your new find_track()
function. You'll need to declare it
here before you call it from main(). → char tracks[][][80] = {
    "I left my heart in Harvard Med School",
    "Newark, Newark - a wonderful town",
    "Dancing with a Dork",
    "From here to maternity",
    "The girl from Iwo Jima",
};

This code will display all
the matching tracks. → void find_track(char search_for[])
{
    int i;
    for (i = 0; i < 5; i++) {
        if (strstr(tracks[i], search_for))
            printf("Track %i: '%s'\n", i, tracks[i]);
    }
}

And this is your main() function,
which is the starting point of
the program. → int main()
{
    char search_for[80];
    printf("Search for: ");
    fgets(search_for, 80, stdin);
    find_track(search_for); ← You're asking for the
    return 0;               search text here.
                           Now you call your new
                           find_track() function and
                           display the matching tracks.
}
```

It's important that you assemble the code in this order.

The headers are included at the top so that the compiler will have all the correct functions before it compiles your code.

Then you define the tracks before you write the functions.

This is called putting the tracks array in global scope.

A global variable is one that lives outside any particular function.

Global variables like tracks are available to all of the functions in the program.

Finally, you have the functions: find_track() first, followed by main().

The find_track() function needs to come first, before you call it from main().

And the great news is, the program works!

Even though this program is a little longer than any code you've written so far, it's actually doing a lot more.

It creates an array of strings and then uses the string library to search through all of them to find the music track that the user was looking for.

More information about the functions available in `string.h`.

QUESTIONS

Why is the list of tracks defined as `tracks[][][80]`? Why not `tracks[5][80]`? You could have defined it that way, but the compiler can tell there are five items in the list, so you can skip the [5] and just put [].

But in that case, why couldn't we just say `tracks[][]`? The track names are all different lengths, so you need to tell the compiler to allocate enough space for even the largest.

Does that mean each string in the tracks array is 80 characters, then? The program will allocate 80 characters for each string, even though each of them is much smaller.

So the tracks array takes 80×5 characters = 400 characters' worth of space in memory? Yes.

What happens if I forget to include a header file like `string.h`? For some header files, the compiler will give you a warning and then include them anyway. For other header files, the compiler will simply give a compiler error.

Why did we put the tracks array definition outside of the functions? We put it into global scope. Global variables can be used by all functions in the program.

Now that we've created two functions, how does the computer know which one to run first? The program will always run the `main()` function first.

Why do I have to put the `find_track()` function before `main()`? C needs to know what parameters a function takes and what its return type is before it can be called.

What would happen if I put the functions in a different order? In that case, you'd just get a few warnings.

You can create an array of arrays with `char strings[...][...]`.  

The first set of brackets is used to access the outer array.  

The second set of brackets is used to access the details of each of the inner arrays.

The `string.h` header file gives you access to a set of string manipulation functions in the C Standard Library.  

You can create several functions in a C program, but the computer will always run `main()` first.

USING `strlen()`

The guys are working on a new piece of code for a game.

They've created a function that will display a string backward on the screen.

Unfortunately, some of the fridge magnets have moved out of place.

Do you think you can help them to reassemble the code?

```

void print_reverse(char *s)
{
    size_t len = strlen(s);

    char *t = ..... + ..... - 1;
    while ( ..... >= ..... ) {
        printf("%c", *t);

        t = ..... - ..... 1; ← Calculating addresses like this is
    }                                called "pointer arithmetic."
    puts("");
}

```

ARRAY OF ARRAYS VS ARRAY OF POINTERS

You've seen how to use an array of arrays to store a sequence of strings, but another option is to use an array of pointers.

An array of pointers is actually what it sounds like: a list of memory addresses stored in an array.

It's very useful if you want to quickly create a list of string literals:

```

char *names_for_dog[] = {"Bowser", "Bonza", "Snodgrass"};

```

↑ This is an array that stores pointers.

↑ There will be one pointer pointing at each string literal.

You can access the array of pointers just like you accessed the array of arrays.

Now that the guys have the `print_reverse()` function working, they've used it to create a crossword.

The answers are displayed by the output lines in the code.

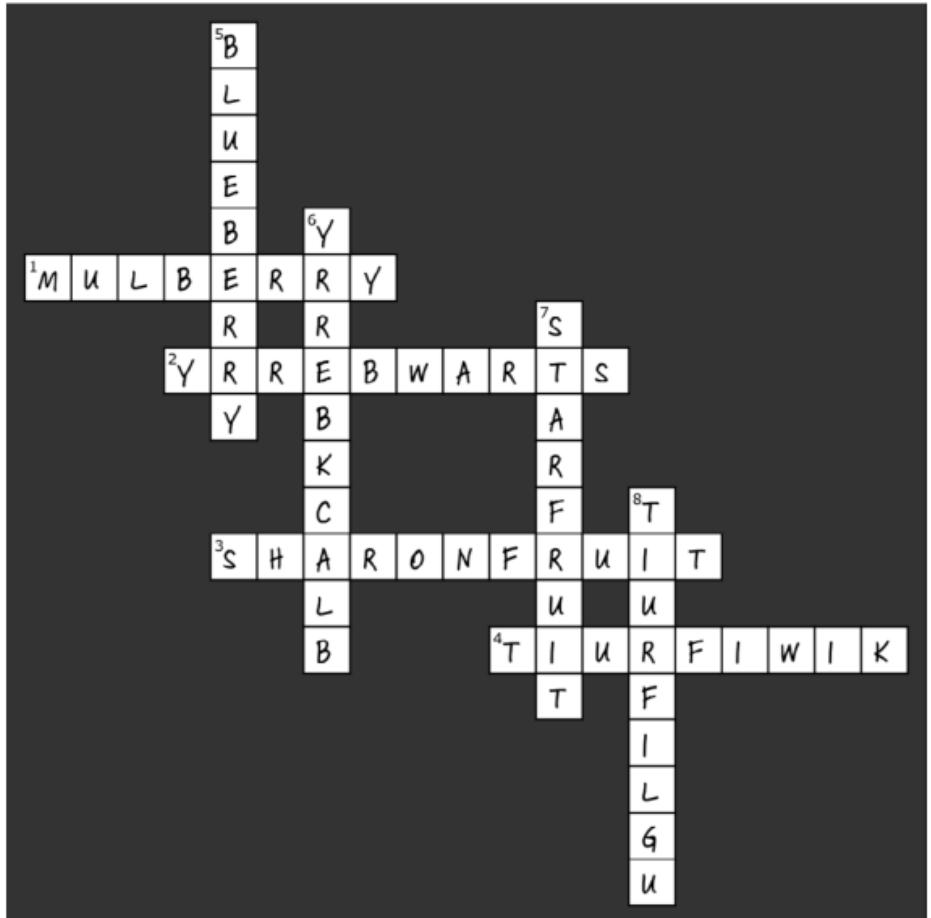
```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main()
5 {
6     char *juices[] =
7     {
8         "dragonfruit", "waterberry", "sharonfruit", "uglifruit", "rumberry", "kiwifruit",
9         "mulberry", "strawberry", "blueberry", "blackberry", "starfruit"
10    };
11
12     char *a;
13     puts(juices[6]);
14     print_reverse(juices[7]);
15     a = juices[2];
16     juices[2] = juices[8];
17     juices[8] = a;
18     puts(juices[8]);
19     print_reverse(juices[(18+7)/5]);
20
21     puts(juices[2]);
22     print_reverse(juices[9]);
23     juices[1] = juices[3];
24     puts(juices[10]);
25     print_reverse(juices[1]);
26     return 0;
27
28
29 }
```

Line 21 => crossword down



C-Cross Solution

Now that the guys have the `print_reverse()` function working, they've used it to create a crossword. The answers are displayed by the output lines in the code.



Across

```
int main()
{
```



Your C Toolbox

You've got Chapter 2.5 under your belt, and now you've added strings to your toolbox.

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

The `string.h` header contains useful string functions.

`strstr(a, b)` will return the address of string `b` in string `a`.

`strcpy()` copies one string to another.

`strcmp()` compares two strings.

`strcat()` concatenates two strings.

An array of strings is an array of arrays.

You create an array of arrays using `char strings [...][][]`

`strchr()` finds the location of a character inside a string.

`strlen()` finds the length of a string.

Pointers in C++ first

Data type is a set of values together with a set of operations.

Recall that the set of values is called the domain of the data type.

To manipulate numeric integer data in the range -2147483648 to 2147483647, you

can declare variables using the word `int`.

The name of the data type allows you to declare a variable of that type.

Next, we describe the pointer data type.

The values belonging to pointer data types are the memory addresses of your computer.

Because the domain—that is, the set of values of a pointer data type—is the addresses (locations) in memory, a pointer variable is a variable whose content is an address, that is, a memory location and the pointer variable is said to point to that memory location.

DECLARING POINTER VARIABLES

As remarked previously, there is no name associated with pointer data types.

Moreover, pointer variables store memory addresses.

So the obvious question is: If no name is associated with a pointer data type, how do you declare pointer variables?

The value of a pointer variable is an address or memory space that typically contains some data.

Therefore, when you **declare a pointer variable**, you also **specify the data type** of the value to be stored in the memory location pointed to by the pointer variable.

For example, if a pointer variable contains the address of a memory location containing an `int` value, it is said to be an `int` pointer or a pointer (variable) of type `int`.

As with regular variables, pointers are bound to a data type and they can only contain the addresses of (or point to) variables of the specific data type they were created to hold.

In *C++*, you declare a pointer variable by using the asterisk symbol (*) between the

data type and the variable name.

The general syntax to declare a pointer variable is:

```
dataType * identifier;
```

As an example, consider the following statements:

```
int *p;  
char *ch;
```

In these statements, both p and ch are pointer variables.

The content of p (when properly assigned) points to a memory location of type int, and the content of ch points to a memory location of type char.

So, p is a pointer variable of type int, and ch is a pointer variable of type char.

Before discussing how pointers work, let us make the following observations. The statement:

```
int *p;
```

```
int *p;      ||      int * p;      ||      int* p; //are similar
```

Thus, the character * can appear anywhere between the data type name and the variable name.

Now, consider the following statements:

```
int *p, q;
```

p is a pointer to an integer data stored in some memory location , q is a variable that will hold an integer.

```
int *p,*q;
```

declares both p and q to be pointer variables of type int.

Now that you know how to declare pointers, next we will discuss how to make a pointer point to a memory space and how to manipulate the data stored in these memory locations.

Because the value of a pointer is a memory address, a pointer can store the address of a memory space of the designated type.

For example, if p is a pointer of type int, p can store the address of any memory space of type int.

C++ provides two operators—the **address of operator (&)** and the **dereferencing operator (*)**—to work with pointers.

The next two sections describe these operators.

ADDRESS OF AN OPERATOR(&)

In C++, the ampersand, &, called the address of operator, is a unary operator that returns the address of its operand. For example, given the statements:

In C, the & operator is called the "address-of" operator. It is used to obtain the memory address of a variable.

```
int x = 5;  
int *ptr = &x; // ptr now points to the memory address of x
```

*ptr holds the data in the address held by &x.

```
int x;  
int *p;  
p = &x;
```

Assigns the address of x to p.

That is, x and the value of p refer to the same memory location.

Let's compare names and pointers.

In the following two declarations, num1 and num2 are int variables and numptr is the name of an int pointer:

```
int num1, num2;  
int *numptr;
```

The name num1 now stands in place of an int memory location and can be assigned the value of any integer between -2147483648 and 2147483647.

For example, the following statement stores 100 in num1:

```
num1 = 100;
```

The value held at the memory location named num1 can easily be copied directly to another integer memory location by using the assignment operator.

For instance, if we want to copy the integer 100 at the memory location num1 to the memory location num2, we can use this statement:

The name num1 in a sense means "**the value held in the memory location named num1.**"

This value is copied into the memory location named num2.

On the other hand, when we use the statement:

```
int num1, num2;  
int *numptr;
```

```
num1 = 100;  
num2 = num1;  
numptr = &num1;
```

We are asking the program to copy the address of the memory location of num1 to numptr, not the integer value it is holding.

The pointer numptr will not contain 100, but the actual address num1 has been assigned by the operating system.

In a sense, since the name num1 stands for the memory address of num1's value, and numptr contains the address of num1, numptr is really holding the name num1.

This makes numptr a reference to num1.

Now, if we output both num1 and numptr using the statements.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6
7     int num1, num2;
8     int *numptr;
9
10    num1 = 100;
11    num2 = num1;
12    numptr = &num1;
13
14    printf("Num1 printed is : %i\n", num1); //prints number
15    printf("numptr printed is : %i\n", numptr); //hexadecimal
16
17 }
18
19 "C:\Users\Nixon\Downloads\C Code\41 Pointers1\bin\Debug\41 Pointers1.exe"
20 Num1 printed is : 100
21 numptr printed is : 6422028
22 Process returned 0 (0x0) execution time : 0.219 s
23 Press any key to continue.

```

What if i want to print out the contents of the *numptr??

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int num1, num2;
7     int *numptr;
8
9     num1 = 100;
10    num2 = 200;
11    numptr = &num1; //hold the data in that address
12
13    printf("The first number is: %d\n", num1);
14    printf("The pointer to first number is: %d\n", *numptr); //don't forget the *
15    return 0;
16
17 }
18

```

When you use the ***** operator with numptr in the printf statement, you are **dereferencing the pointer**, which means that you are accessing the value stored at the memory address that numptr points to. So, *numptr gives you the value of num1, which is 100 in this case.

In the next section, after discussing the dereference operator, we will explain how to output the value of the memory location whose address is stored in `numptr`.

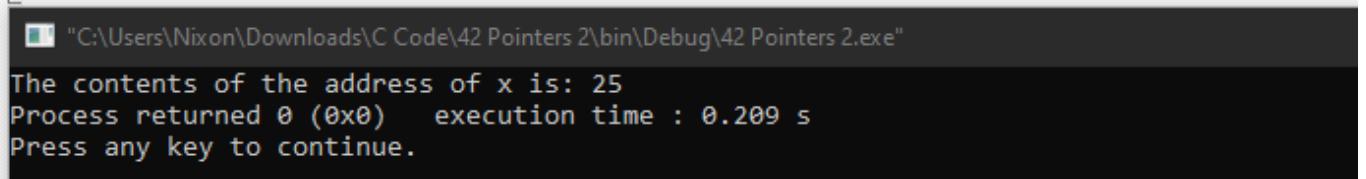
DEREFERENCING OPERATOR(*)

Every chapter until now has used the asterisk character, `*`, as the binary multiplication operator.

C/C++ also uses `*` as a unary operator.

When used as a unary operator, `*`, commonly referred to as the dereferencing operator or indirection operator, refers to the object to which its operand (that is, the pointer) points.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int x = 25;
7      int *p;
8      p = &x; //store address of x in p
9      printf("The contents of the address of x is: %i", *p);
10
11 }
```


"C:\Users\ Nixon\ Downloads\ C Code\ 42 Pointers 2\ bin\ Debug\ 42 Pointers 2.exe"
The contents of the address of x is: 25
Process returned 0 (0x0) execution time : 0.209 s
Press any key to continue.

At this point, `p` contains the memory address of `x`. This is called "referencing" `x` because `p` now refers to the memory location where `x` is stored.

Prints the value stored in the memory space pointed to by `p`, which is the value of `x`.

Also, the statement:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int x = 25;
7     int *p;
8     p = &x; //store address of x in p
9     printf("The contents of the address of x is: %i\n", *p);
10    *p = 55; //store 55 in the location pointed to by p
11    printf("The contents of the address of x is: %i\n", *p);
12
13
14 }
15
```

"C:\Users\Nixon\Downloads\C Code\42 Pointers 2\bin\Debug\42 Pointers 2.exe"

```
The contents of the address of x is: 25
The contents of the address of x is: 55
```

EXAMPLE 2:

In these statements, p is a pointer variable of type int, and num is a variable of type int.

Let us assume that memory location 1200 is allocated for p, and memory location 1800 is allocated for num.

```
int *p;
int num;
```

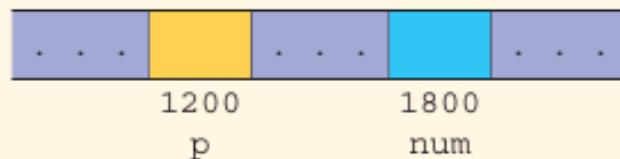
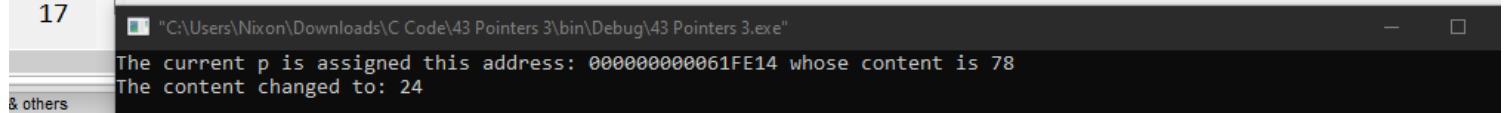


FIGURE 12-1 Variables p and num

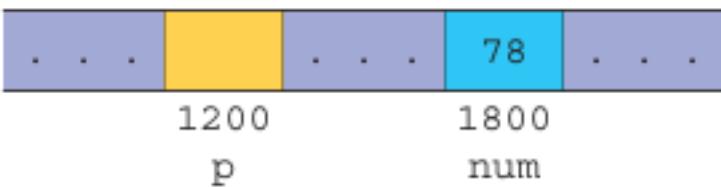
```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6
7      int *p;
8      int num;
9
10     num = 78; //store a value into this box
11     p = &num; //assign address of num to p
12     printf("The current p is assigned this address: %p whose content is %i\n", &num, num);
13     *p = 24; //change the content of what's in that address p is pointing to
14     printf("The content changed to: %i\n", *p);
15
16 }
17

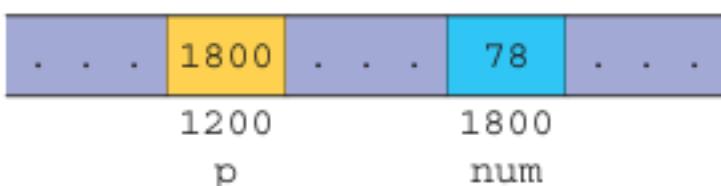
```



The statement `num = 78;` stores 78 into num.



The statement `p = #` stores the address of num, which is 1800, into p.



The statement `p = #` assigns the memory address of num to the pointer variable p. This means that p now points to the memory location where num is stored.

If you were to write `*p = num;` instead, you would be dereferencing the pointer variable p and attempting to assign the value of num to the memory location that p points to. However, since p has not been initialized with a valid memory address, dereferencing it could result in undefined behavior.

If you write `*p = #` instead of `p = #`, you will get a compilation error because the type of `&num` is a pointer to an integer (`int *`), and you are trying to assign it to an integer variable through the dereference operator (`*`).

The expression `*p` represents the value stored in the memory location pointed to by the pointer variable `p`. So, when you write `*p = &num`, you are trying to assign the memory address of `num` (which has type `int *`) to the value stored in the memory location pointed to by `p` (which has type `int`). This is not valid because you are trying to store a pointer value in an integer variable.

To assign the memory address of `num` to `p`, you should use the address-of operator (`&`) to get the memory address of `num`, and assign it directly to the pointer variable `p`, like this:

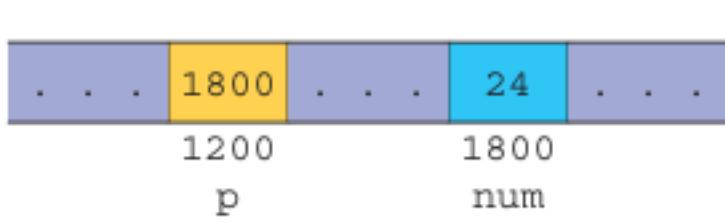
```
int *p;  
int num;  
  
num = 80;  
p = &num; // assign the memory address of num to p
```

This will assign the memory address of `num` to `p`, so that `p` points to the memory location where `num` is stored.

The statement `*p = 24;` stores 24 into the memory location to which `p` points.

Because the value of `p` is 1800, statement 3 stores 24 into memory location 1800.

Note that the value of `num` is also changed.



Let us summarize the preceding discussion.

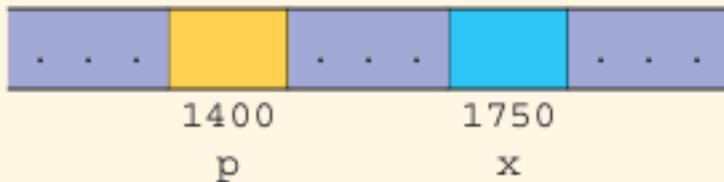
- A declaration such as `int *p;` allocates memory for `p` only, not for `*p`.

- Later, you will learn how to allocate memory for `*p`.
- The content of `p` points only to a memory location of type `int`.
- `&p`, `p`, and `*p` all have different meanings.
- `&p` means the address of `p`—that is, 1200.
- `p` means the content of `p`, which is 1800, after the statement `p = #` executes.
- `*p` means the content of the memory location to which `p` points.
- Note that after the statement `p = #` executes, the value of `*p` is 78, after the statement `*p = 24;` executes, the value of `*p` is 24.

EXAMPLE 3:

```
int *p;
int x;
```

Suppose that we have the memory allocation for `p` and `x` as shown:



The values of `&p`, `p`, `*p`, `&x`, and `x` are as follows:

<code>&p</code>	<code>1400</code>	<code>p</code>	<code>? (unknown)</code>	<code>*p</code>	<code>Does not exist (undefined)</code>	<code>&x</code>	<code>1750</code>	<code>x</code>	<code>? (unknown)</code>
---------------------	-------------------	----------------	--------------------------	-----------------	---	---------------------	-------------------	----------------	--------------------------

Suppose that the following statements are executed in the order given:

```
void pointers()
{
    int *p;
    int x;
    x = 50;
    p = &x;
    *p = 38;
}
```

The values of `&p`, `p`, `*p`, `&x`, and `x` are shown after each of these statements executes.

After the statement `x = 50;` executes, the values of `&p`, `p`, `*p`, `&x`, and `x` are as follows:

<code>&p</code>	1400	<code>p</code>	? (unknown)	<code>*p</code>	Does not exist (undefined)	<code>&x</code>	1750	<code>x</code>	50
---------------------	------	----------------	-------------	-----------------	----------------------------	---------------------	------	----------------	----

After the statement `p = &x;` executes, the values of `&p`, `p`, `*p`, `&x`, and `x` are as follows:

<code>&p</code>	1400	<code>p</code>	1750	<code>*p</code>	50	<code>&x</code>	1750	<code>x</code>	50
---------------------	------	----------------	------	-----------------	----	---------------------	------	----------------	----

After the statement `*p = 38;` executes, the values of `&p`, `p`, `*p`, `&x`, and `x` are as follows.

Because `*p` and `x` refer to the same memory space, the value of `x` is also changed to 38.

`&p` 1400

`p` 1750

`*p` 38

`&x` 1750 `x` 38

Let us note the following:

- `p` is a pointer variable.
- The content of `p` points only to a memory location of type int.
- Memory location `x` exists and is of type int.
- Therefore, the assignment statement: `p = &x;` is legal.
- After this assignment statement executes, `*p` is valid and meaningful.

FUNCTIONS AND POINTERS

A pointer variable can be passed as a parameter to a function either by value or by reference.

To declare a **pointer as a value** parameter in a function heading, you use the same mechanism as you use to declare a variable.

To make a formal parameter be a **reference parameter**, you use `&` when you declare the formal parameter in the function heading.

Therefore, to declare a formal parameter as a reference pointer parameter, between the data type name and the identifier name, you must include `*` to make the identifier a pointer and `&` to make it a reference parameter.

The obvious question is: In what order should `&` and `*` appear between the data type name and the identifier to declare a pointer as a reference parameter?

In C/C++, to make a pointer a reference parameter in a function heading, * appears before the & between the data type name and the identifier.

The following example illustrates this concept:

```
void pointersParameters(int * &p , double *q)
{
    .
    .
    .
}
```

In the function pointerParameters, both p and q are pointers.

The parameter p is a **reference parameter**; the parameter q is a **value parameter**.

Furthermore, the function pointerParameters can change the value of *q, but not the value of q.

However, the function pointerParameters can change the value of both p and *p.

Pointers and Function Return Values

In C++, the return type of a function can be a pointer.

For example, the return type of the function, is a pointer type int.

```
int* testExp(....)
{
    .
    .
    .
}
```

Small Tools in C

GPS CODE

The passage discusses the concept of small tools, which are C programs that perform a single task effectively. These tasks might include displaying file contents, listing running processes, or printing files.

The author suggests that using small tools can help solve big problems by breaking them down into a series of smaller tasks.

Many operating systems come with small tools that can be run from the command prompt or terminal. Overall, the passage emphasizes the importance of simple and focused programming to solve complex problems.

Someone's written me a map web application, and I'd love to publish my route data with it. Trouble is, the format of the data coming from my GPS is wrong.

This is the data from the cyclist's GPS. It's a comma-separated format.

This is the data format the map needs. It's in JavaScript Object Notation, or JSON.

Someone's written me a map web application, and I'd love to publish my route data with it. Trouble is, the format of the data coming from my GPS is wrong.



This is the data from the cyclist's GPS. It's a comma-separated format.

This is a latitude.

This is a longitude.

```
42.363400,-71.098465,Speed = 21  
42.363327,-71.097588,Speed = 23  
42.363255,-71.096710,Speed = 17
```

This is the data format the map needs. It's in JavaScript Object Notation, or JSON.

```
data=[  
  {latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},  
  {latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},  
  {latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},  
  ...  
]
```

The data's the same, but the format's a little different.

If one small part of your program needs to convert data from one format to another, that's the perfect kind of task for a small tool.

Hey, who hasn't taken a code printout on a long ride only to find that it soon become unreadable? Sure, we all have. But with a little thought, you should be able to piece together the original version of some code. This program can read comma-separated data from the command line and then display it in JSON format. See if you can figure out what the missing code is.

CODE DESCRIBED:

char info[][] is a two-dimensional character array in C. The first dimension of the array info[] represents the number of rows, and the second dimension [80] represents the number of columns.

Each element in the array info is a character array of size 80. This means that info[0] is a character array of size 80, info[1] is a character array of size 80, and so

on.

You can use this array to store a table of strings, where each row represents a separate string. For example, you might use this to store information about a group of people, where each row represents a separate person and the columns contain information such as name, age, address, etc.

Here's an example of how you might initialize and use a two-dimensional character array:

```
char info[3][80] = { "John Smith, 35, 123 MainSt.", "JaeDe, 28, 456 Elm St.", "Doe, 27, 436 Elm St." };
```

In this example, we initialize `info` with three rows of data, each containing a string of up to 80 characters.

You can access individual elements of the array using two indices, like this:

```
printf("%s\n", info[0]); // prints "John Smith, 35, 123 Main St."
printf("%s\n", info[1]); // prints "Jane Doe, 28, 456 Elm St."
printf("%s\n", info[2]); // prints "Bob Johnson, 42, 789 Oak St."
```

Note that each element of the array is a null-terminated string, so you can use string functions like `strlen()` and `strcmp()` to manipulate the data.

GPS CONTINUED

`char info[80]` is a one-dimensional character array in C that has 80 elements. This means that `info` can hold a string of up to 80 characters. You can use this array to store a single string or a series of related strings, like a list of names, phone numbers, or addresses.

We're using `scanf()` to enter more than one piece of data.

```
#include <stdio.h>

int main()
{
    float latitude;
    float longitude;
    char info[80];
    int started = .....;

    puts("data=[");
    while (scanf("%f,%f,%79[^\\n]", ..... , ..... , ..... ) == 3) {
        if (started)
            printf(",\n");
        else
            started = .....; Be careful how you set "started."
        printf("{latitude: %f, longitude: %f, info: '%s'}", ..... , ..... , ..... );
    }
    puts("\n]");
    return 0;
}
```

We're using `scanf()` to enter more than one piece of data.

What will these values be? Remember: `scanf()` always uses pointers.

The `scanf()` function returns the number of values it was able to read.

This is just a way of saying, "Give me every character up to the end of the line."

What values need to be displayed?

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      float latitude;
7      float longitude;
8      char info[80];
9      int started = 0 ; //begin with "started" set to zero, meaning false
10
11     puts("data=[");
12     while(scanf("%f,%f,%79[^\\n]", &latitude , &longitude, info) = 3)
13     {
14         if(started)
15             printf(",\\n"); //display a comma only if you've already displayed a previous line
16         else
17             started = 1 ; //once the loop has started, you can set "started" to 1, meaning true
18
19         //you don't need & here because printf is using the values, not addresses
20         printf("{latitude: %f, longitude: %f, info: '%s'}", latitude , longitude ,info);
21     }
22
23     puts("\\n");
24     return 0;
25
26 }

```

This code reads input from the user in a specific format and then prints that data to the console in JSON format. Here's how it works:

The program **initializes several variables** - `latitude`, `longitude`, and `info` - to hold data input by the user, and `started` to keep track of whether we've already printed a line of output.

The program **prints the string "data=[** to the console, indicating the start of the JSON data.

The program **enters a loop that reads input from the user using the `scanf()` function**. The format string `%f,%f,%79[^\\n]` specifies that we expect three inputs from the user - two floats (latitude and longitude) and a string of up to 79 characters (info).

The format string uses `%[^\\n]` to read the string up to a newline character, which is discarded.

If `scanf()` successfully reads three inputs from the user (i.e., returns the value 3), the program enters the if statement.

If `started` is true (i.e., not zero), the program prints a comma and newline character to separate the previous line of output from the current line. Otherwise, the program sets `started` to true and does not print a comma.

The program prints the latitude, longitude, and info to the console in JSON format using the `printf()` function. The values of latitude, longitude, and info are used directly in the format string, without needing to use the address-of operator `&`.

Once the loop is finished, the program prints a newline character and the string "]" to indicate the end of the JSON data.

The program returns 0 to indicate that it completed successfully.

Overall, this code reads input from the user in a specific format and then prints that data to the console in a standardized format that can be easily processed by other programs.

Note the use of `==` instead of `=`. The `scanf` function returns the number of items it successfully read and assigned, so `==` is used to compare it with the expected value of 3.

So what happens when you compile and run this code? What will it do?

This is the data that's printed out.

This is the data you type in.

The input and the output are mixed up.

```
File Edit Window Help JSON
>./geo2json
data=[42.363400,-71.098465,Speed = 21
(latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21')42.363327,-71.097588,Speed = 23
(latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23')42.363255,-71.096710,Speed = 17
(latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17')42.363182,-71.095833,Speed = 22
,
...
...
(latitude: 42.363182, longitude: -71.095833, info: 'Speed = 22')42.362385,-71.086182,Speed = 21
(latitude: 42.362385, longitude: -71.086182, info: 'Speed = 21')^D
```

Several more hours' worth of typing...

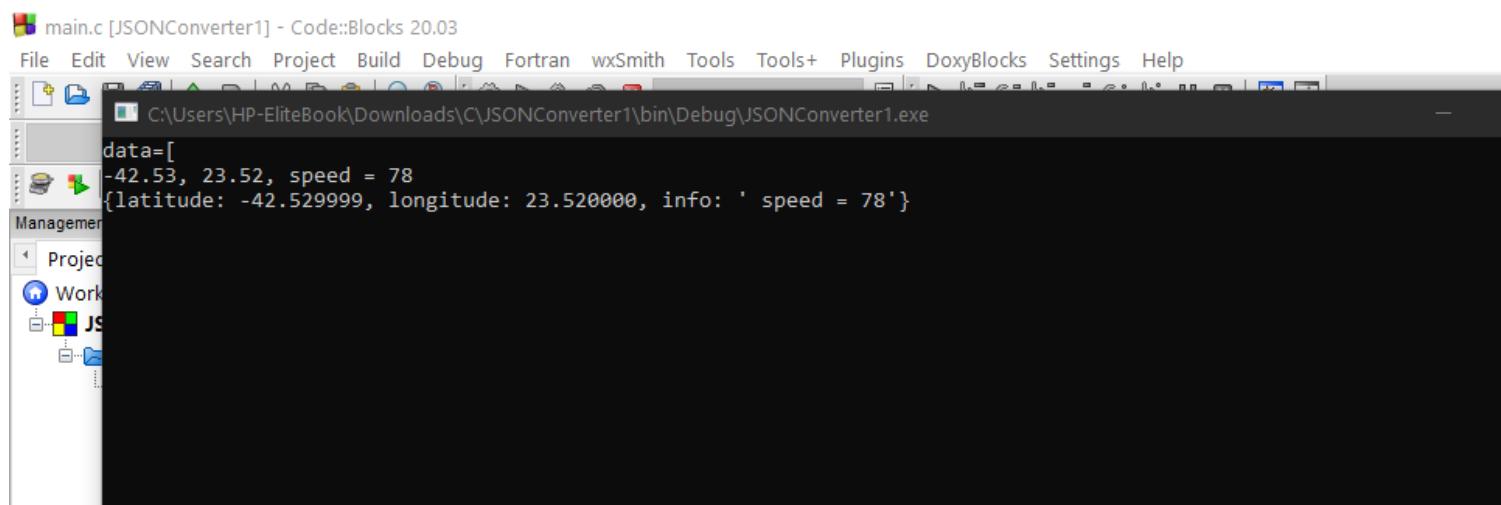
The program lets you enter GPS data at the keyboard and then it

In the end, you need to press `Ctrl-D` just to stop the program.

The program lets you enter GPS data at the keyboard and then it displays the

JSON-formatted data on the screen. Problem is, the input and the output are all mixed up together. Also, there's a lot of data.

If you are writing a small tool, you don't want to type in the data; you want to get large amounts of data by reading a file. Also, how is the JSON data going to be used? Surely it can't be much use on the screen? So is the program running OK? Is it doing the right thing? Do you need to change the code?



main.c [JSONConverter1] - Code::Blocks 20.03

File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help

C:\Users\HP-EliteBook\Downloads\C\JSONConverter1\bin\Debug\JSONConverter1.exe

```
data=[  
-42.53, 23.52, speed = 78  
{latitude: -42.529999, longitude: 23.520000, info: ' speed = 78'}
```

Management

Project

Work

JS

We really don't want the output on the screen. We need it in a file so we can use it with the mapping application. Here, let me show you...



Here's how the program should work:

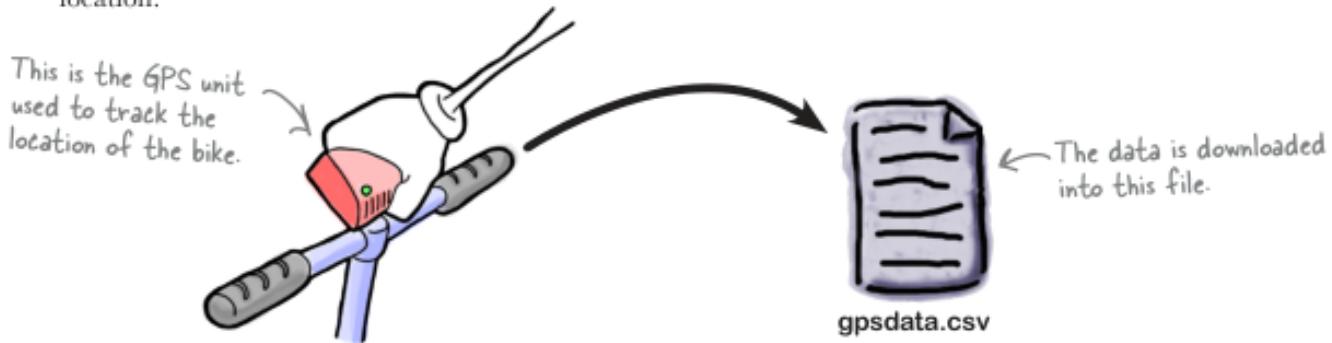
Take the GPS from the bike and download the data. It creates a file called `gpsdata.CSV` with one line of data for every location.

The `geo2json` tool needs to read the contents of the `gpsdata.CSV` line by line, and then write that data in JSON format into a file called `output.JSON`.

Here's how the program should work

1 Take the GPS from the bike and download the data.

It creates a file called `gpsdata.csv` with one line of data for every location.



2 The `geo2json` tool needs to read the contents of the `gpsdata.csv` line by line...

This is our `geo2json` tool.

`geo2json`

3 ...and then write that data in JSON format into a file called `output.json`.

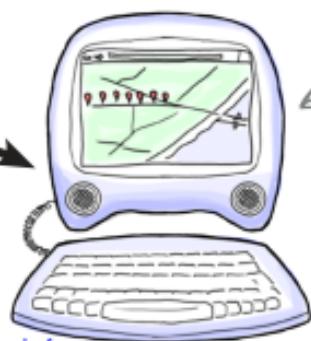
Writing this file.

Your tool will write data to this file.

`output.json`

4 The web page that contains the map application reads the `output.json` file. It displays all of the locations on the map.

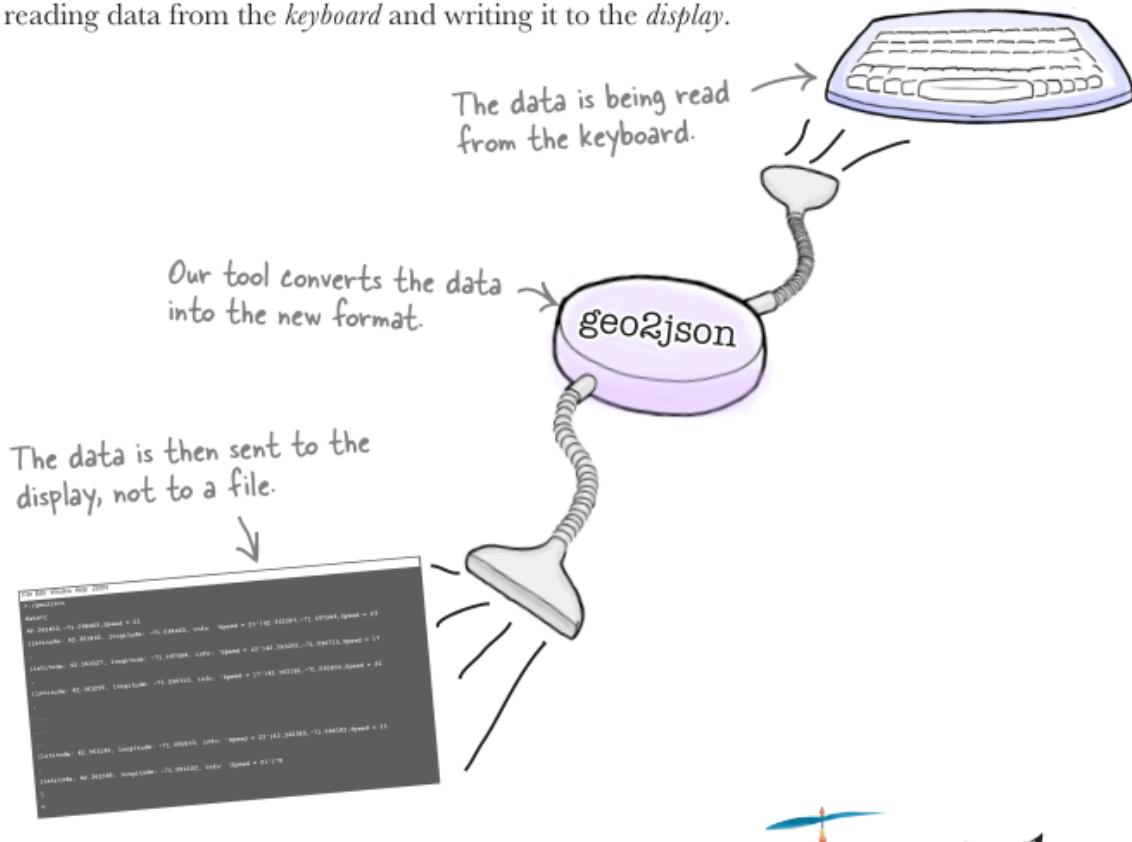
The mapping application reads the data from `output.json` and displays it on a map inside a web page.



But how do you do that? If you want to use files instead of the keyboard and the display, what code will you have to change? Will you have to change any code at all?

But you're not using files...

The problem is, instead of reading and writing files, your program is currently reading data from the *keyboard* and writing it to the *display*.



But you're not using file. The problem is, instead of reading and writing files, your program is currently reading data from the keyboard and writing it to the display.

But that isn't good enough. The user won't want to type in all of the data if it's already stored in a file somewhere. And if the data in JSON format is just displayed on the screen, there's no way the map within the web page will be able to read it. You need to make the program work with files.

Tools that read data line by line, process it, and write it out again are called **filters**. If you have a Unix machine, or you've installed **Cygwin on Windows**, you already have a few filter tools installed.

Head: This tool displays the first few lines of a file.

Tail: This filter displays the lines at the end of a file.

Sed: The stream editor lets you do things like search and replace text.

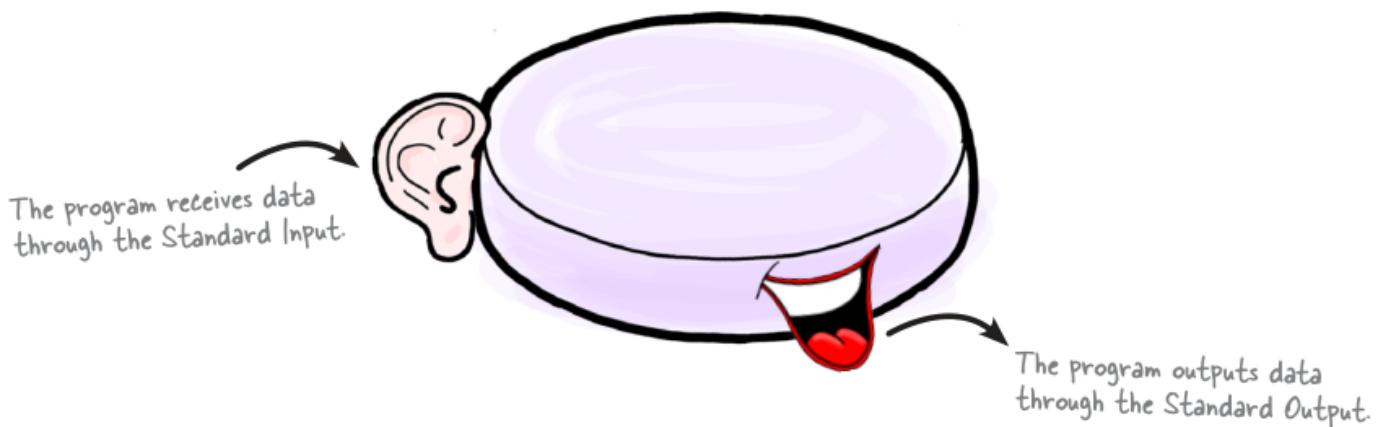
You'll see later how to combine filters together to form **filter chains**.

REDIRECTION

Is there a way of making our program use files without changing code? Without even recompiling it?

You can use redirection. You're using `scanf()` and `printf()` to read from the keyboard and write to the display. But the truth is, they don't talk directly to the keyboard and display.

Instead, they use the Standard Input and Standard Output. The Standard Input and Standard Output are created by the operating system when the program runs.



The operating system controls how data gets into and out of the Standard Input and Output. If you run a program from the command prompt or terminal, the operating system will send all of the keystrokes from the keyboard into the Standard Input.

If the operating system reads any data from the Standard Output, by default it will send that data to the display. The `scanf()` and `printf()` functions don't know, or care, where the data comes from or goes to. They just read and write Standard Input and the Standard Output.

Now this might sound like it's kind of complicated. After all, why not just have your program talk directly to the keyboard and screen? Wouldn't that be simpler?

Well, there's a very good reason why operating systems communicate with programs using the Standard Input and the Standard Output: **You can redirect the Standard Input and Standard Output** so that they read and write data somewhere else, such as to and from files.

REDIRECT STANDARD INPUT WITH <

You can redirect the Standard Input with < Instead of entering data at the keyboard, you can use the < operator to read the data from a file.

```
42.363400,-71.098465,Speed = 21
42.363327,-71.097588,Speed = 23
42.363255,-71.096710,Speed = 17
42.363182,-71.095833,Speed = 22
42.363110,-71.094955,Speed = 14
42.363037,-71.094078,Speed = 16
42.362965,-71.093201,Speed = 18
42.362892,-71.092323,Speed = 22
42.362820,-71.091446,Speed = 17
42.362747,-71.090569,Speed = 23
42.362675,-71.089691,Speed = 14
```

← This is the file containing the data from the GPS device.

This is telling the operating system to send the data from the file into the Standard Input of the program. You don't have to type in the GPS data, so you don't see it mixed up with the output. Now you just see the JSON data coming from the program.

```
42.362602,-71.088814,Speed = 19  
42.362530,-71.087936,Speed = 16  
42.362457,-71.087059,Speed = 16  
42.362385,-71.086182,Speed = 21
```



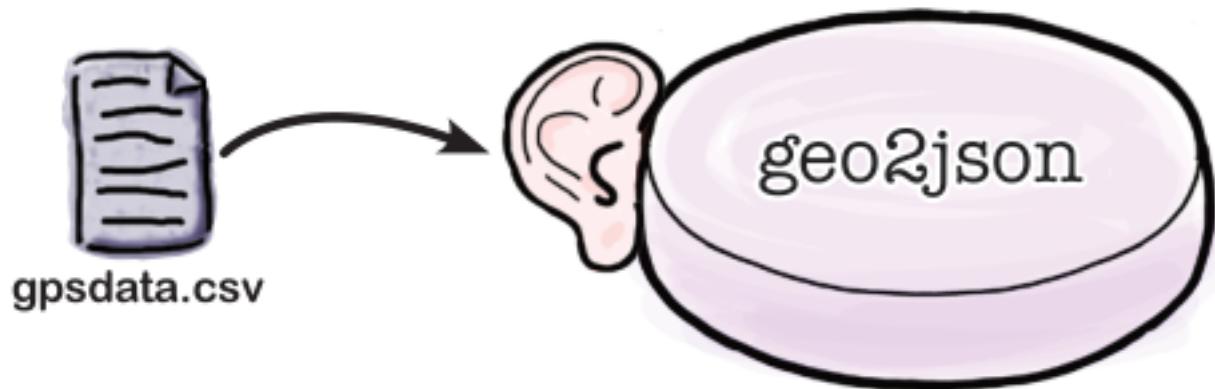
This is telling the operating system to send the data from the file into the Standard Input of the program.

You don't have to type in the GPS data, so you don't see it mixed up with the output.

```
File Edit Window Help Don'tCrossTheStreams  
> ./geo2json < gpsdata.csv  
data=[  
  {latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},  
  {latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},  
  {latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},  
  {latitude: 42.363182, longitude: -71.095833, info: 'Speed = 22'},  
  {latitude: 42.363110, longitude: -71.094955, info: 'Speed = 14'},  
  {latitude: 42.363037, longitude: -71.094078, info: 'Speed = 16'},  
  ...  
  ...  
  {latitude: 42.362385, longitude: -71.086182, info: 'Speed = 21'}  
>
```

Now you just see the JSON data coming from the program.

The **< operator** tells the operating system that the Standard Input of the program should be connected to the `gps-data.csv` file instead of the keyboard. So you can send the program data from a file. Now you just need to redirect its output.



REDIRECT STANDARD OUTPUT WITH >

Now you are redirecting both the Standard Input and the Standard Output.

Now you are redirecting both the Standard Input and the Standard Output.

```
File Edit Window Help Don CrossTheStreams
> ./geo2json < gpsdata.csv > output.json
>
```

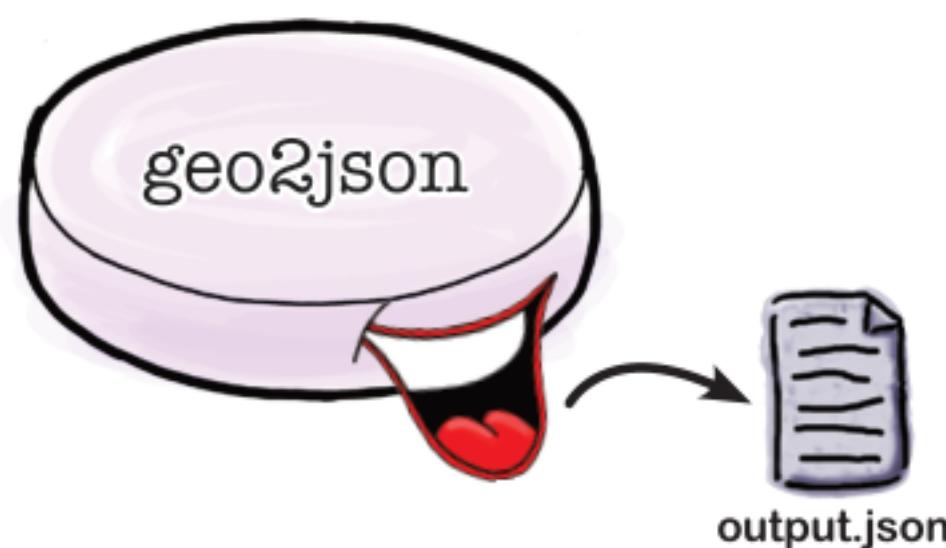
The output of the program will now be written to output.json.

There's no output on the display at all; it's all gone to the output.json file.

```
data=[  
  {latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},  
  {latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},  
  {latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},  
  {latitude: 42.363182, longitude: -71.095833, info: 'Speed = 22'},  
  {latitude: 42.363110, longitude: -71.094955, info: 'Speed = 14'},  
  {latitude: 42.363037, longitude: -71.094078, info: 'Speed = 16'},  
  {latitude: 42.362965, longitude: -71.093201, info: 'Speed = 18'},  
  {latitude: 42.362892, longitude: -71.092323, info: 'Speed = 22'},  
  {latitude: 42.362820, longitude: -71.091446, info: 'Speed = 17'},  
  {latitude: 42.362747, longitude: -71.090569, info: 'Speed = 23'},  
  {latitude: 42.362675, longitude: -71.089691, info: 'Speed = 14'},  
  {latitude: 42.362602, longitude: -71.088814, info: 'Speed = 19'},  
  {latitude: 42.362530, longitude: -71.087936, info: 'Speed = 16'},  
  {latitude: 42.362457, longitude: -71.087059, info: 'Speed = 16'},  
  {latitude: 42.362385, longitude: -71.086182, info: 'Speed = 21'}  
]
```



Because you've redirected the Standard Output, you don't see any data appearing on the screen at all. But the program has now created a file called output.json. The output.json file is the one you needed to create for the mapping application. Let's see if it works.



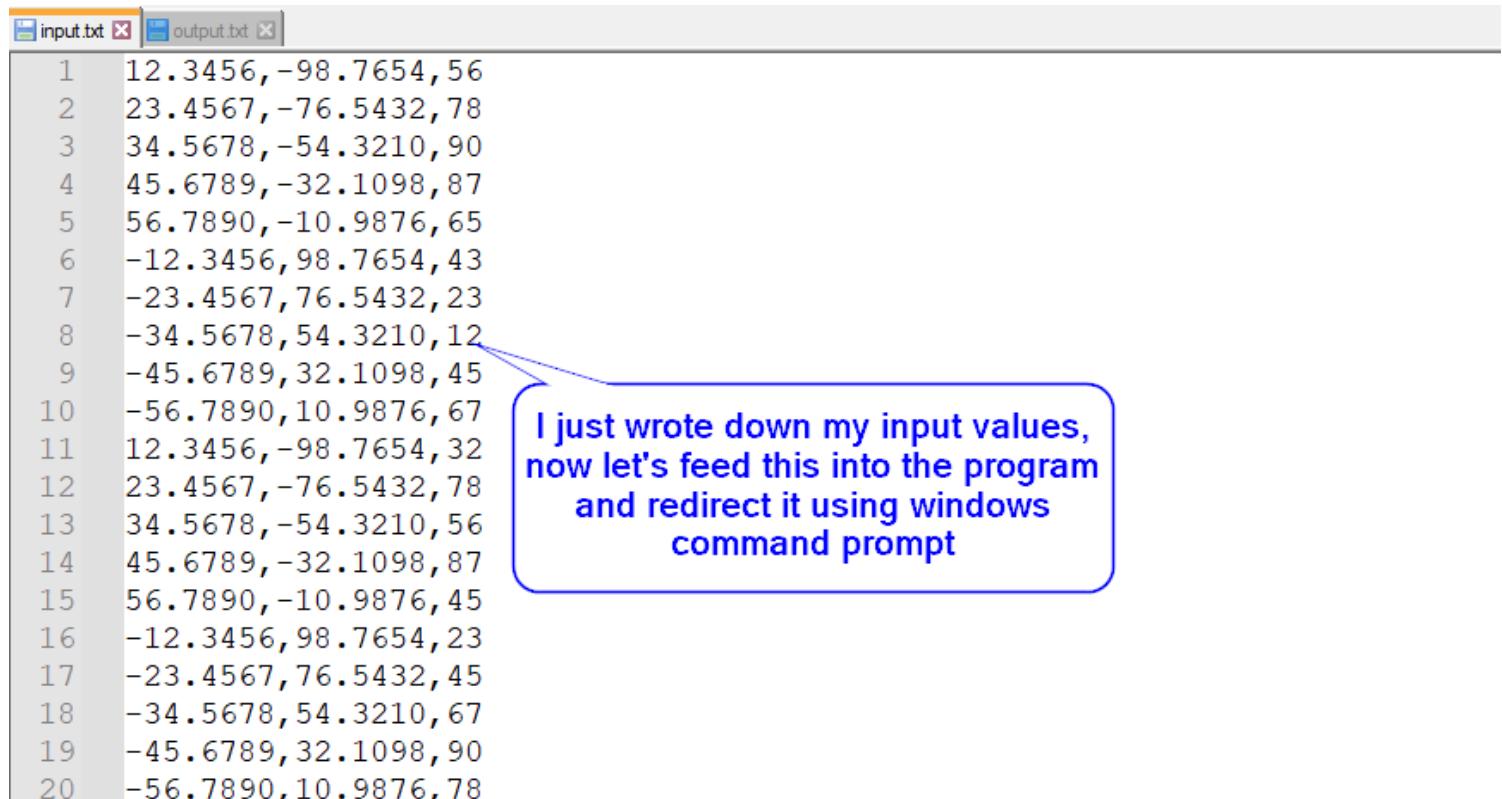
Now it's time to see if the new data file you've created can be used to plot the location data on a map. You'll take a copy of the web page containing the mapping program and put it into the same folder as the output.json file. Then you need to open the web page in a browser:

For windows users:

To use this modified code with the input and output redirection, you would run the command,

```
JSONConverter1.exe < input.txt > output.json
```

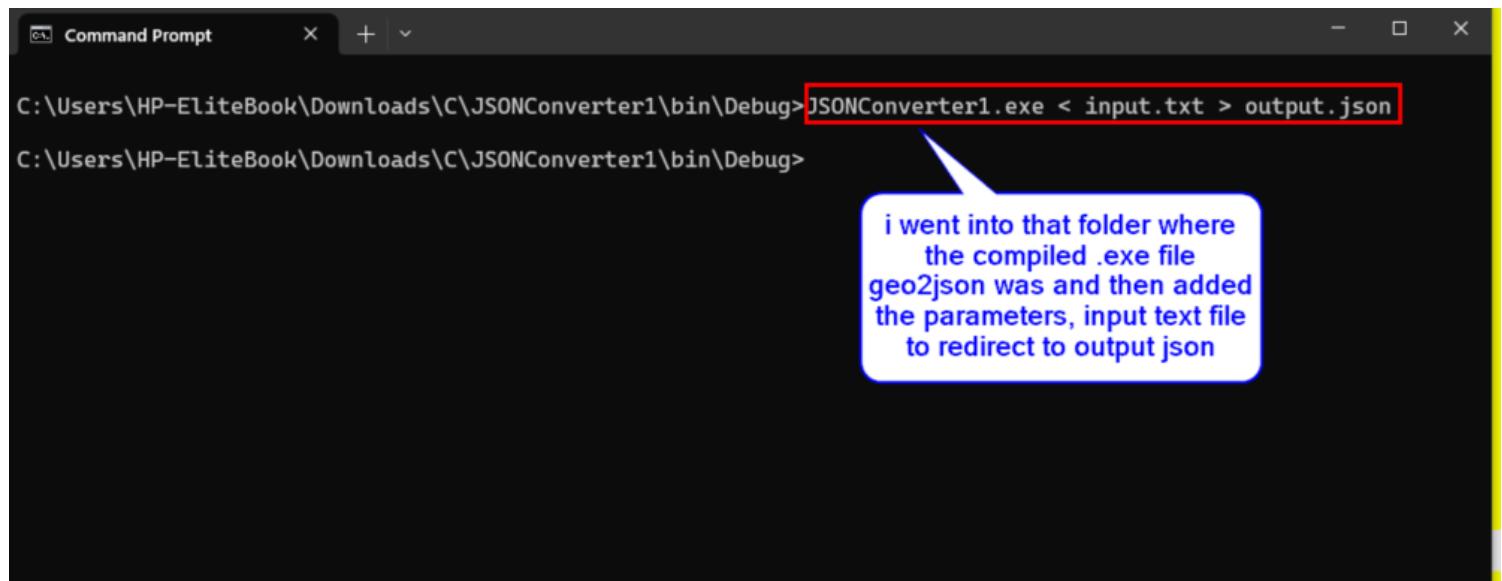
In the terminal while in the directory where the compiled JSONConverter1 executable is located. This will read the contents of gpsdata.csv and write the output to output.json.



input.txt output.txt

```
1 12.3456,-98.7654,56
2 23.4567,-76.5432,78
3 34.5678,-54.3210,90
4 45.6789,-32.1098,87
5 56.7890,-10.9876,65
6 -12.3456,98.7654,43
7 -23.4567,76.5432,23
8 -34.5678,54.3210,12
9 -45.6789,32.1098,45
10 -56.7890,10.9876,67
11 12.3456,-98.7654,32
12 23.4567,-76.5432,78
13 34.5678,-54.3210,56
14 45.6789,-32.1098,87
15 56.7890,-10.9876,45
16 -12.3456,98.7654,23
17 -23.4567,76.5432,45
18 -34.5678,54.3210,67
19 -45.6789,32.1098,90
20 -56.7890,10.9876,78
```

I just wrote down my input values, now let's feed this into the program and redirect it using windows command prompt



```
C:\Users\HP-EliteBook\Downloads\C\JSONConverter1\bin\Debug>JSONConverter1.exe < input.txt > output.json
```

The json file generated is :

Name	Date modified	Type	Size
input.txt	3/27/2023 2:52 PM	TXT File	1 KB
JSONConverter1.exe	3/27/2023 2:03 PM	Application	55 KB
output.json	3/27/2023 2:54 PM	JSON Source File	2 KB

The code for generating the JSON file is missing a closing bracket. You can add the closing bracket after the while loop ends to properly close the JSON object. Here is the modified code with the closing bracket added:

```
started = 1; //once the loop has started, you can  
//you don't need & here because printf is using the val  
printf("{latitude: %f, longitude: %f, info: '%s'}", la  
}  
  
puts("\n ]\n"); ←  
return 0;
```

The final program should be this, if you want to include the {} braces.

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

int main()
{
    float latitude;
    float longitude;
    char info[80];
    int started = 0;

    puts("{");
    puts("  \"data\": [");

    while (scanf("%f,%f,%79[^\\n]", &latitude, &longitude, info) == 3)
    {
        if (started)
            printf(",\\n");
        else
            started = 1;

        printf("      \\n");
        printf("      \"latitude\": %f,\\n", latitude);
        printf("      \"longitude\": %f,\\n", longitude);
        printf("      \"info\": \"%s\\n\", info");
        printf("    }");
    }

    puts("\\n  ]");
    puts("}");

    return 0;
}
```

Or you can do it like me, and just use the [] braces.

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

int main()
{
    float latitude;
    float longitude;
    char info[80];
    int started = 0; //begin with "started" set to zero, meaning false
```

```
puts("data=[");
while(scanf("%f,%f,%79[^\\n]",&latitude, &longitude ,info) == 3) //  
equals not assingment
{
    if(started)
        printf(",\\n");
        //display a comma only if you've already displayed a
previous line

    else
        started = 1 ;
        //once the loop has started, you can set "started to 1,
meaning true

        //you don't need & here because printf is using the values, not
addresses
        printf("{latitude: %f, longitude: %f, info: '%s'}",
latitude , longitude ,info);
    }

puts("\\n  ]\\n");
return 0;
}
```

And the final output should be:

```
data=[  
  {latitude: 12.345600, longitude: -98.765404, info: '56'},  
  {latitude: 23.456699, longitude: -76.543198, info: '78'},  
  {latitude: 34.567799, longitude: -54.320999, info: '90'},  
  {latitude: 45.678902, longitude: -32.109798, info: '87'},  
  {latitude: 56.789001, longitude: -10.987600, info: '65'},  
  {latitude: -12.345600, longitude: 98.765404, info: '43'},  
  {latitude: -23.456699, longitude: 76.543198, info: '23'},  
  {latitude: -34.567799, longitude: 54.320999, info: '12'},  
  {latitude: -45.678902, longitude: 32.109798, info: '45'},  
  {latitude: -56.789001, longitude: 10.987600, info: '67'},  
  {latitude: 12.345600, longitude: -98.765404, info: '32'},  
  {latitude: 23.456699, longitude: -76.543198, info: '78'},  
  {latitude: 34.567799, longitude: -54.320999, info: '56'},  
  {latitude: 45.678902, longitude: -32.109798, info: '87'},  
  {latitude: 56.789001, longitude: -10.987600, info: '45'},  
  {latitude: -12.345600, longitude: 98.765404, info: '23'},  
  {latitude: -23.456699, longitude: 76.543198, info: '45'},  
  {latitude: -34.567799, longitude: 54.320999, info: '67'},  
  {latitude: -45.678902, longitude: 32.109798, info: '90'},  
  {latitude: -56.789001, longitude: 10.987600, info: '78'}  
]
```

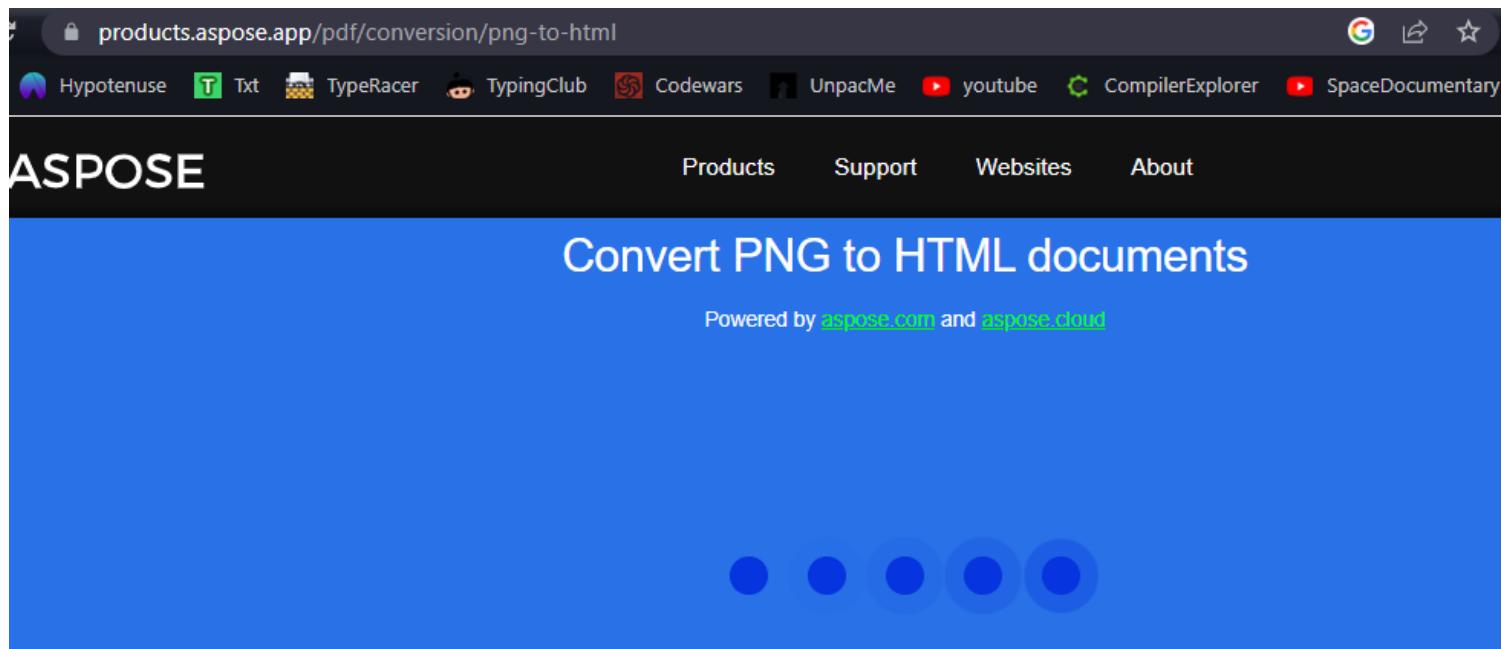
Now it's time to see if the new data file you've created can be used to plot the location data on a map. You'll take a copy of the web page containing the mapping program and put it into the same folder as the output.json file. Then you need to open the web page in a browser:

Get a screenshot of a place you want to use:

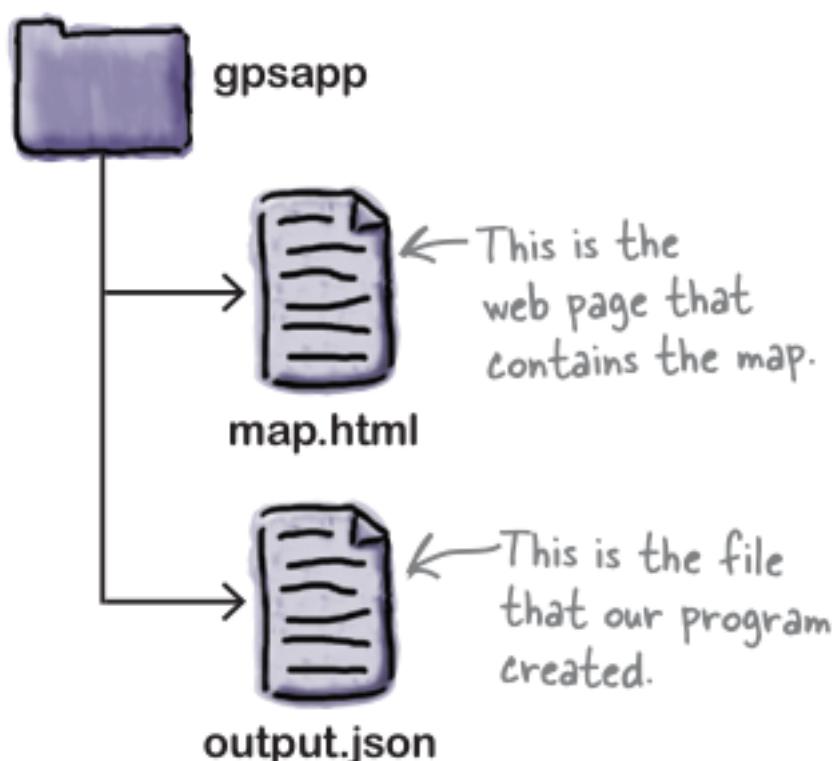
Google Earth

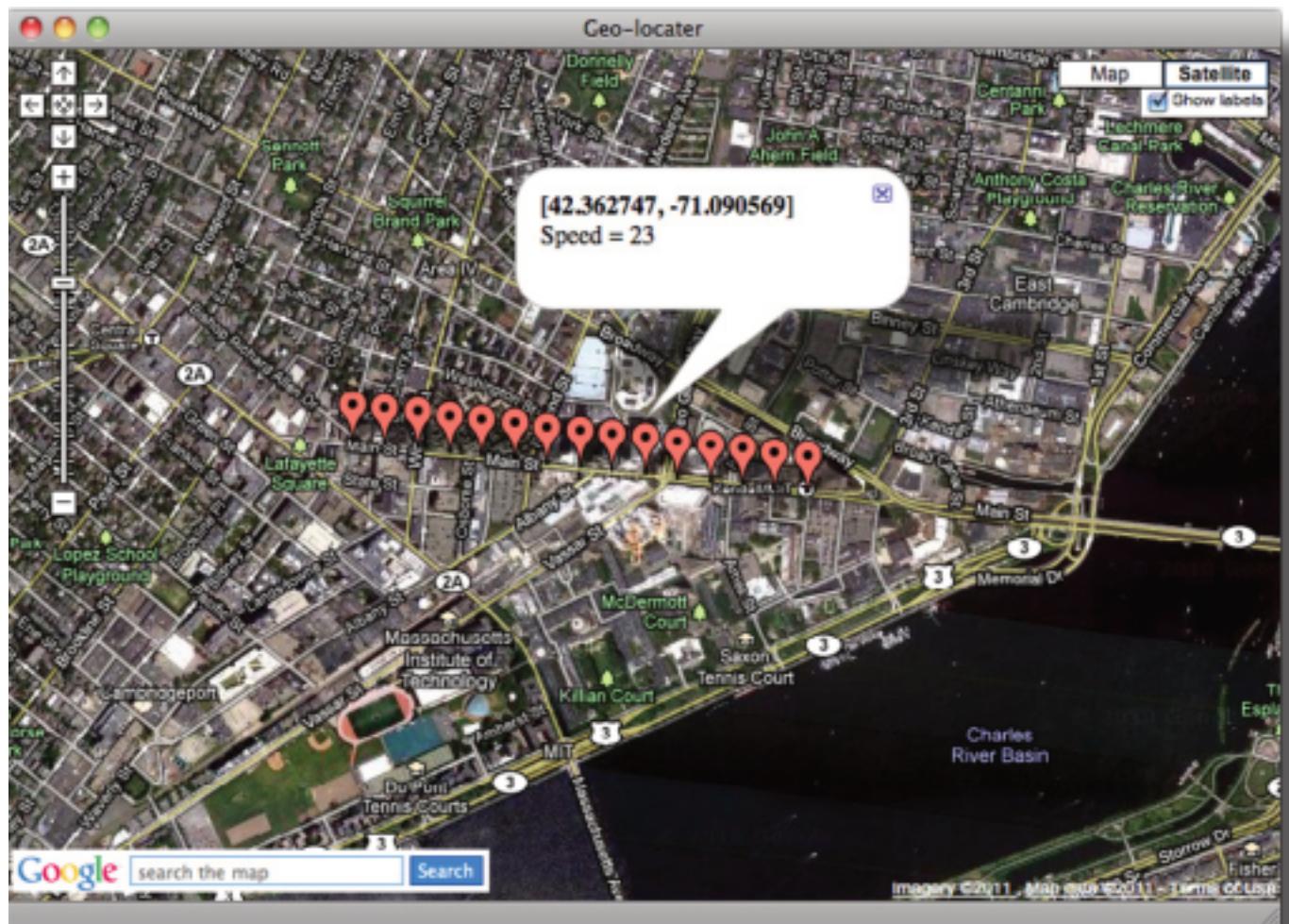


Convert the screenshot to html.



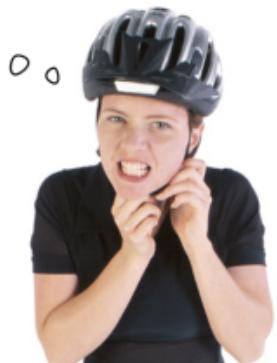
View it from the browser to confirm it works.





The map works. The map inside the web page is able to read the data from the output file. Great! Now I can publish my journeys on the Web!

But there's a problem with some of the data. Your program seems to be able to read GPS data and format it correctly for the mapping application. But after a few days, a problem creeps in.



I dropped the GPS unit on a ride a couple of times, and now the map won't display.

So what happened here? The problem is that there was some bad data in the GPS data file.

```
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},  
{latitude: 423.63182, longitude: -71.095833, info: 'Speed = 22'},
```

The decimal point is in the wrong place in this number.

The decimal point is in the wrong place in this number. But the geo2json program doesn't do any checking of the data it reads; it just reformats the numbers and sends them to the output. That should be easy to fix. You need to validate the data.

You need to add some code to the geo2json program that will check for bad latitude and longitude values. You don't need anything fancy. If a latitude or longitude falls outside the expected numeric, just display an error message and exit the program.

with an error status of 2:

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

int main()
{
    float latitude;
    float longitude;
    char info[80];
    int started = 0;

    puts("data=[");
    while (scanf("%f %f %79[^\\n]", &latitude, &longitude, info) == 3)
    {
        if (started)
            printf(",\\n");
        else
            started = 1;
        if ((latitude < -90.0) || (latitude > 90.0))
        {
            printf("invalid latitude: %f\\n", latitude);
            return 2;
        }
        if ((longitude < -180.0)) || (longitude > 180.0))
        {
            printf("invalid longitude: %f\\n", longitude);
            return 2;
        }

        printf("{latitude: %f, longitude: %f, info: '%s'}", latitude,
longitude, info);
    }
    puts("\\n    ]\\n");
    return 0;
}
```

```

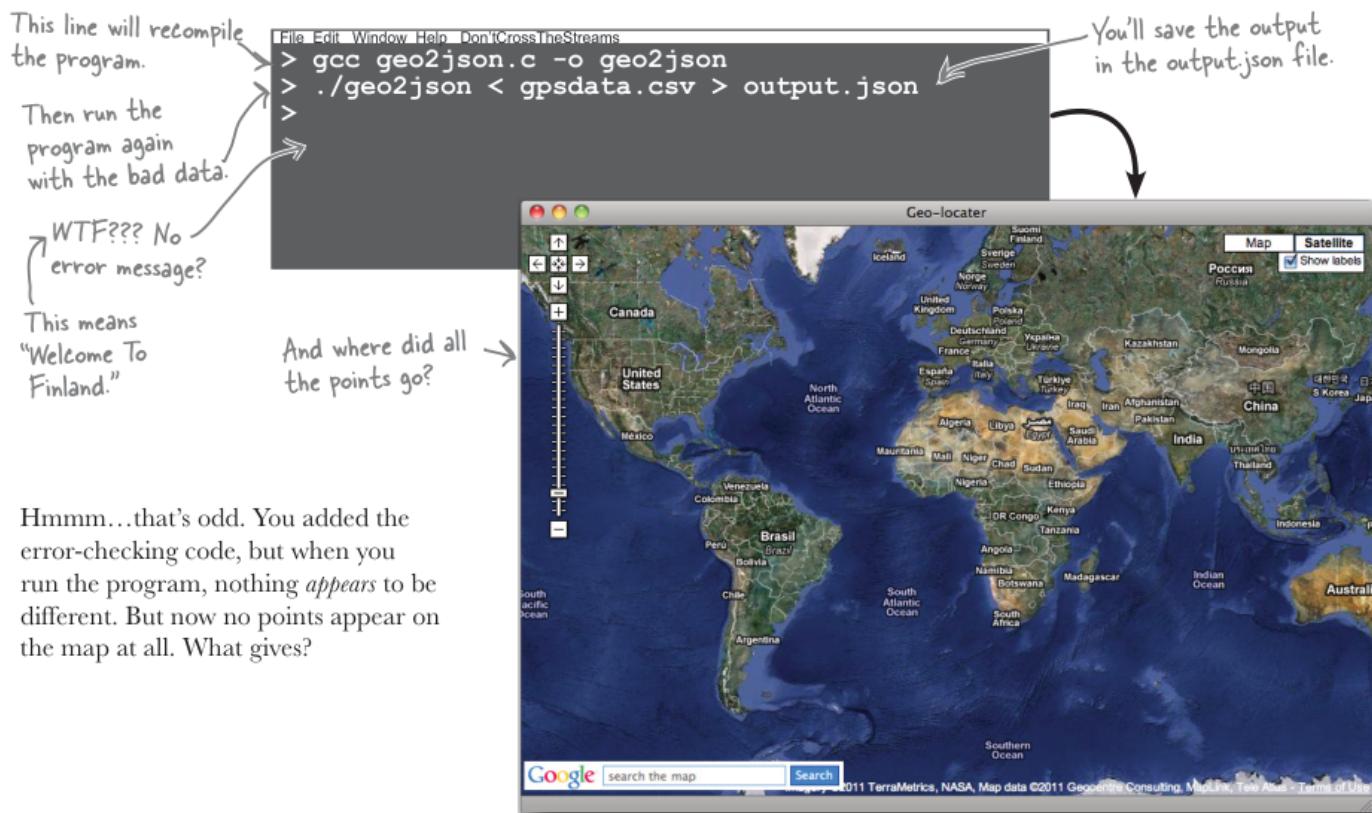
10
11     puts("data=[]");
12     while(scanf("%f %f %79[^\\n]", &latitude, &longitude, info) == 3)
13     {
14         if(started)
15             printf(",\\n");
16         else
17             started = 1;
18         if((latitude < -90.0) || (latitude > 90.0))
19         {
20             printf("invalid latitude: %f\\n", latitude);
21             return 2;
22         }
23         if((longitude < -180.0) || (longitude > 180.0))
24         {
25             printf("invalid longitude: %f\\n", longitude);

```

note that this is the equals operator
not the assignment operator

OK, so you now have the code in place to check that the latitude and longitude are in range. But will it be enough to make our program cope with bad data? Let's see. Compile the code and then run the bad data through the program:

Hmmm...that's odd. You added the error-checking code, but when you run the program, nothing appears to be different. But now no points appear on the map at all. What gives?



Hmmm...that's odd. You added the error-checking code, but when you run the program, nothing *appears* to be different. But now no points appear on the map at all. What gives?

Study the code. What do you think happened? Is the code doing what you asked it to? Why weren't there any error messages? Why did the mapping program think that the entire `output.json` file was corrupt?

CODE DECONSTRUCTION

The mapping program is complaining about the `output.json` file, so let's open it up and see what's inside: Once you open the file, you can see exactly what happened. The program saw that there was a problem with some of the data, and it exited right away.

`code deconstruction`

CODE DECONSTRUCTION

The mapping program is complaining about the `output.json` file, so let's open it up and see what's inside:

↙ This is the `output.json` file.

```
data=[  
  {latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},  
  {latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},  
  {latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},  
  Invalid latitude: 423.631805
```



↑ Oh, the error message was also redirected to the output file.

It didn't process any more data and it did output an error message. Problem is, because you were redirecting the Standard Output into the `output.json`, that meant you were also redirecting the error message. So the program ended silently, and you never saw what the problem was.

Now, you could have checked the exit status of the program, but you really want to be able to see the error messages. But how can you still display error messages if you are redirecting the output? Oh, the error message was also redirected to the output file. This is the `output.json` file

If your program finds a problem in the data, it exits with a status of 2. But how can you check that error status after the program has finished? Well, it depends on what operating system you're using. If you're using the Command Prompt in Windows,

then it's a little different:

```
C:\Users\HP-EliteBook>echo %ERRORLEVEL%  
0
```

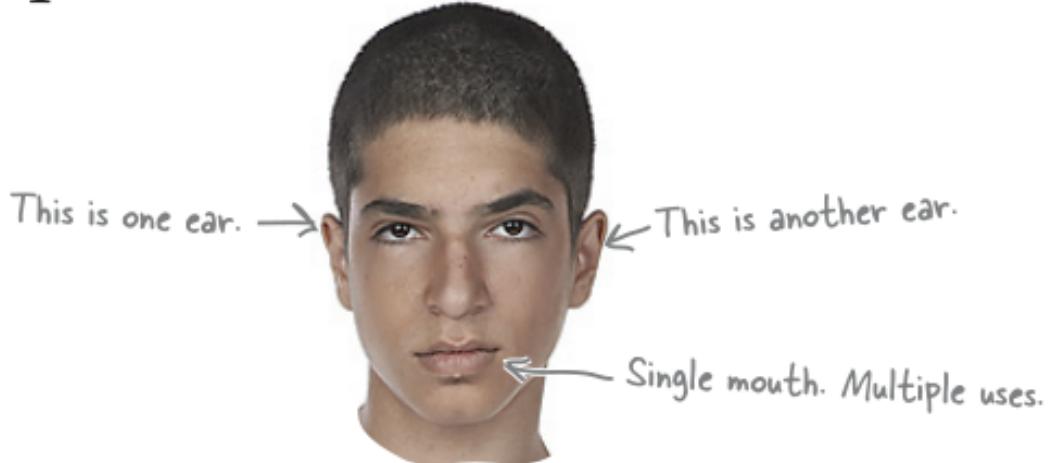
STANDARD ERROR

Wouldn't it be dreamy if there were a special output for errors so that I didn't have to mix my errors in with Standard Output? But I know it's just a fantasy...



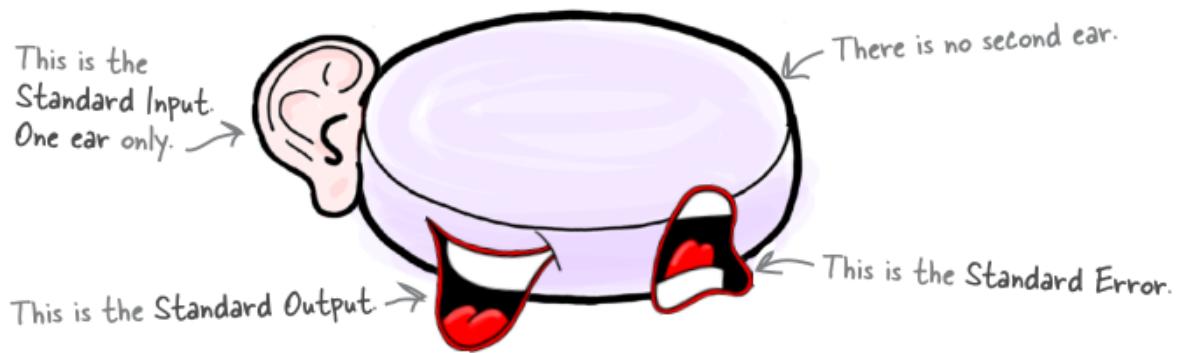
Introducing the Standard Error - The Standard Output is the default way of outputting data from a program. But what if something exceptional happens, like an error? You'll probably want to deal with things like error messages a little differently from the usual output.

Human



That's why the Standard Error was invented. The Standard Error is a second output that was created for sending error messages. Human beings generally have two ears and one mouth, but processes are wired a little differently. Every process has one ear (the Standard Input) and two mouths (the Standard Output and the Standard Error).

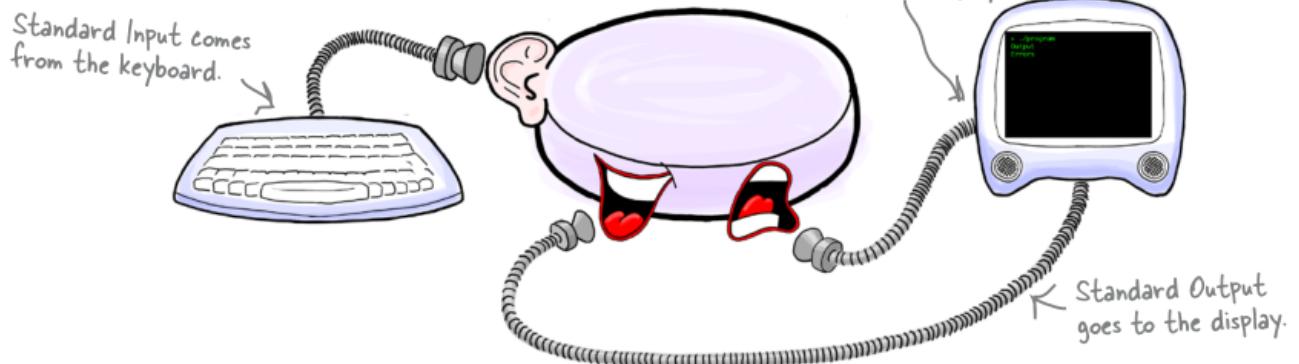
Process



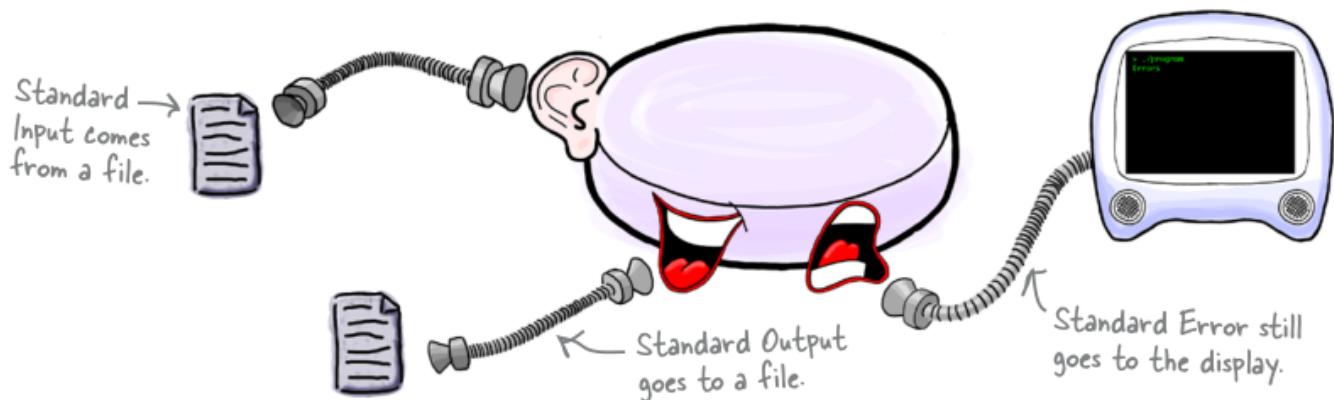
Remember how when a new process is created, the operating system points the Standard Input at the keyboard and the Standard Output at the screen?

Well, the operating system creates the Standard Error at the same time and, like the Standard Output, the **Standard Error is sent to the display by default**.

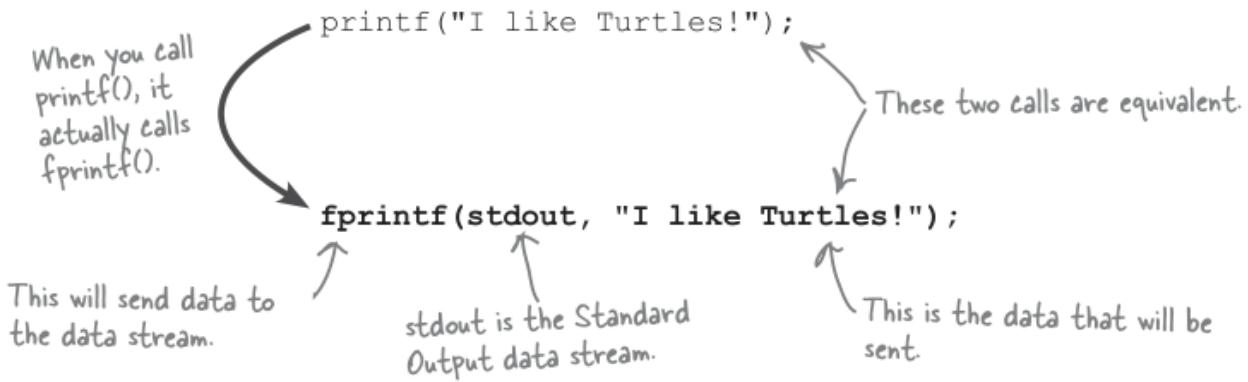
Output at the screen: even, the operating system creates the Standard Error at the same time and, like the Standard Output, the Standard Error is sent to the display by default.



That means that if someone redirects the Standard Input and Standard Output so they use files, **the Standard Error will continue to send data to the display**. And that's really cool, because it means that even if the Standard Output is redirected somewhere else, by default, any messages sent down the Standard Error will still be visible on the screen. So you can fix the problem of our hidden error messages by simply displaying them on the Standard Error.



fprintf() prints to a data stream You've already seen that the **printf()** function sends data to the Standard Output. What you didn't know is that the **printf()** function is just a version of a more general function called **fprintf()**:



NB:

The `fprintf()` function allows you to choose where you want to send text to. You can tell `fprintf()` to send text to `stdout` (the Standard Output) or `stderr` (the Standard Error).

By using `fscanf()`, which is just like `scanf()`, but you can specify the data stream.

In C programming, `STDOUT`, `STDIN`, and `STDERR` are special file descriptors representing the standard output, standard input, and standard error streams respectively. They are typically associated with the console or terminal that a program is running in.

A **data stream** is a channel or pathway through which data can be transferred, either to or from a program. The data can be text, binary data, or other types of data, and the data stream can be a file, a network connection, or another type of data source or destination. In C programming, data streams are typically associated with the standard input, output, and error streams (`stdin`, `stdout`, and `stderr`), which are predefined channels for input, output, and error messages, respectively.

STDOUT is a standard output stream where the program writes its output. It is usually redirected to a file or a pipe when the program is run from a command line. `printf()` function writes its output to the `STDOUT` stream by default.

STDIN is a standard input stream where the program reads its input. It is usually provided by the user through a keyboard or redirected from a file or a pipe.

STDERR is a standard error stream where the program writes its error messages or diagnostic information. It is typically used for printing error messages or

debugging information that should not be mixed with the regular program output. By default, error messages written with `fprintf()` to `STDERR` are not buffered, so they are immediately printed to the console without waiting for the buffer to fill up.

In summary, `STDOUT`, `STDIN`, and `STDERR` are special streams that allow a program to interact with its environment by reading input, writing output, and reporting errors respectively. They are not modes of output or input, but rather file descriptors that point to specific streams.

FSCANF() and FPRINTF() : Yes, `fscanf(stdin, ...)` and `scanf(...)` are essentially the same, since `stdin` is the standard input stream and `scanf` reads input from the standard input stream by default. However, using `fscanf` explicitly with `stdin` can be useful in certain situations where you want to read input from a different stream or file.

```
//scanf function

#include <stdio.h>

int main() {
    int num;
    printf("Enter a number: ");
    fscanf(stdin, "%d", &num);
    printf("You entered: %d\n", num);
    return 0;
}

//fscanf() function

#include <stdio.h>

int main() {
    int num;
    printf("Enter a number: ");
    fscanf(stdin, "%d", &num);
    printf("You entered: %d\n", num);
    return 0;
}
```

Redirecting the standard error:

Yes; **>** redirects the Standard Output. But **2>** redirects the Standard Error.

Example of changing the json file to redirect to errors.

So I could write **geo2json 2> errors.txt**

```
#include <stdio.h>

int main() {
    FILE* err_stream;

    // Redirect the standard error output to a file
    err_stream = freopen("error.log", "w", stderr);
    if (err_stream == NULL) {
        perror("Failed to redirect stderr");
        return 1;
    }

    // Now all error messages will be written to the "error.log" file
    fprintf(stderr, "This is an error message\n");

    // Close the error log file when done
    fclose(err_stream);

    return 0;
}
```

In this example, the `freopen()` function is used to open the "error.log" file in write mode ("w") and redirect the standard error output to it (stderr). The function returns a pointer to the new output stream, which can be used just like the stderr stream in the rest of the program.

Note that the `perror()` function is used to print an error message if `freopen()` fails to

redirect the standard error output. This can happen if the file can't be opened or if there's a problem with the file system.

UPDATING THE PROGRAM TO USE FPRINTF()

```
#include <stdio.h>

int main()
{
    float latitude;
    float longitude;
    char info[80];
    int started = 0;

    puts("data=[");
    while (scanf("%f, %f, %79[^\\n]", &latitude, &longitude, info) == 3)
    {

        if (started)
            printf(",\\n");
        else
            started = 1;
        if ((latitude < -90.0) || (latitude > 90.0))
        {

            fprintf(stderr, "Invalid latitude: %f\\n", latitude);
            return 2;
        }

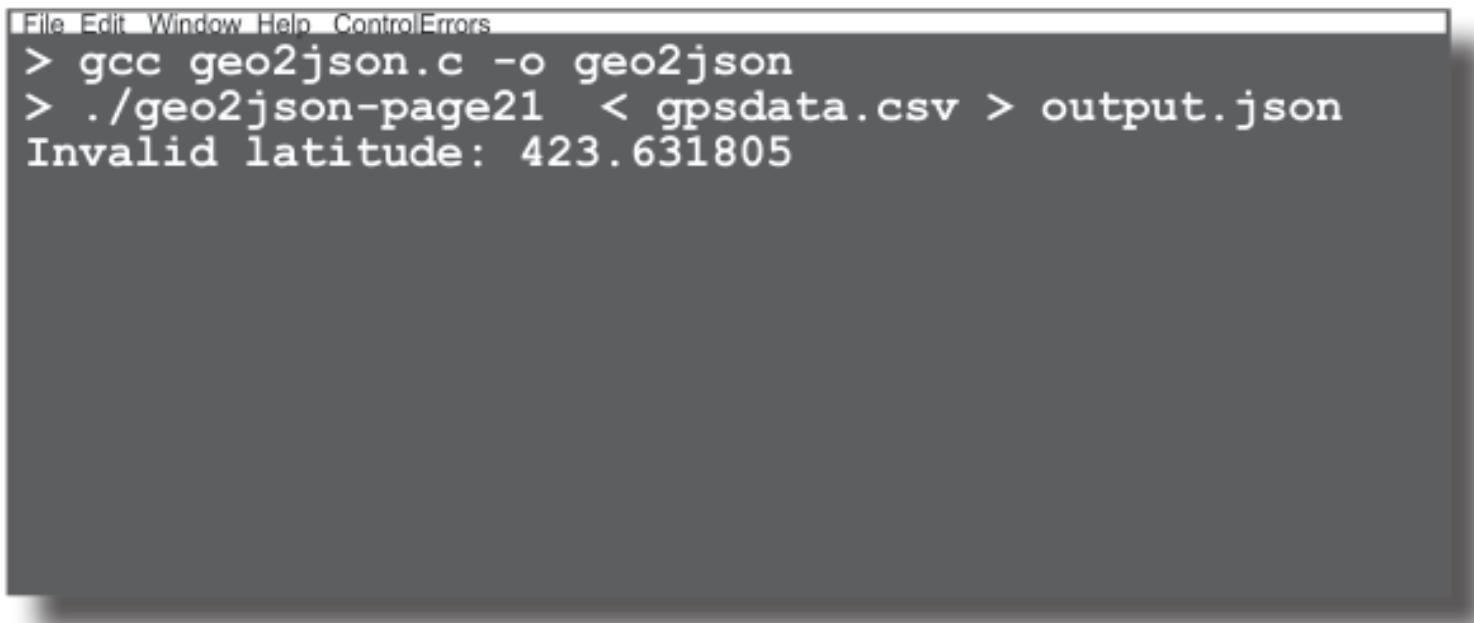
        if ((longitude < -180.0) || (longitude > 180.0))
        {
            fprintf(stderr, "Invalid longitude: %f\\n", longitude);
            return 2; //we need to specify stderr as the first
parameter
        }

        printf("{latitude: %f, longitude: %f, info: '%s'}", latitude,
longitude, info);
    }

    puts("\\n]");
    return 0;
}
```

That means that the code should now work in exactly the same way, except the error messages should appear on the Standard Error instead of the Standard Output.

If you recompile the program and then run the corrupted GPS data through it again, this happens.



```
File Edit Window Help Control Errors
> gcc geo2json.c -o geo2json
> ./geo2json-page21 < gpsdata.csv > output.json
Invalid latitude: 423.631805
```

That's excellent. This time, even though you are redirecting the Standard Output into the output.json file, the error message is still visible on the screen.

The Standard Error was created with exactly this in mind: to separate the error messages from the usual output.

But remember: stderr and stdout are both just output streams. And there's nothing to prevent you from using them for anything.

Let's try out your newfound Standard Input and Standard Error skills.

SUMMARY

The `printf()` function sends data to the Standard Output.

The Standard Output goes to the display by default.

You can redirect the Standard Output to a file by using `>` on the command line.

`scanf()` reads data from the Standard Input.

The Standard Input reads data from the keyboard by default.

You can redirect the Standard Input to read a file by using `<` on the command line.

The Standard Error is reserved for outputting error messages.

You can redirect the Standard Error using `2>`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 //intercepting a file secret.txt and scan of paper with instructions.
6 int main()
7 {
8     char word[10];
9     int i = 0;
10    while (scanf("%9s", word) == 1)
11    {
12        i = i + 1;
13        if(i % 2) //the remainder left when you divide by 2.
14            fprintf(stdout, "%s\n", word);
15        else
16            fprintf(stderr, "%s\n", word);
17    }
18    return 0;
19 }
```

We have intercepted a file called `secret.txt` and a scrap of paper with instructions:

THE BUY SUBMARINE
SIX WILL EGGS
SURFACE AND AT
SOME NINE MILK PM

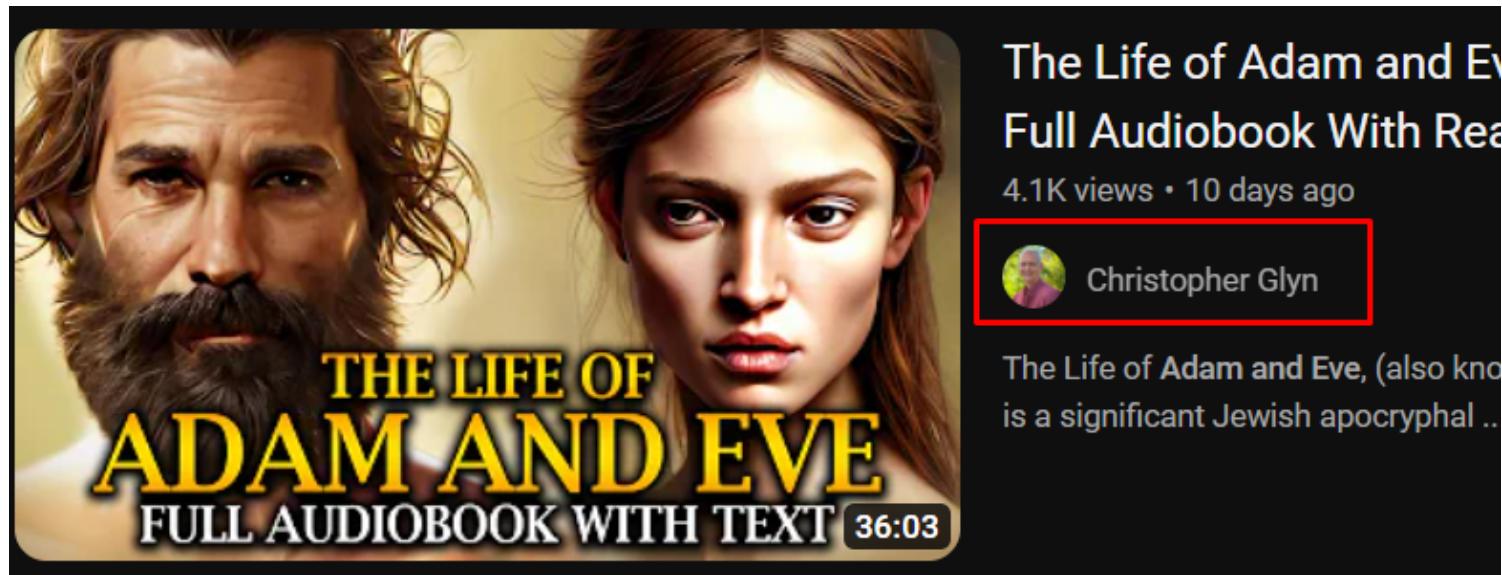
`secret.txt`

Run with:

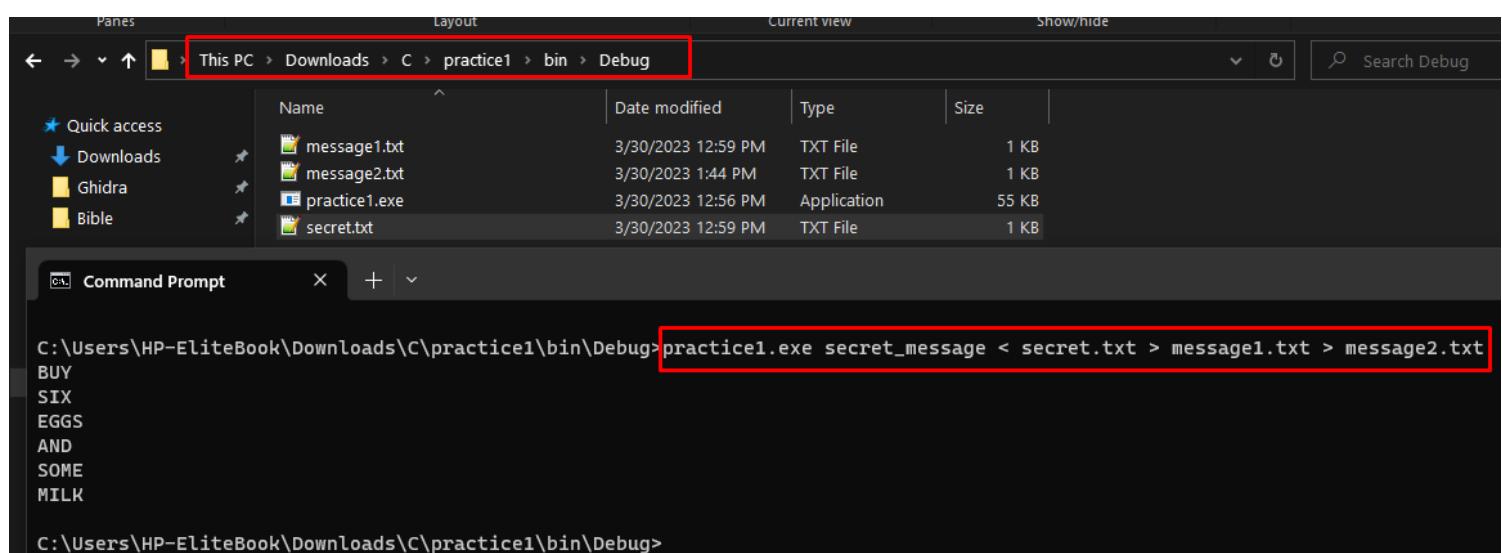
`secret_message < secret.txt > message1.txt 2> message2.txt`

> will redirect the Standard Output.

2> will redirect the Standard Error.



Let's run the decoder.



Panes Layout Current view Show/hide

← → ⌂ ↑ This PC > Downloads > C > practice1 > bin > Debug

Quick access Downloads Ghidra Bible

Name	Date modified	Type	Size
message1.txt	3/30/2023 12:59 PM	TXT File	1 KB
message2.txt	3/30/2023 1:44 PM	TXT File	1 KB
practice1.exe	3/30/2023 12:56 PM	Application	55 KB
secret.txt	3/30/2023 12:59 PM	TXT File	1 KB

Command Prompt

```
C:\Users\HP-EliteBook\Downloads\C\practice1\bin\Debug>practice1.exe secret_message < secret.txt > message1.txt > message2.txt
```

BUY
SIX
EGGS
AND
SOME
MILK

```
C:\Users\HP-EliteBook\Downloads\C\practice1\bin\Debug>
```

The results:

secret.txt

1	THE	BUY	SUBMARINE
2	SIX	WILL	EGGS
3	SURFACE	AND	AT
4	SOME	NINE	MILK PM

message2.txt

1	THE
2	SUBMARINE
3	WILL
4	SURFACE
5	AT
6	NINE
7	PM
8	PM

Head First: Does it matter what kind of O/S you are?

O/S: A lot of people get pretty heated over which operating system to use. But for simple C programs, we all behave pretty much the same way.

Head First: Because of the C Standard Library?

O/S: Yeah, if you're writing C, then the basics are the same everywhere. Like I always say, we're all the same with the lights out. Know what I'm saying?

Head First: Oh, of course. Now, you are in charge of loading programs into memory?

O/S: I turn them into processes, that's right.

Head First: Important job?

O/S: I like to think so. You can't just throw a program into memory and let it struggle, you know? There's a whole bunch of setup. I need to allocate memory for the programs and connect them to their standard data streams so they can use things like displays and keyboards.

Head First: Like you just did for the geo2json program?

O/S: That guy's a real tool.

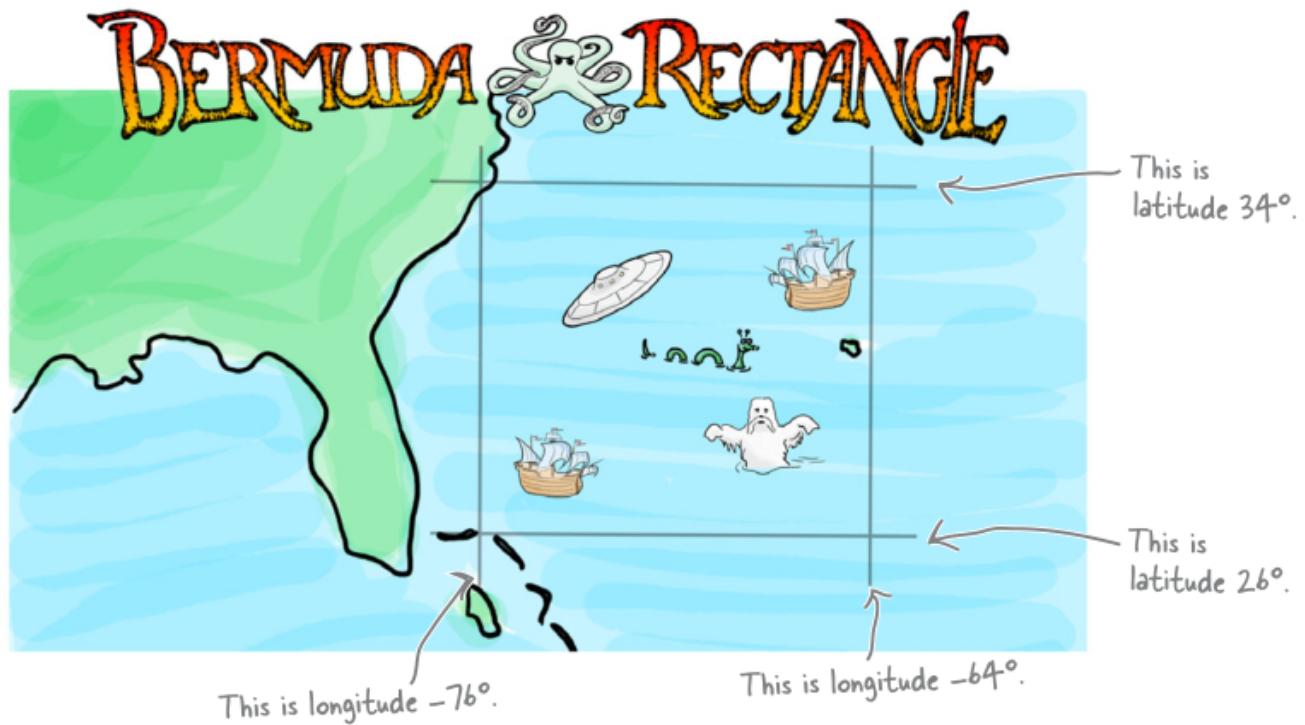
Head First: Oh, I'm sorry.

O/S: No, I mean he's a real tool: a simple, text-based program.

To see how flexible our tool is, let's use it for a completely different problem. Instead of just displaying data on a map, let's try to use it for something a little more complex.

Say you want to read in a whole set of GPS data like before, but instead of just displaying everything, let's just display the information that falls inside the Bermuda Rectangle.

That means you will display only data that matches these conditions:



```
((latitude > 26) && (latitude < 34))
```

```
((longitude > -76) && (longitude < -64))
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4
5
6 int main()
7 {
8     float latitude;
9     float longitude;
10    char info[80];
11    while (scanf("%f, %f, %79[^\\n]", &latitude, &longitude, info) == 3)
12        if((latitude > 26) && (latitude < 34))
13        {
14            if((longitude > -76) && (longitude < -64))
15                printf("%f, %f, %s\\n", latitude, longitude, info);
16        }
17    return 0;
18
19 }
```

Opening, closing and reading and writing files.

```
5  int main()
6  {
7      char line[80];
8      FILE *in = fopen("spooky.csv", 'r');
9      FILE *file1 = fopen("ufos.csv", 'w');
10     FILE *file2 = fopen("disappearances.csv", 'w');
11     FILE *file3 = fopen("others.csv", 'w');
12
13     while(fscanf(in, "%79[^\\n]\\n", line) == 1)
14     {
15         if(strstr(line, "UFO"))
16             fprintf(file1, "%s\\n", line);
17         else if(strstr(line, "Disappearance"))
18             fprintf(file2, "%s\\n", line);
19         else
20             fprintf(file3, "%s\\n", line);
21     }
22
23     fclose(file1);
24     fclose(file2);
25     fclose(file3);
26     return 0;
27
28 }
```

ARGV AND ARGC

Yes, if those are the command-line parameters, then argc will be 6, and argv will contain the following values:

- argv[0] will be the string "./categorize"
- argv[1] will be the string "mermaid"
- argv[2] will be the string "mermaid.csv"
- argv[3] will be the string "Elvis"
- argv[4] will be the string "elvises.csv"
- argv[5] will be the string "the_rest.csv"

So if you want to access the values of the command-line arguments in your program, you can use the argc and argv parameters of the main function.

For example, the following code prints the values of the command-line arguments:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;

    for (i = 0; i < argc; i++) {
        printf("Argument %d is %s\n", i, argv[i]);
    }

    return 0;
}
```

```
Argument 0 is ./categorize
Argument 1 is mermaid
Argument 2 is mermaid.csv
Argument 3 is Elvis
Argument 4 is elvises.csv
Argument 5 is the_rest.csv
```

In C and C++, `argv` is an array of strings (i.e., an array of arrays of characters). Each element of the array `argv` is a pointer to a null-terminated string.

So in the example you gave, `argv[0]`, `argv[1]`, `argv[2]`, `argv[3]`, `argv[4]`, and `argv[5]` are all pointers to null-terminated strings.

- `argv[0]` is a pointer to the string `"/categorize"`
- `argv[1]` is a pointer to the string `"mermaid"`
- `argv[2]` is a pointer to the string `"mermaid.csv"`
- `argv[3]` is a pointer to the string `"Elvis"`
- `argv[4]` is a pointer to the string `"elvises.csv"`

- `argv[5]` is a pointer to the string "the_rest.csv"

Note that `argv[0]` is typically the name of the program being executed (i.e., the first command-line argument passed to the program).

So `argv` is an array of pointers to strings, and you can access each individual string by using array subscript notation (i.e., `argv[i]` to access the *i*-th string).

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

int main(int argc, char *argv[])
{
    char line[80];
    if (argc != 5) {
        fprintf(stderr, "You need to give 5 arguments\n");
        return 1;
    }
    FILE *in = fopen("spooky.csv", "r");
    FILE *file1 = fopen(argv[1], "w");
    FILE *file2 = fopen(argv[2], "w");
    FILE *file3 = fopen(argv[3], "w");

    while (fscanf(in, "%79[^\\n]\\n", line) == 1) {
        if (strstr(line, argv[1]))
            fprintf(file1, "%s\n", line);
        else if (strstr(line, argv[2]))
            fprintf(file2, "%s\n", line);
        else
            fprintf(file3, "%s\n", line);
    }

    fclose(file1);
    fclose(file2);
    fclose(file3);
    return 0;
}
```

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

These are include statements that allow the program to use functions from the standard input/output library (stdio.h), standard library (stdlib.h), and string library (string.h).

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

This is the starting point of the program. It declares the main function with two arguments: argc (an integer that represents the number of arguments passed to the program from the command line), and argv (an array of strings that contains the actual arguments). The main function returns an integer, which in this case indicates whether the program ran successfully or not.

```
char line[80];
```

This declares a character array named line that can hold up to 80 characters.

```
if (argc != 5) {
    fprintf(stderr, "You need to give 5 arguments\n");
    return 1;
}
```

This checks if the number of arguments passed to the program is not equal to 5. If it's not, the program outputs an error message to the standard error stream using fprintf() and returns a non-zero value, indicating that the program failed.

```
FILE *in = fopen("spooky.csv", "r");
FILE *file1 = fopen(argv[2], "w");
FILE *file2 = fopen(argv[3], "w");
FILE *file3 = fopen(argv[4], "w");
```

These lines declare four FILE pointers that are used to handle file input and output. The first pointer (in) is used to read data from a file named "spooky.csv". The remaining three pointers are used to write data to three different files whose names are specified by the command line arguments argv[2], argv[3], and argv[4], respectively. The "w" argument passed to the fopen() function specifies that the file should be opened in write mode.

```
while (fscanf(in, "%79[^\\n]\\n", line) == 1) {
```

This starts a loop that reads lines from the in file using fscanf(). The "%79[^\\n]" format specifier tells fscanf() to read up to 79 characters or until it encounters a newline character (\\n). The newline character is then consumed by the fscanf() call, so that it doesn't get included in the line array. The while loop continues as long as fscanf() successfully reads a line from the input file (i.e., it returns a value of 1).

"%79[^\\n]\\n" is a format string used by the fscanf() function to read input from a file. The string consists of two parts:

1. "%79[^\\n]" is a conversion specifier that tells fscanf() to read up to 79 characters from the input file, excluding any newline (\\n) characters. The [^\\n] part specifies that any character other than a newline character is allowed.
2. "\\n" matches a newline character in the input file. This serves to consume the newline character so that it's not included in the resulting string stored in the line array.

So when fscanf() is called with this format string, it reads up to 79 characters from

the input file until it reaches a newline character (\n). It stores the resulting string in the line array, and then consumes the newline character. This process is repeated until fscanf() can no longer read any more input lines from the file.

```
if (strstr(line, argv[1]))
    fprintf(file1, "%s\n", line);
else if (strstr(line, argv[2]))
    fprintf(file2, "%s\n", line);
else
    fprintf(file3, "%s\n", line);
```

Inside the loop, this code uses the strstr() function to search for a substring in the line array that matches the first command-line argument argv[1]. If a match is found, the line is written to the first output file using fprintf().

If there's no match with argv[1], the code searches for a match with the second command-line argument argv[2]. If a match is found with argv[2], the line is written to the second output file. If there's no match with either argv[1] or argv[2], the line is written to the third output.

FSCANF AND FPRINTF

fscanf() and fprintf() are functions from the standard I/O library in C that are used for reading input from and writing output to a file, respectively.

In the code provided, fscanf() is used to read input lines from the `spooky.csv` file. The format string "%79[^\\n]\\n" specifies how the input lines should be read and parsed.

The function reads up to 79 characters from each input line, excluding the newline character, and stores the resulting string in the line array. The return value of fscanf() is used to determine whether there are more lines to read from the file.

If the return value is 1, it means that fscanf() was able to successfully read and parse one line from the file, so the loop continues to read more lines. If the return value is not 1, it means that there are no more lines to read from the file, so the loop exits.

fprintf() is used to write output lines to three different files: file1, file2, and file3.

The fprintf() function takes a file pointer as its first argument, followed by a format string and any additional arguments to be written to the file.

In the code provided, the if statement checks whether each input line contains a specific substring using the strstr() function.

If the substring is found in the input line, the line is written to the appropriate output file using fprintf().

If the substring is not found, the line is written to file3.

The format string "%s\n" specifies how each output line should be formatted: as a string followed by a newline character.

FSCANF AND FPRINTF, PRINTF AND SCANF AND STDIN, STDOUT, STDERR

stdin, stdout, and stderr are the standard input/output streams in C. printf() and scanf() are functions that write to and read from these streams, respectively.

stdin is the standard input stream, which is typically connected to the keyboard or some other input device. scanf() reads input from stdin.

stdout is the standard output stream, which is typically connected to the console or some other output device. printf() writes output to stdout.

stderr is the standard error stream, which is also typically connected to the console or some other output device. It is used for writing error messages and diagnostic

output.

In the code provided, `fscanf()` is used to read input from the `in` file pointer, which points to a file named `spooky.csv`. The `scanf()` function can also be used to read input from `stdin`.

`fprintf()` is used to write output to three different files, `file1`, `file2`, and `file3`, which are opened using `fopen()`. The `fprintf()` function can also be used to write output to `stdout`.

`printf()` and `fprintf()` are similar, but `printf()` writes output to `stdout` while `fprintf()` writes output to a specified file. Similarly, `scanf()` and `fscanf()` are similar, but `scanf()` reads input from `stdin` while `fscanf()` reads input from a specified file.

`stderr` is used in the code to write an error message to the console if the number of command line arguments is not equal to 5.

EXAMPLES

Here's an example of how `printf()` and `scanf()` could be used in the code to read input from `stdin` and write output to `stdout`:

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

int main() {
    char input[80];
    printf("Enter a line of text:\n");
    scanf("%79[^\\n]\\n", input);
    printf("You entered: %s\n", input);
    return 0;
}
```

In this modified version of the code, `scanf()` reads a line of input from `stdin` using the same format string as in the original code.

The resulting string is stored in the `input` array. `printf()` is then used to write a message to `stdout` that displays the input that was just read.

Similarly, here's an example of how `fprintf()` and `fscanf()` could be used in the code to read input from a file and write output to a file:

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

int main() {
    char input[80];
    FILE *in = fopen("input.txt", "r");
    FILE *out = fopen("output.txt", "w");
    while (fscanf(in, "%79[^\\n]\\n", input) == 1) {
        fprintf(out, "You entered: %s\\n", input);
    }
    fclose(in);
    fclose(out);
    return 0;
}
```

In this modified version of the code, `fopen()` is used to open two files, `input.txt` and `output.txt`, for reading and writing, respectively.

`fscanf()` is used to read input lines from the `in` file pointer using the same format string as in the original code.

The resulting string is stored in the `input` array.

`fprintf()` is then used to write a message to the `out` file pointer that displays the

input that was just read.

Finally, `fclose()` is used to close both file pointers.

OTHER LIBRARIES

GETOPT() FUNCTION

The `getopt` function gets the next option argument from the argument list specified by the `argv` and `argc` arguments.

Normally these values come directly from the arguments received by `main`.

The `options` argument is a string that specifies the option characters that are valid for this program.

The `getopt` function returns the option character for the next command line option. When no more option arguments are available, it returns `-1`.

There may still be more non-option arguments; you must compare the external variable `optind` against the `argc` parameter to check this.

Chances are, any program you write is going to need options. If you create a chat program, it's going to need preferences.

If you write a game, the user will want to change the shape of the blood spots.

And if you're writing a command-line tool, you are probably going to need to add command-line options.

Command-line options are the little switches you often see with command-line tools:

`ps -ae` =>> Display all the processes, including their environments.

`tail -f logfile.txt` =>> Display the end of the file, but wait for new data to be added to the end of the file.

Many programs use command-line options, so there's a special library function you can use to make dealing with them a little easier.

It's called `getopt()`, and each time you call it, it returns the next option it finds on the command line.

This statement is generally true. The `getopt()` function is a standard library function that is commonly used in C and C++ programs to parse command-line options.

It allows the programmer to specify which options the program should recognize, and then handles the parsing of the command line arguments to extract those options.

When you call `getopt()`, it returns the next option it finds on the command line, along with any associated arguments.

The programmer can then take action based on the option and its arguments.

For example, if the program is designed to accept a `-h` option to display help information, the programmer could use `getopt()` to detect the presence of the `-h` option and display the appropriate help text.

However, it's worth noting that the specifics of how `getopt()` is used can vary depending on the programming language and the operating system being used.

For example, the POSIX version of `getopt()` behaves slightly differently than the GNU version of `getopt()`.

Additionally, some languages may have their own built-in libraries for parsing command-line arguments, rather than relying on `getopt()`.

Let's see how it works. Imagine you have a program that can take a set of different options:

Use four engines. ↗ Awesomeness mode enabled.
`rocket_to -e 4 -a Brasilia Tokyo London`

This program needs one option that will take a value (-e = engines) and another that is simply on or off (-a = awesomeness). You can handle these options by calling getopt() in a loop like this:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main()
6  {
7      while((ch = getopt(argc, argv, "ae:")) != EOF)
8          switch(ch)
9          {
10             case 'e':
11                 engine_count = optarg;
12             }
13
14             argc -= optind;
15             argv += optind;
16 }
```

This code is an example of using the getopt() function to parse command line arguments in a C or C++ program.

The getopt() function is used to parse command line arguments that are passed to a program when it is run, and it requires two arguments: `argc` and `argv`.

In this example, the `getopt()` function is called in a **loop** to process each option that appears on the command line.

The loop continues until `getopt()` returns `EOF`, indicating that there are no more

options to process.

The arguments to getopt() are:

- argc: The number of arguments passed to the program.
- argv: An array of strings containing the arguments passed to the program.
- "ae": A string containing the valid options that the program can accept. In this case, the program can accept options -a and -e, where the -e option requires an argument.

The loop switches on the value returned by getopt() for each option, which will be one of the valid options specified in the third argument to getopt().

If the option is -e, the value of optarg is assigned to the engine_count variable, which is presumably used elsewhere in the program.

After all options have been processed, **argc** and **argv** are adjusted to point to the remaining arguments that were not options.

Overall, this code demonstrates a basic use of getopt() to parse command line arguments and extract option values from them.

REAL WORLD EXAMPLE FOR GETOPT:

Imagine you're going to a restaurant and the menu has many options, but you want to order only what you like.

The waiter brings you a paper where you can write your order.

On this paper, you write the name of the dish you want to order and some additional instructions, like whether you want it with or without cheese, or with extra sauce.

In computer programs, we also have something similar. When we run a program, we can give it some instructions, called "command line arguments", that tell the program what to do.

For example, we can tell the program to open a file, or to show us some information on the screen.

The getopt() function is like the paper you get at the restaurant. It helps the program read the instructions you give it on the command line.

The getopt() function looks at the options you give it, like -a or -e, and figures out what you want to do with them.

Just like you can write additional instructions on your restaurant order, some options require more information to be useful.

For example, if you order a pizza and you want extra cheese, you have to specify that you want extra cheese.

In programs, some options require more information too, and you give this information by adding an argument after the option, like -e 10 (where 10 is the additional information).

So, getopt() helps the program read your instructions and understand what you want to do. It's like a helper that takes your order and translates it into something the chef can understand.

You will need to → `#include <unistd.h>`
include this header.

This means "The a option is valid; so is the e option."

The code to handle each option goes here. → `while ((ch = getopt(argc, argv, "ae:")) != EOF)`

The ":" means that the e option needs an argument.

You're reading the argument for the "e" option here. → `switch(ch) {`

```
while ((ch = getopt(argc, argv, "ae:")) != EOF)
    switch(ch) {
        ...
        case 'e':
            engine_count = optarg;
        ...
    }
```

These final two lines make sure we skip past the options we read. → `argc -= optind;` → `argv += optind;`

optind stores the number of strings read from the command line to get past the options.

Inside the loop, you have a switch statement to handle each of the valid options. The string `ae:` tells the `getopt()` function that `a` and `e` are valid options.

The `e` is followed by a colon to tell `getopt()` that the `-e` needs to be followed by an extra argument.

`getopt()` will point to that argument with the `optarg` variable. When the loop finishes, you tweak the `argv` and `argc` variables to skip past all of the options and get to the main command-line arguments.

That will make your `argv` array look like this:

The diagram shows a horizontal line representing an array of strings. Three arrows point to specific elements: an arrow from the text "This is argv[0]." to the first element "Brasilia", an arrow from "This is argv[1]." to the second element "Tokyo", and an arrow from "This is argv[2]." to the third element "London".

Brasilia Tokyo London

This is argv[0]. This is argv[1]. This is argv[2].

After processing the arguments, the 0th argument will no longer be the program name. `argv[0]` will instead point to the first command-line argument that follows the options.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main(int argc, char *argv[])
6  {
7      char *delivery;
8      int thick = 0;
9      int count = 0;
10     char ch;
11
12     while ((ch = getopt(argc, argv, "d:t")) != EOF)
13     {
14         switch(ch)
15         {
16             case 'd':
17                 delivery = optarg; //point delivery variable to the arg supplied with d option.
18                 break;
19             }
20
21             case 't':
22                 thick = 1; //in c setting anything to 1 == true
23                 break;
24             default:
25                 fprintf(stderr, "Unknown option: '%s'\n", optarg);
26
27             return 1;
28         }
29         argc -= optind;
30         argv += optind;
31
32         if(thick)
33             puts("Thick crust!");
34
35         if(delivery[0])
36             printf("To be delivered %s.\n", delivery);
37
38         puts("Ingredients: ");
39
40         for(count=0;count<argc; count++) //keep looping while we're less than argc
41             puts(argv[count]);
42
43         return 0;
44     }

```

This section includes the necessary header files and declares some variables. `argc` is the number of command line arguments, and `argv` is an array of strings containing those arguments.

`delivery` will hold the delivery option, `thick` is a flag indicating whether the crust should be thick or not, `count` is a counter for the loop, and `ch` is a temporary variable for storing the current option being processed.

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

int main(int argc, char *argv[])
{
    char *delivery;
    int thick = 0;
    int count = 0;
    char ch;
```

This is the main loop that processes the command line options.

The getopt() function reads the options and their arguments from argv, and sets ch to the current option character.

The switch statement processes each option, setting delivery if the -d option is given, and setting thick if the -t option is given.

If an unknown option is encountered, the program prints an error message and exits.

```
while ((ch = getopt(argc, argv, "d:t")) != EOF)
{
    switch(ch)
    {
        case 'd':
            delivery = optarg;
            break;
        case 't':
            thick = 1;
            break;
        default:
            fprintf(stderr, "Unknown option: '%s'\n", optarg);
            return 1;
    }
}
```

After processing the options, argc and argv are adjusted to remove the options and their arguments.

optind is the index of the next argument to be processed, and subtracting it from argc and advancing argv discards the options and leaves only the remaining arguments.

```
argc -= optind;
argv += optind;
```

Finally, the program prints out the results of the options processing.

If the **-t** option was given, it prints "Thick crust!".

If the **-d** option was given, it prints the value of **delivery**.

Then, it prints "Ingredients:" and loops over the remaining arguments in argv, printing

each one.

Overall, this program demonstrates how to use command line options to control the behavior of a program.

It reads the options using getopt(), sets variables based on the options, and then uses those variables to control the output of the program.

```
if(thick)
    puts("Thick crust!");

if(delivery)
    printf("To be delivered %s.\n", delivery);

puts("Ingredients: ");

for(count=0;count<argc; count++)
    puts(argv[count]);

return 0;
}
```

Running the program:

```
C:\Users\HP-EliteBook\Downloads\C\getOptPizza\bin\Debug>getOptPizza.exe -d
getOptPizza.exe: option requires an argument -- d
Unknown option: '(null)'

C:\Users\HP-EliteBook\Downloads\C\getOptPizza\bin\Debug>getOptPizza.exe -t
Thick crust!

C:\Users\HP-EliteBook\Downloads\C\getOptPizza\bin\Debug>getOptPizza.exe -d 1
To be delivered 1.

Ingredients:

C:\Users\HP-EliteBook\Downloads\C\getOptPizza\bin\Debug>getOptPizza.exe -d brocolli
To be delivered brocolli.

Ingredients:
brocolli

C:\Users\HP-EliteBook\Downloads\C\getOptPizza\bin\Debug>getOptPizza.exe -d brocolli pineapple cakes sodas
To be delivered brocolli.

Ingredients:
pineapple
cakes
sodas
```

You can combine the commandline options.

Yes, it is possible to combine options like `-td` instead of typing them separately as `-d` and `-t`. The `getopt()` function will handle this automatically.

The order in which options are specified also doesn't matter as `getopt()` will read them and process them in the order they appear.

However, if a value on the command line begins with `"-"`, `getopt()` will treat it as an option.

To avoid ambiguity, the programmer can use `--` to separate the main arguments from the options.

For example, if the command is `set_temperature -c -4`, you can use `set_temperature -c -- -4` to make sure that `getopt()` will treat `-4` as an argument and not an option.

In the program above, the use of `getopt()` allows the user to specify options like `-d` and `-t`.

The program uses the `while` loop to read the options and `switch` statement to process each option.

By using `getopt()`, the program can easily handle different combinations of options and

arguments, making it more versatile and user-friendly.

SUMMARY

The `main()` function in C programming language can have two versions - one with command-line arguments, and one without. When command-line arguments are passed to `main()`, they come in the form of an argument count and an array of pointers to the argument strings. Command-line options are arguments that begin with a hyphen (-) and are used to modify the behavior of the program.

To handle command-line options, C provides the `getopt()` function, which helps in parsing the command-line arguments. You can specify the valid options for the program by passing a string to `getopt()`, like "ae:". The colon following an option in the string means that the option takes an additional argument. `getopt()` records the options argument using the `optarg` variable.

After reading all the options using `getopt()`, you should skip past them using the `optind` variable. This allows you to access the remaining command-line arguments which were not options. It's important to note that the order of the options doesn't matter, and you can even combine them into a single argument. However, to avoid ambiguity when passing negative numbers, you can split your main arguments from the options using "--".

It works! Well, you've learned a lot in this chapter. You got deep into the Standard Input, Standard Output, and Standard Error.

You learned how to talk to files using redirection and your own custom data streams. Finally, you learned how to deal with command-line arguments and options.

A lot of C programmers spend their time creating small tools, and most of the small tools you see in operating systems like Linux are written in C.

If you're careful in how you design them, and if you make sure that you design tools that do one thing and do that one thing well, you're well on course to becoming a kick-ass C coder.

C functions like `printf()` and `scanf()` use the Standard Output and Standard Input to communicate.

The Standard Output goes to the display by default.

The Standard Error is a separate output intended for error messages.

The Standard Input reads from the keyboard by default.

You can print to the Standard Error using `fprintf(stderr, ...)`.

You can change where the Standard Input, Output, and Error are connected to using redirection.

Command-line arguments are passed to `main()` as an array of string pointers.

You can create custom data streams with `fopen("filename", mode)`.

The mode can be "w" to write, "r" to read, or "a" to append.

The `getopt()` function makes it easier to read command-line options.

Data Types

If you create a big program, you don't want a big source file. Can you imagine how difficult and time-consuming a single source file for an enterprise-level program would be to maintain?

In this chapter, you'll learn how C allows you to break your source code into small, manageable chunks and then rebuild them into one huge program. Along the way, you'll learn a bit more about data type subtleties and get to meet your new best friend: `make`.

C can handle quite a few different types of data: characters and whole numbers, floating-point values for everyday values, and floating-point numbers for really precise scientific calculations.

You can see a few of these data types listed on the opposite page. See if you can figure out which data type was used in each example. Remember: each example uses a different data type.

These are numbers containing decimal points.

Floating Points

float
double

Integers

short
int
long
char

That's right! In C, chars are actually stored using their character codes. That means they're just numbers too!

DATATYPES

1. Char: In computer memory, every character is stored as a character code, which is just a number. For example, when the computer sees the letter A, it actually sees the literal number 65. In programming, the char data type is used to represent these character codes. It takes up 1 byte of memory and can hold values from -128 to 127 or 0 to 255, depending on whether it's signed or unsigned. Char is commonly used to store individual characters, such as letters, digits, or punctuation marks.

2. Int: If you need to store a whole number, the int data type is typically used. The exact maximum size of an int can vary depending on the system, but it's guaranteed to be at least 16 bits. In general, an int can store values up to a few million. Integers can be either signed or unsigned, meaning they can hold both positive and negative values or only positive values, respectively. Ints are commonly used for counting, indexing, and performing arithmetic operations.

3. Long: Sometimes, an int is not large enough to hold the value you need to store. In those cases, the long data type is used. On some systems, a long takes up twice the

memory of an int, and it can hold values up in the billions. However, because most computers can handle really large integers, on some machines, the long data type is exactly the same size as an int. The maximum size of a long is guaranteed to be at least 32 bits. Longs are commonly used for storing large numbers or counts.

4. Float: For basic floating-point numbers, such as those used to represent everyday values like the amount of fluid in a drink, the float data type is used. Floats take up 4 bytes of memory and can hold values with up to about 7 decimal places of precision. Floats are commonly used for scientific calculations and simulations.

5. Double: In some cases, greater precision is needed than what a float can provide. For those cases, the double data type is used. Doubles take up twice the memory of a float and can hold values with up to about 15 decimal places of precision. Doubles are commonly used for financial calculations, 3D graphics, and other applications that require high levels of accuracy.

6. Short: The short data type in computer programming is used to store integer values that require less memory than an int. A short typically takes up 2 bytes of memory ranging from -32,768 to 32,767. It is commonly used in situations where memory is a concern, such as an embedded system or when dealing with large arrays. When assigning value to a short variable, the value should be within the range of the datatype to avoid unexpected bugs. When performing arithmetic operations on a short, the result will automatically be promoted to an int, which can lead to an overflow if the result is not explicitly cast back to a short.

Representations of these datatypes:

```
12 void DataRepresentations()
13 {
14     char : %c
15     int : %d or %i
16     long : %ld or %li
17     float : %f
18     double : %lf or %e
19     short: %h or %hd
20     short int: %hi
21 }
```

CONTINUED....

When working with values in a program, it's important to ensure that the data type of the value matches the data type of the variable you plan to store it in.

This is because different data types require different amounts of memory.

Trying to store a value that's too large for the memory allocated to a variable can cause errors in your code.

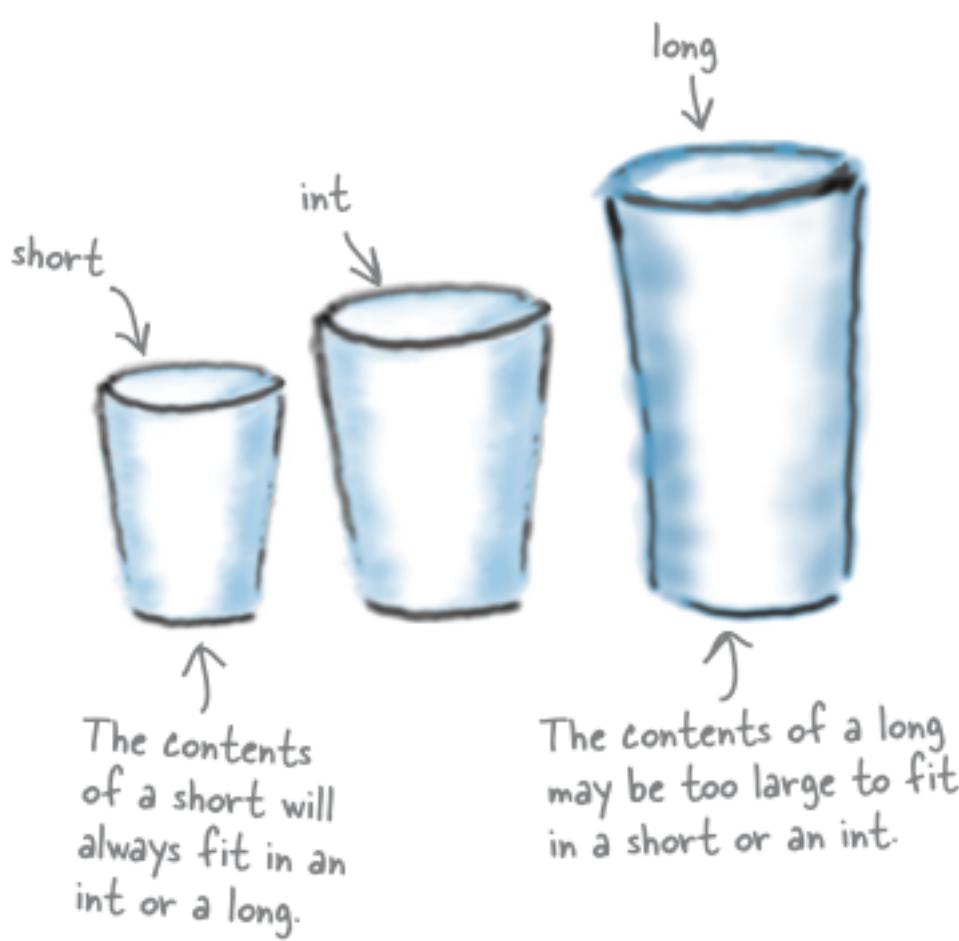
In general, short variables require less memory than ints, and ints require less memory than longs.

However, it's always safe to store a short value inside an int or a long variable.

This is because there is always enough memory available, and your code will function correctly without any issues.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     short x = 15;
7     int y = x;
8     printf("The value of y is: %i\n", y);
9     return 0;
10 }
```

The contents of a short will always be able to fit inside an int and a long.



The problems start to happen if you go the other way around—if, say, you try to store an int value into a short.

Sometimes, the compiler will be able to spot that you're trying to store a really big value into a small variable, and then give you a warning.

But a lot of the time the compiler won't be smart enough for that, and it will compile the code without complaining.

In that case, when you try to run the code, the computer won't be able to store a number 100,000 into a short variable.

The computer will fit in as many 1s and 0s as it can, but the number that ends up stored inside the y variable will be very different from the one you sent it:

```
23 void CompilerComplain()
24 {
25     int x = 100000;
26     short y = x;
27     printf("The value of y is: %hi\n", y);
28 }
```

When a large number is stored in a data type that can only hold a smaller range of values, the number is **truncated or shortened to fit within the available memory**.

In the case of storing a large number in a short, only the rightmost bits of the number are stored, discarding the leftmost bits that exceed the capacity of the short.

In the given example, the binary representation of the number 100,000 is too large to be stored in a short.

The program only stores the rightmost 16 bits of the number, which happen to represent the value -31072 in two's complement notation.

In two's complement notation, the leftmost bit is used to represent the sign of the number, where 1 represents negative and 0 represents positive.

Since the leftmost bit of the truncated number is 1, it represents a negative value.

This is why the value appears as a negative number (-31072) when it is printed out.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     short x = 15;
7     int y = x;
8     printf("The value of y is: %i\n", y);
9     CompilerComplain();
10    return 0;
11 }
12
13 "C:\Users\HP-EliteBook\Downloads\C\DATATYPES 1\bin\Debug\DATATYPES 1.exe"
14 The value of y is: 15
15 The value of y is: -31072
```

USING CASTING TO PUT FLOAT NUMBERS INTO WHOLE NUMBERS.

What do you think this piece of code will display?

```
32 void Casting()
33 {
34     int x = 7;
35     int y = 2;
36     float z = x/y;
37     printf("z = %f\n", z);
38 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     short x = 15;
7     int y = x;
8     printf("The value of y is: %i\n", y);
9     CompilerComplain();
10    Casting();
11    return 0;
12 }
13
14
```

The value of y is: 15
The value of y is: -31072
z = 3.000000

Casting is a technique in programming that allows you to convert one data type to another.

If you want to put a float into a whole number, you can use casting to convert the float to an integer data type.

In C programming, you can use the (int) casting operator to do this. For example:

```
float myFloat = 3.14;
int myInt = (int) myFloat;
```

In this example, the float value of 3.14 is cast to an integer, resulting in the value of 3 being stored in the variable myInt.

Note that when you cast a float to an int, the decimal portion of the float is truncated (not rounded), so 3.9 would become 3, and -3.9 would become -3.

It's also important to be aware of the range of the integer data type you're using, as casting a large float value to an integer may result in overflow and unexpected behavior.

CONTINUED....

The answer? 3.0000. Why is that? Well, x and y are both integers, and if you divide integers you always get a rounded-off whole number—in this case, 3.

What do you do if you want to perform calculations on whole numbers and you want to get floating-point results?

You could store the whole numbers into float variables first, but that's a little wordy.

Instead, you can use a cast to convert the numbers on the fly:

```
41 void Casting2()
42 {
43     int x = 7;
44     int y = 2;
45     float z = (float)x / (float)y;
46     printf("z = %f\n", z);
47 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     short x = 15;
7     int y = x;
8     printf("The value of y is: %i\n", y);
9     CompilerComplain();
10    Casting();
11    Casting2();
12    return 0;
13 }
14
15 "C:\Users\HP-EliteBook\Downloads\C\DATATYPES 1\bin\Debug\DATATYPES 1.exe"
16 The value of y is: 15
17 The value of y is: -31072
18 z = 3.000000
19 z = 3.500000
```

The (float) will cast an integer value into a float value.

The calculation will then work just as if you were using floating-point values the entire time.

In fact, if the compiler sees you are adding, subtracting, multiplying, or dividing a floating-point value with a whole number, it will automatically cast the numbers for you.

That means you can cut down the number of explicit casts in your code:

```
42 void Casting2()
43 {
44     int x = 7;
45     int y = 2;
46     float z = (float)x / (float)y;
47     float automaticCast = (float)x / y; //compiler autocasts this...
48     printf("z = %f\n", z);
49 }
```

```
float z = (float)x / y;
```

The compiler will automatically cast y to a float.

You can put some other keywords before data types to change the way that the numbers are interpreted:

Unsigned. The number will always be positive. Because it doesn't need to worry about recording negative numbers, unsigned numbers can store larger numbers since there's now one more bit to work with. So an unsigned int stores numbers from 0 to a maximum value that is about twice as large as the maximum number that can be stored inside an int. There's also a **signed keyword**, but you almost never see it, because all data types are signed by default.

Long. That's right, you can prefix a data type with the word long and make it longer. So a long int is a longer version of an int, which means it can store a larger range of numbers. And a **long long** is longer than a long. You can also use long with floating-point numbers.

```
--  
52  unsigned char c;  
53  
54  long double b;
```

```
unsigned char c;
```

Chapter 4

This will probably store numbers from 0 to 255.

```
    long double d;
```

A really REALLY precise number.

long long is only in C11 and C99.

EXERCISE

There's a new program helping the waiters bus tables at the Head First Diner. The code automatically totals a bill and adds sales tax to each item.

See if you can figure out what needs to go in each of the blanks.

Note: there are several data types that could be used for this program, but which would you use for the kind of figures you'd expect?

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  float total = 0.0;
6  short count = 0;
7  short tax_percent = 6;
8
9  float add_with_tax(float f) //return a small cash value which is a float.
10 {
11     float tax_rate = 1 + tax_percent/ 100.0;
12     total = total + (f * tax_rate);
13     count = count + 1;
14 }
15 }
```

```
16
17  int main()
18 {
19     float val;
20     printf("Price of item: ");
21     while(scanf("%f", &val) == 1)
22     {
23         printf("Total so far: %2f\n", total);
24         printf("Number of items: %hi\n", count);
25         return 0;
26     }
27 }
28 }
29 }
```

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

```

float total = 0.0;
short count = 0;
short tax_percent = 6;

float add_with_tax(float f) //return a small cash value which is a
float.
{
    float tax_rate = 1 + tax_percent/ 100.0;
    total = total + (f * tax_rate);
    count = count + 1;
    return total;
}

int main()
{
    float val;
    printf("Price of item: ");
    while (scanf("%f", &val) == 1)
    {
        printf("Total so far: %2f\n", total);
        printf("Number of items: %hi\n", count);
        return 0;
    }
}

```

Explained:

```

#include <stdio.h>
#include <stdlib.h>

```

These are **include** statements, which allow the program to use standard input/output and other functionality provided by the C standard library.

```

float total = 0.0;
short count = 0;
short tax_percent = 6;

```

These are variable declarations. total is initialized to 0.0, count is initialized to 0, and tax_percent is initialized to 6.

```
float add_with_tax(float f)
{
    float tax_rate = 1 + tax_percent/ 100.0;
    total = total + (f * tax_rate);
    count = count + 1;
    return total;
}
```

This is a function definition for add_with_tax. It takes a single argument f which is a float.

The function calculates a tax rate based on the tax_percent variable, adds the tax to the total, increments the count variable, and returns the new total.

```
int main()
{
    float val;
    printf("Price of item: ");
    while(scanf("%f", &val) == 1)
    {
        printf("Total so far: %2f\n", total);
        printf("Number of items: %hi\n", count);
        return 0;
    }
}
```

This is the main function of the program. It declares a variable val which will be used to store user input.

It then enters a while loop which repeatedly prompts the user for input using `printf`, reads the input using `scanf`, and displays the current total and count using `printf`.

The loop will continue until the `scanf` function returns something other than 1, which indicates an error or end of input.

Finally, the program returns 0 to indicate successful completion.

However, the return statement is misplaced in the loop and should be moved outside the loop to avoid premature termination of the program.

You need a small floating-point number to total the cash.

```
#include <stdio.h>
float total = 0.0;
short count = 0;
short tax_percent = 6;

float add_with_tax(float f); ← We're returning a small cash value, so it'll be a float.

A float will be OK for this fraction. → float tax_rate = 1 + tax_percent / 100.0;
total = total + (f * tax_rate);
count = count + 1;
return total;
}

int main()
{ ← Each price will easily fit in a float.
float val;
```

There won't be many items on an order, so we'll choose a short.

By adding .0, you make the calculation work as a float. If you left it as 100, it would have returned a whole number.

$1 + \text{tax_percent} / 100$ would return the value 1 because $6 / 100 == 0$ in integer arithmetic.

Data types are different sizes on different platforms.

But how do you find out how big an `int` is, or how many bytes a `double` takes up?

Fortunately, the C Standard Library has a couple of headers with the details.

This program will tell you about the sizes of `ints` and `floats`:

```
++++++
```

Programming is about thinking, not about typing.

we program to advance the currently existing technologies. Current tech applies the new languages that are simpler to use therefore problems can be solved much quickly and have less errors, unlike older tech like assembly.

programming helps simplify

Requires a lot of time to understand and apply a single language.

```
++++++
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <limits.h> //contains the values for integer types like int and char
4 #include <float.h> //contains the values for floats and doubles.
5
6 int main()
7 {
8     printf("The value of INT_MAX is %i\n", INT_MAX);
9     printf("The value of INT_MIN is %i\n", INT_MIN);
10    printf("An int takes %z bytes\n", sizeof(int));
11
12    printf("The value of FLT_MAX is %f\n", FLT_MAX);
13    printf("The value of FLT_MIN is %.50f\n", FLT_MIN);
14    printf("A float takes %z bytes\n", sizeof(float)); //returns the number of bytes
15
16    return 0;
17 }
```

The values you see on your particular machine will probably be different. What if you want to know the details for chars or doubles? Or longs?

No problem. Just replace INT and FLT with CHAR (chars), DBL (doubles), SHRT (shorts), or LNG (longs).

Why are data types different on different operating systems? Wouldn't it be less confusing to make them all the same?

C uses different data types on different operating systems and processors because it allows the language to make the most out of the hardware.

In what way?

When C was first created, most machines were 8-bit. Now, most machines are 32- or 64-bit.

Because C doesn't specify the exact size of its data types, it's been able to adapt over time.

And as newer machines are created, C will be able to make the most of them as well.

What do 8-bit and 64-bit actually mean?

Technically, the bit size of a computer can refer to several things, such as the size of its CPU instructions or the amount of data the CPU can read from memory.

The bit size is really the favored size of numbers that the computer can deal with.

So what does that have to do with the size of ints and doubles?

If a computer is optimized best to work with 32-bit numbers, it makes sense if the basic data type—the int—is set at 32 bits.

Floats and doubles are stored in a more complicated manner.

Most computers use a standard published by the IEEE to represent floating-point numbers.

I understand how whole numbers like ints work, but how are floats and doubles stored? How does the computer represent a number with a decimal point?

It is not necessary to understand how floating-point numbers work to use floats and doubles as a developer.

The vast majority of developers use floats and doubles without worrying about the details.

OUT OF WORK ACTORS(FUNCTION DECLARATIONS)

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 float total = 0.0;
5 short count = 0;
6 //this is 6%. Which is a lot less than my agent takes.
7 short tax_percent = 6;
8
9
10 int main()
11 {
12     //hey-i was up for a movie with Val Kilmer.
13     float val;
14     printf("Price of item: ");
15     while (scanf("%f", &val) == 1)
16     {
17         printf("Total so far: %2f\n", total); //float up to 2 decimals
18         printf("Number of items: %hi\n", count);
19         return 0;
20     }
21 }
22

```

```

24 float add_with_tax(float f) //why are we passing it as a parameter??
25 {
26     float tax_rate = 1 + tax_percent/100.0;
27     //and what about the tip? Voice lessons ain't free
28     total = total + (f * tax_rate);
29     count = count + 1;
30     return total;
31 }

```

This code runs, but its not the code from the book, i made a mistake, but it ran anyways. Now, the code from the book:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 float total = 0.0;
5 short count = 0;
6 //this is 6%. Which is a lot less than my agent takes.
7 short tax_percent = 6;
8
9
10 int main()
11 {
12     //hey-i was up for a movie with Val Kilmer.
13     float val;
14     printf("Price of item: ");
15     while (scanf("%f", &val) == 1)
16     {
17         printf("Total so far: %.2f\n", add_with_tax(val)); //float up to 2 decimals
18         printf("Number of items: ");
19         return 0;
20     }
21
22     printf("\n Final total: %.2f\n", total);
23     printf("Number of items: %hi\n", count);
24     return 0;
25 }
26
27 float add_with_tax(float f) //why are we passing it as a parameter??
28 {
29     float tax_rate = 1 + tax_percent/100.0;
30     //and what about the tip? Voice lessons ain't free
31     total = total + (f * tax_rate);
32     count = count + 1;
33     return total;
34 }
```

Error messages:

```

27 float add_with_tax(float f)//why are we passing it as a parameter??
28 {
29     float tax_rate = 1 + tax_percent/100.0;
30     //and what about the tip? Voice lessons ain't free
31     total = total + (f * tax_rate);
32     count = count + 1;
33     return total;

```

Logs & others

Code::Blocks X Search results X Ccc X Build log X Build messages X CppCheck/Vera++ X CppCheck/Vera...

File	Line	Message
C:\Users\HP-E...	17	warning: format '%f' expects argument of type 'double', but argument 2 has type 'int' ...
C:\Users\HP-E...	17	warning: format '%f' expects argument of type 'double', but argument 2 has type 'int' ...
C:\Users\HP-E...	27	error: conflicting types for 'add_with_tax'
C:\Users\HP-E...	17	note: previous implicit declaration of 'add_with_tax' was here ==== Build failed: 1 error(s), 3 warning(s) (0 minute(s), 0 second(s)) ===

The output 2:

If you open up the console and try to compile the program, this happens:

```

File Edit Window Help SlickToActing
> gcc totaller.c -o totaller && ./totaller
totaller.c: In function "main":
totaller.c:14: warning: format "% .2f" expects type
"double", but argument 2 has type "int"
totaller.c: At top level:
totaller.c:23: error: conflicting types for "add with tax"
totaller.c:14: error: previous implicit declaration of
"add_with_tax" was here

```

That's not good.

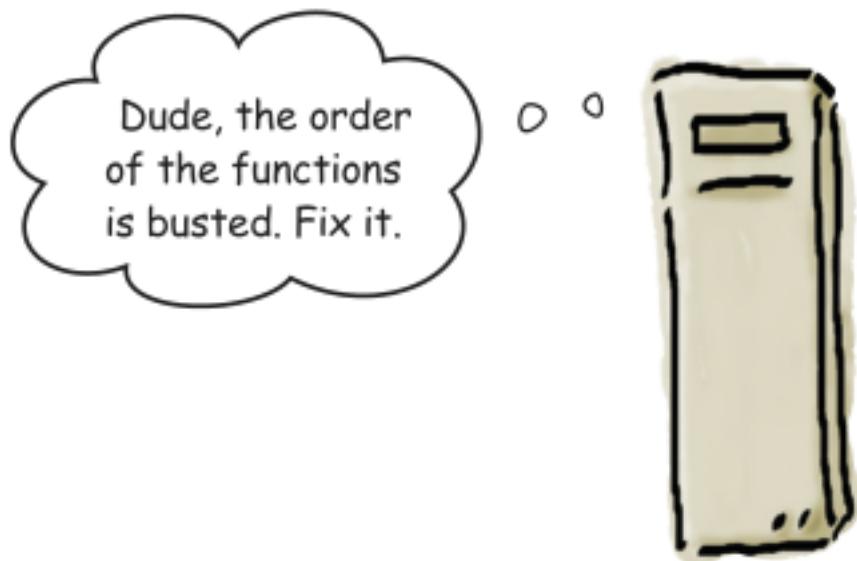
What does error: conflicting types for 'add_with_tax' mean? What is a previous implicit declaration?

And why does it think the line that prints out the current total is now an int? Didn't we design that to be floating point?

The compiler will ignore the changes made to the comments, so that shouldn't make any difference.

That means the problem must be caused by changing the order of the functions.

But if the order is the problem, why doesn't the compiler just return a message saying something like:



Compilers don't like surprises. So what happens when the compiler sees this line of code?

```
printf("Total so far: %.2f\n", add_with_tax(val));
```

The compiler sees a call to a function it doesn't recognize.

Rather than complain about it, the compiler figures that it will find out more about the function later in the source file.

The compiler simply remembers to look out for the function later on in the file. Unfortunately, this is where the problem lies...

The compiler needs to know what data type the function will return.

Of course, the compiler can't know what the function will return just yet, so it makes an assumption.

The compiler assumes it will return an int.

I bet the function returns an int. Most do.

When it reaches the code for the actual function, it returns a "conflicting types for add_with_tax" error.

This is because the compiler thinks it has two functions with the same name.

One function is the real one in the file.

The other is the one that the compiler assumed would return an int.

A function called add_with_tax() that returns a float???

But in my notes it says we've already got one of these returning an int.

The computer makes an assumption that the function returns an int, when in reality it returns a float.

If you were designing the C language, how would you fix the problem?

You could just put the functions back in the correct order and define the function before you call it in main().

Changing the order of the functions means that you can avoid the compiler ever making any dangerous assumptions about the return types of unknown functions.

But if you force yourself to always define functions in a specific order, there are a couple of consequences.

Hello? I really don't care
how the C language solves the
problem. Just put the functions in
the correct freaking order!



Put the functions back in the correct order the function before you call it in main().

Put the functions means that you can avoid the any dangerous assumptions about the return types. But if you force yourself to always define functions there are a couple of consequences.

main

Fixing function order is a pain.

Say you've added a cool new function to your code that everyone thinks is fantastic:

```
36 int do_whatever()
37 {
38     ....
39 }
40
41 float do_something_fantastic(int awesome_level)
42 {
43     ....
44 }
45
46 int do_stuff()
47 {
48     do_something_fantastic(11);
49 }
50
```

What happens if you then decide your program will be even better if you add a call to the do_something_fantastic() function in the existing do_whatever() code?

You will have to move the function earlier in the file.

Most coders want to spend their time improving what their code can do.

It would be better if you didn't have to shuffle the order of the code just to keep the compiler happy.

In some situations, there is no correct order.

OK, so this situation is kind of rare, but occasionally you might write some code that is **mutually recursive**:

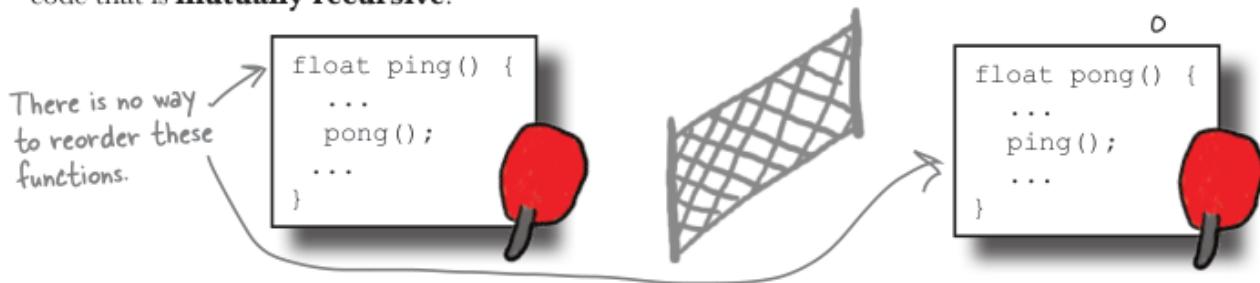
If you have **two functions that call each other**, then one of them will always be called in the file before it's defined.

For both of those reasons, it's really useful to be able to define functions in whatever order is easiest at the time. But how?

code just to keep the compiler happy.

In some situations, there is no correct order

OK, so this situation is kind of rare, but occasionally you might write some code that is **mutually recursive**:



Split the declaration from the definition Remember how the compiler made a note to itself about the function it was expecting to find later in the file?

You can avoid the compiler making assumptions by explicitly telling it what functions it should expect.

When you tell the compiler about a function, it's called a **function declaration**:

The declaration is just a function signature: a record of what the function will be called, what kind of parameters it will accept, and what type of data it will return.

Once you've declared a function, the compiler won't need to make any assumptions, so it won't matter if you define the function after you call it.

So if you have a whole bunch of functions in your code and you **don't want to worry about their order** in the file, you can **put a list of function declarations** at the start of your C program code:

The declaration tells the compiler what return value to expect. → `float add_with_tax();` ↗ A declaration has no body code.
It just ends with a ; (semicolon).

```
36  float do_something_fantastic();
37  double awesomeness_2_dot_0();
38  int stinky_pete();
39  char make_maguerita(int count);
40
41  int do_whatever()
42  {
43  }
44  ....
45
46  float do_something_fantastic(int awesome_level)
47  {
48  }
49  ....
50
51  int do_stuff()
52  {
53  }
54  do_something_fantastic(11);
```

FUNCTION DECLARATIONS IN HEADER FILES

But even better than that, C allows you to take that whole set of declarations out of your code and put them in a header file.

You've already used header files to include code from the C Standard Library:

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

This line will include the contents of the header file called stdio.h.

Declarations
don't have a
body.



Let's go see how you can create your own header files.

Creating your first header file To create a header, you just need to do two things:

Create a new file with a .h extension.

If you are writing a program called totaller, then create a file called totaller.h and write your declarations inside it.

```
float add_with_tax(float f);
```



totaller.h

t

You won't need to include the main() function in the header file, because nothing else will need to call it.

Include your header file in your main program. At the top of your program, you should add an extra include line:

Add this include to your other include lines.

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



totaller.c

at the name of the header file under main.c

When you write the name of the header file, make sure you surround it with double quotes rather than angle brackets.

Why the difference? When the compiler sees an include line with angle brackets, it assumes it will find the header file somewhere off in the directories where the library code lives.

But your **header file** is in the **same directory** as your **.c file**.

By wrapping the header filename in quotes, you are telling the compiler to look for a local file.

When the compiler reads the `#include` in the code, it will read the contents of the header file, just as if it had been typed into the code.

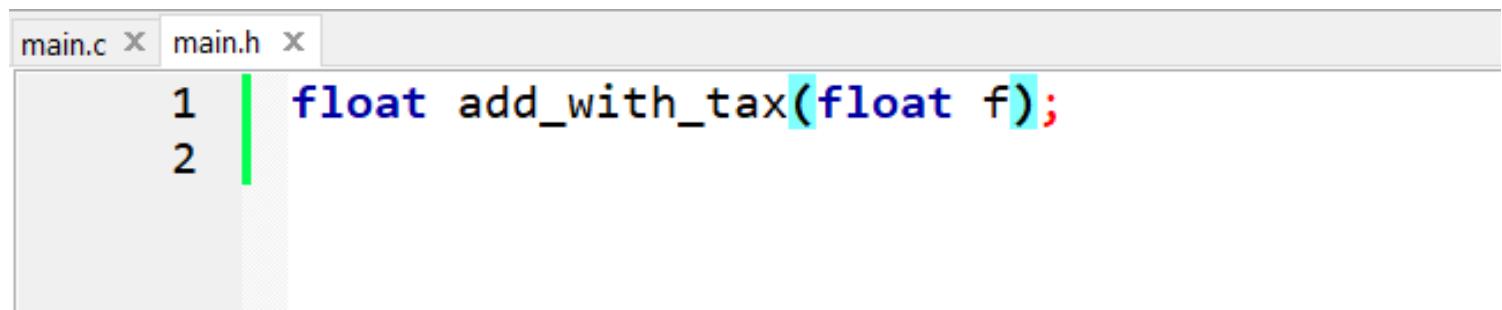
Separating the declarations into a separate header file keeps your main code a little shorter, and it has another big advantage that you'll find out about in a few pages.

For now, let's see if the header file fixed the mess.

- Local header files can also include directory names, but you will normally put them in the same directory as the C file.

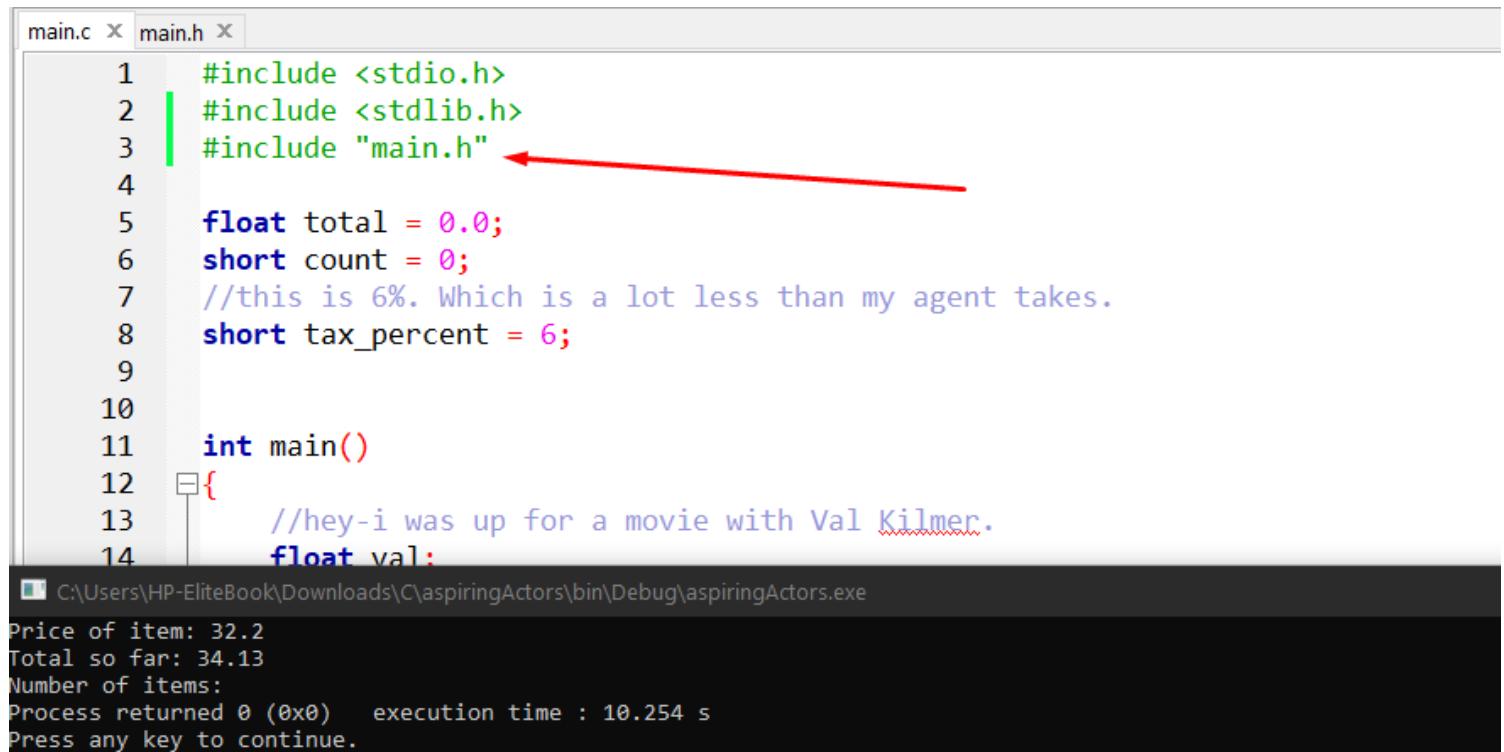
**#include is a
preprocessor
instruction.**

Header file with the declaration:



```
main.c x main.h x
1 float add_with_tax(float f);
2
```

After including this header file:

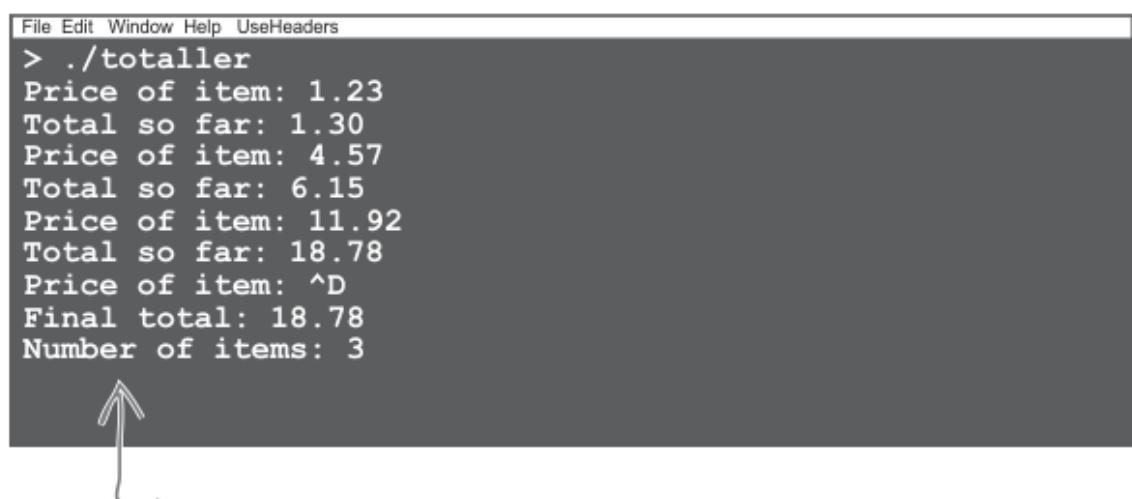


```
main.c x main.h x
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "main.h" ←
4
5 float total = 0.0;
6 short count = 0;
7 //this is 6%. Which is a lot less than my agent takes.
8 short tax_percent = 6;
9
10
11 int main()
12 {
13     //hey-i was up for a movie with Val Kilmer.
14     float val:
```

C:\Users\HP-EliteBook\Downloads\C\aspiringActors\bin\Debug\aspiringActors.exe

```
Price of item: 32.2
Total so far: 34.13
Number of items:
Process returned 0 (0x0)   execution time : 10.254 s
Press any key to continue.
```

The program will keep asking for more prices and items, so you will need to stop it yourself:



```
File Edit Window Help UseHeaders
> ./totaller
Price of item: 1.23
Total so far: 1.30
Price of item: 4.57
Total so far: 6.15
Price of item: 11.92
Total so far: 18.78
Price of item: ^D
Final total: 18.78
Number of items: 3
```



Press Ctrl-D here to stop the program from asking for more prices.

BE THE COMPILER.

Look at the program below. Part of the program is missing.

Your job is to play like you're the compiler and say what you would do if each of the candidate code fragments on the right were slotted into the missing space.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 float mercury_day_in_earth_days();
5 int hours_in_an_earth_day();
6
7 int main()
8 {
9     float length_of_day = mercury_day_in_earth_days();
10    int hours = hours_in_an_earth_day();
11    float day = length_of_day * hours; //mercury * earth
12    printf("A day on Mercury is %f hours\n", day);
13    return 0;
14 }
15 float mercury_day_in_earth_days()
16 {
17     return 58.65;
18 }
19
20 int hours_in_an_earth_day()
21 {
22     return 24;
23 }
```

Try these pieces of code:

```

float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
    float length_of_day = mercury_day_in_earth_days();
    int hours = hours_in_an_earth_day();
    float day = length_of_day * hours;

```



You can compile the code.



You should display a warning.



The program will work.

There will be a warning, because you haven't declared the `hours_in_an_earth_day()` before calling it. The program will still work because it will guess the function returns an int.

```

float mercury_day_in_earth_days();

int main()
{
    float length_of_day = mercury_day_in_earth_days();
    int hours = hours_in_an_earth_day();
    float day = length_of_day * hours;

```



You can compile the code.



You should display a warning.



The program will work.

Try these ones too:

```

int main()
{
    float length_of_day = mercury_day_in_earth_days();
    int hours = hours_in_an_earth_day();
    float day = length_of_day * hours;

```

The program won't compile, because you're calling a float function without declaring it first.



You can compile the code.



You should display a warning.



The program will work.

The program will compile without warnings, but it won't work because there will be a rounding problem.

```

float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
    int length_of_day = mercury_day_in_earth_days();
    int hours = hours_in_an_earth_day();
    float day = length_of_day * hours;

```

The `length_of_day` variable should be a float.



You can compile the code.



You should display a warning.



The program will work.

So I don't need to have declarations for int functions? Not necessarily, unless you are sharing code. You'll see more about this soon.

I'm confused. You talk about the compiler preprocessing? Why does the compiler do that? Strictly speaking, the compiler just does the compilation step: it converts the

C source code into assembly code. But in a looser sense, all of the stages that convert the C source code into the final executable are normally called compilation, and the gcc tool allows you to control those stages. The gcc tool does preprocessing and compilation.

What is the preprocessor? Preprocessing is the first stage in converting the raw C source code into a working executable. Preprocessing creates a modified version of the source just before the proper compilation begins. In your code, the preprocessing step read the contents of the header file into the main file.

Does the preprocessor create an actual file? No, compilers normally just use pipes for sending the stuff through the phases of the compiler to make things more efficient.

Why do some headers have quotes and others have angle brackets? Strictly speaking, it depends on the way your compiler works. Usually quotes mean to simply look for a file using a relative path. So if you just include the name of a file, without including a directory name, the compiler will look in the current directory. If angle brackets are used, it will search for the file along a path of directories.

What directories will the compiler search when it is looking for header files? The gcc compiler knows where the standard headers are stored. On a Unix- style operating system, the header files are normally in places like /usr/local/include, /usr/include, and a few others.

So that's how it works for standard headers like stdio.h? Yes. You can read through the stdio.h file on a Unix-style machine in /usr/include/stdio.h. If you have the MinGW compiler on Windows, it will probably be in C:\MinGW\include\stdio.h.

Can I create my own libraries? Yes; you'll learn how to do that later in the book.

SUMMARY

If the compiler finds a call to a function it hasn't heard of, it will assume the function returns an int.

So if you try to call a function before you define it, there can be problems.

Function declarations tell the compiler what your functions will look like before you define them.

If function declarations appear at the top of your source code, the compiler won't get confused about return types.

Function declarations are often put into header files.

You can tell the compiler to read the contents of a header file using `#include`.

The compiler will treat included code the same as code that is typed into the source file.

RESERVED WORDS(C IS PRETTY SMALL)

C is a very small language. Here is the entire set of reserved words (in no useful order).

Every C program you ever see will break into just these words and a few symbols.

If you use these for names, the compiler will be very, very upset.



This Table's Reserved...

C is a very small language. Here is the entire set of reserved words (in no useful order).

Every C program you ever see will break into just these words and a few symbols. If you use these for names, the compiler will be very, very upset.

auto	if	break
int	case	long
char	register	continue
return	default	short
do	sizeof	double
static	else	struct
entry	switch	extern
typedef	float	union
for	unsigned	goto
while	enum	void
const	signed	volatile

If you have common features.

Chances are, when you begin to write several programs in C, you will find that there are some functions and features that you will want to reuse from other programs.

For example, look at the specs of the two programs on the right.

XOR encryption is a very simple way of disguising a piece of text by XOR-ing each character with some value.

It's not very secure, but it's very easy to do.

And the same code that can encrypt text can also be used to decrypt it.

Here's the code to encrypt some text:

```
4 //void means return nothing
5 //message - give me the address and I'll work on the values inside there.
6 void encrypt(char *message)//pass a pointer to an array into the function
7 {
8     char c;
9     while (*message)//loop through array and update each character with encrypted version
10    {
11        *message = *message ^ 31; //XOR each character with number 31.
12        message++; //you can do math with char coz its a numeric type.
13    }
14 }
```

it's good to share code. Clearly, both of those programs are going to need to use the same `encrypt()` function.

So you could just copy the code from one program to the other, right?

That's not so bad if there's just a small amount of code to copy, but what if there's a really large amount of code?

Or what if the way the `encrypt()` function works needs to change in the future?

If there are two copies of the `encrypt()` function, you will have to change it in more than one place.

For your code to scale properly, you really need to find some way to reuse common pieces of code—some way of taking a set of functions and making them available in a bunch of different programs. How would you do that?

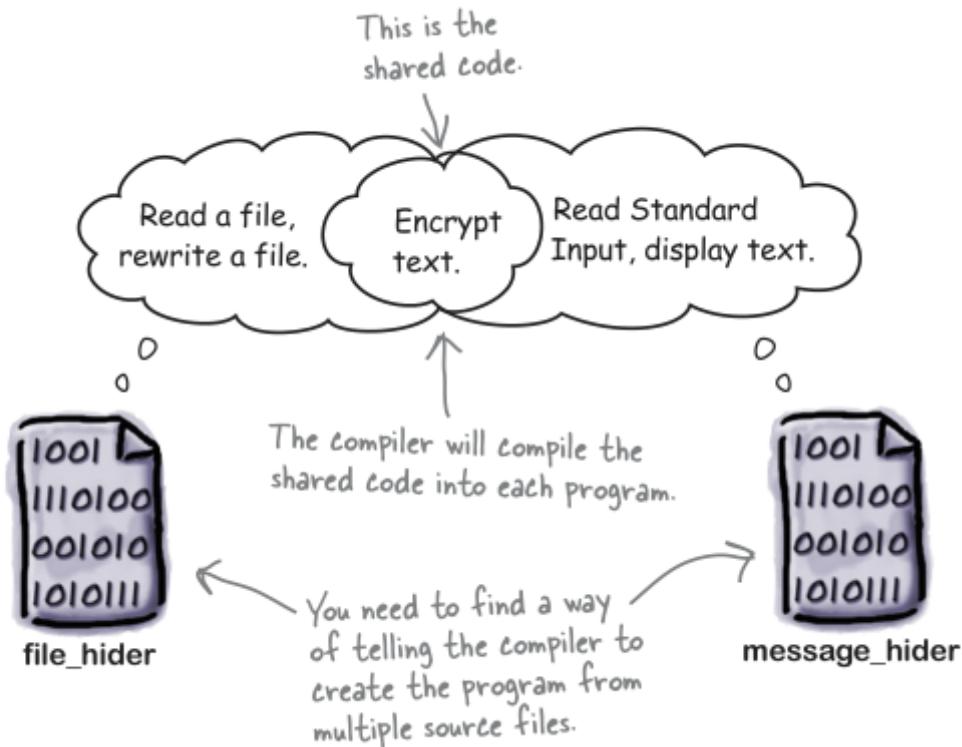


Imagine you have a set of functions that you want to share between programs. If you had created the C programming language, how would you allow code to be shared?

You can split the code into separate files. If you have a set of code that you want to share among several files, it makes a lot of sense to put that shared code into a separate .c file.

If the compiler can somehow include the shared code when it's compiling the program, you can use the same code in multiple applications at once.

So if you ever need to change the shared code, you only have to do it in one place.



If you want to use a separate .c file for the shared code, that gives us a problem.

So far, you have only created programs from single .c source files.

So if you had a C program called `blitz_hack`, you would have created it from a single source code file called `blitz_hack.c`.

But now you want some way to give the compiler a set of source code files and say, "Go make a program from those."

How do you do that? What syntax do you use with the gcc compiler?

And more importantly, what does it mean for a compiler to create a single executable program from several files? How would it work?

How would it stitch them together? To understand how the C compiler can create a

single program from multiple files, let's take a look at how compilation works.

COMPIRATION BEHIND THE SCENES

To understand how a compiler can compile several source files into a single program, you'll need to pull back the curtain and see how compilation really works.

Hmmmm... so I need to compile the source files into a program?

Let's see what I can cook up...

1. Preprocessing: fix the source.

The first thing the compiler needs to do is fix the source. It needs to add in any extra header files it's been told about using the `#include` directive.

It might also need to expand or skip over some sections of the program. Once it's done, the source code will be ready for the actual compilation.

Compilation behind the scenes

To understand how a compiler can compile several source files into a single program, you'll need to pull back the curtain and see how compilation really works.

1 Preprocessing: fix the source.

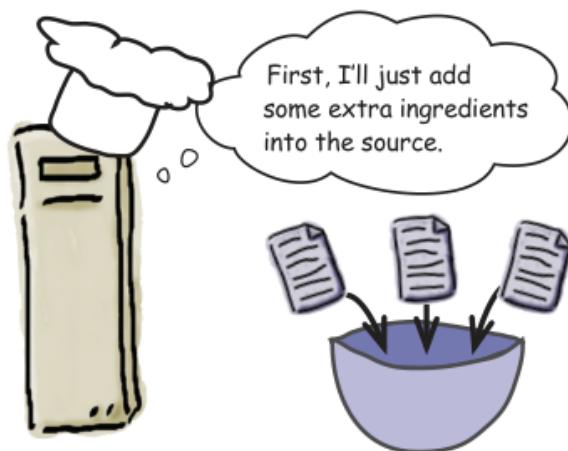
The first thing the compiler needs to do is fix the source. It needs to add in any extra header files it's been told about using the `#include directive`.

It might also need to expand or skip over some sections of the program. Once it's done, the source code will be ready for the actual compilation.

It can do this with commands like `#define` and `#ifdef`. You'll see how to use them later in the book.

First, I'll just add some extra ingredients into the source.

“directive” is just a fancy word for “command.”



It can do this with commands like `#define` and `#ifdef`.

You'll see how to use them later in the book. "directive" is just a fancy word for "command."

2. Compilation: translate into assembly.

The C programming language probably seems pretty low level, but the truth is it's not low level enough for the computer to understand.

The computer only really understands very low-level machine code instructions, and the first step to generate machine code is to convert the C source code into assembly language symbols like this:

```
asm
(
    movq -24(%rbp), %rax
    movzbl(%rax), %eax
    movl %eax, %edx
)
```

Looks pretty obscure?

Assembly language describes the individual instructions the central processor will have to follow when running the program.

The C compiler has a whole set of recipes for each of the different parts of the C language.

These recipes will tell the compiler how to convert an if statement or a function call into a sequence of assembly language instructions.

But even assembly isn't low level enough for the computer. That's why it needs...

3. Assembly: generate the object code.

The compiler will need to assemble the symbol codes into machine or object code.

This is the actual binary code that will be executed by the circuits inside the CPU.

This is a really
dirty joke in
machine code.

10010101 00100101 11010101 01011100



So are you all done? After all, you've taken the original C source code and converted it into the 1s and 0s that the computer's circuits need.

But no, there's still one more step.

If you give the computer several files to compile for a program, the compiler will generate a piece of object code for each source file.

But in order for these separate object files to form a single executable program, one more thing has to occur.

Time to bake that assembly into something edible.

4. Linking: put it all together.

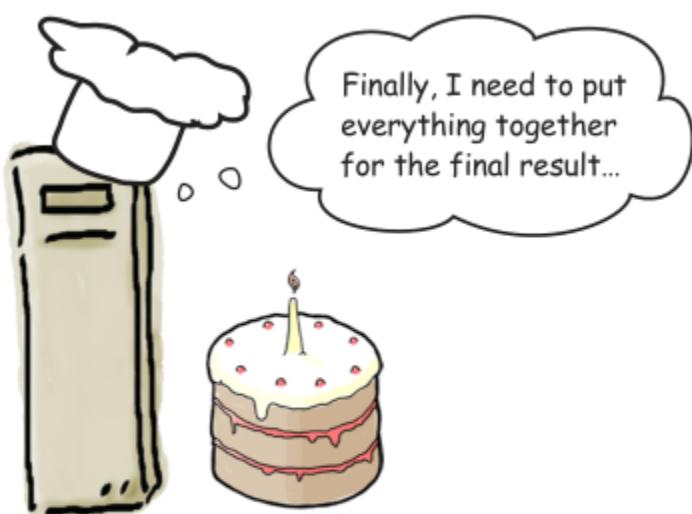
Once you have all of the separate pieces of object code, you need to fit them together like jigsaw pieces to form the executable program.

The compiler will connect the code in one piece of object code that calls a function in another piece of object code.

Linking will also make sure that the program is able to call library code properly.

Finally, the program will be written out into the executable program file using a format that is supported by the operating system.

The file format is important, because it will allow the operating system to load the program into memory and make it run.



So how do you actually tell gcc that we want to make one executable program from several separate source files?

The shared code needs its own header file If you are going to share the `encrypt.c` code between programs, you need some way to tell those programs about the encrypt

code.

You do that with a header file.

If you are going to share the *encrypt.c* code between programs, you need some way to tell those programs about the encrypt code. You do that with a header file.

```
void encrypt(char *message);
```



You'll include the header inside *encrypt.c*.

```
#include "encrypt.h"

void encrypt(char *message)
{
    char c;
    while (*message) {
        *message = *message ^ 31;
        message++;
    }
}
```



encrypt.c

Include encrypt.h in your program

You're not using a header file here to be able to reorder the functions. You're using it to **tell other programs about the encrypt() function**:

Include your *encrypt.h* file in your header declarations:

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

int main()
{
    char msg[80];
    while (fgets(msg, 80, stdin)) {
        encrypt(msg);
        printf("%s", msg);
    }
}
```



You'll include *encrypt.h* so that the program has the declaration of the *encrypt()* function.

Sharing

You've seen how to share code between different programs. You want to share code between different files in a program.

Having *encrypt.h* inside the main program will mean the compiler will know enough about the *encrypt()* function to compile the code.

At the linking stage, the compiler will be able to connect the call to `encrypt(msg)` in `message_hider.c` to the actual `encrypt()` function in `encrypt.h`.

Finally, to compile everything together you just need to pass the source files to `gcc`:

```
gcc message_hider.c encrypt.c -o message_hider.
```

SHARING VARIABLES

You've seen how to share functions between different files. But what if you want to share variables?

Source code files normally contain their own separate variables to prevent a variable in one file affecting a variable in another file with the same name.

But if you genuinely want to share variables, you should declare them in your header file and prefix them with the keyword `extern`: `extern int passcode;`

Let's see what happens when you compile the `message_hider` program:

Let's see what happens when you compile the `message_hider` program:

You need to compile the code with both source files.

```
File Edit Window Help Shhh...
> gcc message_hider.c encrypt.c -o message_hider
> ./message_hider
I am a secret message
V?~r?~?1z|mzk?rzll~xz
> ./message_hider < encrypt.h
ipv{?zq|mfok7|w~m5?rzll~xz6$
```

When you run the program, you can enter text and see the encrypted version.

You can even pass it the contents of the `encrypt.h` file to encrypt it.

The `message_hider` program is using the `encrypt()` function from `encrypt.c`.

The program works. Now that you have the `encrypt()` function in a separate file, you can use it in any program you like.

If you ever change the `encrypt()` function to be something a little more secure, you will need to amend only the `encrypt.c` file.

SUMMARY

You can share code by putting it into a separate *C* file.

You need to put the function declarations in a separate *.h* header file.

Include the header file in every *C* file that needs to use the shared code.

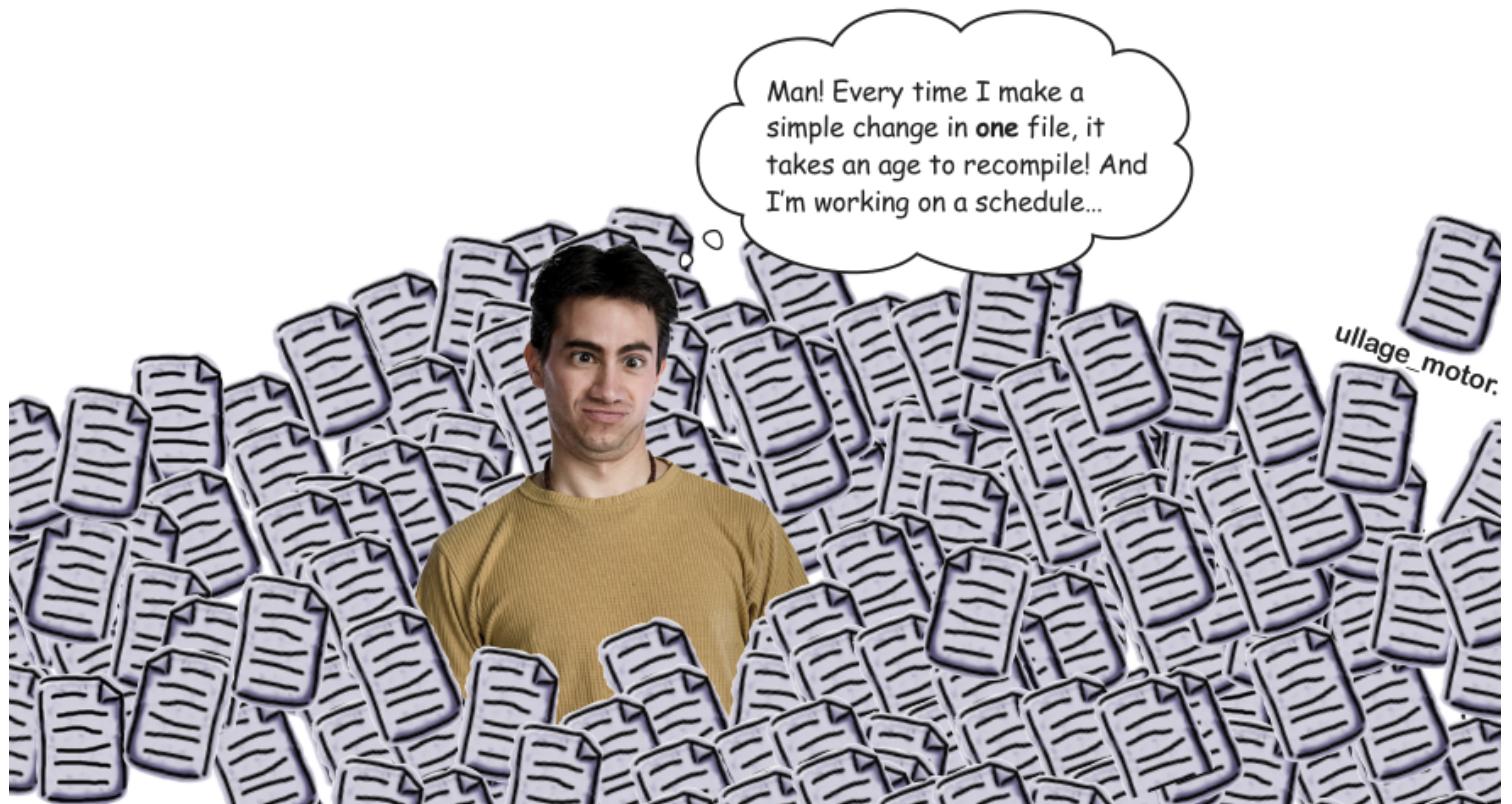
List all of the *C* files needed in the compiler command.



Go Off Piste

Write your own program using the `encrypt()` function. Remember, you can call the same function to decrypt text.

Man! Every time I make a simple change in one file, it takes an age to recompile! And I'm working on a schedule.



It's not rocket science, or is it?

Breaking your program out into separate source files not only means that you can share code between different programs, but it also means you can start to create really large programs.

Why? Well, because you can start to break your program down into smaller self-contained pieces of code.

Rather than being forced to have one huge source file, you can have lots of simpler files that are easier to understand, maintain, and test.

So on the plus side, you can start to create really large programs. The downside?

The downside is you can start to create really large programs.

C compilers are really efficient pieces of software. They take your software through some very complex transformations.

They can **modify your source, link hundreds of files together** without blowing your memory, and even optimize the code you wrote, along the way.

And even though they do all that, they still manage to run quickly.

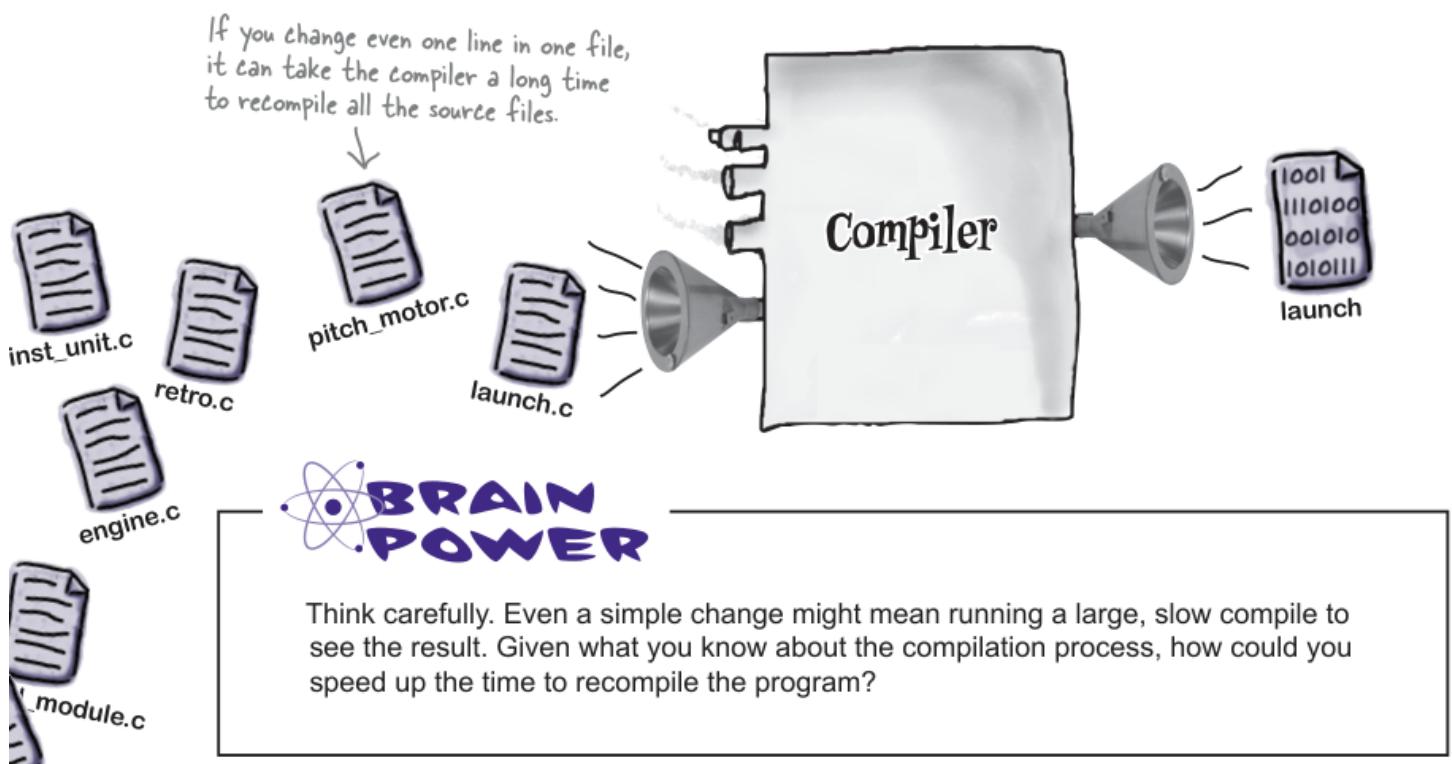
But if you create programs that use more than a few files, the time it takes to compile the code starts to become important.

Let's say it takes a minute to compile a large project.

That might not sound like a lot of time, but it's more than long enough to break your train of thought.

If you try out a change in a single line of code, you want to see the result of that change as quickly as possible.

If you have to wait a full minute to see the result of every change, that will really start to slow you down.



Think carefully. Even a simple change might mean running a large, slow compile to see the result.

Given what you know about the compilation process, how could you **speed up the time** to recompile the program?

DON'T RECOMPILE EVERY FILE

If you've just made a change to one or two of your source code files, it's a waste to recompile every source file for your program.

Think what happens when you issue a command like this:



gcc reaction_control.c pitch_motor.c ... engine.c -o launch

Skipping a few filenames here.

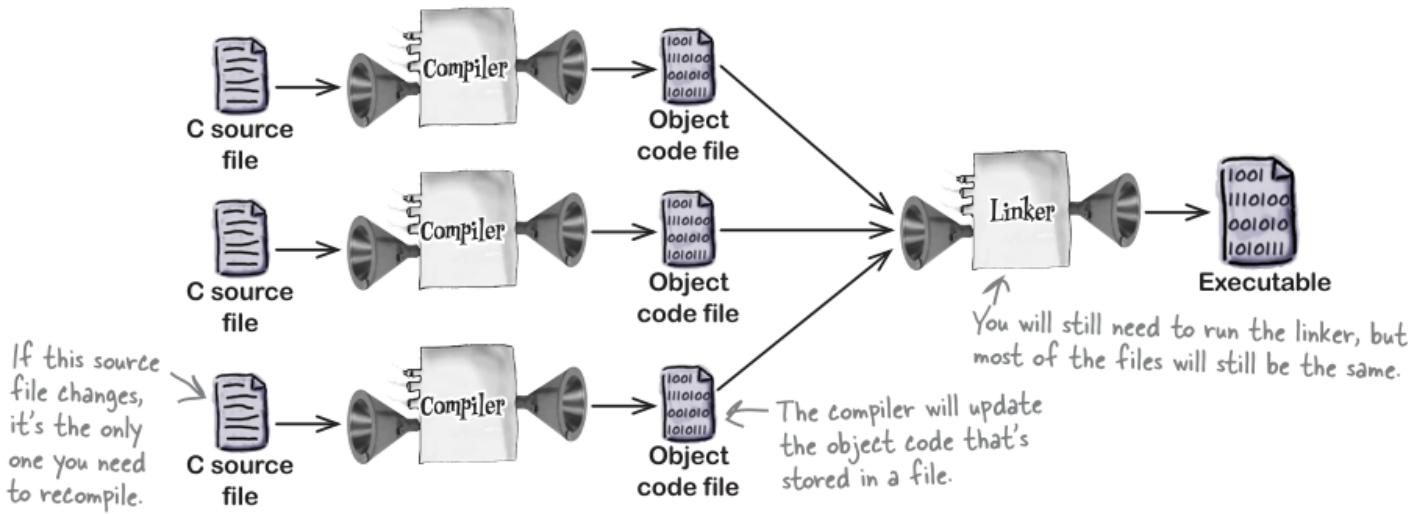
A hand-drawn arrow points from the text "Skipping a few filenames here." to the three dots (...).

What will the compiler do? It will run the preprocessor, compiler, and assembler for each source code file. Even the ones that haven't changed.

And if the source code hasn't changed, the object code that's generated for that file won't change either.

So if the compiler is generating the object code for every file, every time, what do you need to do?

If you change a single file, you will have to recreate the object code file from it, but you won't need to create the object code for any other file.



Then you can pass all the object code files to the linker and create a new version of the program.

So how do you tell `gcc` to save the object code in a file? And how do you then get the compiler to link the object files together?

Save copies of the compiled code If you tell the compiler to save the object code it generates into a file, it shouldn't need to recreate it unless the source code changes.

If a file does change, you can recreate the object code for that one file and then pass the whole set of object files to the compiler so they can be linked.

First, compile the source into object files.

You want object code for each of the source files, and you can do that by typing this command:

This will create object code for every file. $\rightarrow \text{gcc } -c \text{ *.c}$ The operating system will replace `*.c` with all the C filenames.

The `*.c` will match every C file in the current directory, and the `-c` will tell the compiler that you want to create an object file for each source file, but you don't want to link them together into a full executable program.

Now that you have a set of object files, you can **link them together** with a simple compile command. But instead of giving the compiler the names of the C source files, you tell it the names of the object files:

This is similar to the → `gcc *.o -o launch` Instead of C source files, list the object files.
compile commands you've used before. ↑ This will match all the object files in the directory.

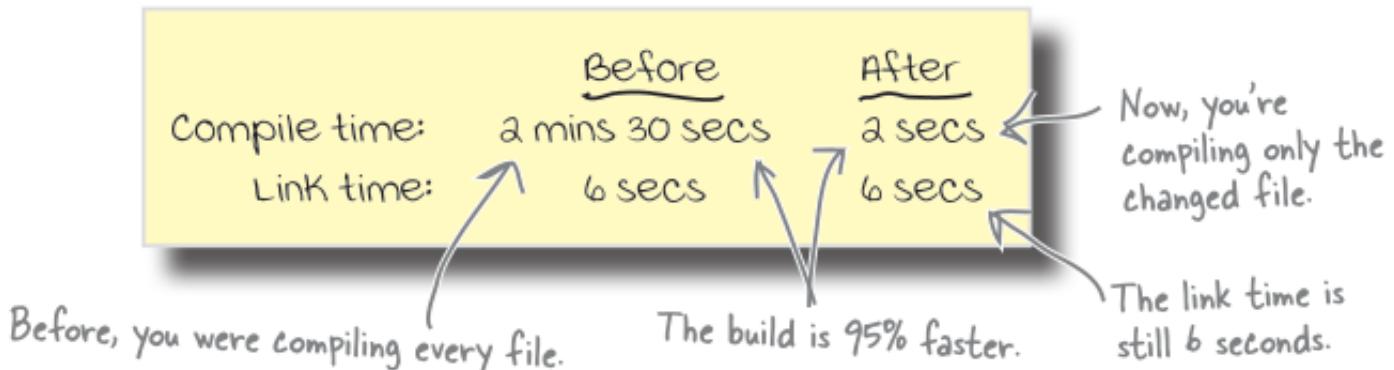
The compiler is smart enough to recognize the files as object files, rather than source files, so it will skip most of the compilation steps and just link them together into an executable program called launch.

OK, so now you have a compiled program, just like before. But you also have a set of object files that are ready to be linked together if you need them again.

So if you change just one of the files, you'll only need to recompile that single file and then relink the program:

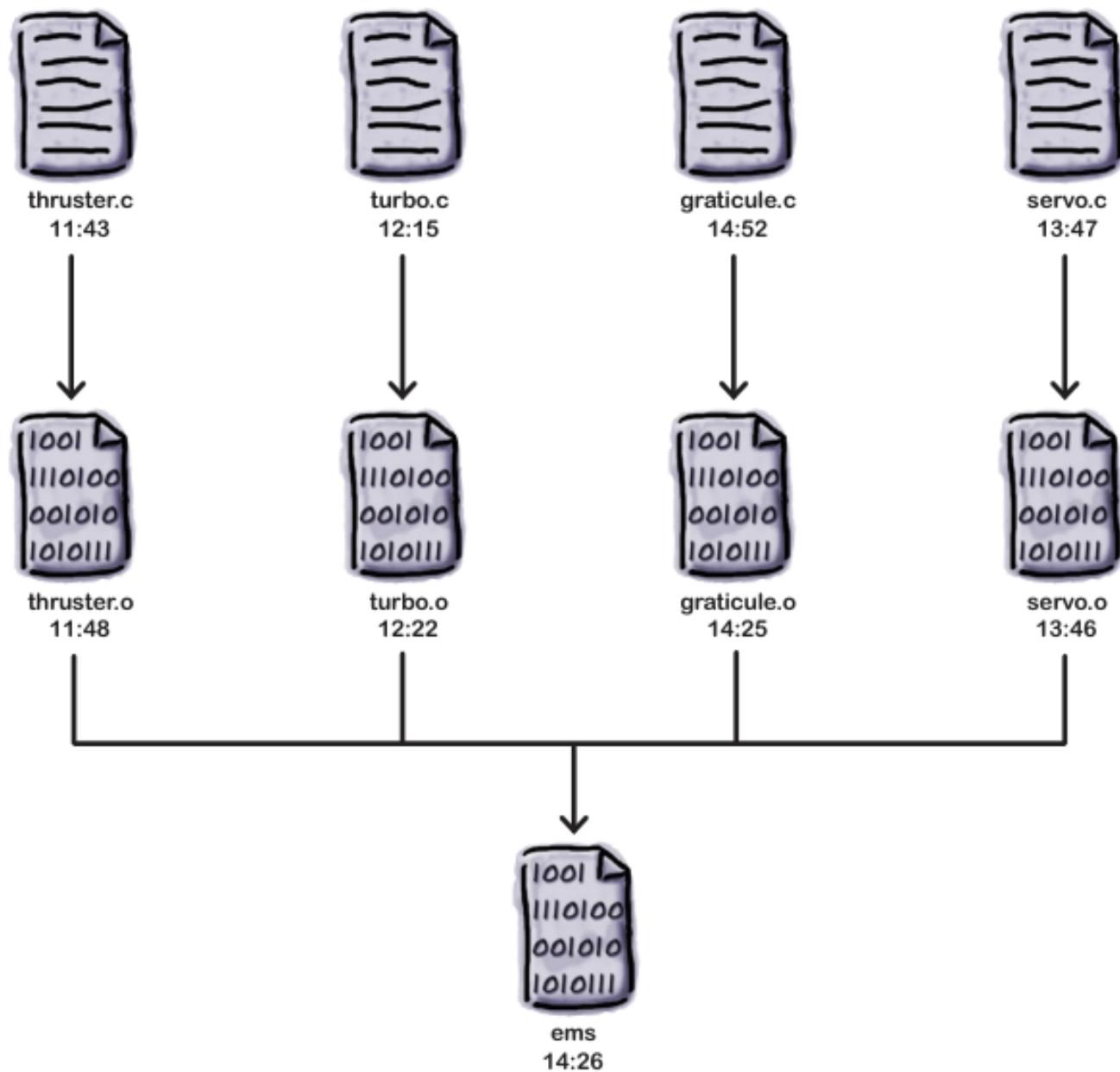
This is the only file that's changed. → `gcc -c thruster.c` ← This will recreate the thruster.o file.
`gcc *.o -o launch` ← This will link everything together.

Even though you have to type two commands, you're saving a lot of time.

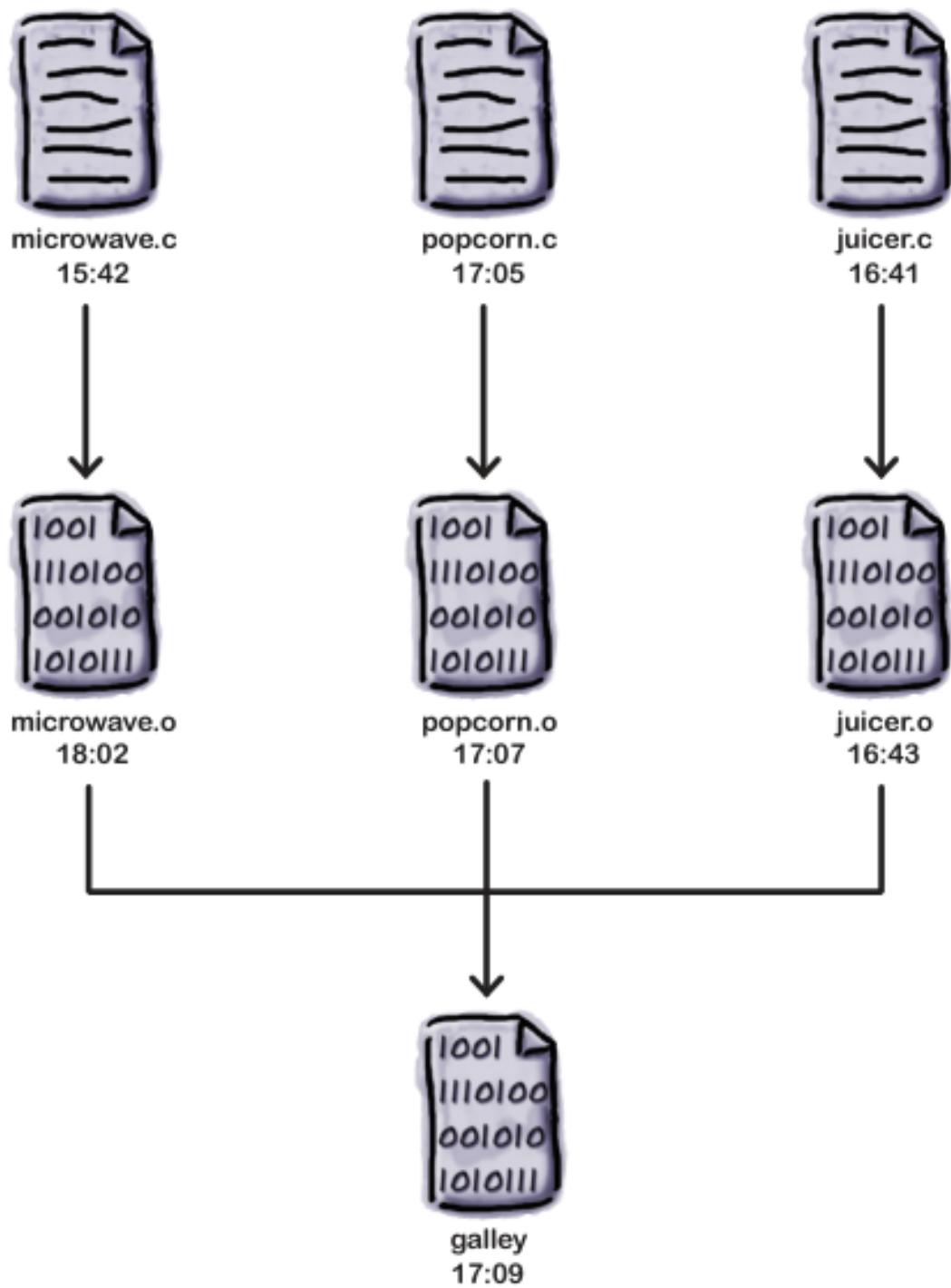


Here is some of the code that's used to control the engine management system on the craft. There's a timestamp on each file.

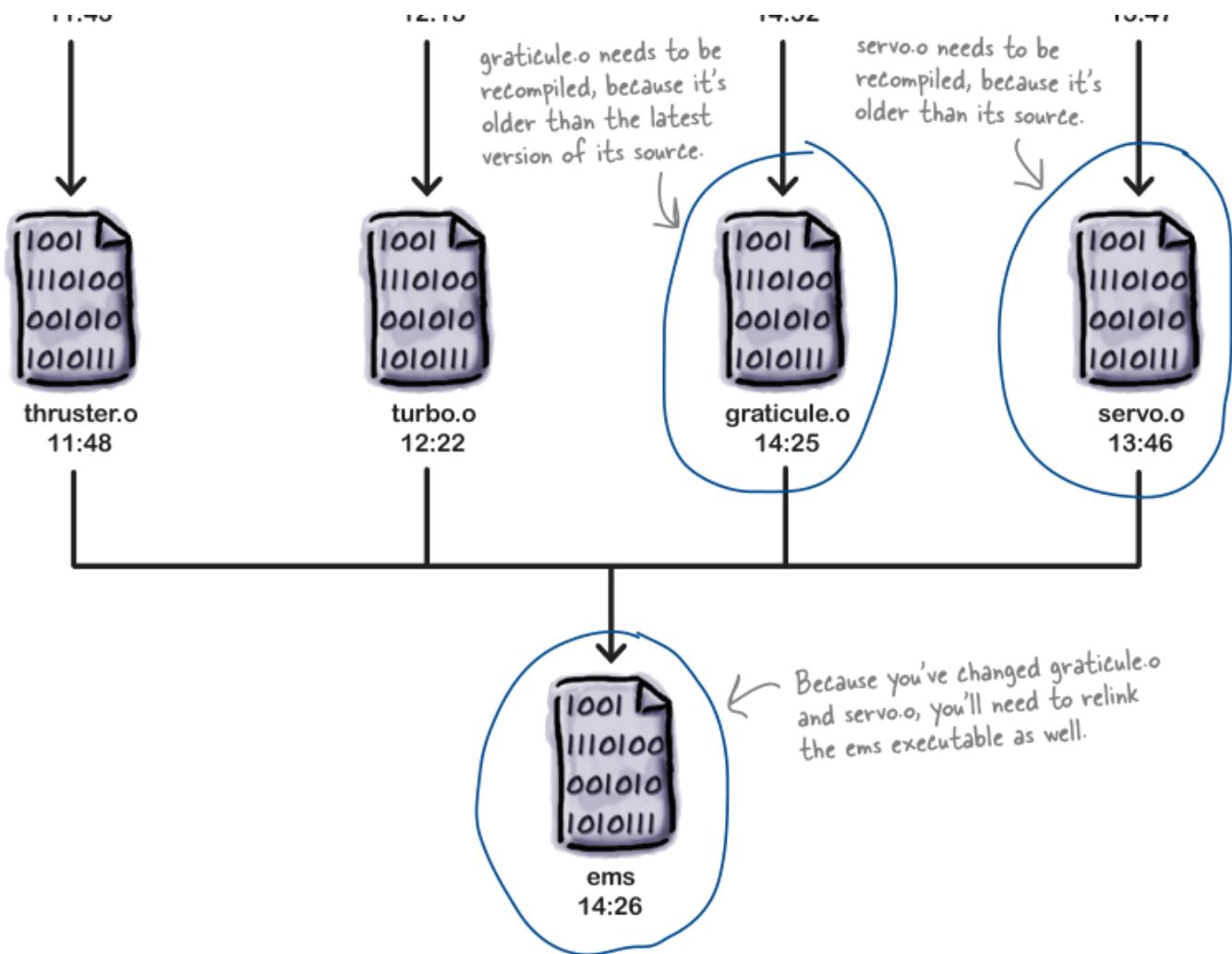
Which files do you think need to be recreated to make the `ems` executable up to date? Circle the files you think need to be updated.



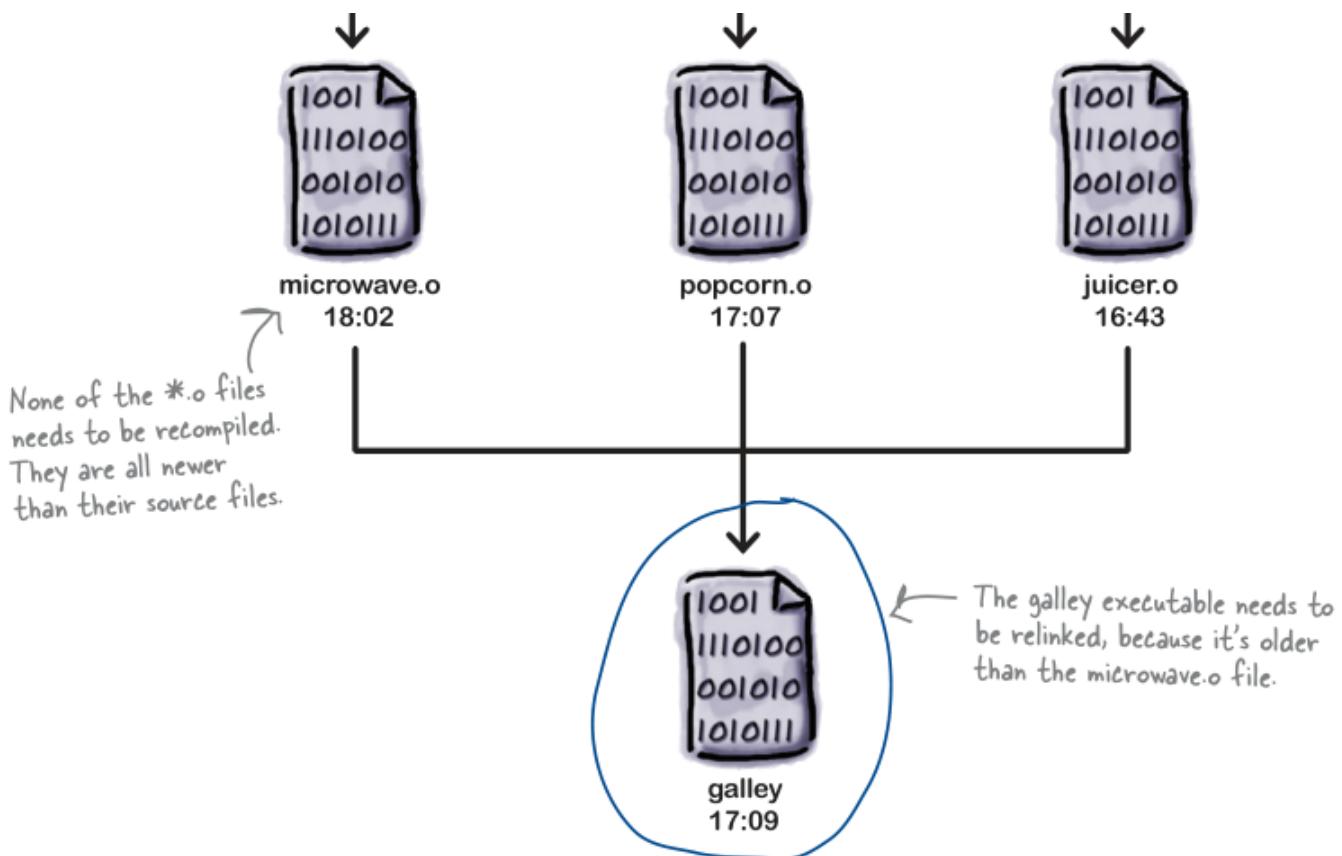
And in the galley, they need to check that their code's up to date as well. Look at the times against the files. Which of these files need to be updated?



Answers for A:



Answers for B:



It's hard to keep track of the files. I thought the whole point of saving time was so I didn't have to get distracted.



I thought the whole point of saving time was so I didn't have to get distracted. Now the compile is faster, but I have to think a lot harder about how to compile my code. Where's the sense in that?

It's true: partial compiles are faster, but you have to think more carefully to make sure you recompile everything you need.

If you are working on just one source file, things will be

Now the compile is faster, but I have to think a lot harder about how to compile my code. Where's the sense in that?

It's true: partial compiles are faster, but you have to think more carefully to make

sure you recompile everything you need.

If you are working on just one source file, things will be pretty simple.

But if you've changed a few files, it's pretty easy to forget to recompile some of them.

That means the newly compiled program won't pick up all the changes you made.

Now, of course, when you come to ship the final program, you can always make sure you can do a full recompile of every file, but you don't want to do that while you're still developing the code.

Even though it's a fairly mechanical process to look for files that need to be compiled, if you do it manually, it will be pretty easy to miss some changes.

Is there something we can use to automate the process?

Wouldn't it be dreamy if there were a tool that could automatically recompile just the source that's changed?

But I know it's just a fantasy.

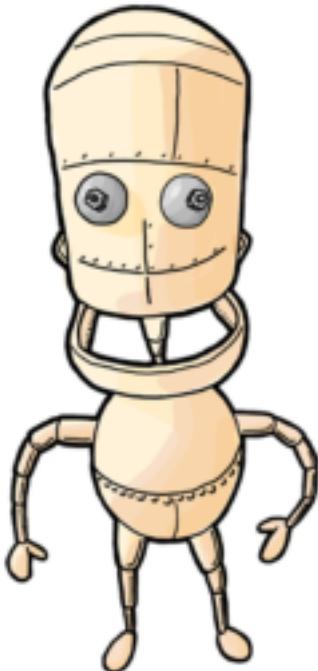


Wouldn't it be dreamy if there were a tool that could automatically recompile just the source that's changed? But I know it's just a fantasy...

Automate your builds with the `make` tool.

You can compile your applications really quickly in `gcc`, as long as you keep track of which files have changed.

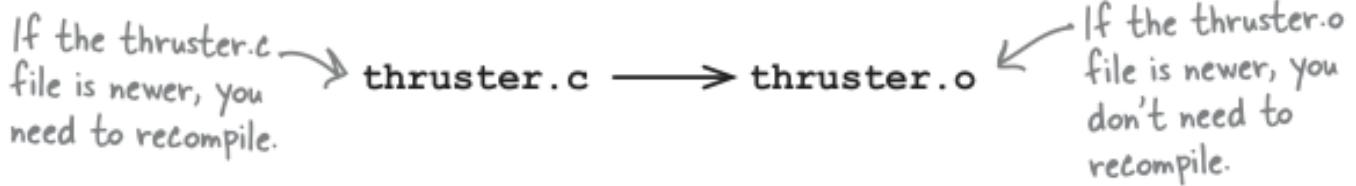
That's a tricky thing to do, but it's also pretty straightforward to automate.



This is make, your new best friend.

Imagine you have a file that is generated from some other file.

Let's say it's an object file that is compiled from a source file:



How do you tell if the `thruster.o` file needs to be recompiled? You just look at the timestamps of the two files.

If the `thruster.o` file is older than the `thruster.c` file, then the `thruster.o` file needs to be recreated.

Otherwise, it's up to date.

That's a pretty simple rule.

And if you have a simple rule for something, then don't think about it—automate it.

`make` is a tool that can run the `compile` command for you.

The `make` tool will check the timestamps of the source files and the generated files, and then it will only recompile the files if things have gotten out of date.

But before you can do all these things, you need to tell `make` about your source code.

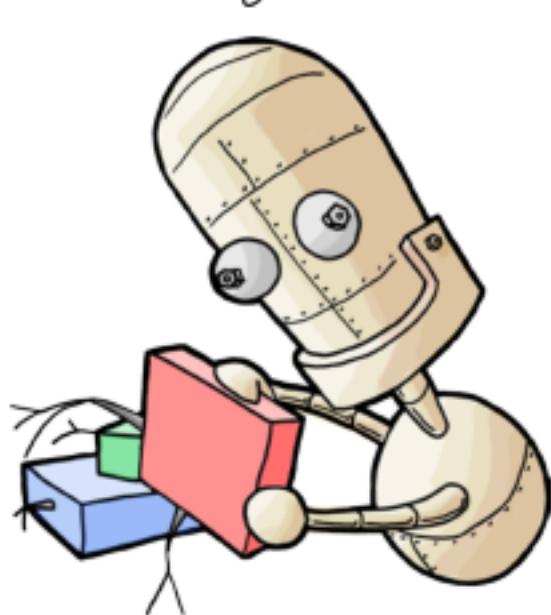
It needs to know the details of which files depend on which files.

And it also needs to be told exactly how you want to build the code.

What does `make` need to know?

Every file that `make` compiles is called a **target**.

Hmm...this file's OK.
And this one. And this one.
And...ah, this one's out of
date. I'd better send that
to the compiler.



Strictly speaking, make isn't limited to compiling files.

A **target** is any file that is generated from some other files.

So a target might be a zip archive that is generated from the set of files that need to be compressed.

For every target, make needs to be told two things:



The dependencies.

Which files the target is going to be generated from.



The recipe.

The set of instructions it needs to run to generate the file.

Together, the dependencies and the recipe form a rule. A **rule** tells make all it needs to know to create the target file.

HOW MAKE WORKS

Let's say you want to compile `thruster.c` into some object code in `thruster.o`.

What are the dependencies and what's the recipe?

The `thruster.o` file is called the **target**, because it's the file you want to generate. `thruster.c` is a **dependency**, because it's a file the compiler will need in order to create `thruster.o`.

And what will the **recipe** be? That's the **compile command** to convert `thruster.c` into `thruster.o`.

`thruster.c` → `thruster.o`

`gcc -c thruster.c` ← This is the rule for
creating `thruster.o`.

Make sense? If you tell the make tool about the dependencies and the recipe, you can leave it to make to decide when it needs to recompile `thruster.o`. But you can go further than that.

Once you build the `thruster.o` file, you're going to use it to create the launch program.

That means the **launch file** can also be set up as a **target**, because it's a file you want to generate.

The dependency files for launch are all of the **.o** object files.

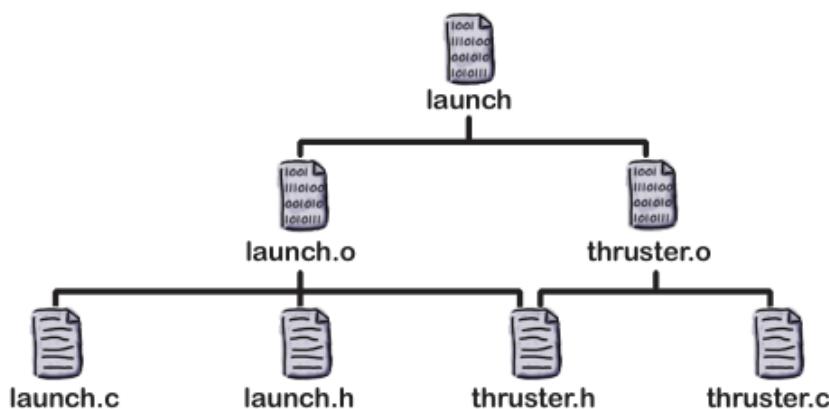
The recipe is this command:

```
gcc *.o -o launch
```

Once make has been given the details of all of the dependencies and rules, all you have to do is tell it to create the launch file.

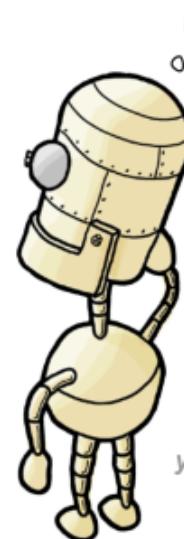
make will work out the details.

Once make has been given the details of all of the dependencies and rules, all you have to do is tell it to create the launch file. make will work out the details.



But how do you tell make about the dependencies and recipes? Let's find out.

So I've got to compile the launch program? Hmm... First I'll need to recompile thruster.o, because it's out of date; then I just need to relink launch.



you are here ▶

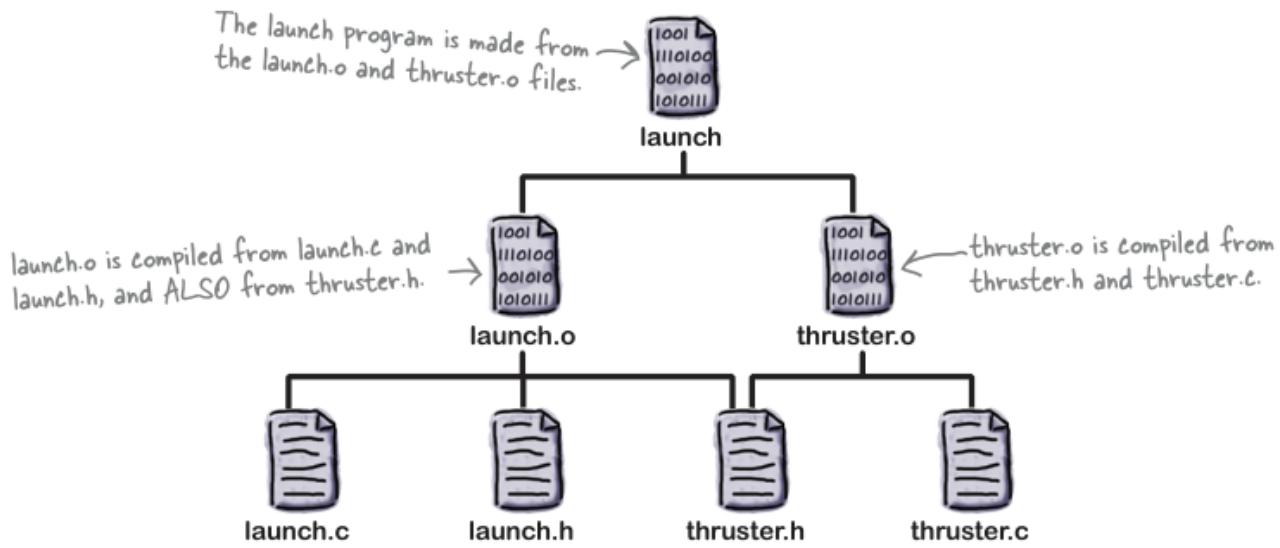
199

www.it-ebooks.info

Tell make about your code with a **makefile**.

All of the details about the targets, dependencies, and recipes need to be stored in a file called either **makefile** or **Makefile**.

To see how it works, imagine you have a pair of source files that together create the launch program:



The launch program is made by linking the launch.o and thruster.o files.

Those files are compiled from their matching C and header files, but the launch.o file also depends on the thruster.h file because it contains code that will need to call a function in the thruster code.

This is how you'd describe that build in a makefile:

This is a target. A target is a file that is going to be generated.

```
launch.o: launch.c launch.h thruster.h  
        gcc -c launch.c
```

There are three RULES.

```
thruster.o: thruster.h thruster.c  
        gcc -c thruster.c
```

```
launch: launch.o thruster.o  
        gcc launch.o thruster.o -o launch
```

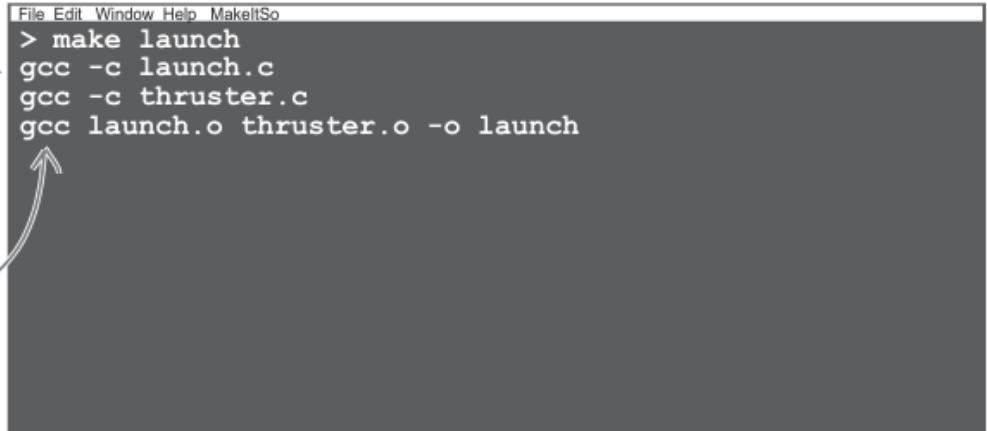
The recipes MUST begin with a tab character.



All of the recipe lines MUST begin with a tab character.

If you just try to indent the recipe lines with spaces, the build won't work.

open up a console and type the following:



```
File Edit Window Help Makefile
> make launch
gcc -c launch.c
gcc -c thruster.c
gcc launch.o thruster.o -o launch
```

You can see that make was able to work out the sequence of commands required to create the launch program.

But what happens if you make a change to the `thruster.c` file and then run make again?



```
File Edit Window Help Makefile
> make launch
gcc -c thruster.c
```

`make` is able to skip creating a new version of `launch.o`!

Instead, it just compiles `thruster.o` and then **relinks** the program.

QUESTIONS

Is make just like ant? It's probably better to say that build tools like ant and rake are like make. make was one of the earliest tools used to automatically build

programs from source code.

This seems like a lot of work just to compile source code. Is it really that useful? Yes, make is amazingly useful. For small projects, make might not appear to save you that much time, but once you have more than a handful of files, compiling and linking code together can become very painful.

If I write a makefile for a Windows machine, will it work on a Mac? Or a Linux machine? Because makefiles calls commands in the underlying operating system, sometimes makefiles don't work on different operating systems.

Can I use make for things other than compiling code? Yes. make is most commonly used to compile code. But it can also be used as a command-line installer, or a source control tool. In fact, you can use make for almost any task that you can perform on the command line.

WHY USE TABS

It worked, it stayed. And then a few weeks later I had a user population of about a dozen, most of them friends, and I didn't want to screw up my embedded base. The rest, sadly, is history."

make takes away a lot of the pain of compiling files. But if you find that even it is not automatic enough, take a look at a tool called autoconf.

Other tools are found on C websites.

USING MAKEFILE

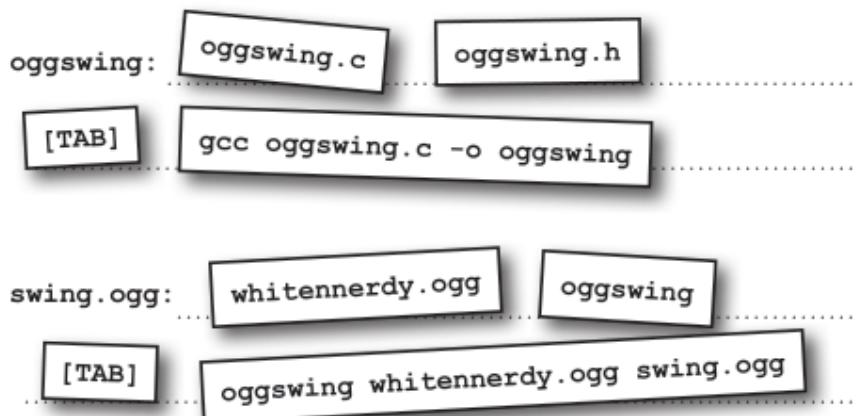
Make Magnets Hey, baby, if you don't groove to the latest tunes, then you'll love the program the guys in the Head First Lounge just wrote!

oggswing is a program that **reads an Ogg Vorbis music file** and **creates a swing version**. Sweet!

See if you can complete the makefile that compiles oggswing and then uses it to convert a .ogg file:

```
oggswing: oggswing.c oggswing.h
          gcc oggswing.c -o oggswing
```

```
swing.ogg: whitennerdy.ogg oggswing
          oggswing.whitennerdy.ogg swing.ogg //remember tabs not spaces
```



It can take a long time to compile a large number of files.

You can speed up compilation time by storing object code in `*.o` files.

The `gcc` can compile programs from object files as well as source files.

The `make` tool can be used to automate your builds.

`make` knows about the dependencies between files, so it can compile just the files that change.

`make` needs to be told about your build with a `makefile`.

Be careful formatting your `makefile`: don't forget to indent lines with tabs instead of spaces.

chars are numbers.

Use shorts for small whole numbers.

Use longs for really big whole numbers.

Use ints for most whole numbers.

Use floats for most floating points.

Use doubles for really precise floating points.

Split function declarations from definitions.

Put declarations in a header file.

#include <> for library headers.

#include " " for local headers.

Save object code into files to speed up your builds.

Use make to

Split function declarations from definitions.

Put declarations in a header file.

#include <> for library headers.

#include " " for local headers.

Save object code into files to speed up your builds.

Use make to manage your builds.

Arduino Lab 1

This lab work:

The spec: make your houseplant talk Ever wished your plants could tell you when they need watering?

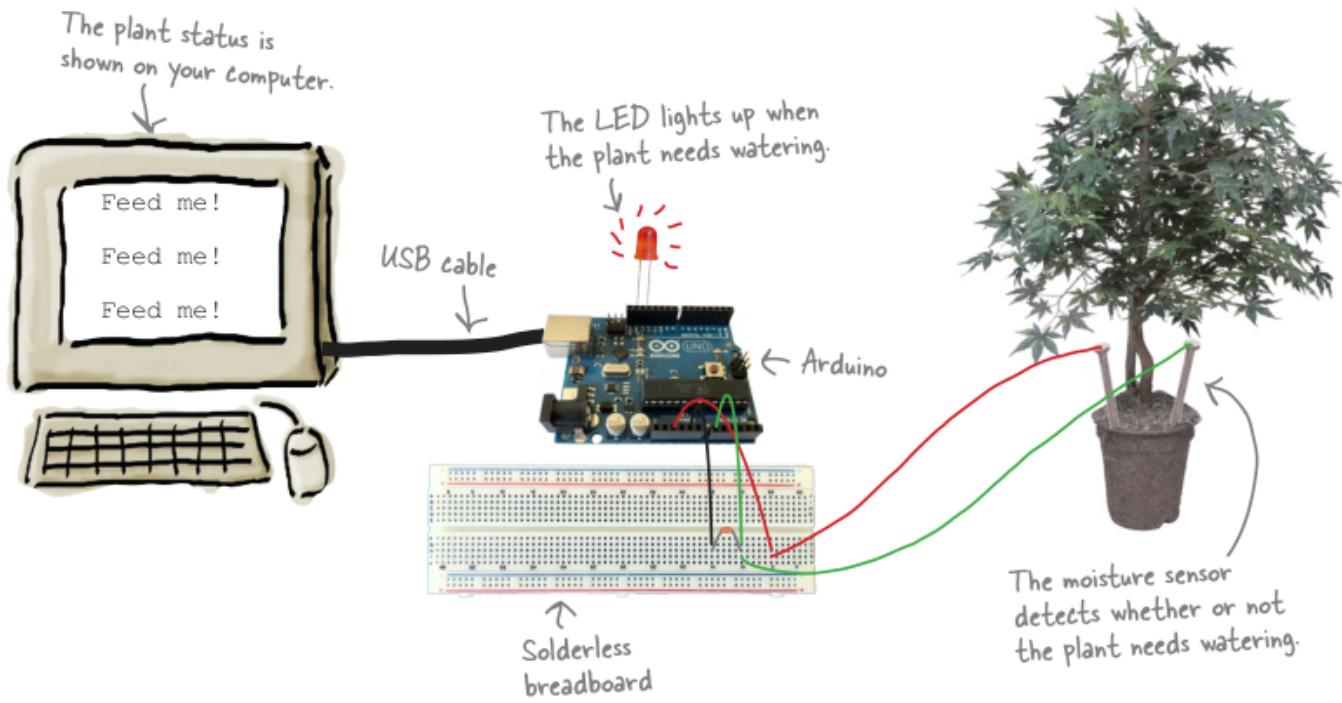
Well, with an Arduino they can! In this lab, you'll create an Arduino-powered plant monitor, all coded in C.

Here's what you're going to build.



The physical device: The plant monitor has a moisture sensor that measures how wet your plant's soil is. If the plant needs watering, an LED lights up until the plant's

been watered, and the string "Feed me!" is repeatedly sent to your computer. When the plant has been watered, the LED switches off and the string "Thank you, Seymour!" is sent once to your computer.



The Arduino The brains of the plant monitor is an Arduino.

An Arduino is a small micro- controller-based open source platform for electronic prototyping.

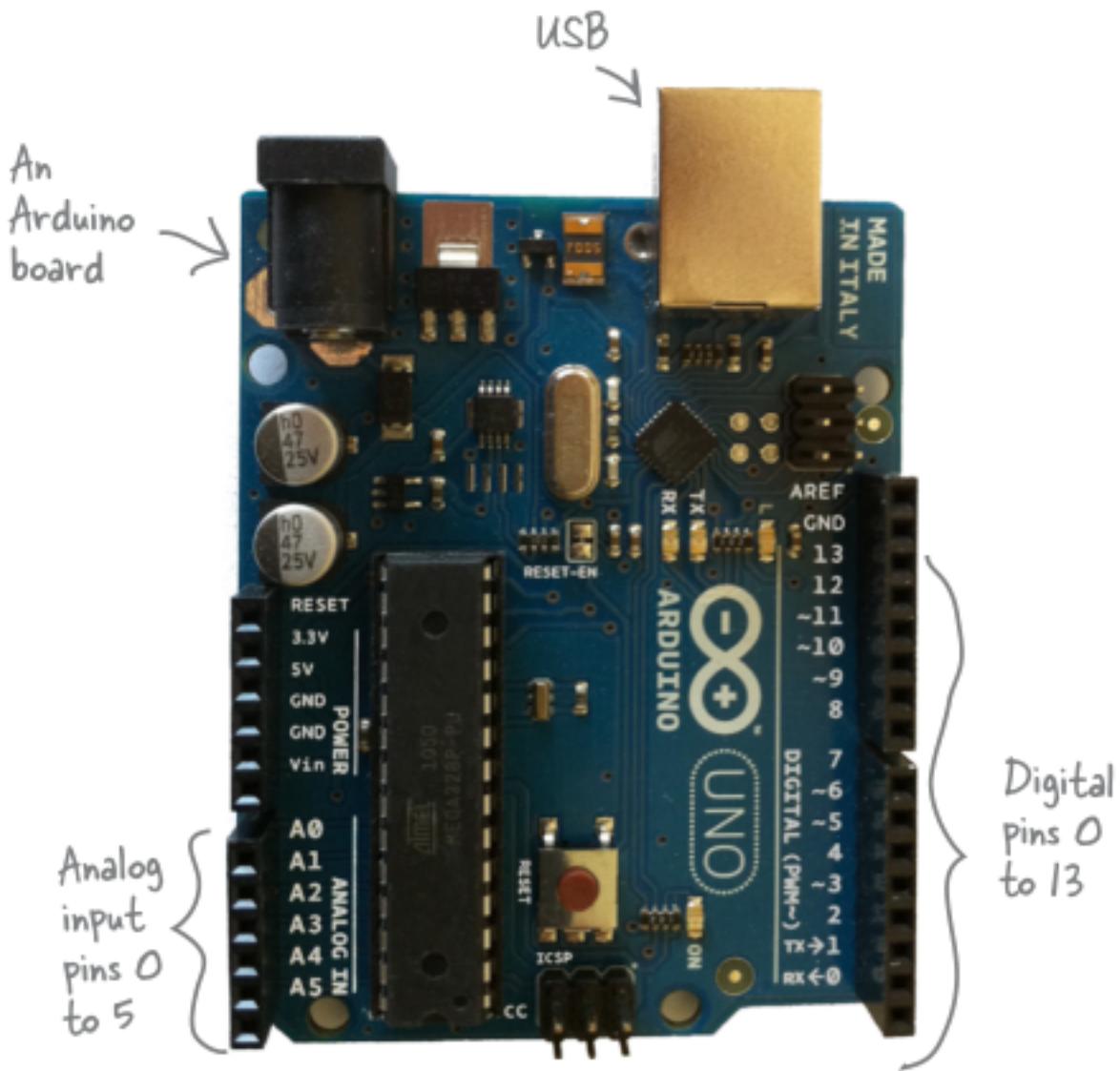
You can connect it to sensors that pick up information about the world around it, and actuators that respond.

All of this is controlled by code you write in C.

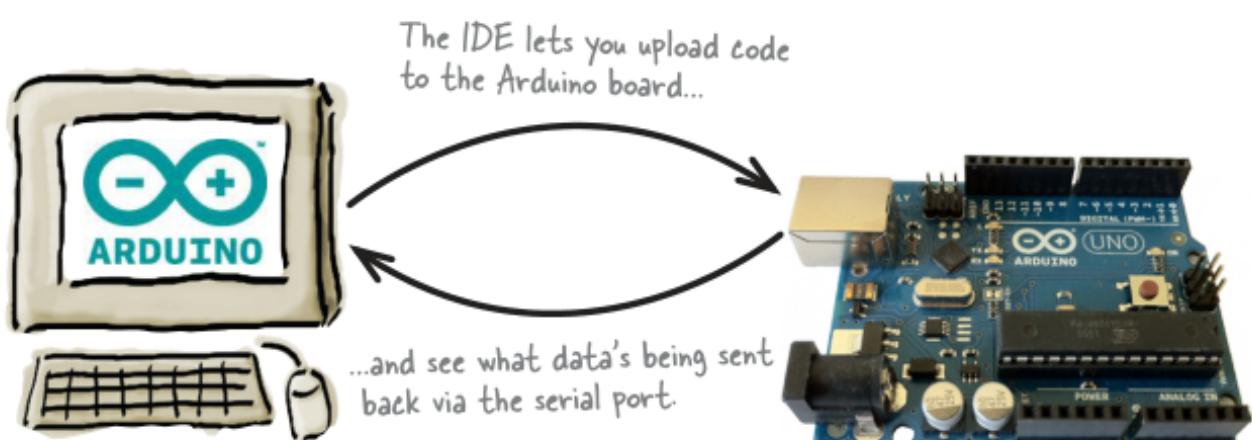
The Arduino board has 14 digital IO pins, which can be inputs or outputs.

These tend to be used for reading on or off values, or switching actuators on or off.

The board also has six analog input pins, which take voltage readings from a sensor.



The board can take power from your computer's USB port. The Arduino IDE You write your C code in an Arduino IDE.



The IDE allows you to verify and compile your code, and then upload it to the Arduino

itself via your USB port.

The IDE also has a built-in serial monitor so that you can see what data the Arduino is sending back (if any).

The Arduino IDE is free, and you can get hold of a copy from arduino website.

Build the physical device.

1 Arduino.

1 solderless.

Breadboard.

1 LED.

1 10K Ohm resistor.

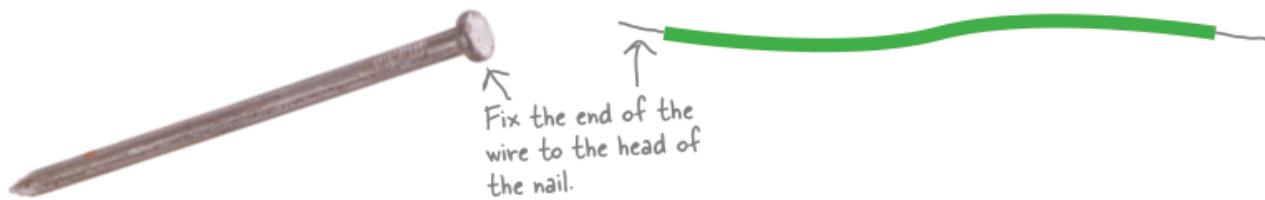
2 galvanized nails.

3 short pieces of jumper wire.

2 long pieces of jumper wire.

We used an Arduino Uno.

Build the moisture sensor Take a long piece of jumper wire and attach it to the head of one of the galvanized nails.



You can either wrap the wire around the nail or solder it in place.

Once you've done that, attach another long piece of jumper wire to the second galvanized nail.

The moisture sensor works by checking the conductivity between the two nails.

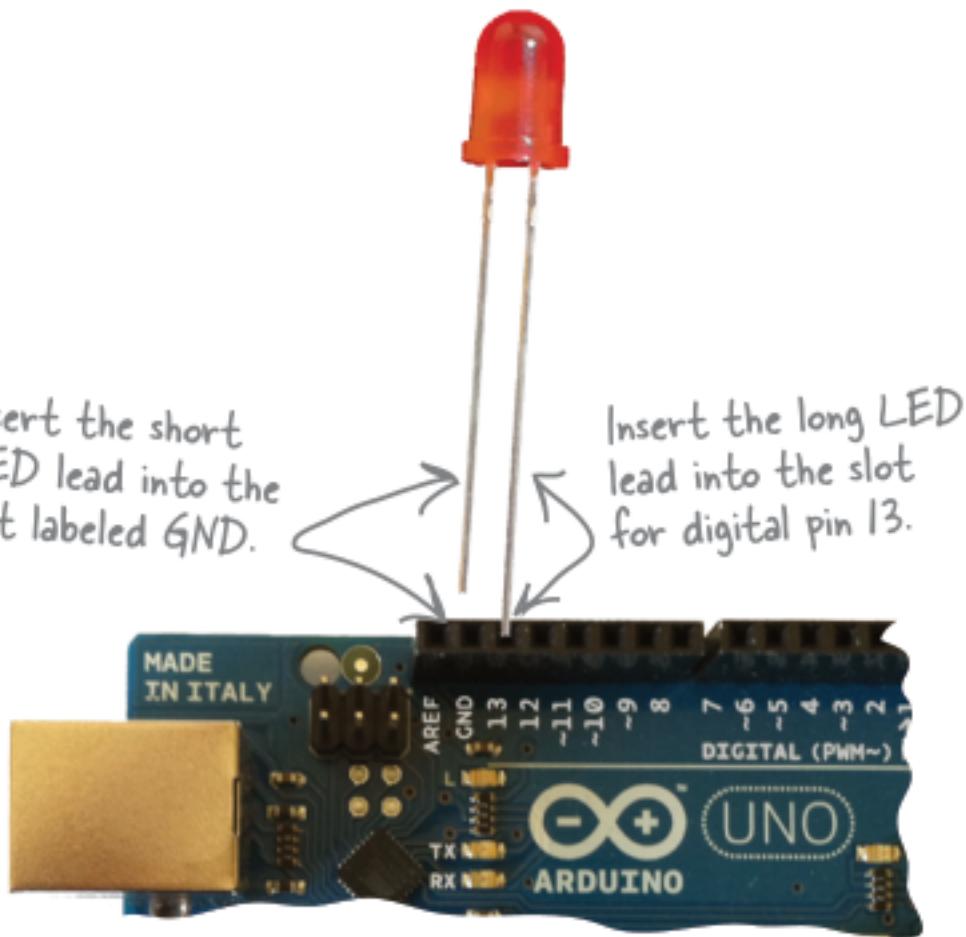
If the conductivity is high, the moisture content must be high. If it's low, the moisture content must be low.

Connect the LED Look at the LED.

You will see that it has one longer (positive) lead and one shorter (negative) lead.

Now take a close look at the Arduino.

You will see that along one edge there are slots for 14 digital pins labeled 0-13, and another one next to it labeled GND.



Put the long positive lead of the LED into the slot labeled 13, and the shorter negative lead into the slot labeled GND.

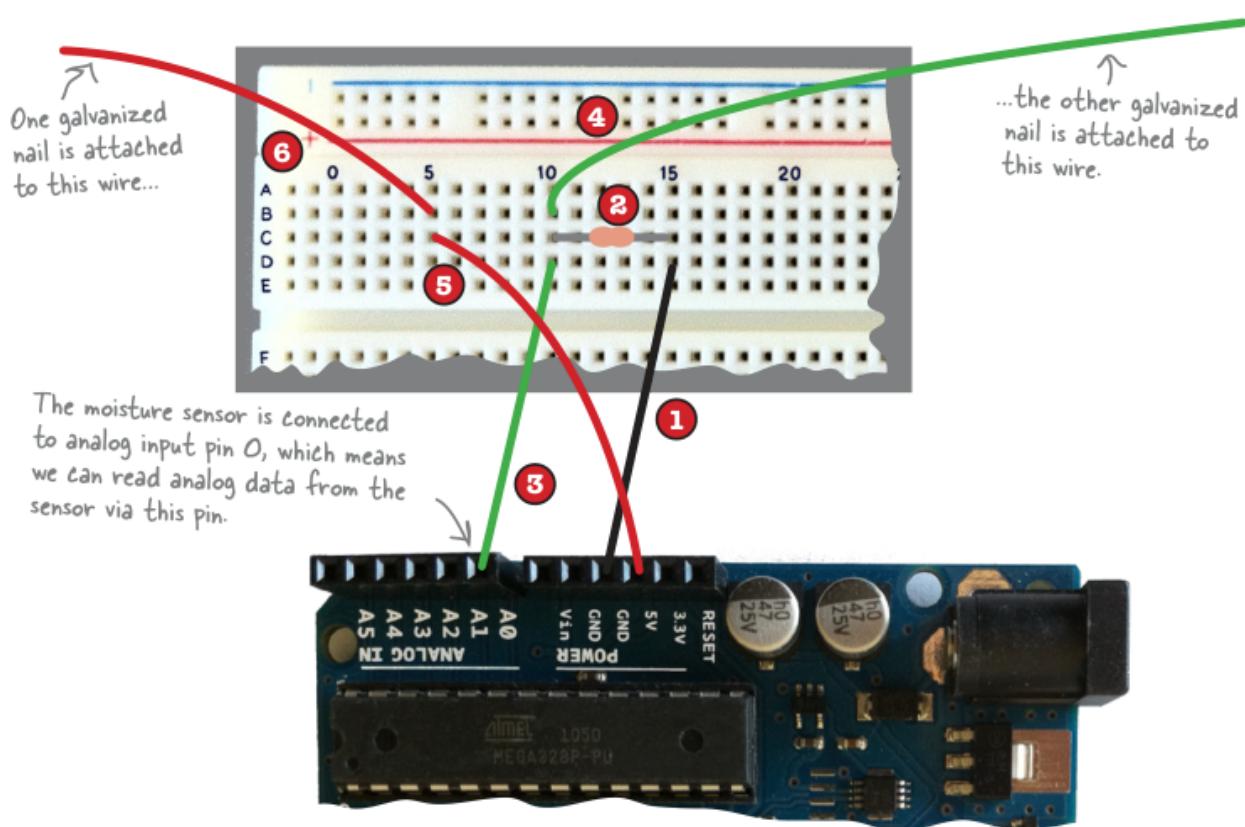
This means that the LED can be controlled through digital pin 13.

Connect the moisture sensor

Connect a short jumper wire from the GND pin on the Arduino to slot D15 on the breadboard.

Connect a short jumper wire from the 0 analog input pin to slot D10 on the breadboard.

Connect the 10K Ohm resistor from slot C15 on the breadboard to slot C10.



That's the physical Arduino built. Now for the C code...

Take one of the galvanized nails, and connect the wire attached to it to slot B10.

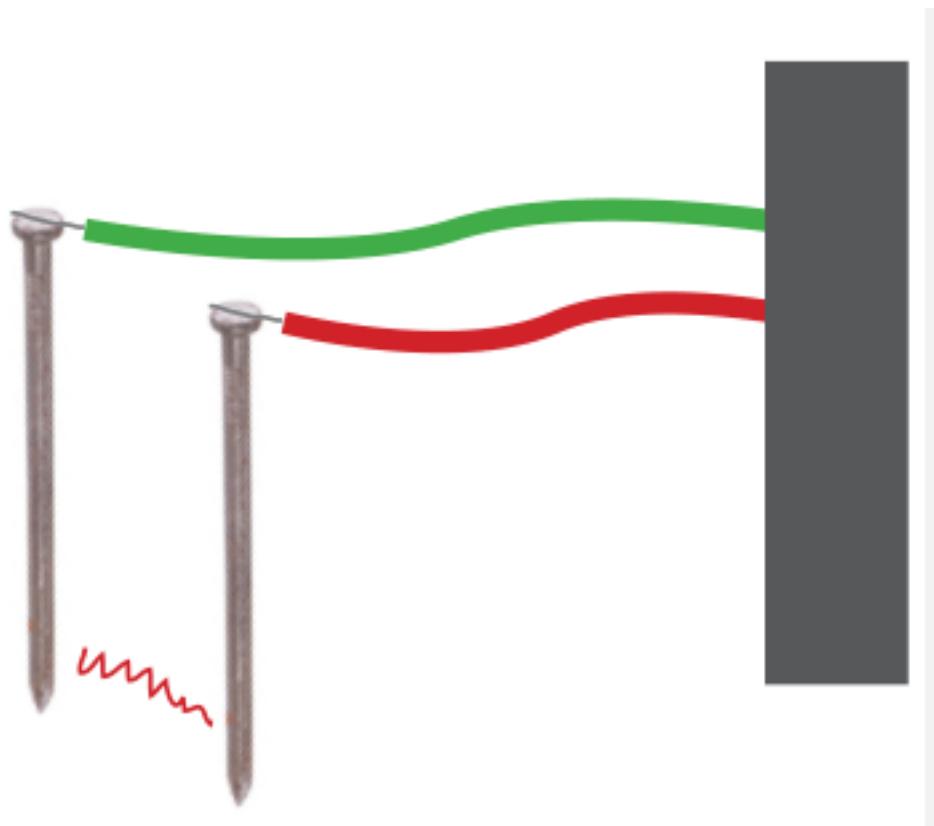
Connect a short jumper wire from the 5V pin on the Arduino to slot C5 on the breadboard.

Take the other galvanized nail, and connect the wire attached to it to slot B5.

Write the code

Read from the moisture sensor: The moisture sensor is connected to an analog input pin. You will need to read analog values from this pin.

Here at the lab, we've found that our plants generally need watering when the value goes below 800, but your plant's requirements may be different—say, if it's a cactus.

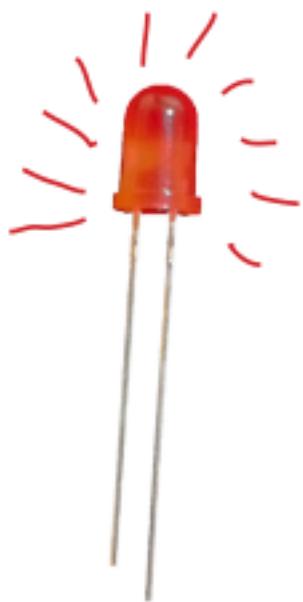


Write to the LED: The LED is connected to a digital pin.

When the plant doesn't need any more water, write to the digital pin the LED is connected to, and get it to switch off the LED.

When the plant needs watering, write to the digital pin and get it to switch on the LED. For extra credit, get it to flash.

Even better, get it to flash when the conditions are borderline.



Write to the serial port: When the plant needs watering, repeatedly write the string "Feed me!" to the computer serial port.

When the plant has enough water, write the string "Thank you, Seymour!" to the serial port once.

Assume that the Arduino is plugged in to the computer USB socket.



```
void setup()
{
  /*This is called when the program starts. It basically sets up the
  board. Put any initialisations in here */
}

void loop()
{
  /*This is where your main code goes. This funciton
  loops over and over, and allows you to respond to input from
  your sensors. It only stops running when
  the board is switched off. */
```

Use the arduino IDE, and upload it.

Functions for arduino:

void pinMode(int pin, int mode): Tells the Arduino whether the digital pin is an input

or output. mode can be either INPUT or OUTPUT.

int digitalRead(int pin): Reads the value from the digital pin. The return value can be either HIGH or LOW.

void digitalWrite(int pin, int value): Writes a value to a digital pin. value can be either HIGH or LOW.

int analogRead(int pin): Reads the value from an analog pin. The return value is between 0 and 1023.

void analogWrite(int pin, int value): Writes an analog value to a pin. value is between 0 and 255.

void Serial.begin(long speed): Tells the Arduino to start sending and receiving serial data at speed bits per second. You usually set speed to 9600.

void Serial.println(val): Prints data to the serial port. val can be any data type.

void delay(long interval): Pauses the program for interval milliseconds.

The finished product.

You'll know your Arduino project is complete when you put the moisture sensor in your plant's soil, connect the Arduino to your computer, and start getting status updates about your plant.



Structs, Unions and Bitfields

Roll your own structures

```
struct tea quila =  
{"tealeaves", "milk",  
"sugar", "water", "tequila"};
```



We've been using basic datatypes, we now want to advance to more complexity.

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

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

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

In this chapter, you'll learn how to combine the basic data types into structs, and even handle life's uncertainties with unions.

And if you're after a simple yes or no, bitfields may be just what you need.

HANDING OUT A LOT OF DATA

You've seen that C can handle a lot of different types of data: small numbers and large numbers, floating-point numbers, characters, and text.

But quite often, when you are recording data about something in the real world, you'll find that you need to use more than one piece of data.

Take a look at this example. Here you have two functions that both need the same set of data, because they are both dealing with the same real-world thing:

```
//print out the catalog entry.
void catalog(const char *name, const char *species, int teeth, int age)
{
    printf("%s is a %s with %i teeth. He is %i\n", name, species, teeth, age);
}

//print label for the tank
void label(const char *name, const char *species, int teeth, int age)
{
    printf("Name: %s\nSpecies: %s\n%iyears old, %i teeth\n", name, species, teeth, age);
}
```

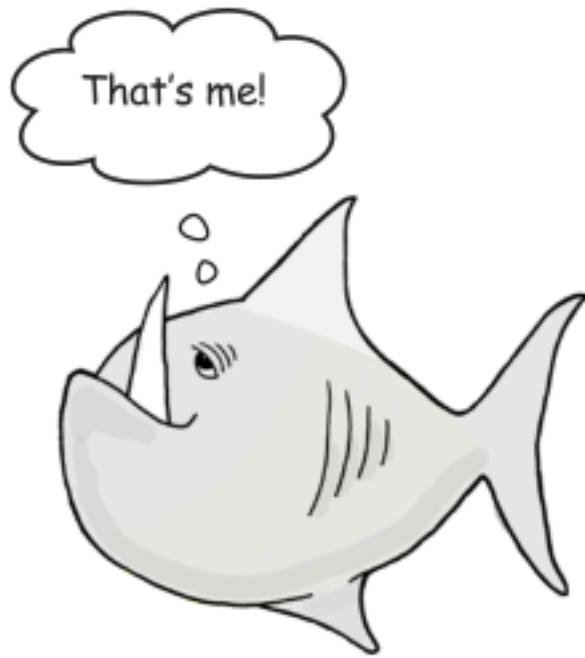
"**const char ****" just means you're going to pass **string literals**.

Both of these functions take the same set of parameters.

Now that's not really so bad, is it?

But even though you're just passing four pieces of data, the code's starting to look a little messy:

```
int main()
{
    catalog("Snappy","Piranha",69,4); //passing the same 4 pieces of data twice.
    label("Snappy","Piranha",69,4); //there's only 1 fish, but we passing 4 pieces of data.
    return 0;
}
```



So how do you get around this problem?

What can you do to avoid passing around lots and lots of data if you're really only using it to describe a single thing?

What if we add another piece of data? We can't group such into an array, coz they only store one datatype!! And over here we are recording strings and ints.

If you have a set of data that you need to bundle together into a single thing, then you can use a struct. The word struct is short for **structured data type**.

A **struct** will let you take all of those different pieces of data into the code and wrap them up into one large new data type, like this:

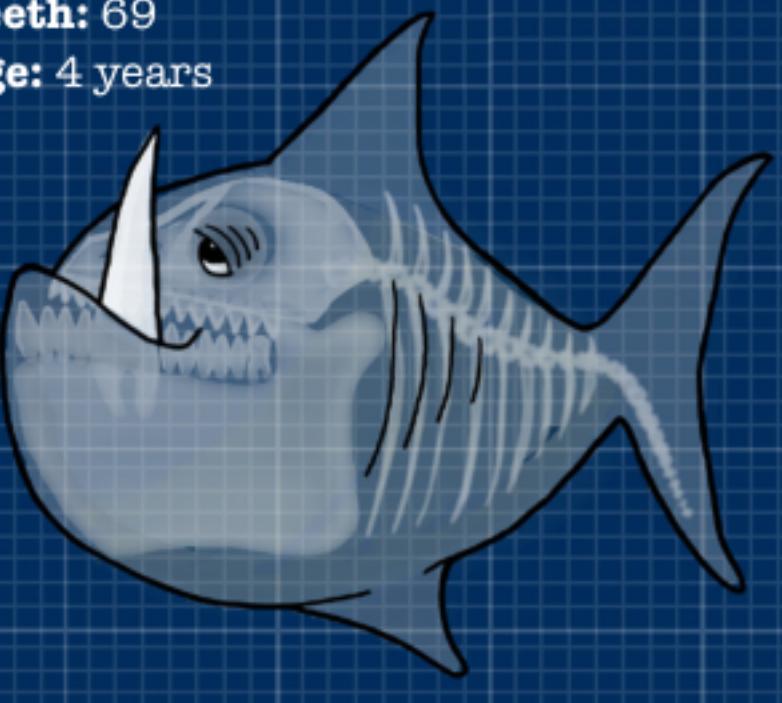
```
struct fish
{
    const char *name;
    const char *species;
    int teeth;
    int age;
};
```

Name: Snappy

Species: Piranha

Teeth: 69

Age: 4 years



This will create a new custom data type that is made up of a collection of other pieces of data.

In fact, it's a little bit like an array, except:

- It's fixed length.
- The pieces of data inside the struct are given names.

But once you've defined what your new struct looks like, how do you create pieces of data that use it?

Well, it's quite similar to creating a new array.

You just need to make sure the **individual pieces of data are in the order that they are defined in the struct**:

order that they are defined in the struct:

"struct fish" is the data type.
"snappy" is the variable name.
This is the name.

This is the species.
This is the number of teeth.
This is Snappy's age.

Hey, wait a minute. What's that `const char` thing again? `const char *` is used for strings that you don't want to change. That means it's often used to record string literals.

OK. So does this struct store the string? In this case, no. The struct here just stores a pointer to a string. That means it's just recording an address, and the string lives somewhere else in memory.

But you can store the whole string in there if you want? Yes, if you define a `char` array in the string, like `char name[20];` eg.

```
struct
{
    int number;
    char name[NAME_LEN + 1]; //name len is going to be an int
    int on_hand;
};
```

Just give them the fish. Now, instead of having to pass around a whole collection of individual pieces of data to the functions, you can just pass your new custom piece of data:

```
//print out the catalog entry.  
void catalog(struct fish f)  
{  
    .....  
}  
  
//print out the label for the tank  
void label(struct fish f)  
{  
    .....  
}
```

Looks a lot simpler, doesn't it? Not only does it mean the functions now only need a single piece of data, but the code that calls them is easier to read:

So that's how you can define your custom data type, but how do you use it? How will our functions be able to read the individual pieces of data stored inside the struct?

Wrapping parameters in a struct makes your code more stable.

One of the great things about data passing around inside structs is that you can change the contents of your struct without having to change the functions that use it. For example, let's say you want to add an extra field to fish:

```
struct fish  
{  
    const char *name;  
    const char *species;  
    int teeth;  
    int age;  
    int favorite_music;  
};
```

All the catalog() and label() functions have been told is they they're going to be handed a fish.

They don't know (and don't care) that the fish now contains more data, so long as it has all the fields they need.

That means that structs don't just make your code easier to read, they also make it better able to cope with change.

READ STRUCTS WITH . OPERATOR

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};  
printf("Name = %s\n", snappy[0]); //would work only if snappy was a pointer to an array
```

You get an error if you try to read a struct field like it's an array.

But you can't. Even though a struct stores fields like an array, the only way to access them is by name.

You can do this using the **".** operator. If you've used another language, like JavaScript or Ruby, this will look familiar:

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};  
printf("Name = %s\n", snappy.name);
```

```
struct fish snappy = {"Snappy", "piranha", 69, 4};  
printf("Name = %s\n", snappy.name);
```

```
File Edit Window Help Fish  
> gcc fish.c -o fish  
> ./fish  
Name = Snappy  
>
```

This is the name attribute in snappy.

↑
This will return the
string "Snappy."

OK, now that you know a few things about using structs, let's see if you can go back and update that code.

Pool Puzzle:

Your job is to write a new version of the catalog() function using the fish struct.

Take fragments of code from the pool and place them in the blank lines below. You may not use the same fragment more than once, and you won't need to use all the fragments.

You've rewritten the catalog() function, so it's pretty easy to rewrite the label() function as well.

Once you've done that, you can compile the program and check that it still works:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct fish
5 {
6     const char *name;
7     const char *species;
8     int teeth;
9     int age;
10 };
11
12 void catalog(struct fish f)
13 {
14     printf("%s is a %s with %i teeth. He is %i\n", f.name, f.species, f.teeth, f.age);
15 }
16
17 int main()
18 {
19     struct fish snappy = {"Snappy", "Piranha", 69, 4};
20     catalog(snappy);
21     //skip calling label for now.
22     return 0;
23 }

```

The output:

```

C:\Users\HP-EliteBook\Downloads\C\CatalogStructs\bin\Debug\CatalogStructs.exe
Snappy is a Piranha with 69 teeth. He is 4

Process returned 0 (0x0)  execution time : 1.881 s
Press any key to continue.

```

Adding the label function():

Hey, look, someone's using make... →

This line is printed out by the catalog() function. →

These lines are printed by the label() function. →

```

File Edit Window Help FishAreFriendsNotFood
> make pool_puzzle && ./pool_puzzle
gcc pool_puzzle.c -o pool_puzzle
Snappy is a Piranha with 69 teeth. He is 4
Name:Snappy
Species:Piranha
4 years old, 69 teeth
>

```

Not only is the code more readable, but if you ever decide to record some extra data in the struct, you won't have to change anything in the functions that use it.

QUESTIONS:

So is a struct just an array? No, but like an array, it groups a number of pieces of data together.

An array variable is just a pointer to the array. Is a struct variable a pointer to a struct? No, a struct variable is a name for the struct itself.

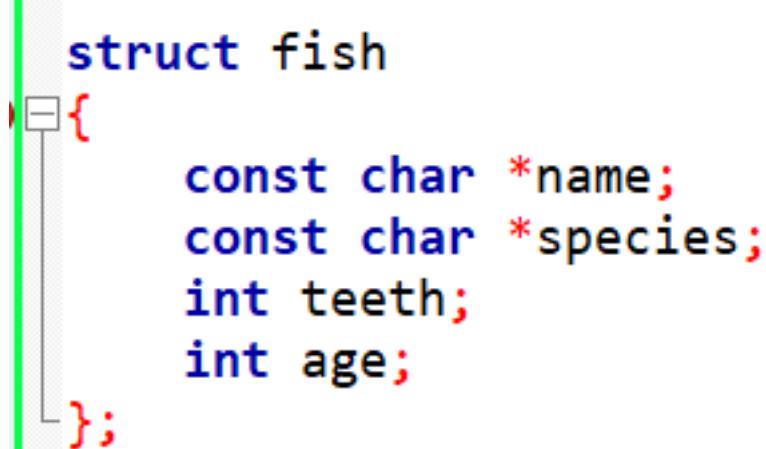
I know I don't have to, but could I use [0], [1] to access the fields of a struct? No, you can only access fields by name.

Are structs like classes in other languages? They're similar, but it's not so easy to add methods to structs.

STRUCTS IN MEMORY

When you define a struct, you're not telling the computer to create anything in memory.

You're just giving it a template for how you want a new type of data to look.

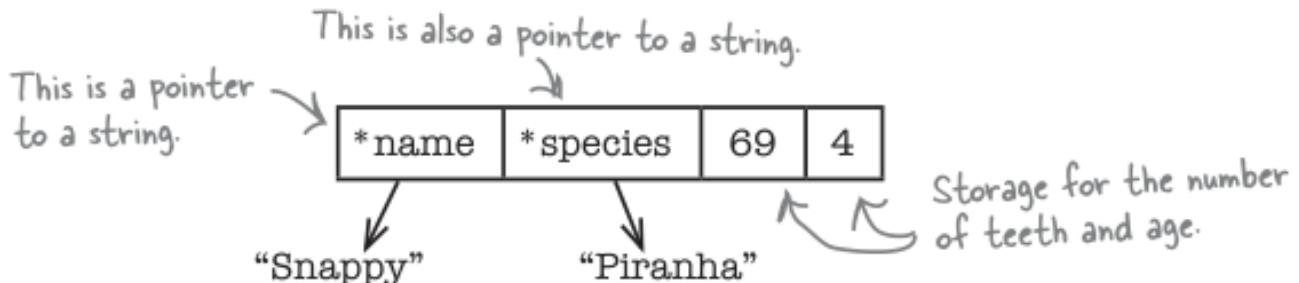


```
struct fish
{
    const char *name;
    const char *species;
    int teeth;
    int age;
};
```

But when you define a new variable, the computer will need to create some space in memory for an instance of the struct.

That space in memory will need to be big enough to contain all of the fields within the struct:

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
```

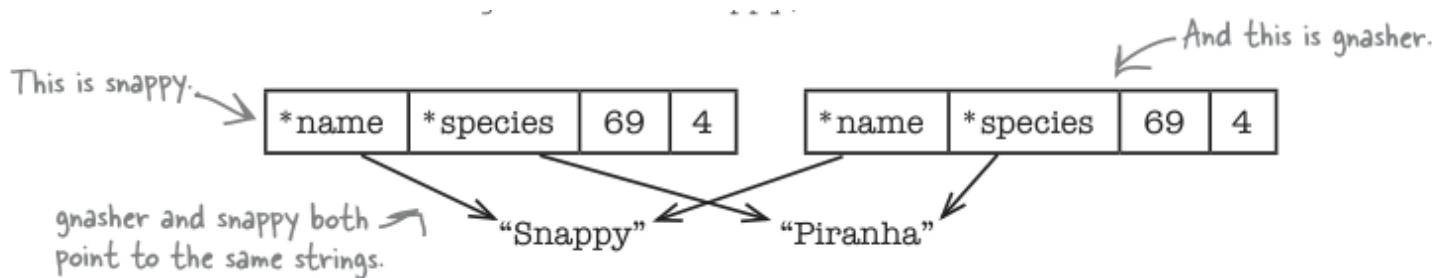


So what do you think happens when you assign a struct to another variable?

Well, the computer will create a brand-new copy of the struct.

That means it will need to allocate another piece of memory of the same size, and then copy over each of the fields.

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};  
struct fish gnasher = snappy;
```



Remember: when you're assigning struct variables, you are telling the computer to copy data.



Watch it!

The assignment **copies the pointers** to strings, not the strings themselves.

When you assign one struct to another, the **contents of the struct will be copied**.

But if, as here, that includes pointers, the assignment will just copy the pointer values.

That means the name and species fields of gnasher and snappy both point to the same strings.

STRUCT INSIDE A STRUCT

Can you put one struct inside another? Remember that when you define a struct, you're actually creating a new data type.

C gives us lots of built-in data types like ints and shorts, but a struct lets us combine existing types together so that you can describe more complex objects to the computer.

But if a struct creates a data type from existing data types, that means you can also create structs from other structs.

To see how this works, let's look at an example.

```
struct preferences
{
    const char *food;
    float exercise_hours;
};

struct fish
{
    const char *name;
    const char *species;
    int teeth;
    int age;
    struct preference care; //called nesting
    //out new field is "care", but it contains fields defined inside preference struct
};
```

```
struct preferences { ← These are things our fish likes.
    const char *food;
    float exercise_hours;
};

struct fish {
    const char *name;
    const char *species;
    int teeth;
    int age; ← This is a struct inside a struct.
    struct preferences care; ← This is called nesting.
};

This is a new field. → };
```

↑ Our new field is called "care," but it will contain fields defined by the "preferences" struct.

This code tells the computer one struct will contain another struct.

You can then create variables using the same array like code as before, but now you can include the data for one struct inside another:

```
struct fish snappy = {"Snappy", "Piranha", 69, 4, {"Meat", 7.5}};  
//meat is value for care.food.  
//value for care.exercise_hours is 7.5  
//struct data for the care field included.
```

```
#include <stdio.h>  
#include <stdlib.h>  
  
struct exercise  
{  
    const char *description;  
    float duration;  
};  
  
struct meal  
{  
    const char *ingredients;  
    float weight;  
};  
  
struct preferences  
{  
    struct meal food;  
    struct exercise exercise;  
};  
  
struct fish  
{  
    const char *name;  
    const char *species;  
    int teeth;  
    int age;  
    struct preferences care;  
};
```

The guys at the Head First Aquarium are starting to record lots of data about each of their fish guests.

Here are their structs:

This is the data that will be recorded for one of the fish:

```
Name: Snappy
Species: Piranha
Food ingredients: meat
Food weight: 0.2 lbs
Exercise description: swim in the jacuzzi
Exercise duration: 7.5 hours
```

How would you write this data in C?

```
struct fish snappy = {"Snappy", "Piranha", 69, 4, {"meat", 0.2}, {"Swim in the jacuzzi", 7.5}};
```

Completing this code label so that it produces this output:

```
Name: snappy
Species: Piranha
4 years old, 69 teeth
Feed with 0.20 lbs of meat and allow to swim in the jacuzzi for 7.50 hours.
```

Code for label:

```
Name: snappy
Species: Piranha
4 years old, 69 teeth
Feed with 0.20 lbs of meat and allow to swim in the jacuzzi for 7.50 hours.

void label (struct fish a)
{
    printf("Name: %s\nSpecies: %s\n%i years old, %i teeth\n", a.name, a.species, a.age, a.teeth);
    printf("Feed with %.2f lbs of %s and allow to %s for %.2f hours\n",
           a.care.food.weight, a.care.food.ingredients, a.care.food.exercise.description,
           a.care.exercise.duration);
}
```

All these struct commands seem kind of wordy.

I have to use the `struct` keyword when I define a struct, and then I have to use it again when I define a variable.

I wonder if there's some way of simplifying this.

Hmm...all these `struct` commands seem kind of wordy. I have to use the `struct` keyword when I define a struct, and then I have to use it again when I define a variable. I wonder if there's some way of simplifying this.



You can give your struct a proper name using `typedef`.

When you create variables for built-in data types, you can use simple short names like `int` or `double`, but so far, every time you've created a variable containing a struct you've had to include the `struct` keyword.

```
struct cell_phone
{
    int cell_no;
    const char *wallpaper;
    float minutes_of_charge;
};

struct cell_phone p = {5557879, "sinatra.png", 1.35};
```

But C allows you to create an alias for any struct that you create. If you add the word `typedef` before the `struct` keyword, and type name after the closing brace, you can call the new type whatever you like:

typedef
means you
are going → **typedef struct cell_phone {**
to give int cell_no;
the struct const char *wallpaper;
type a new float minutes_of_charge;
name. } **phone;** ← phone will become an alias for
 "struct cell_phone."

what the fucking hell am i dealing with here. My goodness is this fuck fucking my fucking mind at 100% accuracy.

```
+++++
struct cell_phone
{
    int cell_no;
    const char *wallpaper;
    float minutes_of_charge;
};

struct cell_phone p = {5557879, "sinatra.png", 1.35};

+++++
typedef struct cell_phone //typedef means you are going to give the struct type a new name
{
    int cell_no;
    const char *wallpaper;
    float minutes_of_charge;
}; phone; // will become an alias for the "struct cell_phone".

phone p = {5557879, "sinatra.png", 1.35};
```

Now, when the compiler sees "phone", it will treat it like "struct cell_phone".

typedefs can shorten your code and make it easier to read. Let's see what your code will look like if you start to add **typedefs** to it.

What should I call my new type? If you use **typedef** to create an alias for a struct,

you will need to decide what your alias will be.

The alias is just the name of your type.

That means there are two names to think about: the name of the struct (struct cell_phone) and the name of the type (phone).

Why have two names? You usually don't need both.

The compiler is quite happy for you to skip the struct name, like this:

```
typedef struct
{
    int cell_no;
    const char *wallpaper;
    float minutes_of_charge;
}; phone; //this is the alias

phone p = {5557879, "s.wallpaper", 1.35};
```

It's time for the scuba diver to make his daily round of the tanks, and he needs a new label on his suit.

Trouble is, it looks like some of the code has gone missing. Can you work out what the missing words are?

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct
5 {
6     float tank_capacity;
7     int tank_psi;
8     const char *suit_material;
9 } equipment;
10
11
12 typedef struct scuba
13 {
14     const char *name;
15     equipment kit;
16 } diver;
17

```

```

-- 19 void badge(diver d)
20 {
21     printf("Name: %s Tank: %2.2f(%i) Suit: %s\n" ,
22             d.name, d.kit.tank_capacity, d.kit.tank_psi, d.kit.suit_material);
23 }
24
25 int main()
26 {
27     diver randy = {"Randy", {5.5, 3500, "Neoprene"}};
28     badge(randy);
29     return 0;
30 }
31

```

SUMMARY OF STRUCTS

A struct is a data type made from a sequence of other data types.

structs are fixed length.

struct fields are accessed by name, using the `<struct>.<field name>` syntax (a.k.a dot notation).

struct fields are stored in memory in the same order they appear in the code.

You can nest structs.

`typedef` creates an alias for a data type.

If you use `typedef` with a struct, then you can skip giving the struct a name.

Do struct fields get placed next to each other in memory? Sometimes there are small gaps between the fields.

Why's that? The computer likes data to fit inside word boundaries. So if a computer uses 32-bit words, it won't want a short, say, to be split over a 32-bit boundary.

So it would leave a gap and start the short in the next 32-bit word? Yes.

Does that mean each field takes up a whole word? No. The computer leaves gaps only to prevent fields from splitting across word boundaries. If it can fit several fields into a single word, it will.

Why does the computer care so much about word boundaries? It will read complete words from the memory. If a field was split across more than one word, the CPU would have to read several locations and somehow stitch the value together.

And that'd be slow? That'd be slow.

In languages like Java, if I assign an object to a variable, it doesn't copy the object, it just copies a reference. Why is it different in C? In C, all assignments copy data. If you want to copy a reference to a piece of data, you should assign a pointer.

I'm really confused about struct names. What's the struct name and what's the alias? The struct name is the word that follows the `struct` keyword. If you write `struct peter_parker { ... }`, then the name is `peter_parker`, and when you create variables, you would say `struct peter_parker x`.

And the alias? Sometimes you don't want to keep using the `struct` keyword when you declare variables, so `typedef` allows you to create a single word alias. In `typedef struct peter_parker { ... } spider_man;`, `spider_man` is the alias.

So what's an anonymous struct? One without a name. So `typedef struct { ... }` `spider_man;` has an alias of `spider_man`, but no name. Most of the time, if you create an alias, you don't need a name.

UPDATING A STRUCT

A struct is really just a bundle of variables, grouped together and treated like a single piece of data.

You've already seen how to create a struct object, and how to access its values using dot notation.

But how do you change the value of a struct that already exists? Well, you can change the fields just like any other variable:

//UPDATING STRUCTS

```
fish snappy = {"Snappy", "Piranha", 69, 4}; //creates a struct
printf("Hello %s\n", snappy.name); //reads the value of the name field.
snappy.teeth = 68; //sets the value of the teeth field.
```

This creates a struct. → `fish snappy = {"Snappy", "piranha", 69, 4};`

This sets the value of the teeth field. → `snappy.teeth = 68;` ← This reads the value of the name field. ← Ouch! Looks like Snappy bit something hard.

That means if you look at this piece of code, you should be able to work out what it does, right?

```

43 #include <stdio.h>
44
45 typedef struct
46 {
47     const char *name;
48     const char *species;
49     int age;
50 } turtle;
51
52 void happy_birthday(turtle t)
53 {
54     t.age = t.age + 1;
55     printf("Happy Birthday %s! You are now %i years old!\n", t.name, t.age);
56 }
57
58 int main()
59 {
60     turtle myrtle = {"Myrtle", "Leatherback sea turtle", 99};
61     happy_birthday(myrtle);
62     printf("%s's age is now %i\n", myrtle.name, myrtle.age);
63     return 0;
64 }

```

But there's something odd about this code.

This is what happens when you compile and run the code.

```

File Edit Window Help ILikeTurtles
> gcc turtle.c -o turtle && ./turtle
Happy Birthday Myrtle! You are now 100 years old!
Myrtle's age is now 99
>

```

Something weird has happened.

The code creates a new struct and then passes it to a function that was supposed to increase the value of one of the fields by 1.

And that's exactly what the code did...at least, for a while.

Inside the `happy_birthday()` function, the `age` field was updated, and you know that it worked because the `printf()` function displayed the new increased age value.

But that's when the weird thing happened.

Even though the `age` was updated by the function, when the code returned to the `main()` function, the `age` seemed to reset itself.

This code is doing something weird. But you've already been given enough information to tell you exactly what happened. Can you work out what it is?

THE CODE IS CLONING THE TURTLE

Let's take a closer look at the code that called the `happy_birthday()` function.

```
void happy_birthday(turtle t)
{
    .....
}

//this is the turtle that we are passing to the function.
happy_birthday(myrtle);
//the myrtle struct will be copied to this parameter.
```

Look at the code that called the
() function:

```
void happy_birthday(turtle t)
{
    ...
}
...
happy_birthday(myrtle);
```

This is the turtle that we are
passing to the function.

passed to functions **by value**. That

When you assign
a struct, its
values get copied
to the new struct.

The myrtle struct will be
copied to this parameter.

In C, parameters are passed to functions by value.

That means that when you call a function, the values you pass into it are assigned to the parameters.

So in this code, it's almost as if you had written something like this:

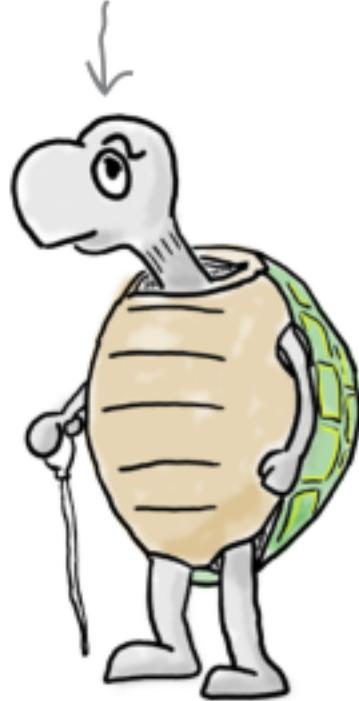
```
turtle t = myrtle;
```

But remember: when you assign structs in C, the values are copied.

When you call the function, the parameter t will contain a copy of the myrtle struct.

It's as if the function has a clone of the original turtle.

This is Myrtle...



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



So the code inside the function does update the age of the turtle, but it's a different turtle.

What happens when the function returns?

The `t` parameter disappears, and the rest of the code in `main()` uses the `myrtle` struct.

But the value of `myrtle` was never changed by the code.

It was always a completely separate piece of data.

So what do you do if you want pass a struct to a function that needs to update it?

You need a pointer to the struct. When you passed a variable to the `scanf()` function, you couldn't pass the variable itself to `scanf()`; you had to pass a pointer:

```
scanf("%f", &length_of_run);
```

Why did you do that? Because if you tell the `scanf()` function where the variable lives in memory, then the function will be able to update the data stored at that place in memory, which means it can update the variable.

And you can do just the same with structs. If you want a function to update a struct variable, you can't just pass the struct as a parameter because that will simply send a copy of the data to the function.

Instead, you can pass the address of the struct:

```
//pointer to a struct (remember: an address is a pointer)
void happy_birthday(turtle *t)
{
    .....
}

happy_birthday(&myrtle); //pass the address of myrtle to the function.
```

See if you can figure out what expression needs to fit into each of the gaps in this new version of the `happy_birthday()` function.

Be careful. Don't forget that `t` is now a pointer variable.

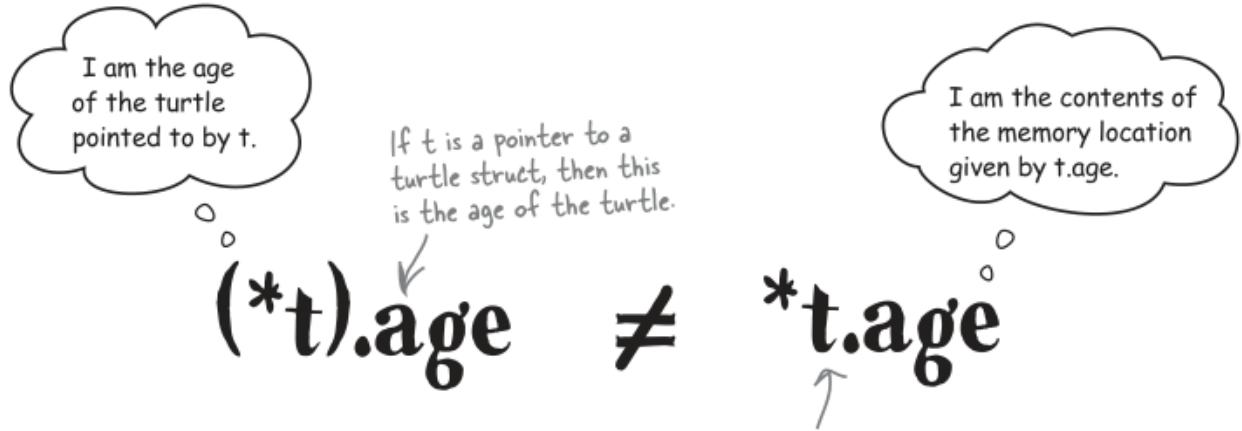
```
+++++
//you need to put a * before the variable name, because you want the value it points to.
void happy_birthday(turtle *t)
{
    (*t).age = (*t).age + 1;
    printf ("Happy Birthday %s! You are now %i years old!\n", (*t).name, (*t).age);
}
//The parentheses are really important. The code will break without them.

+++++
```

$(*t).age$ vs. $*t.age$

So why did you need to make sure that $*t$ was wrapped in parentheses?

It's because the two expressions, $(*t).age$ and $*t.age$, are very different.



So the expression $*t.age$ is really the same as $*(t.age)$. Think about that expression for a moment. It means "the contents of the memory location given by $t.age$." But $t.age$ isn't a memory location.

If t is a pointer to a turtle struct, then this expression is wrong.

So be careful with your parentheses when using structs parentheses really matter.

If t is a pointer to a turtle struct, then this is the age of the turtle.

If t is a pointer to a turtle struct, then this expression is wrong.

The new code:

```
2 #include <stdio.h>
3
4 typedef struct
5 {
6     const char *name;
7     const char *species;
8     int age;
9 } turtle;
10
11 void happy_birthday(turtle *t)
12 {
13     (*t).age = (*t).age + 1;
14     printf ("Happy Birthday %s! You are now %i years old!\n", (*t).name, (*t).age);
15 }
16
17
18 int main()
19 {
20     turtle myrtle = {"Myrtle", "Leatherback sea turtle", 99};
21     happy_birthday(&myrtle);
22     printf("%s's age is now %i\n", myrtle.name, myrtle.age);
23     return 0;
24 }
```

The output:

```
Happy Birthday Myrtle! You are now 100 years old!
Myrtle's age is now 100

Process returned 0 (0x0)    execution time : 1.887 s
Press any key to continue.
```

That's great. The function now works. By passing a pointer to the struct, you allowed the function to update the original data rather than taking a local copy.

I can see how the new code works. But the stuff about parentheses and * notation doesn't make the code all that readable. I wonder if there's something that would help with that.

means
(*)
age

I can see how the new code works. But the stuff about parentheses and * notation doesn't make the code all that readable. I wonder if there's something that would help with that.



It is more readable.

Because you need to be careful to use parentheses in the right way when you're dealing with pointers, the inventors of the C language came up with a simpler and easier-to-read piece of syntax. These two expressions mean the same thing:

Yes, there is another struct pointer notation that is more readable.

Because you need to be careful to use parentheses in the right way when you're dealing with pointers, the inventors of the C language came up with a simpler and easier-to-read piece of syntax.

These two expressions mean the same thing:

//These two mean the same thing:

(*t).age

t -> age

So, t->age means, "The age field in the struct that t points to."

That means you can also write the function like this:

```
34 void happy_birthday(turtle *a)
35 {
36     a->age = a->age + 1;
37     printf("Happy Birthday %s! You are now %i years old!\n", a->name, a->age);
38 }
```

Safe Cracker - Shit, its late at night in the bank vault.

Can you spin the correct combination to crack the safe?

Study these pieces of code and then see if you can find the correct combination that will allow you to get to the gold.

Be careful! There's a swag type and a swag field.

You need to
crack this
combination.

```
#include <stdio.h>

typedef struct {
    const char *description;
    float value;
} swag;

typedef struct {
    swag *swag;
    const char *sequence;
} combination;

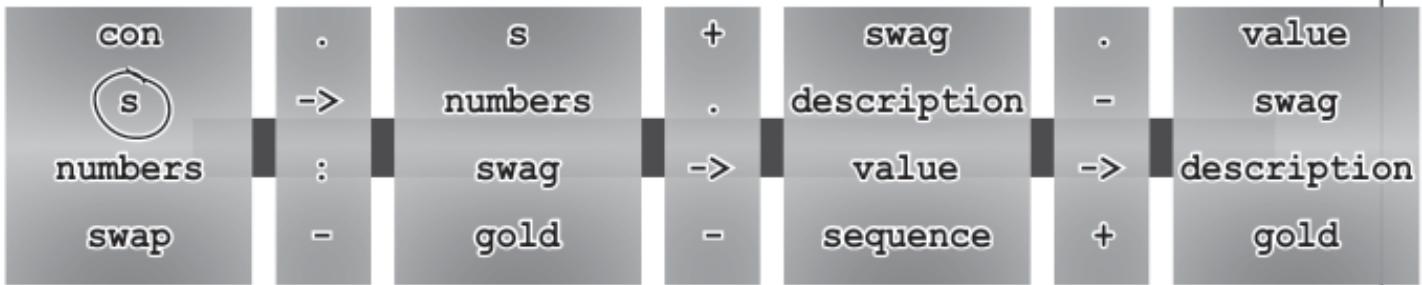
typedef struct {
    combination numbers;
    const char *make;
} safe;
```

```
1 #include <stdio.h>
2
3     typedef struct
4     {
5         const char *description;
6         float value;
7     } swag;
8
9     typedef struct
10    {
11         swag *swag;
12         const char *sequence;
13     } combination;
14
15     typedef struct
16    {
17         combination numbers;
18         const char *make;
19     } safe;
```

The bank created its safe like this:

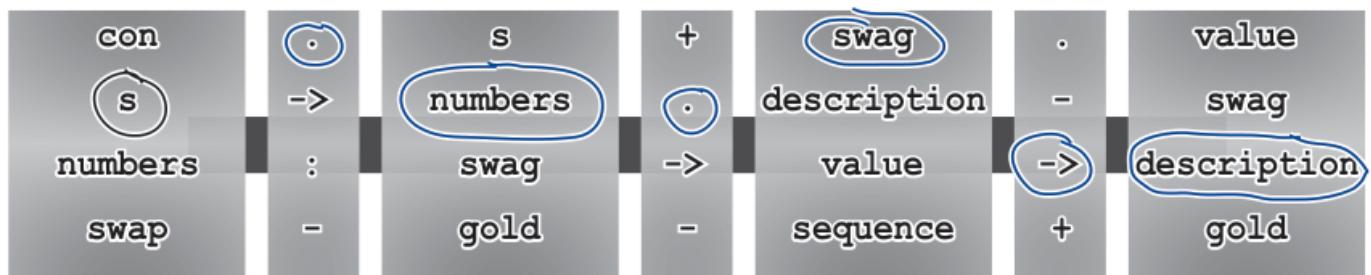
```
swag gold = {"GOLD!", 1000000.0};
combination numbers = {&gold, "6502"};
safe s = {numbers, "RAMACON250"};
```

What combination will get you to the string "GOLD!"? Select one symbol or word from each column to assemble the expression.



Why are values copied to parameter variables? The computer will pass values to a function by assigning values to the function's parameters. And all assignments copy values.

Why isn't `*t.age` just read as `(*t).age`? Because the computer evaluates the dot operator before it evaluates the `*`.



So you can display the gold in the safe with:

```
printf("Contents = %s\n", s.numbers.swag->description);
```

```
printf("Contents = %s\n", s.numbers.swag->description);
```

SUMMARY

When you call a function, the values are copied to the parameter variables.

You can create pointers to structs, just like any other type.

`pointer->field` is the same as `(*pointer).field`.

The `->` notation cuts down on parentheses and makes the code more readable.

`(*pointer).field = pointer -> field`

SAME TYPE OF THING NEEDS DIFFERENT DATA

Structs enable you to model more complex things from the real world.

But there are pieces of data that don't have a single data type.



So if you want to record, say, a quantity of something, and that quantity might be a count, a weight, or a volume, how would you do that?

Well, you could create several fields with a struct, like this:

```
11  typedef struct  
12  {  
13      short count;  
14      float weight;  
15      float volume;  
16      ....  
17  } fruit;
```

But there are a few reasons why this is not a good idea:

But there are a few reasons why this is not a good idea:

- ★ It will take up more space in memory.
- ★ Someone might set more than one value.
- ★ There's nothing called "quantity."

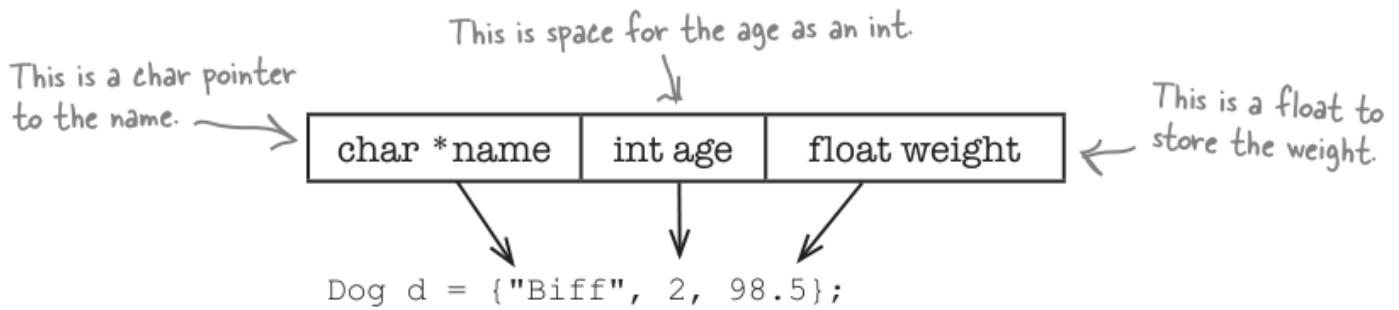
It would be really useful if you could specify something called quantity in a data type and then decide for each particular piece of data whether you are going to record a count, a weight, or a volume against it.

In C, you can do just that by using a union.

UNIONS

A union lets you reuse memory space.

Every time you create an instance of a struct, the computer will lay out the fields in memory, one after the other:



A union is different.

A union will use the space for just one of the fields in its definition.

So, if you have a union called `quantity`, with fields called `count`, `weight`, and `volume`, the computer will give the union enough space for its largest field, and then leave it up to you which value you will store in there.

Whether you set the `count`, `weight`, or `volume` field, the data will go into the same space in memory:

quantity (might be a float or a short)

A union looks like a struct, but it uses the union keyword.

If a float takes 4 bytes, and a short takes 2, then this space will be 4 bytes long.

```
typedef union {  
    short count;  
    float weight;  
    float volume;  
} quantity;
```

Each of these fields will be stored in the same space.

These are all different types, but they're all quantities.



Count oranges.



Weigh grapes.



Measure juice.

`typedef union` //union is like a struct but uses the union keyword.

```
{  
    short count; //all these fields will be stored in the same memory space.  
    float weight; //float = 4 bytes, and short = 2 bytes, the space = 4 bytes.  
    float weight;  
    float volume;  
} quantity;
```

How do you use a union? When you declare a union variable, there are a few ways of setting its value.

C89 style for the first field If the union is going to store a value for the first field, then you can use C89 notation.

To give the union a value for its first field, just wrap the value in braces:

`quantity q = {4};` ← This means the quantity is a count of 4.

Designated initializers set other values: A designated initializer sets a union field value by name, like this.

`quantity q = {.weight=1.5};` ← This will set the union for a floating-point weight value.

Set the value with dot notation: The third way of setting a union value is by creating the variable on one line, and setting a field value on another line:

```
quantity q;  
q.volume = 3.7;
```

Remember: whichever way you set the union's value, there will only ever be one piece of data stored.

The union just gives you a way of creating a **variable that supports several different data types**.

QUESTIONS:

Why is a union always set to the size of the largest field? The computer needs to make sure that a union is always the same size. The only way it can do that is by making sure it is large enough to contain any of the fields.

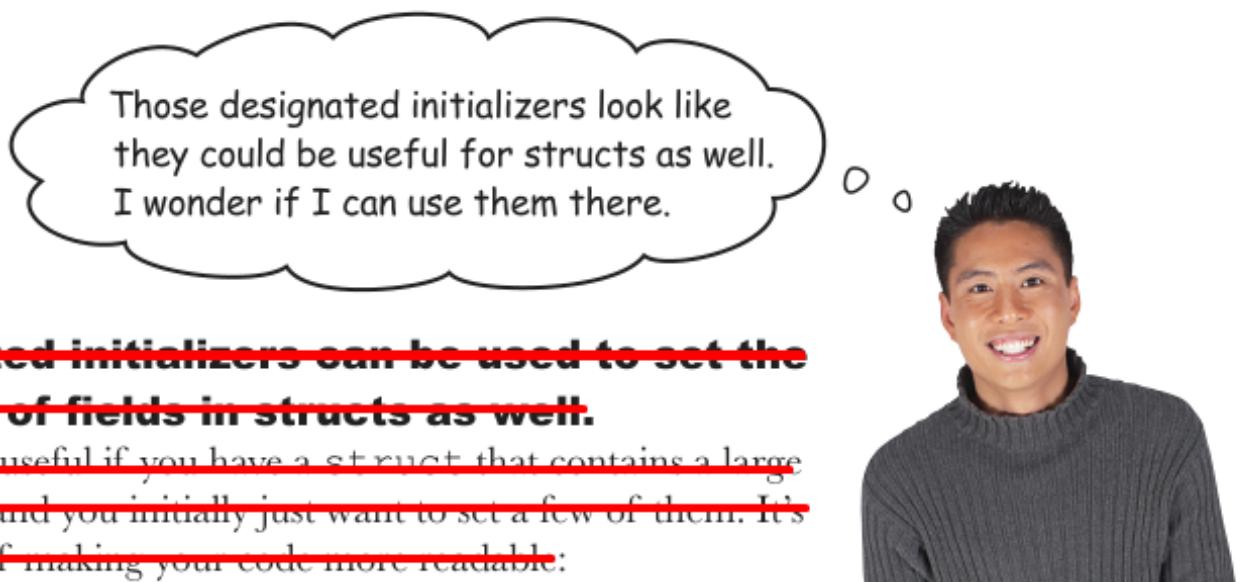
Why does the C89 notation only set the first field? Why not set it to the first float if I pass it a float value? To avoid ambiguity.

If you had, say, a float and a double field, should the computer store {2.1} as a float or a double?

By always storing the value in the first field, you know exactly how the data will be initialized.

The Polite Guide to Standards: Designated initializers allow you to set struct and union fields by name and are part of the C99 C standard. They are supported by most modern compilers, but be careful if you are using some variant of the C language. For example, Objective C supports designated initializers, but C++ does not.

DESIGNATED INITIALIZERS



~~Designated initializers can be used to set the initial values of fields in structs as well.~~

~~They are very useful if you have a struct that contains a large number of fields and you initially just want to set a few of them. It's a good way of making your code more readable:~~

Yes, designated initializers can be used to set the initial values of fields in structs as well.

They can be very useful if you have a struct that contains a large number of fields and you initially just want to set a few of them.

It's also a good way of making your code more readable:

```
32  typedef struct
33  {
34      const char *color;
35      int gears;
36      int height;
37
38  } bike;
39
40  bike b = { .height = 17, .gears = 21};
```

unions are often used with structs:

Once you've created a union, you've created a new data type.

That means you can use its values anywhere you would use another data type like an int or a struct. For example, you can combine them with structs:

```
46  typedef struct
47  {
48      const char *name;
49      const char *country;
50      quantity amount;
51
52  }fruit_order;
```

And you can access the values in the struct/union combination using the dot or `->` notation you used before:

```
55  fruit_order apples = {"apples", "England", .amount.weight=4.2};
56  printf("This order contains %2.2f lbs of %s\n", apples.amount.weight, apples.name);
```

And you can access the values in the struct/union combination using the dot or `->` notation you used before:

Here, you're using a double designated identifier.
.amount for the struct and
.weight for the .amount.

It's `.amount` because that's the name of the struct quantity variable.

```
fruit_order apples = {"apples", "England", .amount.weight=4.2};  
printf("This order contains %2.2f lbs of %s\n", apples.amount.weight, apples.name);
```

↑
This will print "This order contains 4.20 lbs of apples."

Mixed-Up Mixers: It's Margarita Night at the Head First Lounge, but after one too many samples, it looks like the guys have mixed up their recipes.

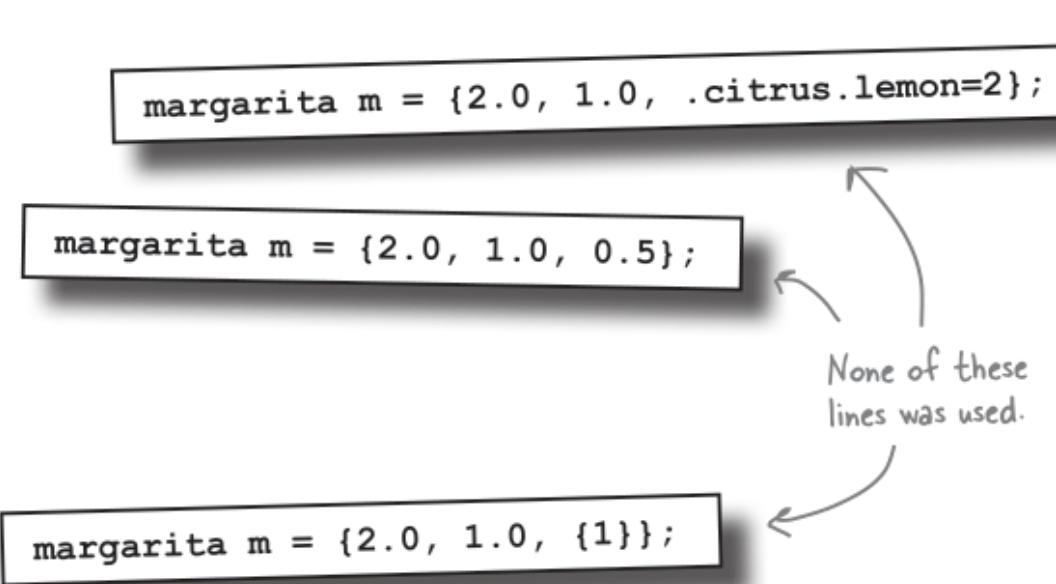
See if you can find the matching code fragments for the different margarita mixes. Here are the basic ingredients:

```
60  typedef union  
61  {  
62      float lemon;  
63      int lime_pieces;  
64  
65  } lemon_lime;  
66  
67  typedef struct  
68  {  
69      float tequila;  
70      float cointreau;  
71      lemon_lime citrus;  
72  
73  } margarita;
```

Here are the values:

```
77  margarita m = {2.0, 1.0, 0.5};  
78  
79  margarita m = {2.0, 1.0, .citrus.lemon = 2};  
80  
81  margarita m = {2.0, 1.0, ,{0.5}};  
82  
83  margarita m = {2.0, 1.0, {.lime_pieces=1}};  
84  
85  margarita m = {2.0, 1.0, {1}};  
86  
87  margarita m = {2.0, 1.0, {2}};
```

Solution 1:



Solution 2 😊:

```
margarita m = {2.0, 1.0, {2}};  
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%2.1f measures of juice\n",  
    m.tequila, m.cointreau, m.citrus.lemon);  
  
2.0 measures of tequila.  
1.0 measures of cointreau.  
2.0 measures of juice.
```

Solution 3:

```
margarita m = {2.0, 1.0, {0.5}};
printf("%2.1f measures of tequila\n %2.1f measures of cointreau\n %2.1f measures of juice\n",
       m.tequila, m.cointreau, m.citrus.lemon);

2.0 measures of tequila.
1.0 measures of cointreau.
0.5 measures of juice.
```

Solution 4:

```
margarita m = {2.0, 1.0, {.lime_pieces=1}};
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%i pieces of lime\n",
       m.tequila, m.cointreau, m.citrus.lime_pieces);

2.0 measures of tequila.
1.0 measures of cointreau.
1 pieces of lime.
```

Be the compiler:



BE the Compiler Solution
One of these pieces of code compiles; the other doesn't. Your job is to play like you're the compiler and say which one compiles, and why the other one doesn't.

margarita m = {2.0, 1.0, {0.5}};
This one compiles perfectly. It's actually just one of the drinks above!

margarita m;
m = {2.0, 1.0, {0.5}};

This one doesn't compile because the compiler will only know that {2.0, 1.0, {0.5}} represents a struct if it's used on the same line that a struct is declared. When it's on a separate line, the compiler thinks it's an array.

you are here ▶

253

Hey, wait a minute... You're setting all these different values with all these different types and you're storing them in the same place in memory.

How do I know if I stored a float in there once I've stored it? What's to stop me

from reading it as a short or something??? Hello?

Hey, wait a minute... You're setting all these different values with all these different types and you're storing them in the same place in memory... How do I know if I stored a float in there once I've stored it? What's to stop me from reading it as a short or something??? Hello?



That's a really good point: you can store lots of possible values in a union, but you have no way of knowing what type it was once it's stored.

The compiler won't be able to keep track of the fields that are set and read in a union, so there's nothing to stop us setting one field and reading another.

Is that a problem? Sometimes it can be a BIG PROBLEM.

```
124 #include <stdio.h>
125
126 typedef union
127 {
128     float weight;
129     int count;
130
131 }cupcake;
132
133 int main()
134 {
135     cupcake order = {2};
136     printf("Cupcakes quantity: %i\n", order.count);
137     return 0;
138 }
```

```
#include <stdio.h>
typedef union {
    float weight;
    int count;
} cupcake;
int main()
{
    cupcake order = {2};
    printf("Cupcakes quantity: %i\n", order.count);
    return 0;
}
```

By mistake, the programmer has set the weight, not the count.

She set the weight, but she's reading the count.

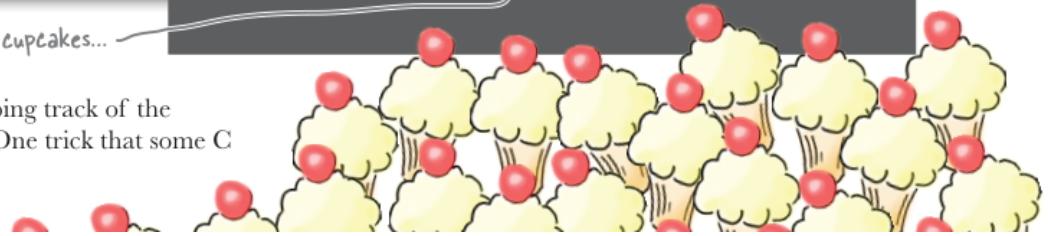


This is what the program did.

```
File Edit Window Help
> gcc badunion.c -o badunion && ./badunion
Cupcakes quantity: 1073741824
```

That's a lot of cupcakes...

You need some way, then, of keeping track of the values we've stored in a union. One trick that some C coders use is to create an **enum**.



You need some way, then, of keeping track of the values we've stored in a union.

One trick that some C coders use is to create an **enum**.

An **enum** variable stores a symbol.

Sometimes you don't want to store a number or a piece of text.

Instead, you want to store something from a list of symbols.

If you want to record a day of the week, you only want to store MONDAY, TUESDAY, WEDNESDAY, etc. You don't need to store the text, because there are only ever going to be seven different values to choose from.

That's why **enums** were invented. **enum** lets you create a list of symbols, like this:

Possible colors
in your enum.

enum colors {RED, GREEN, PUCE};

The values are separated by commas.
You could have given the type a proper name with `typedef`.

Any variable that is defined with a type of `enum colors` can then only be set to one of the keywords in the list.

So you might define an `enum colors` variable like this:

140 | `enum colors favorite = PUCE;`

Under the covers, the computer will just assign numbers to each of the symbols in your list, and the enum will just store a number.

But you don't need to worry about what the numbers are; your C code can just refer to the symbols.

That'll make your code easier to read, and it will prevent storing values like `REB` or `PUSE`:

The computer will spot that this is
not a legal value, so it won't compile.



`enum colors favorite = PUSE;`



So that's how enums work, but how do they help you keep track of unions? Let's look at an example...

Because you can create new data types with enums, you can store them inside structs and unions.

In this program, an enum is being used to track the kinds of quantities being stored.

Do you think you can work out where the missing pieces of code go?

```
#include <stdio.h>
typedef enum
{
    COUNT, POUNDS, PINTS
} unit_of_measure;

typedef union
{
    short count;
    float weight;
    float volume;
} quantity;

typedef struct
{
    const char *name;
    const char *country;
    quantity amount;
    unit_of_measure units;
} fruit_order;
```

```
void display(fruit_order order)
{
    printf("This order contains ");
    if(order.units == PINTS)
        printf("%2.2f pints of %s\n", order.amount.volume, order.name);
    else if(order.units == POUNDS)
        printf("%2.2f lbs of %s\n", order.amount.weight, order.name);
    else
        printf("%i %s\n", order.amount.count, order.name);
}

int main()
{
    fruit_order apples = {"apples", "England", .amount.count=144, COUNT};
    fruit_order strawberries = {"strawberries", "Spain", .amount.weight = 17.6, POUNDS};
    fruit_order oj = {"Orange juice", "U.S.A", .amount.volume = 10.5, PINTS};
    display(apples);
    display(strawberries);
    display(oj);
    return 0;
}
```

Output:

```
C:\Users\HP-EliteBook\Downloads\C\SameTypeDifferentData\bin\Debug\SameTypeDifferentData.exe
This order contains 144 apples
This order contains 17.60 lbs of strawberries
This order contains 10.50 pints of Orange juice

Process returned 0 (0x0)  execution time : 1.936 s
Press any key to continue.
```

The story:

union: ...so I said to the code, “Hey, look. I don’t care if you gave me a `float` or not. You asked for an `int`. You got an `int`.”

struct: Dude, that was totally uncalled for.

union: That’s what I said. It’s totally uncalled for.

struct: Everyone knows you only have one storage location.

union: Exactly. Everything is one. I’m, like, Zen that way...

enum: What happened, dude?

struct: Shut up, `enum`. I mean, the guy was crossing the line.

union: I mean, if he had just left a record. You know, said, I stored this as an `int`. It just needed an `enum` or something.

enum: You want me to do what?

struct: Shut up, `enum`.

union: I mean, if he’d wanted to store several things at once, he should have called you, am I right?

struct: Order. That’s what these people don’t grasp.

Part 2:

struct: Order. That's what these people don't grasp.

enum: Ordering what?

struct: Separation and sequencing. I keep several things alongside each other. All at the same time, dude.

union: That's just my point.

struct: All. At. The. Same. Time.

enum: (Pause) So has there been a problem?

union: Please, enum? I mean these people just need to

Part 3:

make a decision. Wanna store several things, use you. But store just one thing with different possible types? Dude's your man.

struct: I'm calling him.

union: Hey, wait...

enum: Who's he calling, dude?

struct/union: Shut up, enum.

union: Look, let's not cause any more problems here.

struct: Hello? Could I speak to the Bluetooth service, please?

union: Hey, let's just think about this.

struct: What do you mean, he'll give me a callback?

union: I'm just. This doesn't seem like a good idea.

struct: No, let me leave you a message, my friend.

union: Please, just put the phone down.

enum: Who's on the phone, dude?

struct: Be quiet, enum. Can't you see I'm on the phone here? Listen, you just tell him that if he wants to store a float and an int, he needs to come see me. Or I'm going to come see him. Understand me? Hello? Hello?

union: Easy, man. Just try to keep calm.

Part 4:

union: Easy, man. Just try to keep calm.

struct: On hold? They put me on ^*^&^ing hold!

union: They what? Pass me the phone... Oh...that... man. The Eagles! I hate the Eagles...

enum: So if you pack your fields, is that why you're so fat?

struct: You are entering a world of pain, my friend.

Sometimes you want control at the bit level.

Let's say you need a struct that will contain a lot of yes/no values.

You could create the struct with a series of shorts or ints: And that would work.

The problem? The short fields will take up a lot more space than the single bit that you need for true/false values.

It's wasteful. It would be much better if you could create a struct that could hold a sequence of single bits for the values.

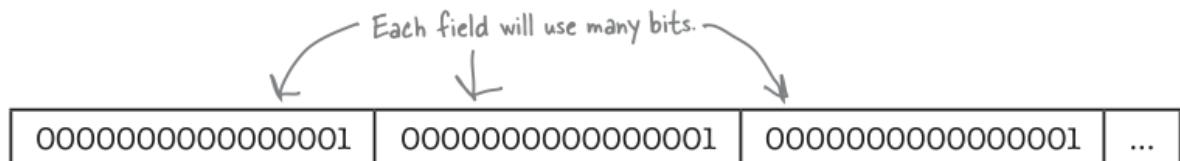
That's why bitfields were created.

Sometimes you want control at the bit level

Let's say you need a struct that will contain a lot of yes/no values. You *could* create the struct with a series of shorts or ints:

```
typedef struct {  
    short low_pass_vcf;  
    short filter_coupler;  
    short reverb;  
    short sequential;  
    ...  
} synth;
```

Each of these fields will contain 1 for true or 0 for false.



When you're dealing with binary value, it would be great if you had some way of specifying the 1s and 0s in a literal, like:

```
int x = 01010100;
```

Unfortunately, C doesn't support binary literals, but it does support hexadecimal literals.

Every time C sees a number beginning with 0x, it treats the number as base 16:

```
int x = 0x54;
```

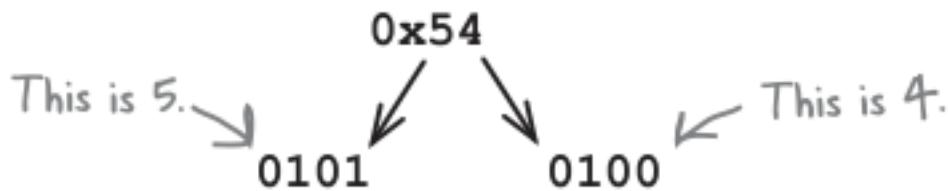
But how do you convert back and forth between hexadecimal and binary?

And is it any easier than converting binary and decimal? The good news is that you can convert hex to binary one digit at a time:

Each hexadecimal digit matches a binary digit of length 4.

All you need to learn are the binary patterns for the numbers 0-15, and you will soon

be able to convert binary to hex and back again in your head within seconds.



Bitfields store a custom number of bits.

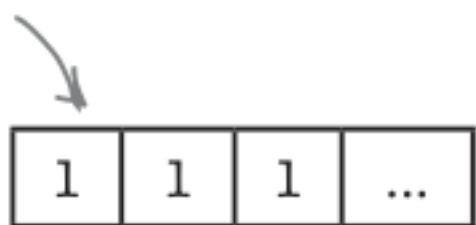
A bitfield lets you specify how many bits an individual field will store.

For example, you could write your struct like this:

```
//By using bitfields, you can make sure that each field takes up only one bit
typedef struct
{
    unsigned int low_pass_vcf: 1; //this means the field will only use 1 bit of storage.
    unsigned int filter_coupler: 1;
    unsigned int reverb: 1; //each field should be an unsigned int.
    unsigned int sequential: 1;

} synth;
```

By using bitfields, you can make sure each field takes up only one bit.



If you have a sequence of bitfields, the computer can squash them together to save space.

So if you have eight single-bit bitfields, the computer can store them in a single byte.



Watch it!

Bitfields can save space if they are collected together in a struct.

But if the compiler finds a single bitfield on its own, it might still have to pad it out to the size of a word. That's why bitfields are usually grouped together.

Let's see how good you are at using bitfields.

How many bits do I need? Bitfields can be used to store a sequence of true/false values, but they're also useful for other short-range values, like months of the year.

If you want to store a month number in a struct, you know it will have a value of, say, 0-11.

You can store those values in 4 bits. Why? Because 4 bits let you store 0-15, but 3 bits only store 0-7.

$$2^4 = 16.$$

```
unsigned int month_no: 4;
```

Back at the Head First Aquarium, they're creating a customer satisfaction survey.

Let's see if you can use bitfields to create a matching struct.



Aquarium Questionnaire

Is this your first visit?	
Will you come again?	
Number of fingers lost in the piranha tank:	
Did you lose a child in the shark exhibit?	
How many days a week would you visit if you could?	

```
typedef struct
{
    unsigned int first_visit: 1 //1 bit can store true or false
    unsigned int come_again: 1 // same to this one.
    unsigned int fingers_lost: 4 //4 bits are needed to store up to 10
    unsigned int shark_attack: 1 //true or false
    unsigned int days_a_week: 3 //3 bits can store up to 7
} survey;
```

Why doesn't C support binary literals? A: Because they take up a lot of space, and it's usually more efficient to write hex values.

Why do I need 4 bits to store a value up to 10? Four bits can store values from 0 to

binary 1111, which is 15. But 3 bits can only store values up to binary 111, which is 7.

So what if I try to put the value 9 into a 3-bit field? The computer will store a value of 1 in it, because 9 is 1001 in binary, so the computer transfers 001.

Are bitfields really just used to save space? No. They're important if you need to read low-level binary information. Such as? If you're reading or writing some sort of custom binary file.

A union allows you to store different data types in the same memory location.

A designated initializer sets a field value by name.

Designated initializers are part of the C99 standard. They are not supported in C++.

If you declare a union with a value in {braces}, it will be stored with the type of the first field.

The compiler will let you store one field in a union and read a completely different field. But be careful! This can cause bugs.

enums store symbols.

Bitfields allow you to store a field with a custom number of bits.

Bitfields should be declared as unsigned int.

bitfields to your toolbox. For a complete list of tooltips in the book, see Appendix ii.

typedef lets you create an alias for a data type.

A struct combines data types together.

You can read struct fields with dot notation.

You can initialize structs with {array, like, notation}.

-> notation lets you easily update fields using a struct pointer.

unions can hold different data types in one location.

Designated initializers let you set struct and union fields by name.

enums let you create a set of symbols.

unions can hold different data types in one location.

Designated initializers let you set struct and union fields by name.

enums let you create a set of symbols.

Bitfields give you control over the exact bits stored in a struct.

UNIONS AND ENUMS REPEATED

A union, like a structure, consists of one or more members, possibly of different types.

However, the compiler allocates only **enough space for the largest of the members**, which overlay each other within this space.

As a result, assigning a new value to one member alters the values of the other members as well.

To illustrate the basic properties of unions, Let's declare a union variable, `u`, with two members:

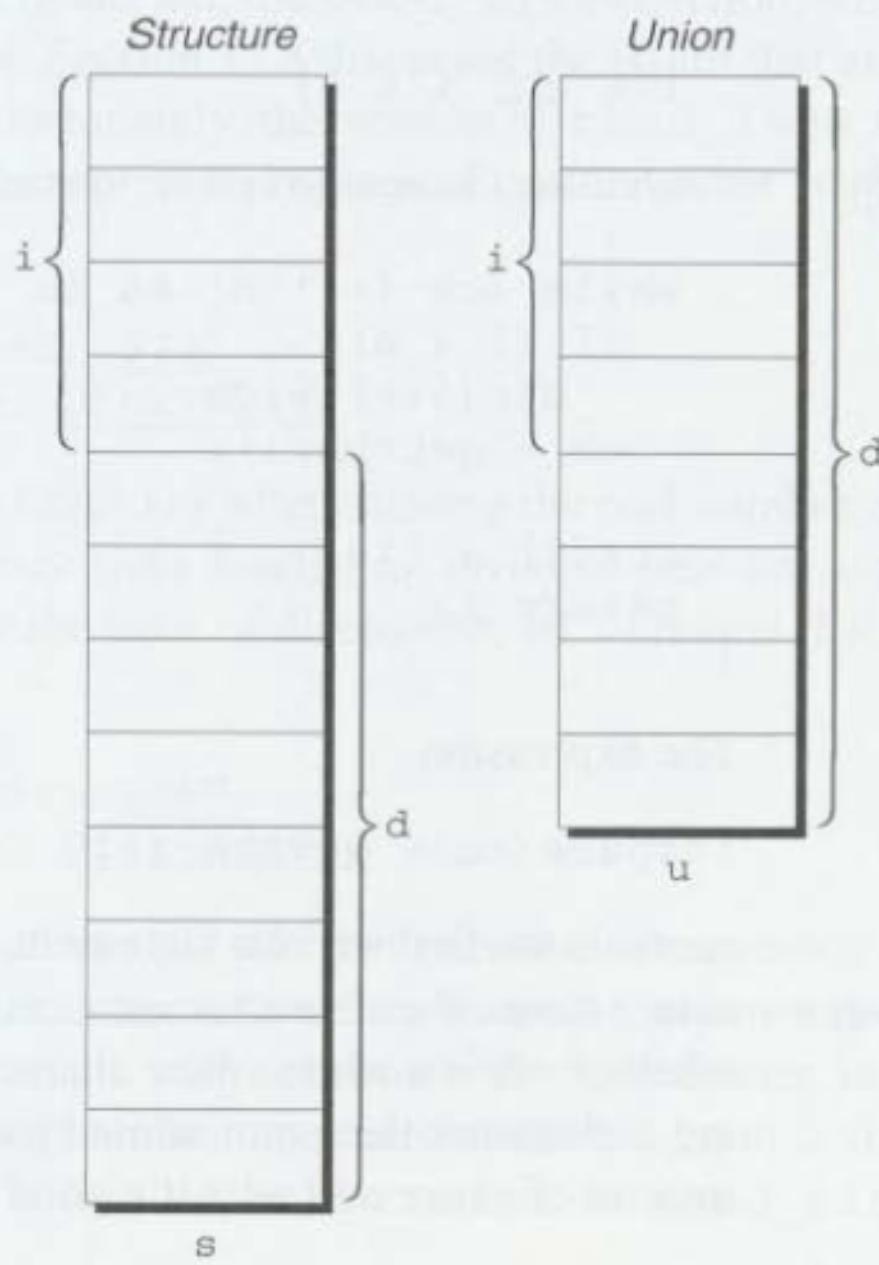
```
typedef union
{
    int i;
    double d;
}
```

```
typedef struct
{
    int i;
    double d;
}
```

The **structure s** and the **union u** differ in just one way.

The members of **s** are stored at **different addresses** in memory, while the members of **u** are stored at **the same address**.

Here's what **s** and **u** will look like in memory (assuming that **int** values require four bytes and **double** values take eight bytes):



In the `s` structure, `i` and `d` occupy different memory locations: the total size of `s` is 12 bytes.

In the `u` union, `i` and `d` overlap (`i` is really the first four bytes of `d`), so `u` occupies only eight bytes.

Also, `i` and `d` have the same address.

Members of a union are accessed in the same way as members of a structure.

To store the number 82 and 74.28384402 in the `i` and `d` member of `u`, we would write:

```
u.i = 82;  
u.d = 74.2838402;
```

Since the compiler overlays storage for the members of a union, changing one member alters any value previously stored in any of the other members.

Thus, if we store a value in `u.d`, any value previously stored in `u.i` will be lost. (If we examine the value of `u.i`, it will appear to be meaningless.)

Similarly, changing `u.i` corrupts `u.d`. Because of this property, we can think of `u` as a place to store either `i` or `d`, not both. (The structure `s` allows us to store `i` and `d`.)

The properties of unions are almost identical to the properties of structures.

We can declare union tags and union types in the same way we declare structure tags and types.

Like structures, unions can be copied using the `=` operator, passed to functions, and returned by functions.

Unions can even be initialized in a manner similar to structures.

However, only the first member of a union can be given an initial value.

For example, we can initialize the `i` member of `u` to 0 in the following way:

```
typedef union
{
    int i;
    double d;
}

} u = {0};
```

Notice the presence of the braces, which are required. The expression inside the braces must be constant.

Designated initializers, a C99 feature that we've previously discussed in the context of arrays and structures, can also be used with unions.

A designated initializer allows us to specify which member of a union should be initialized.

For example, we can initialize the d member of u as follows:

```
typedef union
{
    int i;
    double d;
}

} u = {.d = 10.0};
```

Only one member can be initialized, but it doesn't have to be the first one. There are several applications for unions.

We'll discuss two of these now.

Another application—viewing storage in different ways is highly machine dependent, so I'll postpone it until Section 20.3.

USING UNIONS TO SAVE SPACE

We'll often use unions as a way to save space in structures.

Suppose that we're designing a structure that will contain information about an item that's sold through a gift catalog.

The catalog carries only three kinds of merchandise: books, mugs, and shirts.

Each item has a stock number and a price, as well as other information that depends on the type of the item:

Books: Title, author, number of pages.

Mugs: Design.

Shirts: Design, colors available, sizes available.

Our first design attempt might result in the following structure:

```
struct catalog_item{  
    int stock_number;  
    double price;  
    int item_type;  
    char title[TITLE_LEN + 1];  
    char author[AUTHOR_LEN + 1];  
    int num_pages;  
    char design[DESIGN_LEN + 1];  
    int colors;  
    int sizes;  
};
```

The `item_type` member would have one of the values `BOOK`, `MUG`, or `SHIRT`.

The `colors` and `sizes` members would store encoded combinations of colors and sizes.

Although this structure is perfectly usable, it wastes space, since only part of the information in the structure is common to all items in the catalog.

If an item is a book, for example, there's no need to store `design`, `colors`, and `sizes`.

By putting a union inside the `catalog_item` structure, we can reduce the space required by the structure.

The members of the union will be structures, each containing the data that's needed for a particular kind of catalog item:

```
struct catalog_item
{
    int stock_number;
    double price;
    int item_type;
    union{
        struct{
            char title[TITLE_LEN+1];
            char author[AUTHOR_LEN+1];
            int num_pages;
        } book;
        struct{
            char design[DESIGN_LEN+1];
        } mug;
        struct{
            char design[DESIGN_LEN+1];
            int colors;
            int sizes;
        } shirt;
    } item;
};
```

Notice that the union (named item) is a member of the catalog_item structure, and the book, mug, and shirt structures are members of item.

If c is a catalog_item structure that represents a book, we can print the book's title in the following way:

```
int main()
{
    printf( format: "%s", c.item.book.title);
    return 0;
}
```

As this example shows, accessing a union that's nested inside a structure can be awkward: to locate a book title, we had to specify the name of a structure (c).

The name of the union member of the structure (item), the name of a structure member of the union (book), and then the name of a member of that structure (title).

We can use the catalog_item structure to illustrate an interesting aspect of unions.

Normally, it's not a good idea to store a value into one member of a union and then access the data through a different member, because assigning to one member of a union causes the values of the other members to be undefined.

However, the C standard mentions a special case: two or more of the members of the union are structures, and the structures begin with one or more matching members.

(These members need to be in the same order and have compatible types, but need not have the same name.)

If one of the structures is currently valid, then the matching members in the other structures will also be valid.

Consider the union embedded in the catalog_item structure.

It contains three structures as members, two of which (mug and shirt) begin with a matching member (design).

Now, suppose that we assign a value to one of the design members:

```
int main()
{
    printf( format: "%s", c.item.book.title);
    strcpy( Dest: c.item.mug.design, Source: "Cats");
    return 0;
}
```

The design member in the other structure will be defined and have the same value:

```
int main()
{
    printf( format: "%s", c.item.book.title);
    strcpy( Dest: c.item.mug.design, Source: "Cats");
    printf( format: "%s", c.item.shirt.design);
    return 0;
}
```

USING UNIONS TO BUILD MIXED DATA STRUCTURES

Unions have another important application: creating data structures that contain a mixture of data of different types.

Let's say that we need an **array whose elements are a mixture of int and double values.**

Since the elements of an **array must be of the same type**, it seems impossible to create such an array.

Using unions, though, it's relatively easy.

First, we define a union type whose members represent the different kinds of data to be stored in the array:

```
typedef union{
    int i;
    double d;
} Number;
```

Next, we create an array whose elements are Number values:

```
Number number_array[1000];
```

Each element of number_array is a Number union.

A Number union can store either an int value or a double value, making it possible to store a mixture of int and double values in number array.

For example, suppose that we want element 0 of number_array to store 5, while element 1 stores 8.395.

The following assignments will have the desired effect:

```
Number number_array[1000];
number_array[0].i = 5;
number_array[1].d = 8.395;
```

```
#include <stdio.h>

typedef union
{
    int i;
    double d;
} Numbers;

int main() {

    Numbers numbers_array[3] = { [0].i: 394.343098965, [1].i: 239, [2].i: 284.3996465465};
    printf( format: "%d %d %d\n", numbers_array[0].d, numbers_array[1].i, numbers_array[2].d);
    return 0;
}
```

ADDING A “TAG FIELD” TO UNIONS

Unions suffer from a major problem: there's no easy way to tell which member of a union was last changed and therefore contains a meaningful value.

Consider the problem of writing a function that displays the value currently stored in a Number union.

This function might have the following outline:

```
void print_number(Number n)
{
    if ("n contains an integer")
        printf( format: "%d", n.i);

    else
        printf( format: "%g", n.d);
}
```

Unfortunately, there's no way for `print_number` to determine whether `n` contains an integer or a floating-point number.

In order to keep track of this information, we can embed the union within structure that has one other member:

a "tag field" or "discriminant" whose purpose is to remind us what's currently stored in the union.

In the catalog item structure discussed earlier in this section, `item_type` served this purpose.

Let's convert the `Number` type into a structure with an embedded union:

```
#define INT_KIND 0
#define DOUBLE_KIND 1

typedef struct{
    int kind; //tag field
    union{
        int i;
        double d;
    } u;
} Number;
```

`Number` has **two members**, `kind` and `u`.

The value of `kind` will be either `INT_KIND` or `DOUBLE_KIND`.

Each time we assign a value to a member of `u`, we'll also change `kind` to remind us which member of `u` we modified.

For example, if `n` is a `Number` variable, an assignment to the `i` member of `u` would have the following appearance:

```
n.kind = INT_KIND;  
n.u.i = 82;
```

Notice that assigning to `i` requires that we first select the `u` member of `n`, then the `i` member of `u`.

NB: When we need to retrieve the number stored in a `Number` variable, `kind` will tell us which member of the union was the last to be assigned a value.

The `print_number` function can take advantage of this capability:

```
void print_number(Number n)  
{  
    if(n.kind == INT_KIND)  
        printf( format: "%d", n.u.i);  
    else  
        printf( format: "%g", n.u.d);  
}
```

It's the program's responsibility to change the tag field each time an assignment is made to a member of the union.

ENUMERATIONS

In many programs, we'll need variables that have only a small set of meaningful values.

A Boolean variable, for example, should have only two possible values: "true" and "false."

A variable that stores the suit of a playing card should have only four potential values: "clubs," "diamonds," "hearts," and "spades."

The obvious way to deal with such a variable is to declare it as an integer and have a set of codes that represent the possible values of the variable:

```
int s; //s will store a suit
s = 2; //2 represents hearts
```

Although this technique works, it leaves much to be desired.

Someone reading the program can't tell that s has only four possible values, and the significance of 2 isn't immediately apparent.

Using macros to define a suit "type" and names for the various suits is a step in the right direction:

```
#define SUIT int
#define CLUBS 0
#define DIAMONDS 1
#define HEARTS 2
#define SPADES 3

SUIT s;

s = HEARTS;
```

This technique is an improvement, but it's still not the best solution.

There's no indication to someone reading the program that the macros represent

values of the same "type."

If the number of possible values is more than a few, defining a separate macro for each will be tedious.

Moreover, the names we've defined—CLUBS, DIAMONDS, HEARTS, and SPADES—will be removed by the preprocessor, so they won't be available during debugging.

C provides a special kind of type designed specifically for variables that have a small number of possible values.

An **enumerated type** is a type whose values are listed ("enumerated") by the programmer, who must create a name (an enumeration constant) for each of the values.

The following example enumerates the values (CLUBS, DIAMONDS, HEARTS, and SPADES) that can be assigned to the variables s1 and s2:

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

Although enumerations have little in common with structures and unions, they're declared in a similar way.

Unlike the members of a structure or union, however, the names of enumeration constants must be different from other identifiers declared in the enclosing scope.

Enumeration constants are similar to constants created with the #define directive, but they're not equivalent.

For one thing, enumeration constants are subject to C's scope rules: if an enumeration is declared inside a function, its constants won't be visible outside the function.

Enumeration Tags and Type Names

We'll often need to create names for enumerations, for the same reasons that we name structures and unions.

As with structures and unions, there are two ways to name an enumeration: by declaring a tag or by using `typedef` to create a genuine type name.

Enumeration tags resemble structure and union tags.

To define the tag suit, for example, we could write:

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES}
```

suit variables would be declared in the following way:

```
enum suit sl, 82;
```

As an alternative, we could use `typedef` to make `Suit` a type name:

```
typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;
Suit sl, 82;
```

In C89, using `typedef` to name an enumeration is an excellent way to create a Boolean type:

```
typedef enum {FALSE, TRUE} Bool;
```

C99 has a built-in Boolean type, of course, so there's no need for a C99 programmer to define a `Bool` type in this way.

ENUMERATIONS AS INTEGERS

Behind the scenes, C treats enumeration variables and constants as integers.

By default, the compiler assigns the integers 0, 1, 2, ... to the constants in a particular enumeration.

In our suit enumeration, for example, CLUBS, DIAMONDS, HEARTS, and SPADES represent 0, 1, 2, and 3, respectively.

We're free to choose different values for enumeration constants if we like.

Let's say that we want CLUBS, DIAMONDS, HEARTS, and SPADES to stand for 1, 2, 3, and 4.

We can specify these numbers when declaring the enumeration:

```
enum suit {CLUBS = 1, DIAMONDS = 2, HEARTS = 3, SPADES = 4};
```

The values of enumeration constants may be arbitrary integers, listed in no particular order:

```
enum dept {RESEARCH = 20, PRODUCTION = 10, SALES = 25};
```

It's even legal for two or more enumeration constants to have the same value.

When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant.

(The first enumeration constant has the value 0 by default.)

In the following enumeration, BLACK has the value 0, LT_GRAY is 7, DK_GRAY is

8, and WHITE is 15:

```
enum EGA_colors {BLACK, LT GRAY = 7, DK GRAY, WHITE = 15};
```

Since enumeration values are nothing but thinly disguised integers, C allows us to mix them with ordinary integers:

```
int i;
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;
i = DIAMONDS; //i is now 1
s = 0; //s is now 0(CLUBS)
s++; //s is now 1(DIAMONDS)
i = s + 2; //i is now 3
```

The compiler treats s as a variable of some integer type; CLUBS, DIAMONDS, HEARTS, and SPADES are just names for the integers 0, 1, 2, and 3.

Although it's convenient to be able to use an enumeration value as an integer, it's dangerous to use an integer as an enumeration value.

For example, we might accidentally store the number 4—which doesn't correspond to any suit—into s.

USING ENUMERATIONS TO DECLARE "TAG FIELDS"

Enumerations are perfect for solving a problem that we encountered in Section 16.4: determining which member of a union was the last to be assigned a value.

In the Number structure, for example, we can make the kind member an enumeration instead of an int:

```
typedef struct
{
    enum {INT_KIND, DOUBLE_KIND} kind;
    union
    {
        int i;
        double d;
    } u;
} Number;
```

Buy HDMI cable.

The new structure is used in exactly the same way as the old one.

The advantages are that we've done away with the INT_KIND and DOUBLE_KIND macros (they're now enumeration constants), and we've clarified the meaning of kind, it's now obvious that kind has only two possible values: INT_KIND and DOUBLE_KIND.

Dynamic Memory

Building bridges



Sometimes, a single struct is simply not enough.

To model complex data requirements, you often need to link structs together.

In this chapter, you'll see how to use struct pointers to connect custom data types into large, complex data structures.

You'll explore key principles by creating linked lists.

You'll also see how to make your data structures cope with flexible amounts of data by dynamically allocating memory on the heap, and freeing it up when you're done.

And if good housekeeping becomes tricky, you'll also learn how `valgrind` can help.

FLEXIBLE STORAGES

You've looked at the different kinds of data that you can store in C, and you've also seen how you can store multiple pieces of data in an array.

But sometimes you need to be a little more flexible.

Imagine you're running a travel company that arranges flying tours through the islands.

Each tour contains a sequence of short flights from one island to the next.

For each of those islands, you will need to record a few pieces of information, such as the name of the island and the hours that its airport is open.

So how would you record that?

You could create a struct to represent a single island:

```
typedef struct{
    char *name;
    char *opens;
    char *closes;
} island;
```

Now if a tour passes through a sequence of islands, that means you'll need to record a list of islands, and you can do that with an array of islands:

```
island tour[4];
```

Coconut Airways flies C planes between the islands.



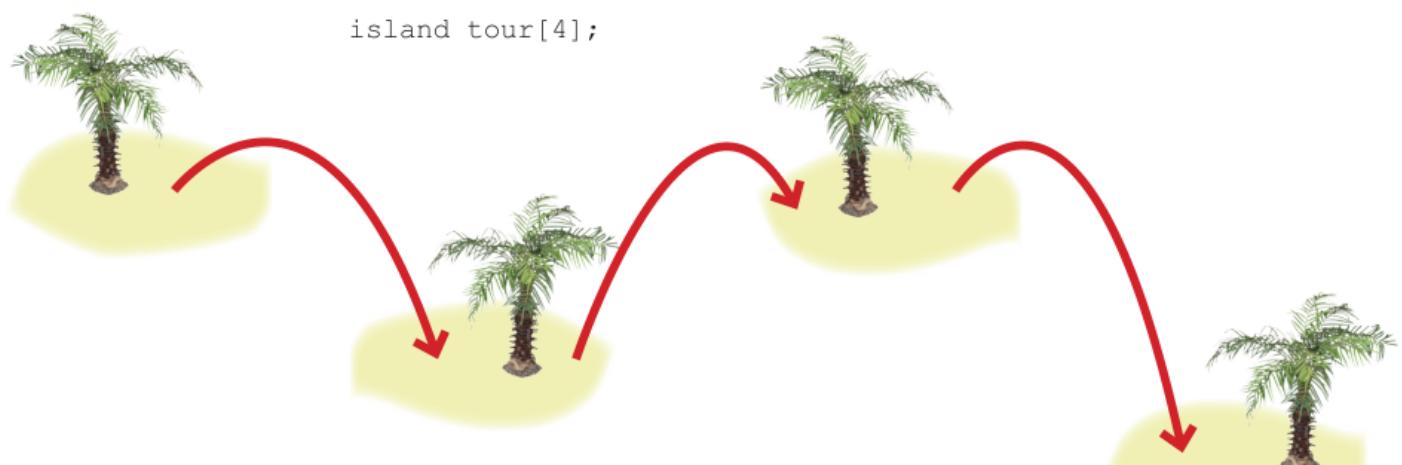
Coconut Airways flies
C planes between the
islands.

But there's a problem. Arrays are fixed length, which means they're not very flexible.

You can use one if you know exactly how long a tour will be. But what if you need to change the tour?

What if you want to add an **extra destination to the middle of the tour?**

To store a flexible amount of data, you need something more extensible than an array. You need a **linked list**.



Linked lists are like chains of data.

A linked list is an example of an abstract data structure.

It's called an abstract data structure because a linked list is general: it can be used to store a lot of different kinds of data.

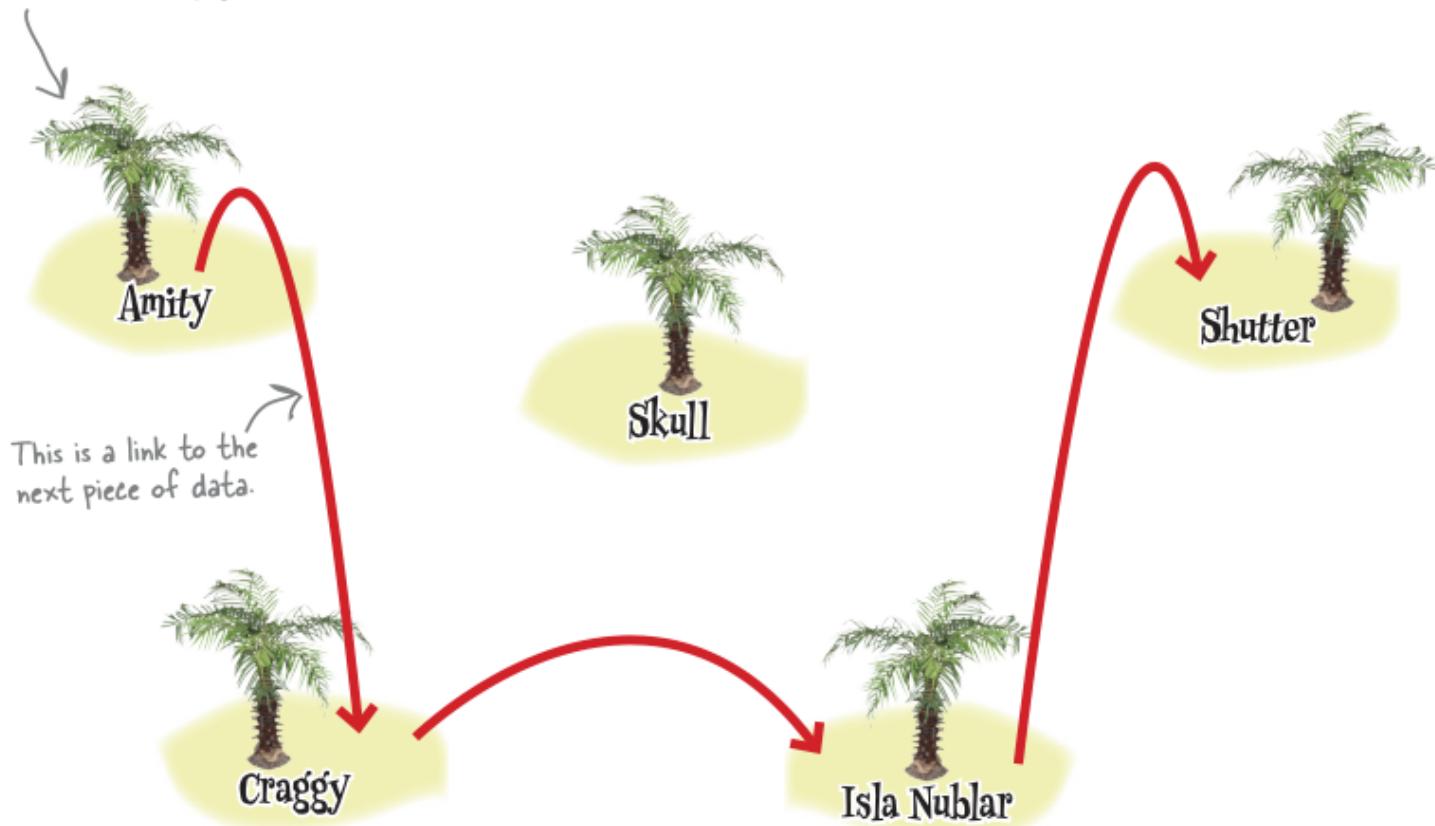
To understand how a linked list works, think back to our tour company.

A linked list stores a piece of data, and a link to another piece of data.

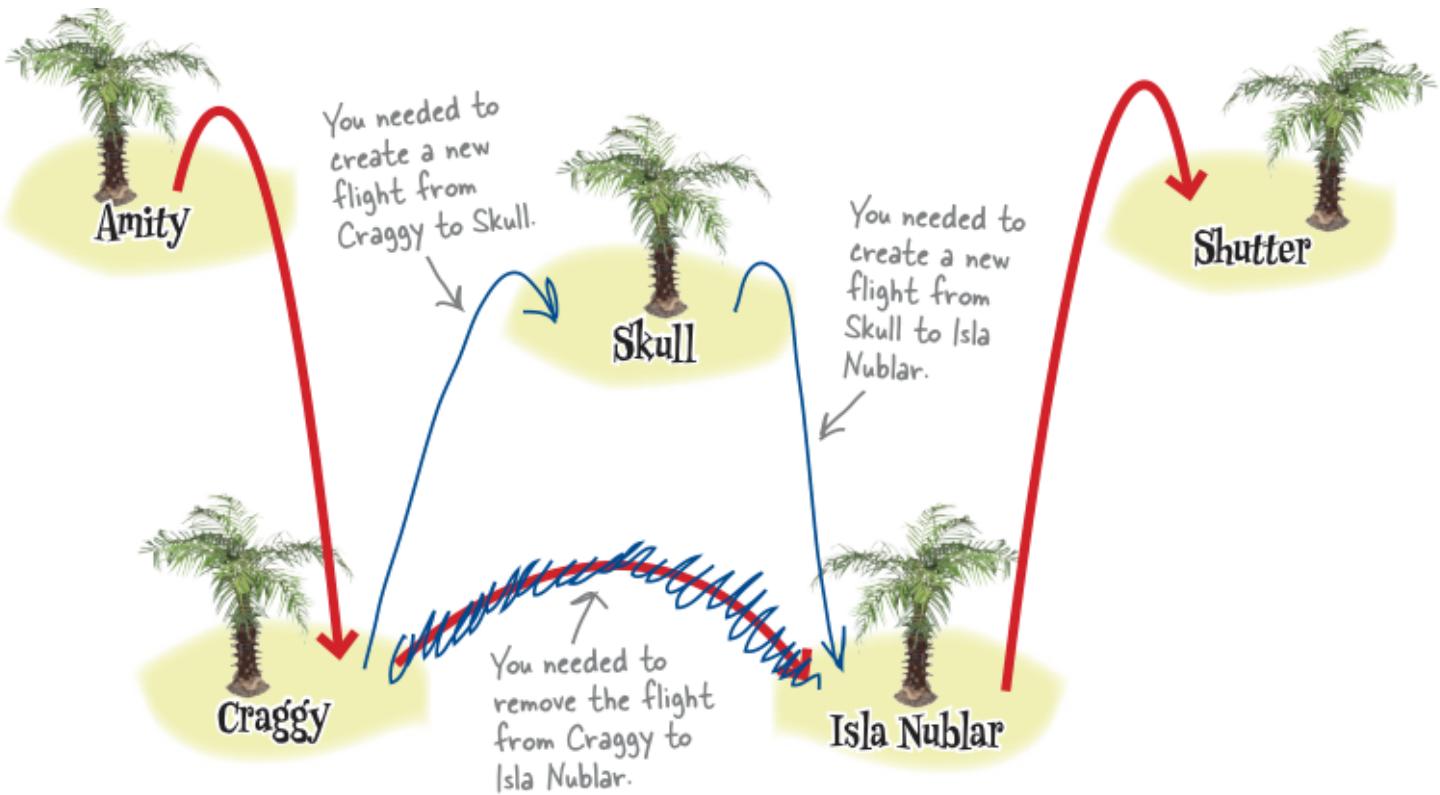
In a linked list, as long as you know where the list starts, you can travel along the list of links, from one piece of data to the next, until you reach the end of the list.

Using a pencil, change the list so that the tour includes a trip to Skull Island between Craggy Island and Isla Nublar.

You are storing a piece of data for each island.



Result:



Linked lists allow inserts. With just a few changes, you were able to add an extra step to the tour.

That's another advantage linked lists have over arrays: inserting data is very quick.

If you wanted to insert a value into the middle of an array, you would have to shuffle all the pieces of data that follow it along by one:

~~That's another advantage linked lists have over arrays: inserting data is very quick. If you wanted to insert a value into the middle of an array, you would have to shuffle all the pieces of data that follow it along by one:~~

If you wanted to insert an extra value after Craggy Island, you'd have to move the other values along one space.

This is an array. →

Amity	Craggy	Isla Nublar	Shutter
-------	--------	-------------	---------

~~linked lists allow you to store a variable amount of data, and they make it simple to add more data.~~

↑
And because an array is fixed length, you'd lose Shutter Island.

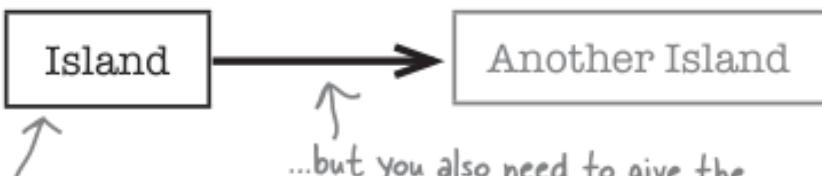
Each one of the structs in the list will need to connect to the one next to it.

A struct that contains a link to another struct of the same type is called a **recursive**

structure.

This is a recursive

structure for an island. →



You need to record all of the usual details for the island...

...but you also need to give the island a link to the next island.

Recursive structures contain pointers to other structures of the same type.

So if you have a flight schedule for the list of islands that you're going to visit, you can use a recursive structure for each island.

Let's look at how that works in more detail:

each island. Let's look at how that works in more detail:

You'll record these details for each island.

Island airport	
Name:	Amity
Opens:	9AM
Closes:	5PM
Next island:	Craggy

For each island, you'll also record the next island.

You must give the struct a name.

```
typedef struct island {  
    char *name;  
    char *opens;  
    char *closes;  
    struct island *next;  
} island;
```

You'll use strings for the name and opening times.

You store a pointer to the next island in the struct.



Recursive structures need names.

Watch it!

If you use the same name...

How do you store a link from one struct to the next? With a pointer.

That way, the island data will contain the address of the next island that we're going to visit.

So, whenever our code is at one island, it will always be able to hop over to the next island.

Let's write some code and start island hopping.

Recursive structures need names.



Watch it!

Recursive structures need names.

If you use the `typedef` command, you can normally skip giving the struct a proper name. But in a recursive structure, you need to include a pointer to the same type. C syntax won't let you use the `typedef alias`, so you need to give the struct a proper name. That's why the struct here is called `struct island`.

If you use the `typedef` command, you can normally skip giving the struct a proper name.

But in a recursive structure, you need to include a pointer to the same type.

C syntax won't let you use the `typedef alias`, so you need to give the struct a proper name.

That's why the struct here is called `struct island`.

```
typedef struct{
    char *name;
    char *opens;
    char *closes;
} island;

island amity = { .name: "Amity", .opens: "09:00", .closes: "17:00", NULL};
island craggy = { .name: "Craggy", .opens: "09:00", .closes: "17:00", NULL};
island isla_nublar = { .name: "Isla_Nublar", .opens: "09:00", .closes: "17:00", NULL};
island shutter = { .name: "Shutter", .opens: "09:00", .closes: "17:00", NULL};
```

Did you notice that we originally set the next field in each island to NULL?

In C, NULL actually has the value 0, but it's set aside specially to set pointers to 0.

```
amity.next = &craggy;
craggy.next = &isla_nublar;
isla_nublar.next = &shutter;
```

You have to be careful to set the next field in each island to the address of the next island.

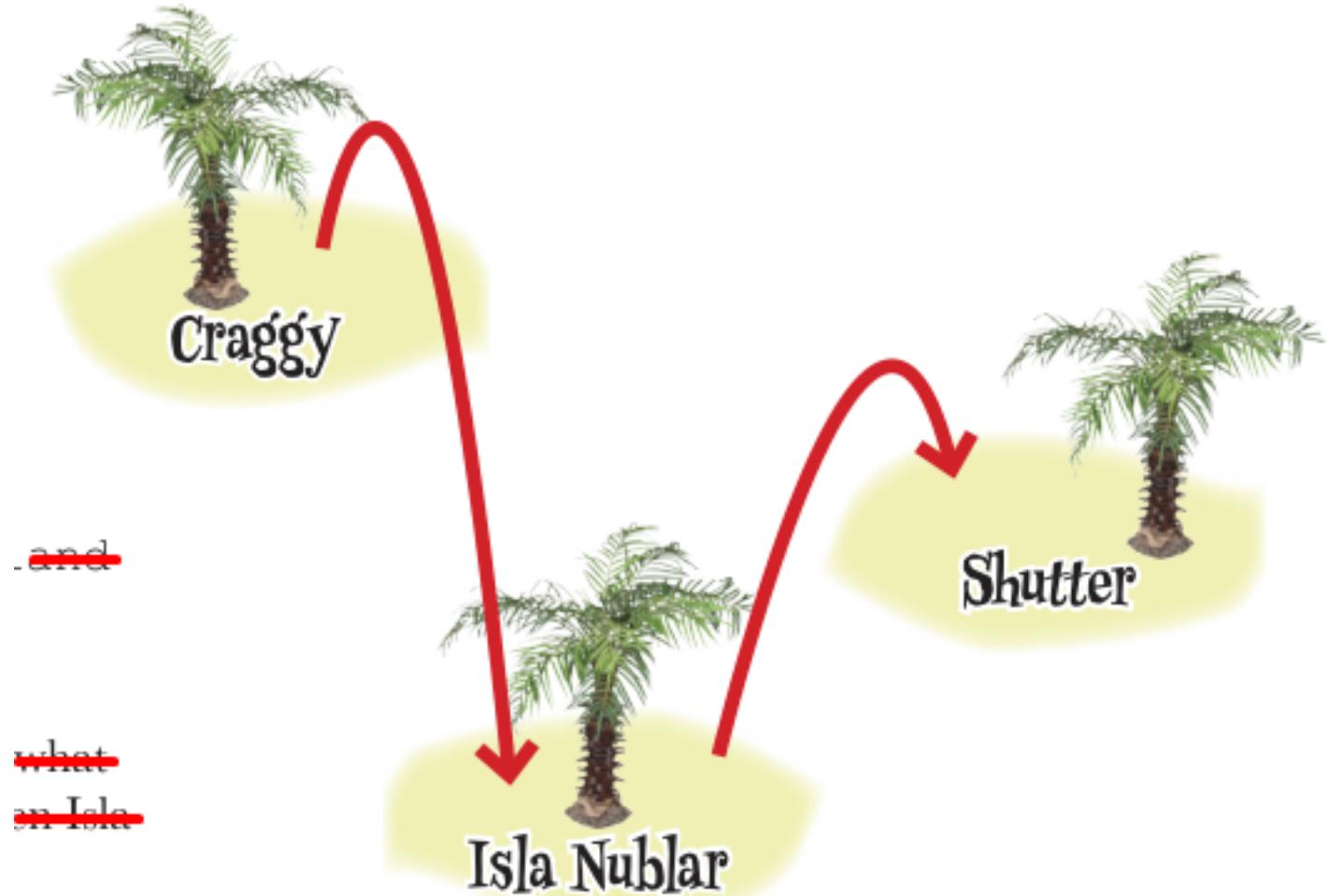
You'll use struct variables for each of the islands.

So now you've created a complete island tour in C, but what if you want to insert an excursion to Skull Island between Isla Nublar and Shutter Island?

Craggy Isla Nublar Shutter ...and link them together to form a tour

Once you've created each island, you can then connect them together:

```
amity.next = &craggy;
craggy.next = &isla_nublar;
isla_nublar.next = &shutter;
```



INSERTING INTO THE LIST

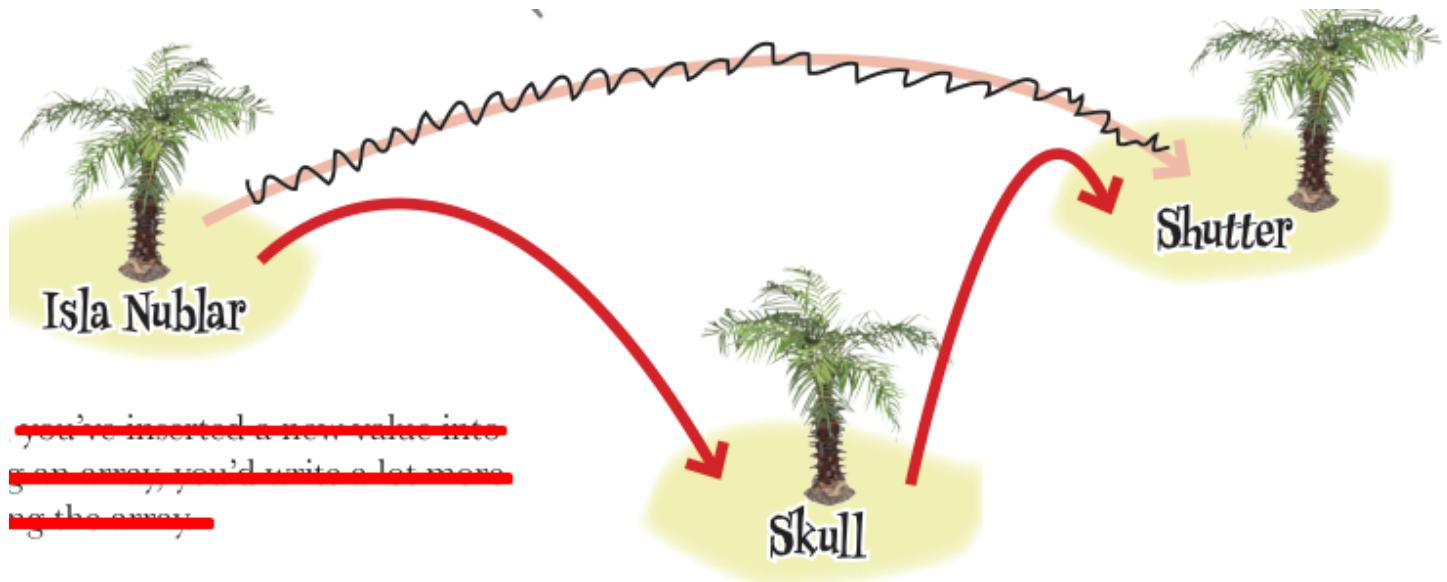
You can insert islands just like you did earlier, by changing the values of the pointers between islands:

```
island skull = {"Skull", "09:00", "17:00", NULL};  
isla_nublar.next = &skull;  
skull.next = &shutter;
```

In just two lines of code, you've inserted a new value into the list.

If you were using an array, you'd write a lot more code to shuffle items along the array.

OK, you've seen how to create and use linked lists. Now let's try out your new skills.



Other languages, like Java, have linked lists built in. Does C have any data structures? C doesn't really come with any data structures built in. You have to create them yourself.

What if I want to use the 700th item in a really long list? Do I have to start at the first item and then read all the way through? Yes, you do.

That's not very good. I thought a linked list was better than an array. You shouldn't think of data structures as being better or worse. They are either appropriate or inappropriate for what you want to use them for.

So if I want a data structure that lets me insert things quickly, I need a linked list, but if I want direct access I might use an array? Answer: Exactly.

You've shown a struct that contains a pointer to another struct. Can a struct contain a whole recursive struct inside itself? Answer: No

Why not? Answer: C needs to know the exact amount of space a struct will occupy in memory. If it allowed full recursive copies of the same struct, then one piece of data would be a different size than another.

The full linked-list:

```
#include <stdio.h>

typedef struct island{
    char *name;
    char *opens;
    char *closes;
    struct island *next;
}

} island;
```

```
int main() {
    island amity = { .name: "Amity", .opens: "09:00", .closes: "17:00", .next: NULL};
    island craggy = { .name: "Craggy", .opens: "09:00", .closes: "17:00", .next: NULL};
    island isla_nublar = { .name: "Isla Nublar", .opens: "09:00", .closes: "17:00", .next: NULL};
    island shutter = { .name: "Shutter", .opens: "09:00", .closes: "17:00", .next: NULL};

    amity.next = &craggy;
    craggy.next = &isla_nublar;
    isla_nublar.next = &shutter;
    island skull = { .name: "Skull", .opens: "09:00", .closes: "17:00", .next: NULL};
    isla_nublar.next = &skull;
    skull.next = &shutter;
    display(&amity);
    return 0;
}
```

```
void display(island *start)
{
    //keep looping until the current island has no next value.
    //at the end of each loop, skip to the next island.
    island *i = start;
    for(;i != NULL; i = i -> next){
        printf( format: "Name: %s open: %s-%s\n", i -> name, i -> opens, i -> closes);
    }
}
```

Result:

```
Name: Amity open: 09.00-17:00
Name: Craggy open: 09:00-17:00
Name: Isla Nublar open: 09:00-17:00
Name: Skull open: 09:00-17:00
Name: Shutter open: 09:00-17:00
```

OK, so now that you know the basics of how to work with recursive structs and lists, you can move on to the main program.



The Polite Guide to Standards

The code on this page declares a new variable, `skull`, right in the middle of the code. This is allowed only in C99 and C11. In ANSI C, you need to declare all your local variables at the top of a function.

You need to read the tour data from a file that looks like this:

There will
be some
more lines
after this.

Delfino Isle
Angel Island
Wild Cat Island
Neri's Island
Great Todday

The folks at the airline are still creating the file, so you won't know how long it is until runtime.

Each line in the file is the name of an island. It should be pretty straightforward to turn this file into a linked list. Right?

Hmm... So far, we've used a separate variable for each item in the list. But if we don't know how long the file is, how do we know how many variables we need? I wonder if there's some way to generate new storage when we need it.



Hmm... So far, we've used a separate variable for each item in the list.

But if we don't know how long the file is, how do we know how many variables we need?

I wonder if there's some way to **generate new storage when we need it**.

Yes, you need some way to create dynamic storage.

All of the programs you've written so far have used static storage.

Every time you wanted to store something, you've added a variable to the code.

Those variables have generally been stored in the stack.

Remember: **The stack** is the area of memory set aside for storing local variables.

So when you created the first four islands, you did it like this:

```
island amity = { .name: "Amity", .opens: "09.00", .closes: "17:00", .next: NULL};  
island craggy = { .name: "Craggy", .opens: "09:00", .closes: "17:00", .next: NULL};  
island isla_nublar = { .name: "Isla Nublar", .opens: "09:00", .closes: "17:00", .next: NULL};  
island shutter = { .name: "Shutter", .opens: "09:00", .closes: "17:00", .next: NULL};
```

Each island struct needed its own variable.

This piece of code will always create exactly four islands.

If you wanted the code to store more than four islands, you would need another local variable.

That's fine if you know how much data you need to store at compile time, but quite often, programs don't know how much storage they need until runtime.

If you're writing a web browser, for instance, you won't know how much data you'll need to store a web page until, well, you read the web page.

So C programs need some way to tell the operating system that they need a little extra storage, at the moment that they need it.

Programs need dynamic storage.



Use the heap for dynamic storage.

Most of the memory you've been using so far has been in the stack.

The **stack** is the area of memory that's used for **local variables**.

Each piece of data is stored in a variable, and each variable disappears as soon as you leave its function.

The trouble is, it's harder to get more storage on the stack at runtime, and that's where the heap comes in.

The **heap** is the place where a program stores data that will need to be available longer term.

It won't automatically get cleared away, so that means it's the perfect place to store data structures like our linked list.

You can think of heap storage as being a bit like reserving a locker in a locker room.

First, get your memory with **malloc()**.

Imagine your program suddenly finds it has a large amount of data that it needs to store at runtime.

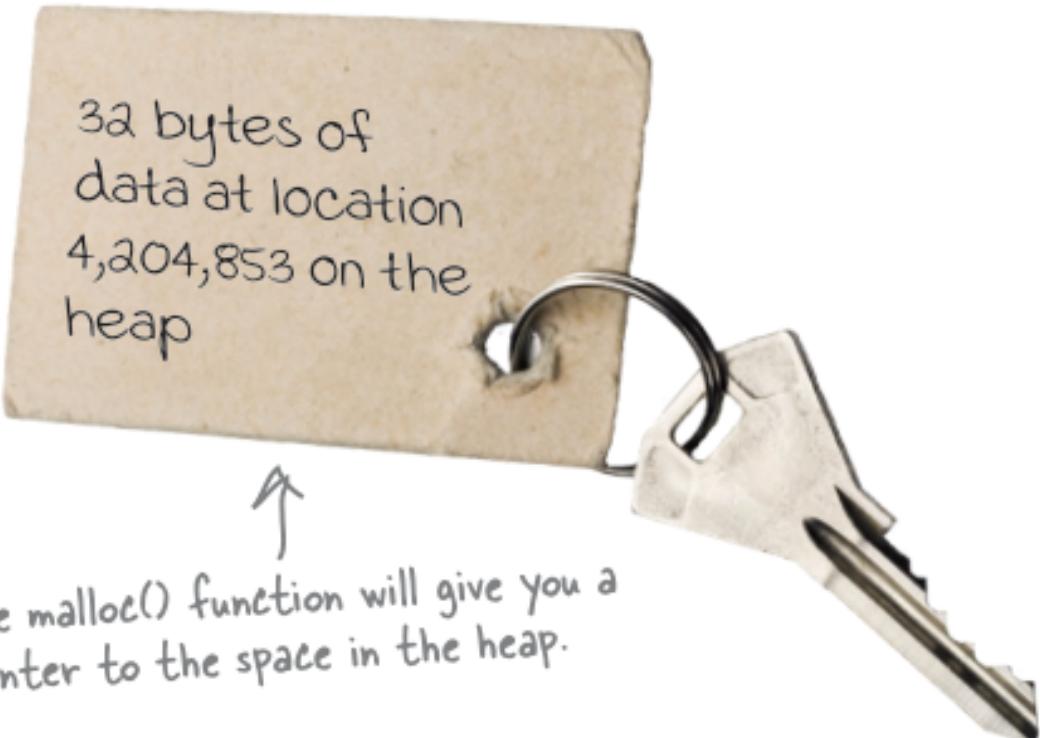
This is a bit like asking for a large storage locker for the data, and in C you do that with a function called **malloc()**.



Heap storage is like saving
valuables in a locker.

You tell the **malloc()** function exactly how much memory you need, and it asks the operating system to set that much memory aside in the heap.

The **malloc()** function then **returns a pointer to the new heap space**, a bit like getting a key to the locker.



It allows you access to the memory, and it can also be used to keep track of the storage locker that's been allocated.



Give the memory back when you're done.

The good news about heap memory is that you can keep hold of it for a really long time.

The bad news is you can keep hold of it for a really long time.

When you were just using the stack, you didn't need to worry about returning memory; it all happened automatically.

Every time you leave a function, the local storage is freed from the stack.

The heap is different.

Once you've asked for space on the heap, it will never be available for anything else until you tell the C Standard Library that you're finished with it.

There's only so much heap memory available, so if your code keeps asking for more and more heap space, your program will quickly start to develop memory leaks.

A memory leak happens when a program asks for more and more memory without releasing the memory it no longer needs.

**The heap has
only a fixed
amount of
storage available,
so be sure you
use it wisely.**

Memory leaks are among the most common bugs in C programs, and they can be really hard to track down.

The heap has only a fixed amount of storage available, so be sure you use it wisely.

Free memory by calling the `free()` function.

The `malloc()` function allocates space and gives you a pointer to it.

You'll need to use this pointer to access the data and then, when you're finished with the storage, you need to release the memory using the `free()` function.

It's a bit like handing your locker key back to the attendant so that the locker can be reused.

Thanks for the storage.



I'm done with it now.

Every time some part of your code requests heap storage with the `malloc()` function, there should be some other part of your code that hands the storage back with the `free()` function.

When your program stops running, all of its heap storage will be released automatically, but it's always good practice to **explicitly call `free()`** on every piece of dynamic memory you've created.

Let's see how `malloc()` and `free()` work.

Ask for memory with `malloc()`.

The function that asks for memory is called `malloc()` for **memory allocation**.

malloc() takes a single parameter: the number of bytes that you need.

Most of the time, you probably don't know exactly how much memory you need in bytes, so `malloc()` is almost always used with an operator called `sizeof`, like this:

```
malloc(sizeof(island)); //give me enough storage to store an island struct
```

`sizeof` tells you how many bytes a particular data type occupies on your system.

It might be a struct, or it could be some base data type, like `int` or `double`.

The `malloc()` function sets aside a chunk of memory for you, then **returns a pointer** containing the start address.

But what kind of pointer will that be? `malloc()` actually returns a **general-purpose pointer**, with type `void *`.

```
island *p = malloc( Size: sizeof(island)); ///give me enough storage to store an island struct
```

This means, "Create enough space for an island, and store the address in variable `p`."

Once you've created the memory on the heap, you can use it for as long as you like.

But once you've finished, you need to release the memory using the `free()` function.

`free()` needs to be given the address of the memory that `malloc()` created.

As long as the library is told where the chunk of memory starts, it will be able to check its records to see how much memory to free up.

So if you wanted to free the memory you allocated above, you'd do it like this:

```
free(p);
```

Remember: if you allocated memory with malloc() in one part of your program, you should always release it later with the free() function.

**Remember: if you
allocated memory
with malloc() in one
part of your program,
you should always
release it later with
the free() function.**

The aspiring actors are currently between jobs, so they've found some free time in their busy schedules to help you out with the coding.

They've created a utility function to create a new island struct with a name that you pass to it. The function looks like this:

```

island * create(char *name)
{
    island *i = malloc( Size: sizeof(island));
    i -> name = name; //a pointer to the name member of struct var i, is assigned the name array
    i -> opens = "09:00";
    i -> next = NULL;
    return i;
}

```

This is the new function.

This will create a new island struct on the heap.

These lines set the fields on the new struct.

The name of the island is passed as a char pointer.

island* create(char *name)

{

island *i = malloc(sizeof(island));

i->name = name;

i->opens = "09:00";

i->closes = "17:00";

i->next = NULL;

return i;

}

The function returns the address of the new struct.

It's using the malloc() function to create space on the heap.

The sizeof operator works out how many bytes are needed.



That's a pretty cool-looking function.

The actors have spotted that most of the island airports have the same opening and closing times, so they've set the opens and closes fields to default values.

The function returns a pointer to the newly created struct.

Look carefully at the code for the create() function. Do you think there might be any problems with it?

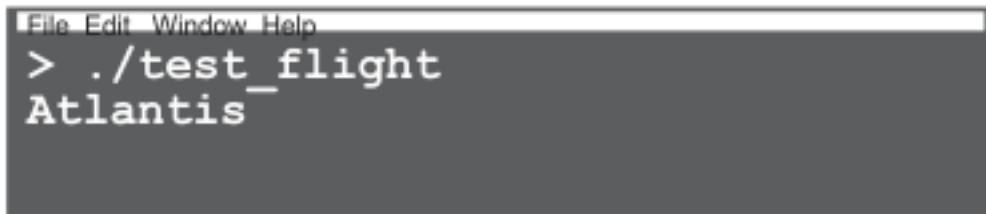
Once you've thought about it good and hard, turn the page to see it in action.

The Case of the Vanishing Island Captain's Log. 11:00. Friday. Weather clear.

A `create()` function using dynamic allocation has been written, and the coding team

says it is ready for air trials.

```
char name[80]; //create an array to store the island name
fgets( Buf: name,  MaxCount: 80,  File: stdin); //ask the user for the name of an island
island *p_island0 = create(name);
```



```
File Edit Window Help
> ./test_flight
Atlantis
```

15:35. Arrival at second island. Weather good. No wind. Entering details into new program.

```
fgets( Buf: name,  MaxCount: 80,  File: stdin); //ask user for the name of second island
island *p_island1 = create(name); //create the second island
p_island0 -> next = p_island1; //connects the first island to the second island
```



```
File Edit Window Help
Titchmarsh Island
```

17:50 Back at headquarters tidying up on paperwork. Strange thing. The flight log produced by the test program appears to have a bug.

When the details of today's flight are logged, the trip to the first island has been mysteriously renamed. Asking software team to investigate.

This will display the details of the list of islands using the function we created earlier.

What happened to Atlantis????

```
File Edit Window Help
Name: Titchmarsh Island
open: 09:00-17:00
Name: Titchmarsh Island
open: 09:00-17:00
```

The first island now has the same name as the second island!!!

What happened to the name of the first island?

Is there a bug in the create() function?

Does the way it was called give any clues?

When the code records the name of the island, it doesn't take a copy of the whole name string; it just records the address where the name string lives in memory.

Is that important? Where did the name string live?

We can find out by looking at the code that was calling the function: The program asks the user for the name of each island, but both times it uses the same local char array to store the name.

That means that the two islands share the same name string.

As soon as the local name variable gets updated with the name of the second island, the name of the first island changes as well.

```
char name[80]; //create an array to store the island name
fgets( Buf: name,  MaxCount: 80,  File: stdin); //ask the user for the name of an island
island *p_island0 = create(name);
fgets( Buf: name,  MaxCount: 80,  File: stdin); //ask user for the name of second island
island *p_island1 = create(name); //create the second island
p_island0 -> next = p_island1; //connects the first island to the second island
```

In C, you often need to make **copies of strings**.

You could do that by calling the `malloc()` function to create a little space on the heap and then manually copying each character from the string you are copying to the space on the heap. But guess what?

Other developers got there ahead of you. They created a function in the `string.h` header called **`strup()`**.

Let's say that you have a pointer to a character array that you want to copy:

```
char *s = "MONA LISA";
```



The `strup()` function can reproduce a complete copy of the string somewhere on the heap:

The `strup()` function can reproduce a complete copy of the string somewhere on the heap:

The `strup()` function works out how long the string is, and then calls the `malloc()` function to allocate the correct number of characters on the heap.

That's 10 characters from position `s` to the `\0` character, and `malloc(10)` tells me I've got space starting on the heap at location 2,500,000.

It then copies each of the characters to the new space on the heap.



2,500,000 is an
M; 2,500,001 is
an O; ...

That means that `strdup()` always creates space on the heap.

It can't create space on the stack because that's for local variables, and local variables get cleared away too often.

But because `strdup()` puts new strings on the heap, that means you must always remember to release their storage with the `free()` function.

Let's fix the code using the `strdup()` function:

```
island * create(char *name)
{
    island *i = malloc( Size: sizeof(island));
    i -> name = strdup( Src: name); //a pointer to the name member of struct var i, is assigned the name an
    i -> opens = "09:00";
    i -> next = NULL;

    char name[80]; //create an array to store the island name
    fgets( Buf: name, MaxCount: 80, File: stdin); //ask the user for the name of an island
    island *p_island0 = create(name);
    fgets( Buf: name, MaxCount: 80, File: stdin); //ask user for the name of second island
    island *p_island1 = create(name); //create the second island
    p_island0 -> next = p_island1; //connects the first island to the second island

    return i;
}
```

You can see that we only need to put the `strdup()` function on the name field. Can you figure out why that is?

It's because we are setting the `opens` and `closes` fields to string literals.

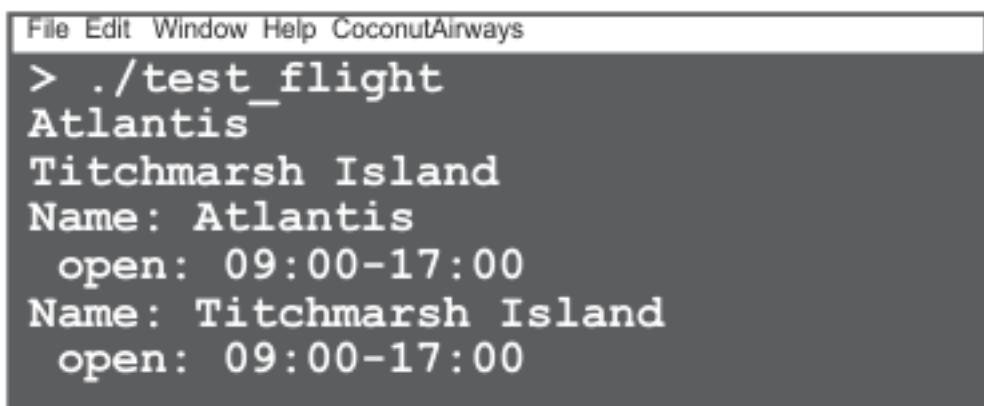
Remember way back when you saw where things were stored in memory?

String literals are stored in a **read-only area of memory** set aside for constant values.

Because you **always set the `opens` and `closes` fields to constant values**, you don't need to take a **defensive copy** of them, because they'll never change.

But you had to take a defensive copy of the `name` array, because something might come and update it later. So does it fix the code?

To see if the change to the `create()` function fixed the code, let's run your original code again:



```
File Edit Window Help CoconutAirways
> ./test_flight
Atlantis
Titchmarsh Island
Name: Atlantis
  open: 09:00-17:00
Name: Titchmarsh Island
  open: 09:00-17:00
```

Now that code works. Each time the user enters the name of an island, the `create()` function is storing it in a brand-new string.

OK, now that you have a function to create island data, let's use it to create a linked list from a file.

Catastrophe!

Your goal is to reconstruct the program so that it can read a list of names from Standard Input and then connect them together to form a linked list.

```

int main() {
    island *start = NULL;
    island *i = NULL;
    island *next = NULL;
    char *name[80];
    for(; fgets(Buf: name, MaxCount: 80, File: stdin) != NULL; i = next)
    {
        next = create(name);
        if(start == NULL)
            start = next;
        if(i != NULL)
            i -> next = next;
    }
    display(start);
    return 0;
}

```

Explained:

```

island *start = NULL;
island *i = NULL;
island *next = NULL;
char name[80];
for(; fgets(name, 80, stdin) != NULL; i = next) {
    next = create(name);
    if (start == NULL)
        start = next;
    if (i != NULL)
        i -> next = next;
}
display(start);

```

At the end of each loop, set *i* to the next island we created.

This creates *next* = *create(name)*; Read a string from the Standard Input.

We'll keep looping until we don't get any more strings.

The first time through, *start* is set to *NULL*, so set it to the first island.

Don't forget: *i* is a pointer, so we'll use *->* notation.

But wait! You're not done yet. Don't forget that if you ever allocate space with the

malloc() function, you need to release the space with the free() function.

The program you've written so far creates a linked list of islands in heap memory using malloc(), but now it's time to write some code to release that space once you're done with it.

Here's a start on a function called release() that will release all of the memory used by a linked list, if you pass it a pointer to the first island:

```
void release(island *start) {
    island *i = start;
    island *next = NULL;
    for (; i != NULL; i = next)
    {
        next = i -> next; //set next to point to the next island
        free( Memory: i -> name); //free the name string that you created with strdup()
        free( Memory: i); //only after freeing the name, should you free the island struct var i
        //if you've freed the island first you might not be able to reach the name to free it.
    }
}
```

Free the memory when you're done Now that you have a function to free the linked list, you'll need to call it when you've finished with it.

Your program only needs to display the contents of the list, so once you've done that, you can release it:

```
display(start);
release(start);
```

Once that's done, you can test the code.

```
File Edit Window Help FreeSpaceYouDon'tNeed
> ./tour < trip1.txt
Name: Delfino Isle
  Open: 09:00-17:00
Name: Angel Island
  Open: 09:00-17:00
Name: Wild Cat Island
  Open: 09:00-17:00
Name: Neri's Island
  Open: 09:00-17:00
Name: Great Todday
  Open: 09:00-17:00
Name: Ramita de la Baya
  Open: 09:00-17:00
Name: Island of the Blue Dolphins
  Open: 09:00-17:00
Name: Fantasy Island
  Open: 09:00-17:00
Name: Farne
  Open: 09:00-17:00
Name: Isla de Muert
  Open: 09:00-17:00
Name: Tabor Island
  Open: 09:00-17:00
Name: Haunted Isle
  Open: 09:00-17:00
Name: Sheena Island
  Open: 09:00-17:00
```

It works. Remember: you had no way of knowing how long that file was going to be.

In this case, because you are just printing out the file, you could have simply printed it out without storing it all in memory.

But because you do have it in memory, you're free to manipulate it. You could add in extra steps in the tour, or remove them.

You could reorder or extend the tour. Dynamic memory allocation lets you create the memory you need at **RUNTIME**.

And the way you access dynamic heap memory is with `malloc()` and `free()`.

Stack:

Heap? Are you there? I'm home.

Deep regression. Oops...excuse me... Just tidy that up...

The code just exited a function. Just need to free up the storage from those local variables.

Perhaps you're right. Mind if I sit?

I...think this is yours?

You really should consider getting somebody in to take care of this place.

How do you know? I mean, how do you know it hasn't just forgotten about it?

Hmmmm? Are you sure? Wasn't it written by the same woman who wrote that dreadful Whack-a-bunny game? Memory leaks everywhere. I could barely move for rabbit structs. Droppings everywhere. It was terrible.

Heap:

Don't see you too often this time of day. Got a little something going on?

What're you doing?

You should take life a little easier. Relax a little...

Beer? Don't worry about the cap; throw it anywhere.

Hey, you found the pizza! That's great. I've been looking for that all week.

Don't worry about it. That online ordering application left it lying around. It'll probably be back for it.

He'd have been back in touch. He'd have called `free()`.

Hey, it's not my responsibility to clear up the memory. Someone asks me for space, I give him space. I'll leave it there until he tells me to clean it up.

That's irresponsible.

Yeah, maybe. But I'm easy to use. Not like you and your...fussing.

Fussing? I don't fuss! You might want to use a napkin...

<belches>What? I'm just saying you're difficult to keep track of.

I just believe that memory should be properly maintained.

Whatever. I'm a live-and-let-live type. If a program wants to make a mess, it's not my responsibility.

You're messy.

I'm easygoing.

Why don't you do garbage collection?!

Ah, here we go again...

I mean, just a little...tidying up. You don't do anything!!!

Easy, now.

Q: Why is the heap called the heap? A: Because the computer doesn't automatically organize it. It's just a big heap of data.

Q: What's garbage collection? A: Some languages track when you allocate data on a heap and then, when you're no longer using the data, they free the data from the heap.

Q: Why doesn't C contain garbage collection? A: C is quite an old language; when it was invented, most languages didn't do automatic garbage collection.

Q: I understand why I needed to copy the name of the island in the example. Why didn't I need to copy the opens and closes values? A: The opens and closes values are set to string literals. String literals can't be updated, so it doesn't matter if several data items refer to the same string.

Q: Does strdup() actually call the malloc() function? A: It will depend on how the C Standard Library is implemented, but most of the time, yes.

Q: Do I need to free all my data before the program ends? A: You don't have to; the operating system will clear away all of the memory when the program exits. But it's good practice to always explicitly free anything you've created.

- Dynamic data structures allow you to store a variable number of data items.
- A linked list is a data structure that allows you to easily insert items.
- Dynamic data structures are normally defined in C with recursive structs.
- A **recursive struct** contains one or more pointers to a **similar struct**.
- The stack is used for local variables and is managed by the computer.
- The heap is used for long-term storage. You allocate space with malloc().
- The sizeof operator will tell you how much space a struct needs.
- Data will stay on the heap until you release it with free().

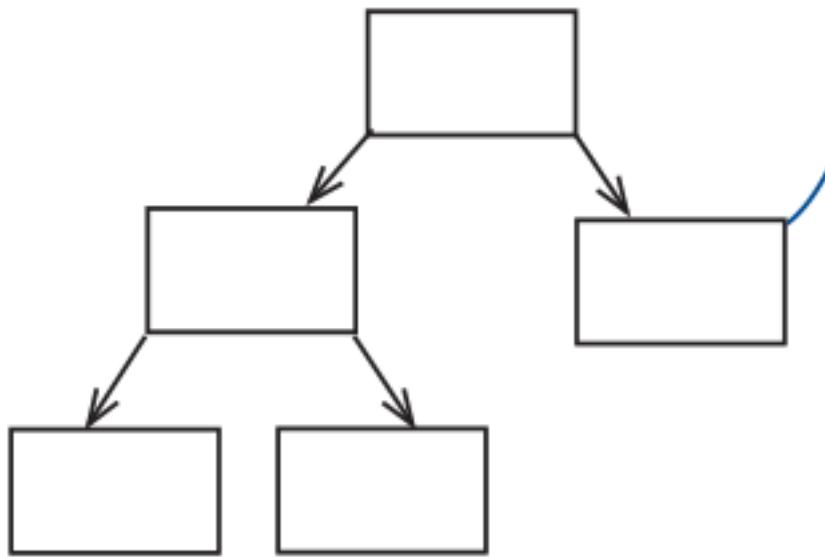
WHAT'S MY DATA STRUCTURES

I can be used to store a sequence of items, and I make it easy to insert new items. But you can process me in only one direction.



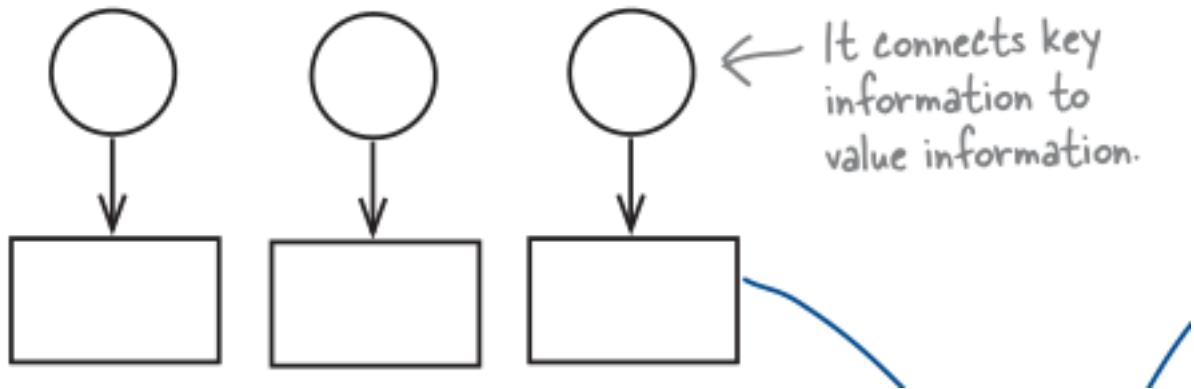
Each item I store can connect to up to two other items. I am useful for storing hierarchical information.

Binary tree



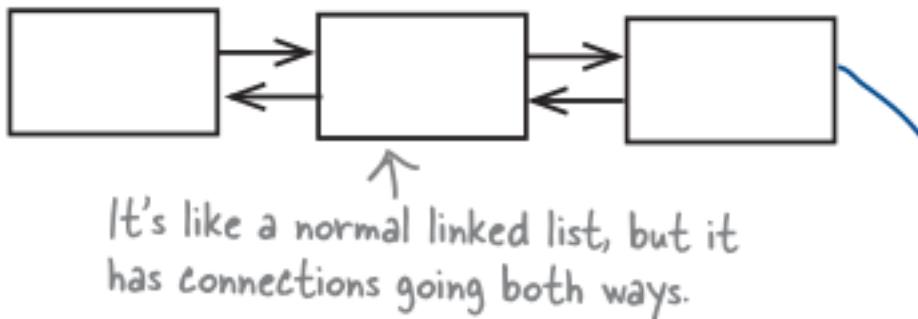
I can be used to associate two different types of data. For example, you could use to me to associate people's names to their phone numbers.

Associated array or map



Each item I store connects to up to two other items. You can process me in two directions.

Doubly linked list



Data structures are useful, but be careful! You need to be careful when you create these data structures using C.

If you don't keep proper track of the data you are storing, there's a risk that you'll leave old dead data on the heap.

Over time, this will start to eat away at the memory on your machine, and it might cause your program to crash with memory errors.

That means it's really important that you learn to track down and fix memory leaks in your code.

FBI LEAK SOURCE CODE

Exhibit A: the source code What follows is the source code for the Suspicious Persons Identification Expert System (SPIES).

This software can be used to record and identify persons of interest.

You are not required to read this code in detail now, but please keep a copy in your records so that you may refer to it during the ongoing investigation.

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

typedef struct node {
    char *question;
    struct node *no;
    struct node *yes;
} node;

int yes_no(char *question)
{
    char answer[3];
    printf(format: "%s? (y/n): ", question);
    fgets(Buf: answer, MaxCount: 3, File: stdin);
    return answer[0] == 'y';
}
```

```
node *create(char *question)
{
    node *n = malloc(Size: sizeof(node));
    n -> question = strdup(Src: question);
    n -> no = NULL;
    n -> yes = NULL;
    return n;
}
```

```
void release(node *n)
{
    if(n){
        if(n -> no)
            release( n: n -> no);
        if(n -> yes)
            release( n: n -> yes);
        if(n -> question)
            free( Memory: n -> question);
        free( Memory: n);
    }
}
```

```
int main()
{
    char question[80];
    char suspect[20];
    node *start_node = create( question: "Does suspect have a mustache");
    start_node -> no = create( question: "Loretta Barnsworth");
    start_node -> yes = create( question: "Vinny the Spoon");

    node *current;
    do{
        current = start_node;
        while(1)
        {
            if(yes_no( question: current -> question))
            {
                current = current -> yes;
            }
            else{
                printf( format: "SUSPECT IDENTIFIED\n");
                break;
            }
        }
    }
```

```

        else if(current -> no)
    {
        current = current -> no;
    }
    else
    {
        //make the yes-node the new suspect name
        printf("Who's the suspect?");
        fgets(suspect, 20, stdin);
        node *yes_node = create(suspect);
        current -> yes = yes_node;

        //then replace this question with the new question
        printf("Give me a question that is TRUE for %s but not for %s?", suspect, current -> question);
        fgets(question, 80, stdin);
        current -> question = strdup(question);
        break;
    }
}
} while(yes_no( question: "Run again"));

release( n: start_node);
return 0;
}

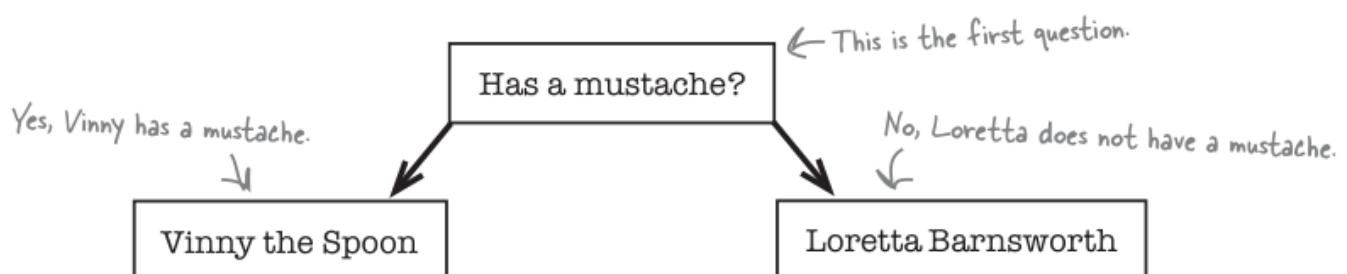
```

An overview of the SPIES system The SPIES program is an expert system that learns how to identify individuals using distinguishing features.

The more people you enter into the system, the more the software learns and the smarter it gets.

The program builds a tree of suspects The program records data using a **binary tree**.

A binary tree allows each piece of data to connect to two other pieces of data like this:



This is what the data looks like when the program starts.

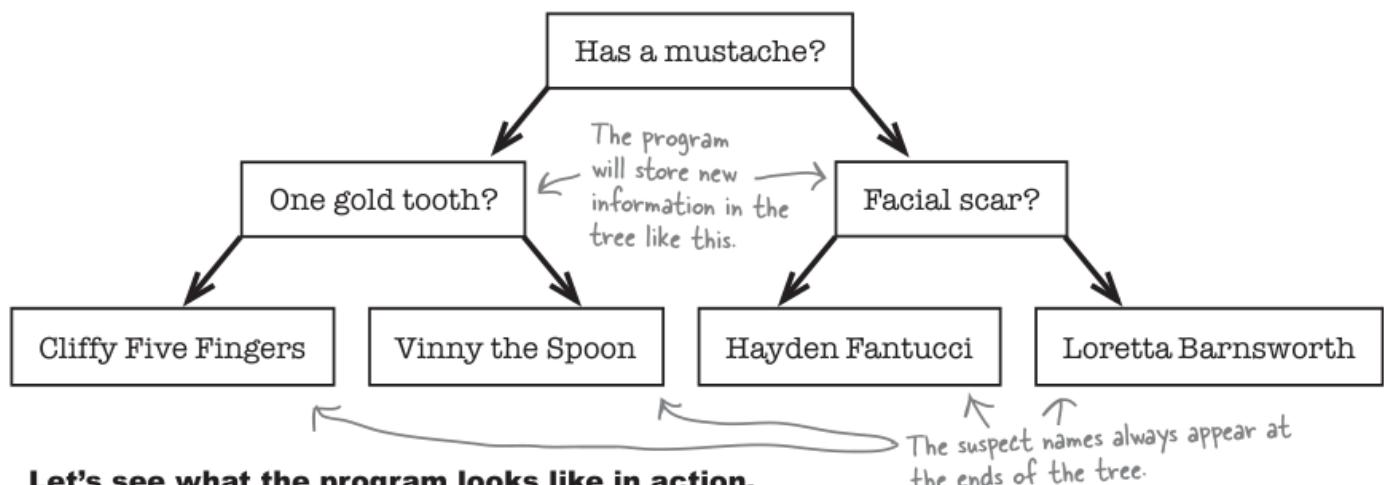
The first item (or node) in the tree stores a question: "Does the suspect have a mustache?"

That's linked to two other nodes: one if the answer's yes, and another if the answer's no. The yes and no nodes store the name of a suspect.

The program will use this tree to ask the user a series of questions to identify a suspect.

If the program can't find the suspect, it will ask the user for the name of the new suspect and some detail that can be used to identify him or her.

It will store this information in the tree, which will gradually grow as it learns more things.



```
> gcc spies.c -o spies && ./spies
Does suspect have a mustache? (y/n): n
Loretta Barnsworth? (y/n): n
Who's the suspect? Hayden Fantucci
Give me a question that is TRUE for Hayden Fantucci
but not for Loretta Barnsworth? Has a facial scar
Run again? (y/n): y
Does suspect have a mustache? (y/n): n
Has a facial scar
? (y/n): y
Hayden Fantucci
? (y/n): y
SUSPECT IDENTIFIED
Run again? (y/n): n
>
```

The first time through, the program fails to identify the suspect Hayden Fantucci. But once the suspect's details are entered, the program learns enough to identify Mr. Fantucci on the second run. Pretty smart.

So what's the problem? Someone was using the system for a few hours in the lab and noticed that even though the program appeared to be working correctly, it was using almost twice the amount of memory it needed.

That's why you have been called in. Somewhere deep in the source code, something is allocating memory on the heap and never freeing it.

Now, you could just sit and read through all of the code and hope that you see what's causing the problem. But memory leaks can be awfully difficult to track down.

VALGRIND

Software forensics: using valgrind Prepare your code: add debug info You don't need to do anything to your code before you run it through valgrind. You don't even need to recompile it.

But to really get the most out of valgrind, you need to make sure your executable contains **debug information**.

Debug information is extra data that gets packed into your executable when it's compiled—things like the line number in the source file that a particular piece of code was compiled from.

If the debug info is present, valgrind will be able to give you a lot more details about the source of your memory leak.

To add debug info into your executable, you need to recompile the source with the `-g` switch:

```
gcc -g spies.c -o spies
```

It can take an achingly long time to track down bugs in large, complex programs like SPIES.

So C hackers have written tools that can help you on your way. One tool used on the Linux operating system is called valgrind. valgrind can monitor the pieces of data that are allocated space on the heap.

It works by creating its own fake version of `malloc()`.

When your program wants to allocate some heap memory, valgrind will intercept your calls to `malloc()` and `free()` and run its own versions of those functions.

The valgrind version of `malloc()` will take note of which piece of code is calling it and which piece of memory it allocated.

When your program ends, valgrind will report back on any data that was left on the heap and tell you where in your code the data was created.

Interrogate your code To see how valgrind works, let's fire it up on a Linux box and use it to interrogate the SPIES program a couple times.

The first time, use the program to identify one of the built-in suspects: Vinny the Spoon.

You'll start valgrind on the command line with the `--leak-check=full` option and then

pass it the program you want to run:

```
File Edit Window Help valgrindRules
> valgrind --leak-check=full ./spies
==1754== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
Does suspect have a mustache? (y/n): y
Vinny the Spoon? (y/n): y
SUSPECT IDENTIFIED
Run again? (y/n): n
==1754== All heap blocks were freed -- no leaks are possible
```

Use valgrind repeatedly to gather more evidence When the SPIES program exited, there was nothing left on the heap.

But what if you run it a second time and teach the program about a new suspect called Hayden Fantucci?

```
File Edit Window Help valgrindRules
> valgrind --leak-check=full ./spies
==2750== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
Does suspect have a mustache? (y/n): n
Loretta Barnsworth? (y/n): n
Who's the suspect? Hayden Fantucci
Give me a question that is TRUE for Hayden Fantucci
  but not for Loretta Barnsworth? Has a facial scar
Run again? (y/n): n
==2750== HEAP SUMMARY:
==2750==     in use at exit: 19 bytes in 1 blocks
==2750==     total heap usage: 11 allocs, 10 frees, 154 bytes allocated
==2750== 19 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2750==    at 0x4026864: malloc (vg_replace_malloc.c:236)
==2750==    by 0x40B3A9F: strdup ( strdup.c:43 )
==2750==    by 0x8048587: create (spies.c:22)
==2750==    by 0x804863D: main (spies.c:46)
==2750== LEAK SUMMARY:
==2750==    definitely lost: 19 bytes in 1 blocks
>
```

↑
Why 19 bytes? Is that a clue?

19 bytes was left on the heap. Up allocated new pieces of memory 11 times, but only freed 10 of them.

Do these lines give us any clues?

This time, valgrind found a memory leak It looks like there were 19 bytes of information left on the heap at the end of the program. valgrind is telling you the following things:

19 bytes of memory were allocated but not freed.

That's quite a few pieces of information.

Let's take these facts and analyze them.

Looks like we allocated new pieces of memory 11 times, but freed only 10 of them.

Do these lines give us any clues? Why 19 bytes? Is that a clue?

Look at the evidence.

OK, now that you've run valgrind, you've collected quite a few pieces of evidence. It's time to analyze that evidence and see if you can draw any conclusions.

1. Location:

You ran the code two times. The first time, there was no problem. The memory leak only happened when you entered a new suspect name.

Why is that significant? Because that means the leak can't be in the code that ran the first time.

Looking back at the source code, that means the problem lies in this section of the code:

```

else if(current -> no)
{
    current = current -> no;
}
else
{
    //make the yes-node the new suspect name
    printf("Who's the suspect?");
    fgets(suspect, 20, stdin);
    node *yes_node = create(suspect);
    current -> yes = yes_node;

    //then replace this question with the new question
    printf("Give me a question that is TRUE for %s but not for %s?", suspect, current -> question);
    fgets(question, 80, stdin);
    current -> question = strdup(question);
    break;
}

```

2. Clues from valgrind:

When you ran the code through valgrind and added a single suspect, the program allocated memory 11 times, but only released memory 10 times. What does that tell you?

valgrind told you that there were 19 bytes of data left on the heap when the program ended.

If you look at the source code, what piece of data is likely to take up 19 bytes of space? Finally, what does this output from valgrind tell you?

- How many pieces of data were left on the heap?

There is one piece of data.

- What was the piece of data left on the heap?

The string "Loretta Barnsworth". It's 18 characters with a string terminator.

- Which line or lines of code caused the leak?

The `create()` functions themselves don't cause leaks because they didn't on the first pass, so it must be this `_strdup()` line:

`current->question = strdup(question);`

- How do you plug the leak?

If `current->question` is already pointing to something on the heap, free that before allocating a new question:

`free(current->question);`

`current->question = strdup(question);`

The fix on trial

Now that you've added the fix to the code, it's time to run the code through valgrind again.

```
File Edit Window Help valgrindRules
> valgrind --leak-check=full ./spies
==1800== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
Does suspect have a mustache? (y/n): n
Loretta Barnsworth? (y/n): n
Who's the suspect? Hayden Fantucci
Give me a question that is TRUE for Hayden Fantucci
  but not for Loretta Barnsworth? Has a facial scar
Run again? (y/n): n
==1800== All heap blocks were freed -- no leaks are possible
>
```

The leak is fixed. You ran exactly the same test data through the program, and this time the program cleared everything away from the heap. How did you do? Did you crack the case?

Don't worry if you didn't manage to find and fix the leak this time. Memory leaks are some of the hardest bugs to find in C programs.

The truth is that many of the C programs available probably have some memory bugs buried deep inside them, but that's why tools like valgrind are important.

Spot when leaks happen. Identify the location where they happen. Check to make sure the leak is fixed.

Q: valgrind said the leaked memory was created on line 46, but the leak was fixed on a completely different line. How come?

A: The "Loretta..." data was put onto the heap on line 46, but the leak happened when the variable pointing to it (`current->question`) was reassigned without freeing it. Leaks don't happen when data is created; they happen when the program loses all references to the data.

Q: Can I get valgrind on my Mac/ Windows/FreeBSD system? A: Check <http://valgrind.org> for details on the latest release.

Q: How does valgrind intercept calls to malloc() and free()? A: The malloc() and free() functions are contained in the C Standard Library.

But valgrind contains a library with its own versions of malloc() and free(). When you run a program with valgrind, your program will be using valgrind's functions, rather than the ones in the C Standard Library.

Q: Why doesn't the compiler always include debug information when it compiles code? A: Because debug information will make your executable larger, and it may also make your program slightly slower.

Q: Where did the name valgrind come from? A: Valgrind is the name of the entrance to Valhalla. valgrind (the program) gives you access to the computer's heap.

- Valgrind checks for memory leaks.
- Valgrind works by intercepting the calls to malloc() and free().
- When a program stops running, valgrind prints details of what's left on the heap.
- If you compile your code with debug information, valgrind can give you more information.
- If you run your program several times, you can narrow the search for the leak.
- Valgrind can tell you which lines of code in your source put the data on the heap.
- Valgrind can be used to check that you've fixed a leak.

~~book, see Appendix II.~~

A linked list is more extensible than an array.

Data can be inserted easily into a linked list.

A linked list is a dynamic data structure.

Dynamic data structures use recursive structs.

Recursive structs contain one or more links to similar data.

- A linked list is more extensible than an array.
- Data can be inserted easily into a linked list.

- A linked list is a dynamic data structure.
- Dynamic data structures use recursive structs.
- Recursive structs contain one or more links to similar data.

`malloc()`
allocates
memory on the
heap.

`free()`
releases
memory on
the heap.

Unlike the
stack, heap
memory is not
automatically
released.

`strdup()` will
create a copy
of a string on
the heap.

A memory leak
is allocated
memory you can
no longer access.

The stack
is used
for local
variables.

valgrind can
help you
track down
memory leaks.

- `malloc()` allocates memory on the heap.
- `free()` releases memory on the heap.
- Unlike the stack, heap memory is not automatically released.
- The stack is used for local variables.
- `strdup()` will create a copy of a string on the heap.
- A memory leak is allocated memory you can no longer access.
- valgrind can help you track down memory leaks.

Advanced Functions

Basic functions are great, but sometimes you need more.

So far, you've focused on the basics, but what if you need even more power and flexibility to achieve what you want?

In this chapter, you'll see how to up your code's IQ by passing functions as parameters. You'll find out how to get things sorted with **comparator functions**.

7 advanced functions

And finally, you'll discover how to make your code super stretchy with **variadic functions**.

* *Turn your functions up to 11*



Looking for Mr. Right.

You've used a lot of C functions in the book so far, but the truth is that there are still some ways to make your C functions a lot more powerful.

If you know how to use them correctly, C functions can make your code do more things but without writing a lot more code. To see how this works, let's look at an example.

Imagine you have an array of strings that you want to filter down, displaying some strings and not displaying others.

```
int main() {
    int NUM_AMS = 7;
    char *ADS[] = {
        "William: SBM GSOH likes sports, TV, dining",
        "Matt: SWM NS likes art, movies, theater",
        "Luis: SLM ND likes books, theater, art",
        "Mike: DWM DS likes trucks, sports and bieber",
        "Peter: SAM likes chess, working out and art",
        "Josh: SJM likes sports, movies and theater",
        "Jed: DBM likes theater, books and dining"
    };
    return 0;
}
```

I want someone into sports, but definitely not into Bieber...



Let's write some code that uses string functions to filter this array down.

Complete the find() function so it can track down all the sports fans in the list who don't also share a passion for Bieber.

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

//global variables
int NUM_AMS = 7;
char *ADS[] = {
    [0]: "William: SBM GOSH likes sports, TV, dining",
    [1]: "Matt: SWM NS likes art, movies, theater",
    [2]: "Luis: SLM ND likes books, theater, art",
    [3]: "Mike: DWM DS likes trucks, sports and bieber",
    [4]: "Peter: SAM likes chess, working out and art",
    [5]: "Josh: SJM likes sports, movies and theater",
    [6]: "Jed: DBM likes theater, books and dining"
};
```

```

void find()
{
    int i;
    puts( Str: "Search results: ");
    puts( Str: "-----");

    for(i = 0; i < NUM_AMS; i++)
    {
        if(strstr( Str: ADS[i], SubStr: "sports") && ! strstr( Str: ADS[i], SubStr: "bieber" ) )
        {
            printf( format: "%s\n", ADS[i]);
        }
    }
    puts( Str: "-----");
}

int main()
{
    find();
    return 0;
}

```

While *i* is less than *NUM_AMS*, increment it.

Look for "sports" in the lines inside array of arrays *ADS[]*, they shouldn't contain "b-eiber".

Print the line.

Call the function from *main*.

```

Search results:
-----
William: SBM GOSH likes sports, TV, dining
Mike: DWM DS likes trucks, sports and bieber
Josh: SJM likes sports, movies and theater
-----
```

And sure enough, the *find()* function loops through the array and finds the matching

strings.

Now that you have the basic code, it would be easy to create clones of the function that could perform different kinds of **searches**.

And sure enough, the `find()` function loops through the array and finds the matching strings.

Now that you have the basic code, it would be easy to create clones of the function that could perform different kinds of searches.

Find someone
who likes
sports or
working out.

I want a non-
smoker who
likes the
theater.

Find someone
who likes the
art, theater, or
dining.

Hey, wait! Clone? Clone the function???? That's dumb. Each version would only vary by, like, one line.

Hey, wait! Clone? **Clone the function???** That's dumb. Each version would only vary by, like, one line.

o o



~~right. If you clone the function, you'll get a lot of duplicated code.~~

~~so it's better to pass in a function that can do whatever test you want.~~

Exactly right. If you clone the function, you'll have a lot of duplicated code. C programs often have to perform tasks that are almost identical except for some small detail.

At the moment, the `find()` function runs through each element of the array and applies a simple test to each string to look for matches.

But the test it makes is hardwired. It will always perform the same test. Now, you could pass some strings into the function so that it could search for different substrings.

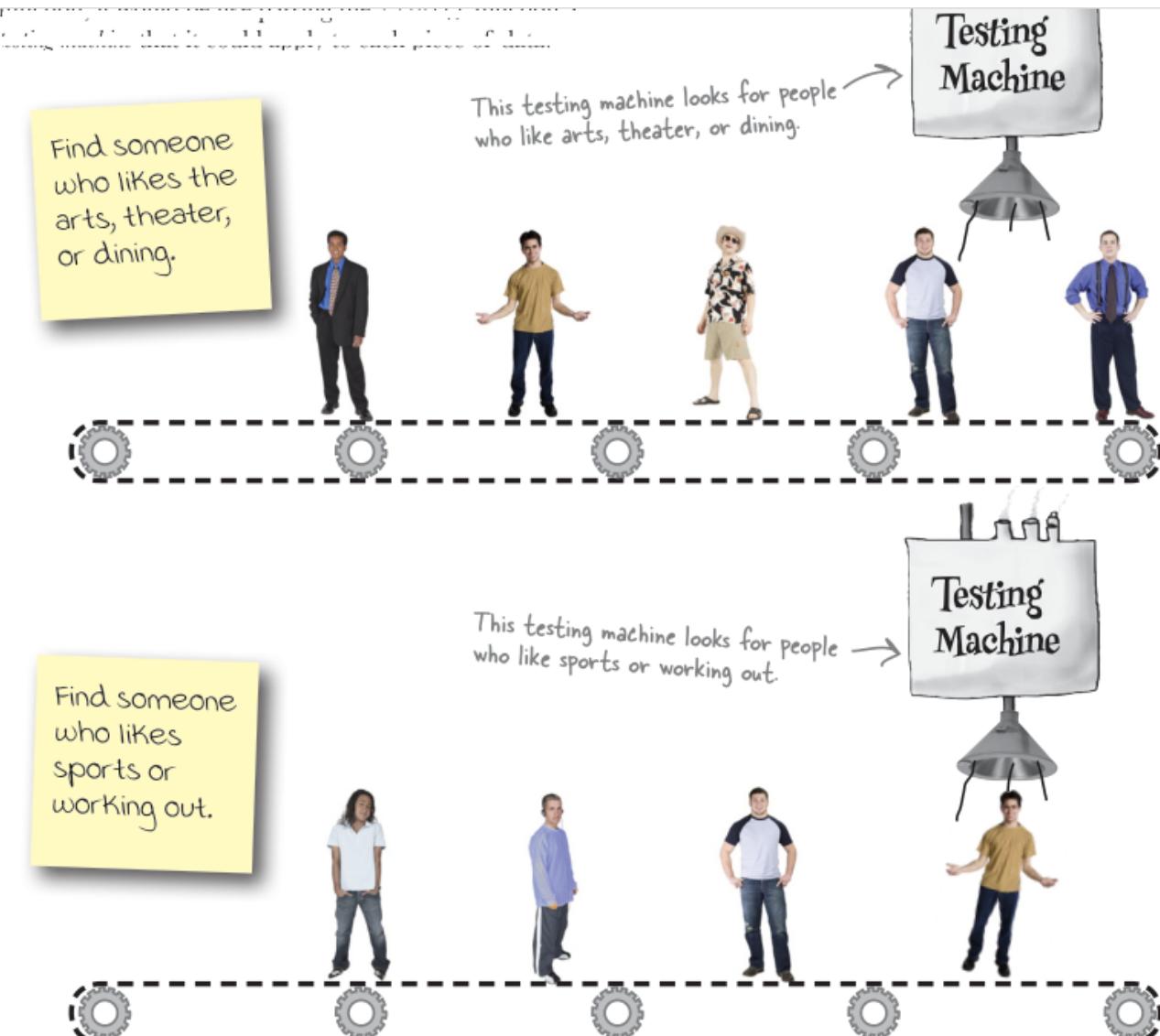
The trouble is, that wouldn't allow `find()` to check for three strings, like "arts," "theater," or "dining."

And what if you needed something wildly different? You need something a little more sophisticated.

Pass code to a function

What you need is some way of passing the code for the test to the `find()` function.

If you had some way of wrapping up a piece of code and handing that code to the function, it would be like passing the `find()` function a testing machine that it could apply to each piece of data.



This means the bulk of the `find()` function would stay exactly the same.

It would still contain the code to check each element in an array and display the same kind of output.

But the test it applies against each element in the array would be done by the code that you pass to it.

You need to tell `find()` the name of a function.

Imagine you take our original search condition and rewrite it as a function:

```
int sport_no_bieber(char *s)
{
    return strstr( Str: s, SubStr: "sports") && !strstr( Str: s, SubStr: "ieber");
}
```

Now, if you had some way of passing the name of the function to `find()` as a parameter, you'd have a way of injecting the test:

```
void find( function-name match )
{
    int i;
    puts("Search results:");
    puts("-----");
    for (i = 0; i < NUM_ADS; i++) {
        if ( call-the-match-function (ADS[i])) {
            printf("%s\n", ADS[i]);
        }
    }
    puts("-----");
}
```

match would specify the name of the function containing the test.

Here, you'd need some way of calling the function whose name was given by the match parameter.

If you could find a way of passing a function name to `find()`, there would be no limit to the kinds of tests that you could make in the future.

As long as you can write a function that will return true or false to a string, you can reuse the same `find()` function.

```
find(sports_no_bieber);  
find(sports_or_workout);  
find(ns_theater);  
find(arts_theater_or_dining);
```

But how do you say that a parameter stores the name of a function?

And if you have a function name, how do you use it to call the function?

Every function name is a pointer to the function.

You probably guessed that pointers would come into this somewhere, right?

Think about what the name of a function really is. It's a way of referring to the piece of code.

And that's just what a pointer is: a way of referring to something in memory.

That's why, in C, **function names are also pointer variables**.

When you create a function called `go_to_warp_speed(int speed)`, you are also creating a pointer variable called `go_to_warp_speed` that contains the **address of the function**.

So, if you give `find()` a parameter that has a function pointer type, you should be able to use the parameter to call the function it points to.

```

int go_to_warp_speed(int speed)
{
    dilithium_crystals(ENGAGE);
    warp = speed;
    reactor_core(c, 125000 * speed, PI);
    clutch(ENGAGE);
    brake(DISENGAGE);
    return 0;
}

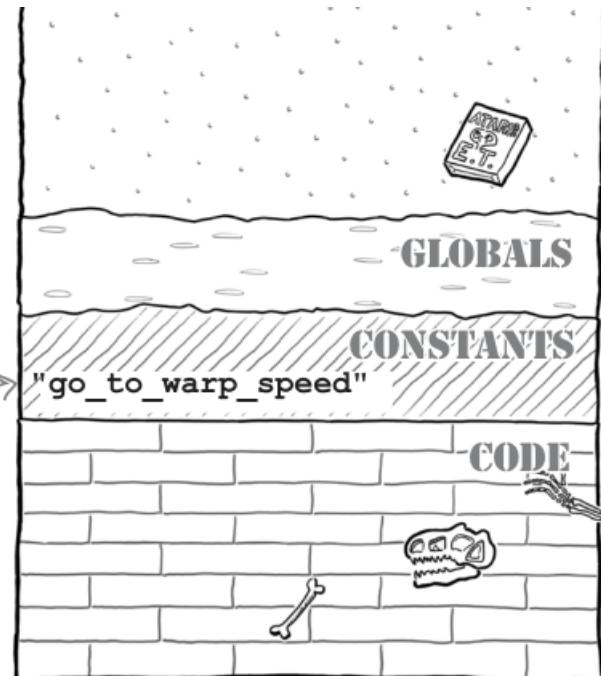
```

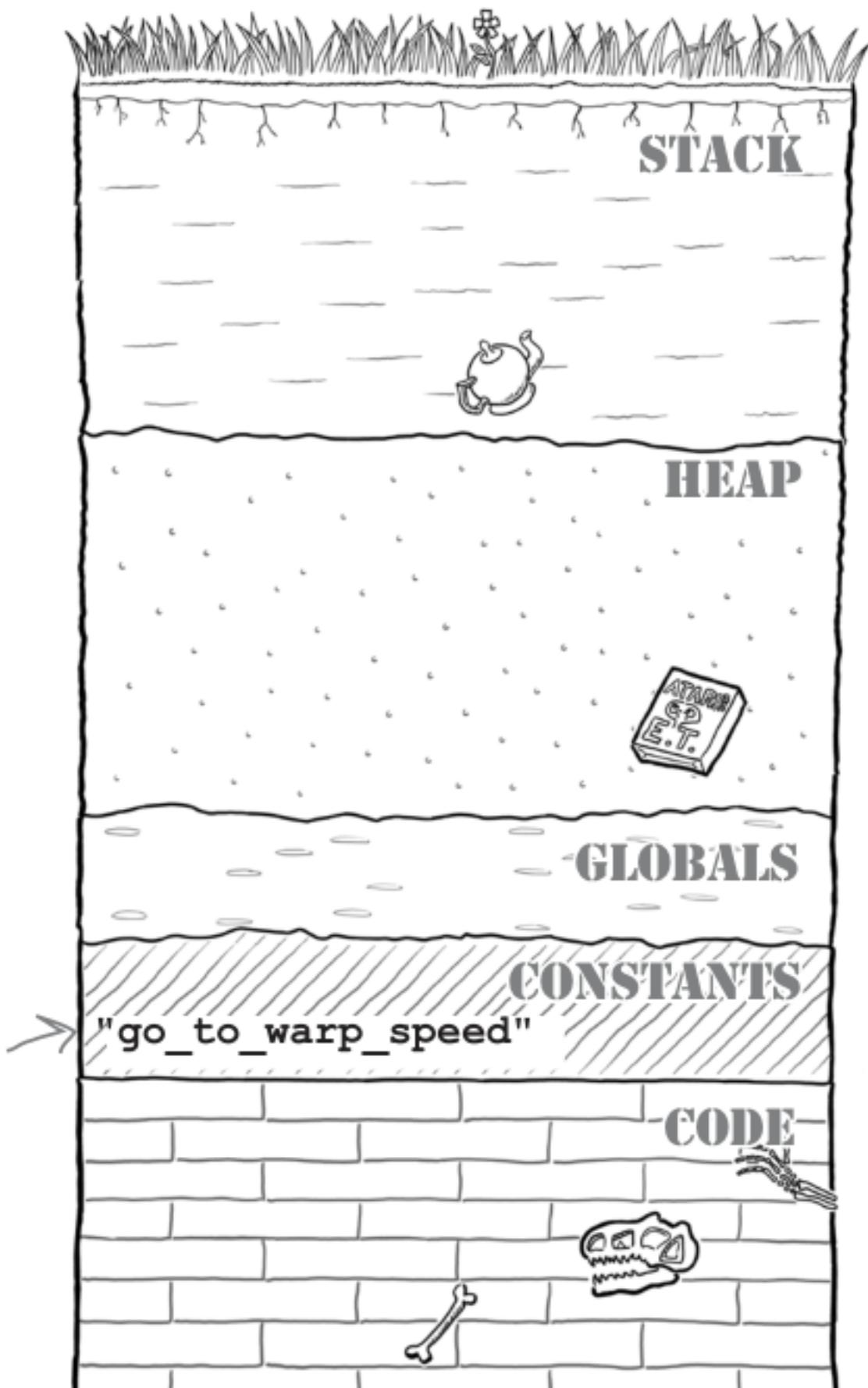
Whenever you create a function, you also create a function pointer with the same name.

The pointer contains the address of the function.

```
go_to_warp_speed(4);
```

When you call the function, you are using the function pointer.





but there's no function data type. Usually, it's pretty easy to declare pointers in C.

If you have a data type like `int`, you just need to add an asterisk to the end of the data type name, and you declare a pointer with `int *`.

Unfortunately, C doesn't have a function data type, so you can't declare a function pointer with anything like `function *`.

`int *a;` ← This declares an int pointer...

`function *f;` ← ...but this won't declare a function pointer.

Why doesn't C have a function data type? C doesn't have a function data type because there's not just one type of function.

When you create a function, you can vary a lot of things, such as the return type or the list of parameters it takes.

That combination of things is what defines the type of the function.

```
int go_to_warp_speed(int speed)
{
    ...
}

char** album_names(char *artist, int year)
{
    ...
}
```

There are many different types of functions. These functions are different types because they have different return types and parameters.

So, for function pointers, you'll need to use slightly more complex notation.

CREATING FUNCTION POINTERS

How to create function pointers Say you want to create a pointer variable that can store the address of each of the functions on the previous page.

You'd have to do it like this:

```
int (*warp_fn)(int);  
warp_fn = go_to_warp_speed;  
warp_fn(4);
```

This will create a variable called warp_fn that can store the address of the go_to_warp_speed() function.

This is just like calling go_to_warp_speed(4).

```
char** (*names_fn)(char*, int);  
names_fn = album_names;  
char** results = names_fn("Sacha Distel", 1972);
```

This will create a variable called names_fn that can store the address of the album_names() function.

That looks pretty complex, doesn't it?

Unfortunately, it has to be, because you need to tell C the return type and the parameter types the function will take.

But once you've declared a function pointer variable, you can use it like any other variable.

You can assign values to it, you can add it to arrays, and you can also pass it to functions, which brings us back to your find() code.

What does char mean? Is it a typing error? A:** char** is a pointer normally used to point to an array of string.

Writing the code:

```

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

//global variables
int NUM_AMS = 7;
char *ADS[] = {
    [0]: "William: SBM GOSH likes sports, TV, dining",
    [1]: "Matt: SWM NS likes art, movies, theater",
    [2]: "Luis: SLM ND likes books, theater, art",
    [3]: "Mike: DWM DS likes trucks, sports and bieber",
    [4]: "Peter: SAM likes chess, working out and art",
    [5]: "Josh: SJM likes sports, movies and theater",
    [6]: "Jed: DBM likes theater, books and dining"
};

```

```

int sport_no_bieber(char *s) //pass it the strings from the array of arrays
{
    return strstr( Str: s, SubStr: "sports") && !strstr( Str: s, SubStr: "bieber");
}

int sports_or_workout(char *s)
{
    return strstr( Str: s, SubStr: "sports") || strstr( Str: s, SubStr: "workout");
}

int ns_theater(char *s)
{
    return strstr( Str: s, SubStr: "theater") && strstr( Str: s, SubStr: "non-smoker");
}

```

```

int arts_theater_or_dining(char *s)
{
    return strstr( Str: s, SubStr: "arts") || strstr( Str: s, SubStr: "theater") || strstr( Str: s, SubStr: "dining");
}

```

```
//new find function
void find(int(*match)(char*))
{
    int i;
    puts( Str: "Search results: ");
    puts( Str: "-----");
    for(i = 0; i < NUM_ADS; i++){
        if(match(ADS[i]))
            printf( format: "%s\n", ADS[i]);
    }
    puts( Str: "-----");
}
```

Let's take those functions out on the road and see how they perform.

You'll need to create a program to call `find()` with each function in turn:

```
int main()
{
    find( match: sport_no_bieber);
    find( match: sports_or_workout);
    find( match: ns_theater);
    find( match: arts_theater_or_dining);
    return 0;
}
```

Results:

```
Search results:
```

```
-----  
William: SBM GOSH likes sports, TV, dining  
Mike: DWM DS likes trucks, sports and bieber  
Josh: SJM likes sports, movies and theater  
-----
```

```
Search results:
```

```
-----  
William: SBM GOSH likes sports, TV, dining  
Mike: DWM DS likes trucks, sports and bieber  
Josh: SJM likes sports, movies and theater  
-----
```

```
Search results:
```

```
-----  
-----  
-----
```

```
Search results:
```

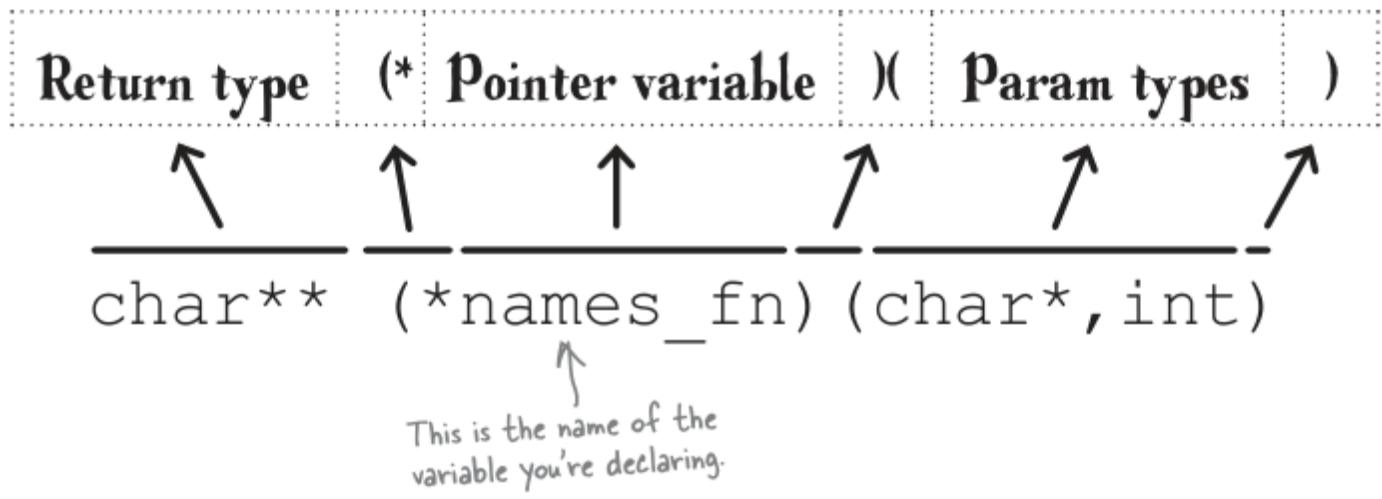
```
-----  
William: SBM GOSH likes sports, TV, dining  
Matt: SWM NS likes art, movies, theater  
Luis: SLM ND likes books, theater, art  
Josh: SJM likes sports, movies and theater  
Jed: DBM likes theater, books and dining  
-----
```

Each call to the `find()` function is performing a very different search. That's why function pointers are one of the most powerful features in C: they allow you to mix functions together.

Function pointers let you build programs with a lot more power and a lot less code.



When you're out in the reeds, identifying those function pointers can be pretty tricky. But this simple, easy-to-carry guide will fit in the ammo pocket of any C user.



Q: If function pointers are just pointers, why don't you need to prefix them with a * when you call the function?

A: You can. In the program, instead of writing `match(ADS[i])`, you could have written `(*match)(ADS[i])`.

Q: And could I have used & to get the address of a method?

A: Yes. Instead of `find(sports_or_workout)`, you could have written `find(&sports_or_workout)`.

Q: Then why didn't I?

A: Because it makes the code easier to read. If you skip the * and &, C will still understand what you're saying.

SORTING

Get it sorted with the C Standard Library: Lots of programs need to sort data.

And if the data's something simple like a set of numbers, then sorting is pretty easy.

Numbers have their own natural order.

But it's not so easy with other types of data.

Imagine you have a set of people. How would you put them in order? By height? By intelligence? By hotness?



When the people who wrote the C Standard Library wanted to create a sort function, they had a problem: How could a sort function sort any type of data at all?

Use function pointers to set the order. You probably guessed the solution: the C Standard Library has a `sort` function that accepts a pointer to a comparator function, which will be used to decide if one piece of data is the same as, less than, or greater than another piece of data.

This is what the `qsort()` function looks like:

This is what the `qsort()` function looks like: This is a pointer

`qsort(void *array,`

to an array.

This is the length → `size_t length,`

`size_t item_size,`

This is the size of each element in the array.

Remember, a `void*` pointer can point to anything.

`int (*compar)(const void *, const void *));`

This is a pointer to a function that compares two items in the array.