

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***

My `go_on_date()`
is awesome now
that I've discovered
variadic functions.



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_ADS = 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_ADS = 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_ADS; i++)
    {
        if(strstr( Str: ADS[i], SubStr: "sports") &&! strstr( Str: ADS[i], SubStr: "beiber" ) )
        {
            printf( format: "%s\n", ADS[i]);
        }
    }
    puts( Str: "-----");
}

int main()
{
    find();
    return 0;
}

```

While i is less than NUM_ADS, 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
-----

```

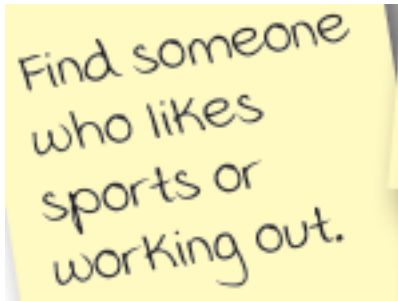
And sure enough, the find() function loops through the array and finds the matching

strings.

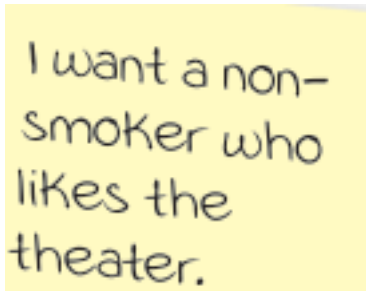
Now that you have the basic code, it would be easy to create clones of the function that could perform different kinds of **searches**.

And sure enough, the `find()` function loops through the array and finds the matching strings.

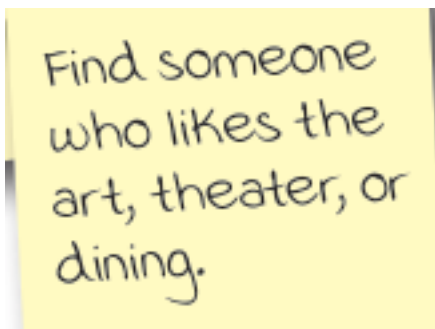
Now that you have the basic code, it would be easy to create clones of the function that could perform different kinds of searches.



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.



$\frac{G}{S} = \frac{1}{S} = \frac{1}{1} = 1$ $\frac{C}{S} = \frac{1}{S} = \frac{1}{1} = 1$ $\frac{I}{S} = \frac{1}{S} = \frac{1}{1} = 1$ $\frac{I'}{S} = \frac{1}{S} = \frac{1}{1} = 1$
 So there are 10 possible combinations that are almost identical
 $\frac{D}{S} = \frac{1}{S} = \frac{1}{1} = 1$ $\frac{A}{S} = \frac{1}{S} = \frac{1}{1} = 1$ $\frac{C}{S} = \frac{1}{S} = \frac{1}{1} = 1$ $\frac{I}{S} = \frac{1}{S} = \frac{1}{1} = 1$

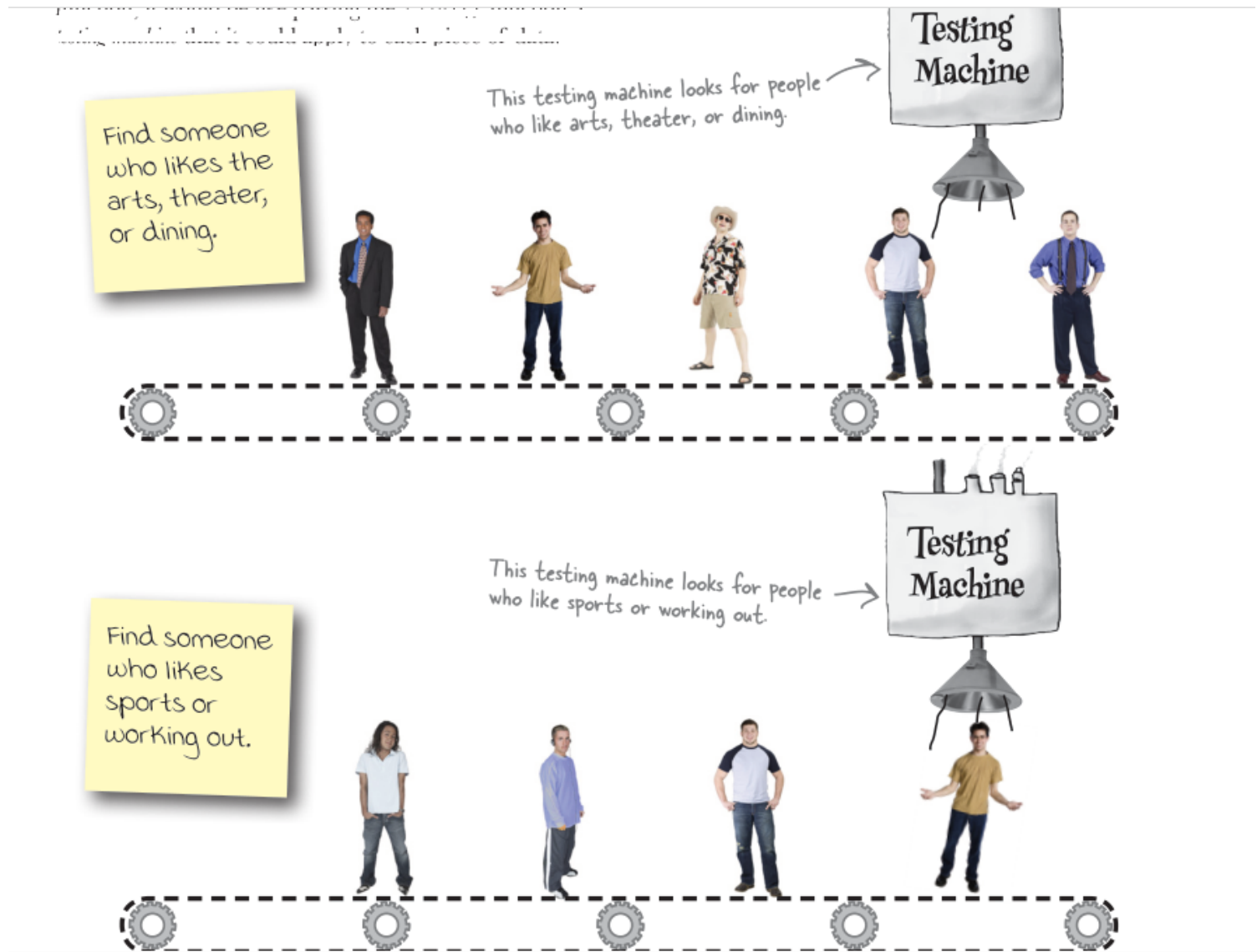
At the moment, the find() function runs through each element of the array and applies a simple test to each string to look for matches.

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.

What you need is some way of passing the code for the test to the find() function.

6/46



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: "beiber");
}
```

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;
}

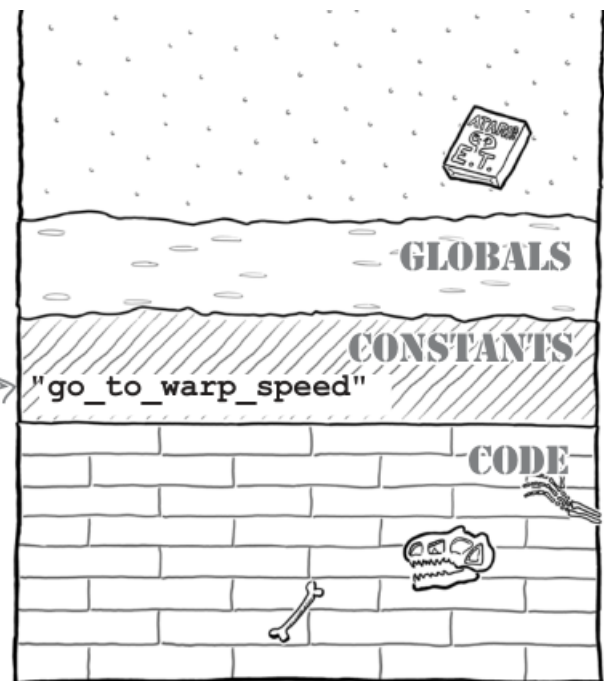
```

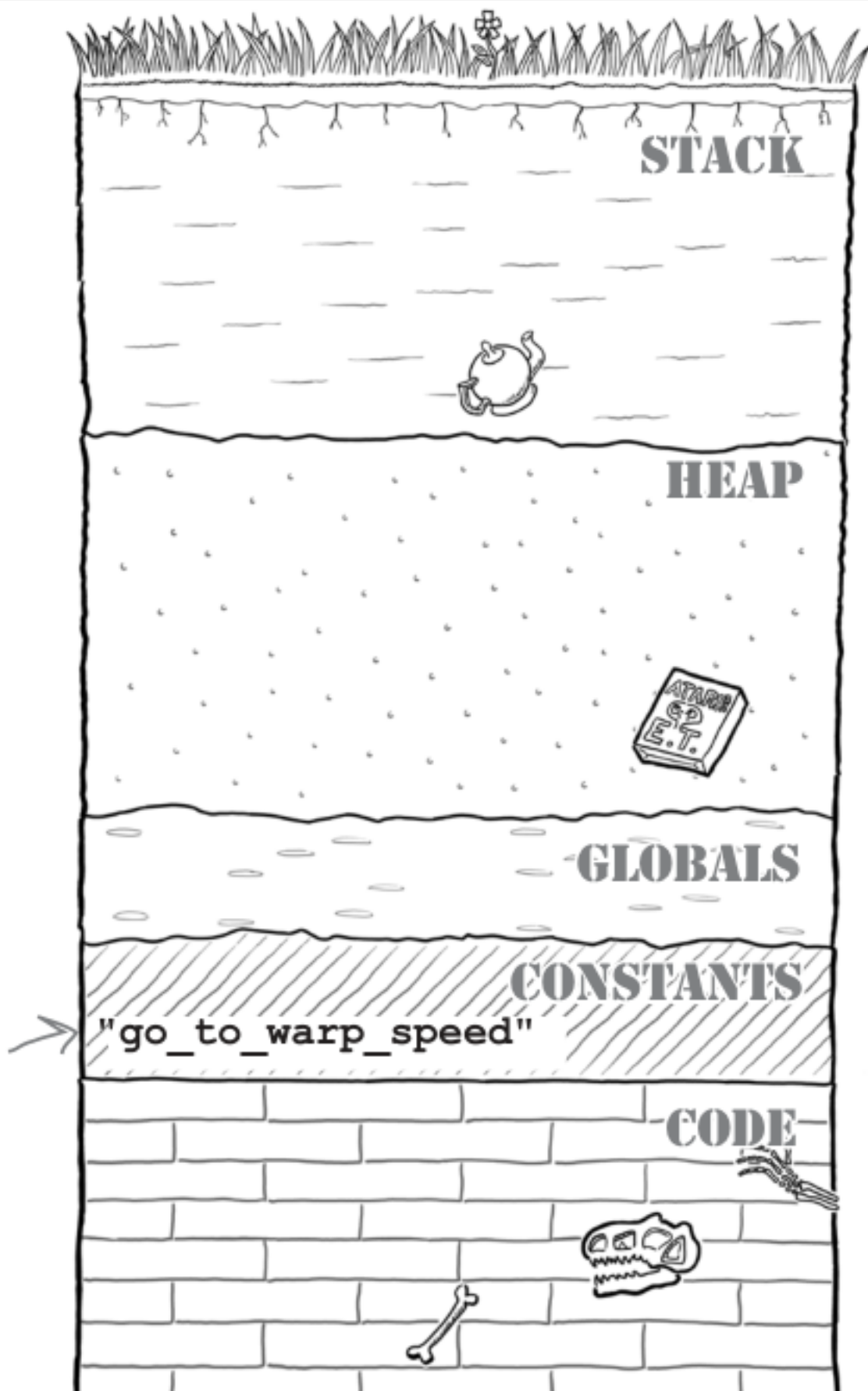
Whenever you create a function,
you also create a function pointer
with the same name.

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_ADS = 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: "beiber");
}

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
```

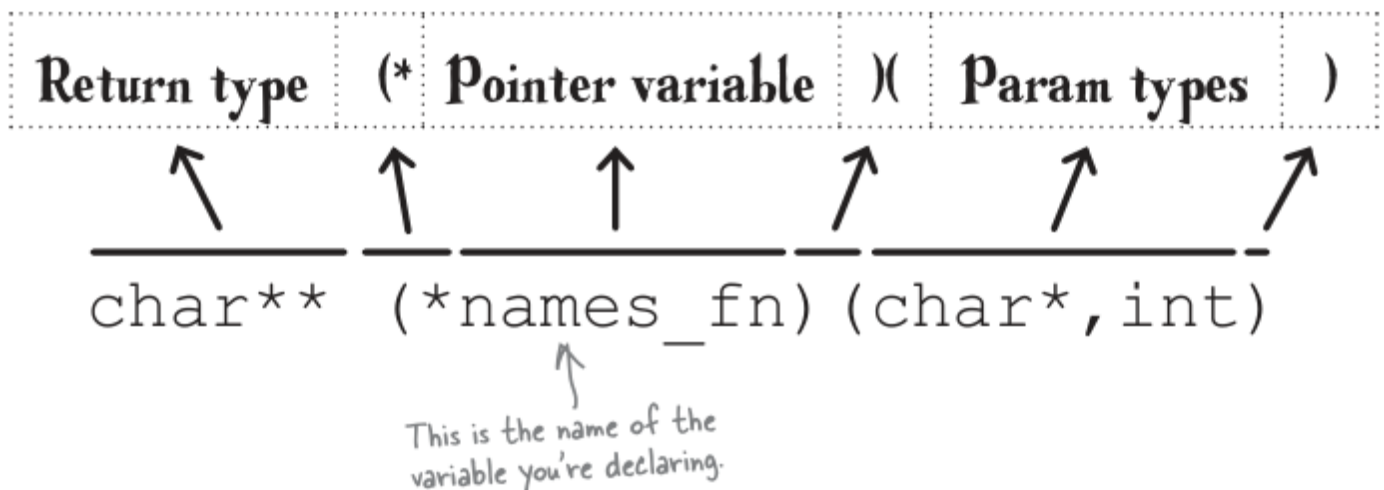
```
-----
```

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:

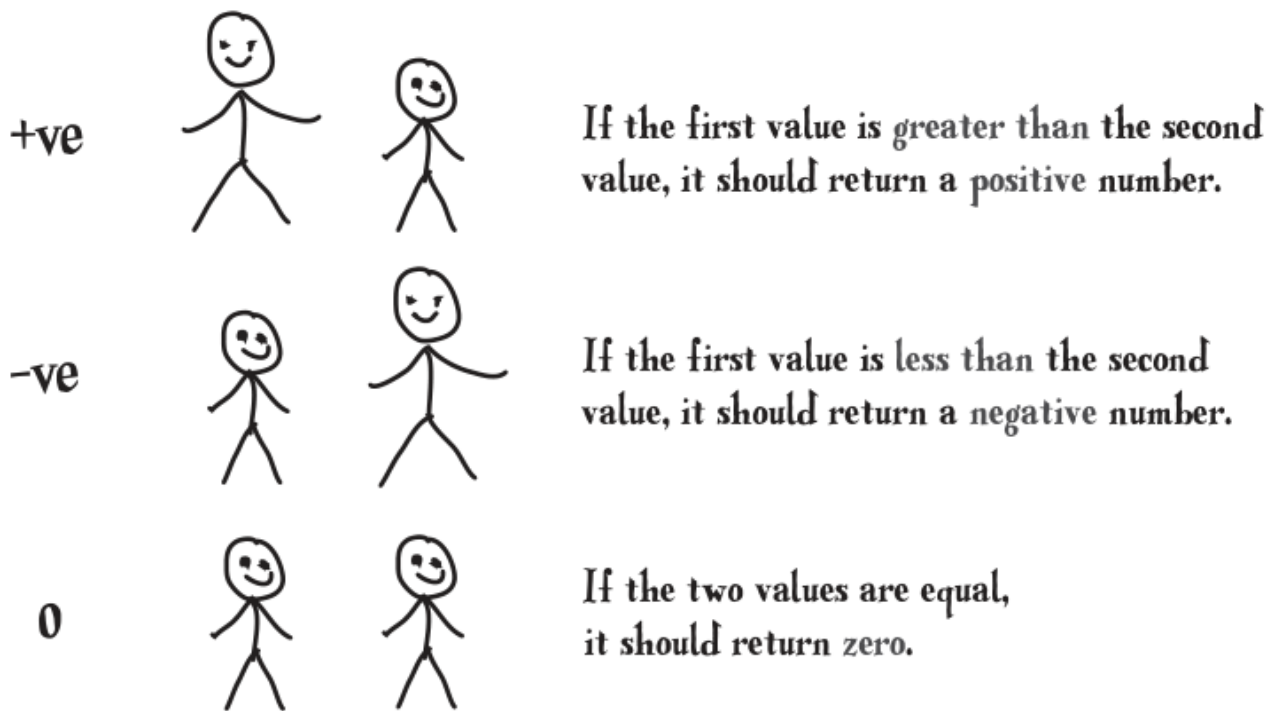
This is what the `qsort()` function looks like:

```
qsort(void *array,
      size_t length,
      size_t item_size,
      int (*compar)(const void *, const void *));
```

Annotations:

- `void *array`: This is a pointer to an array.
- `size_t length`: This is the length of the array.
- `size_t item_size`: This is the size of each element in the array.
- `int (*compar)(const void *, const void *)`: This is a pointer to a function that compares two items in the array. Remember, a `void*` pointer can point to anything.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 qsort(void *array, size_t length, size_t item_size,
5       int (*compar)(const void *, const void *));
```



To see how this works in practice, let's look at an example.

Let's say you have an array of integers and you want to sort them in increasing order. What does the comparator function look like?

```
int scores[] = {543, 323, 32, 554, 11, 3, 112};
```

If you look at the signature of the comparator function that `qsort()` needs, it takes two void pointers given by `void*`.

Remember `void*` when we used `malloc()`?

A void pointer can store the address of any kind of data, but you always need to cast it to something more specific before you can use it.

The `qsort()` function works by comparing pairs of elements in the array and then placing them in the correct order.

It compares the values by calling the comparator function that you give it.

```
int compare_scores(const void* score_a, const void* score_b)
{
    ....
}
```

This code snippet is using the `qsort()` function in C to sort an array of integers called `scores`.

The `qsort()` function is a built-in function in C which can be used to sort arrays. It takes four arguments:

- The base address of the array to be sorted.
- The number of elements in the array.
- The size of each element in the array.
- A pointer to a comparison function that will be used to determine the order of the elements in the sorted array.

In this code, the first argument passed to `qsort()` is `scores`, which is the array to be sorted. The second argument is the number of elements in the array, which can be calculated using the `sizeof()` operator divided by the size of a single element in the array.

The third argument is the size of each element in the array, which can be calculated using the `sizeof()` operator on a single element in the array.

Finally, the fourth argument is a pointer to a comparison function called `compare_scores()`, which will be used by `qsort()` to determine the order of the elements in the sorted array.

The `compare_scores()` function takes two **`const void*` arguments**, which are pointers to the two elements being compared. It then casts these pointers to pointers to integers and returns the difference between the two integers.

The difference between two integers is used to determine their relative order in the sorted array.

If the difference is positive, the first element is considered to be greater than the

second element and will be placed after it in the sorted array.

If the difference is negative, the first element is considered to be less than the second element and will be placed before it in the sorted array.

If the difference is zero, the two elements are considered equal and can be placed in any order relative to each other.

Overall, this code is using the `qsort()` function to sort the scores array in ascending order.

Values are always passed to the function as pointers, so the first thing you need to do is get the integer values from the pointers:

```
int a = *(int*)score_a;  
int b = *(int*)score_b;
```

In the `compare_scores()` function passed as the comparison function to `qsort()`, the lines `int a = *(int*)score_a;` and `int b = *(int*)score_b;` are used to extract the integer values being pointed to by the `score_a` and `score_b` pointers, respectively.

The `qsort()` function is designed to work with arrays of any data type, so the pointers passed to the comparison function `compare_scores()` are of type `const void*`. This means that they are pointers to a memory location of an unspecified data type.

To use these pointers as integer values, we must first cast them to a pointer to an integer type, which is done with `(int*)score_a` and `(int*)score_b`.

Then, we dereference these pointers by using the `*` operator, which means we are accessing the value being pointed to by the pointer. Since we have casted these pointers to `int*`, this means we are accessing the integer values being pointed to by these pointers.

So the lines `int a = *(int*)score_a;` and `int b = *(int*)score_b;` are simply extracting the integer values being pointed to by `score_a` and `score_b`, respectively, so that we

can compare them and determine their relative order in the sorted array.

You need to cast the void pointer to an integer pointer.

```
int a = *(int*)score_a;  
int b = *(int*)score_b;
```

This first * then gets the int stored at address score_b.

Then you need to return a positive, negative, or zero value, depending on whether a is greater than, less than, or equal to b.

For integers, that's pretty easy to do - you just subtract one number from the other:

```
return a - b;
```

← If $a > b$, this is positive. If $a < b$, this is negative. If a and b are equal, this is zero.

The comparator function returned the value -21. That means 11 needs to be before 32.

And this is how you ask `qsort()` to sort the array:

```
qsort(scores, 7, sizeof(int), compare_scores);
```

Now it's your turn. Look at these different sort descriptions. See if you can write a comparator function for each one. To get you started, the first one is already completed.

Sort integer scores, with the smallest first.

Sort integer scores, with the largest first.

Sort the rectangles in area order, smallest first.

Sort a list of names in alphabetical order. Case-sensitive.

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

int compare_scores(const void* score_a, const void* score_b)
{
    int a = *(int*)score_a;
    int b = *(int*)score_b;
    return a - b;
}

int compare_scores_desc(const void* score_a, const void* score_b)
{
    //sort integer score with the largest first.
    int a = *(int*)score_a;
    int b = *(int*)score_b;
    return b - a;
    //subtract the numbers the other way around, you'll reverse the order of final sort.
}
```

```
typedef struct
{
    //rectangle type
    int width;
    int height;
} rectangle;

int compare_areas(const void* a, const void* b)
{
    //sort the rectangles in area order, smallest first.
    //first convert pointers to correct types.
    rectangle *ra = (rectangle*)a;
    rectangle *rb = (rectangle*)b;
    //then calculate the areas.
    int area_a = (ra->width * ra->height);
    int area_b = (rb->width * rb->height);
    return area_a - area_b;
}
```

```
int compare_names(const void* a, const void* b)
{
    //sort a list of names in alphabetical order. Case-sensitive
    //Hint: strcmp("Abc", "Def") < 0
    //if a string is a pointer to a char, what will a pointer to it be?
    char **sa = (char**)a;
    char **sb = (char**)b;
    return strcmp(*sa, *sb);
    //use * operator to find the actual strings.
}
```

Don't worry if this exercise caused you a few problems.

It involved pointers, function pointers, and even a little math.

If you found it tough, take a break, drink a little water, and then try it again in an hour or two.

Some of the comparator functions were really pretty gnarly, so it's worth seeing how they run in action.

This is the kind of code you need to call the functions.

```
int main()
{
    int scores[] = {543, 323, 32, 554, 11, 3, 112};
    int i;
    qsort(scores, 7, sizeof(int), compare_scores_desc);
    puts("These are the scores in order:");
    for (i = 0; i < 7; i++) {
        printf("Score = %i\n", scores[i]);
    }
    char *names[] = {"Karen", "Mark", "Brett", "Molly"};
    qsort(names, 4, sizeof(char*), compare_names);
    puts("These are the names in order:");
    for (i = 0; i < 4; i++) {
        printf("%s\n", names[i]);
    }
    return 0;
}
```

This is the line that sorts the scores.

This will print out the array once it's been sorted.

This sorts the names.

Remember: an array of names is just an array of char pointers, so the size of each item is `sizeof(char*)`.

`qsort()` changes the order of the elements in the array.

This prints the sorted names out.

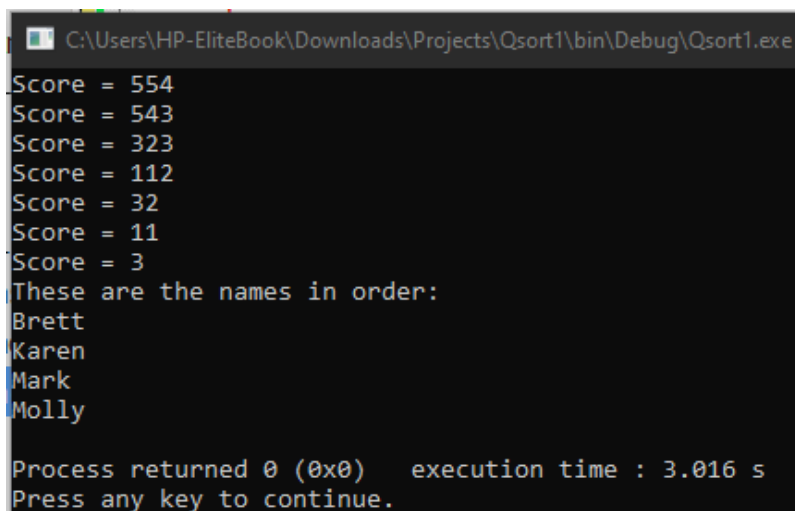
```

int main()
{
    int scores[] = {543, 323, 32, 554, 11, 3, 112};
    int i;
    //the line that sorts the scores.
    //qsort() changes the order of elements in the array
    qsort(scores, 7, sizeof(int), compare_scores_desc);
    for(i = 0; i < 7; i++)
    {
        printf("Score = %i\n", scores[i]);
    }

    char *names[] = {"Karen", "Mark", "Brett", "Molly"};
    qsort(names, 4, sizeof(char*), compare_names);
    puts("These are the names in order: ");
    for(i = 0; i < 4; i++)
    {
        printf("%s\n", names[i]); //prints the sorted names out.
    }

    return 0;
}

```



```

C:\Users\HP-EliteBook\Downloads\Projects\Qsort1\bin\Debug\Qsort1.exe
Score = 554
Score = 543
Score = 323
Score = 112
Score = 32
Score = 11
Score = 3
These are the names in order:
Brett
Karen
Mark
Molly

Process returned 0 (0x0)   execution time : 3.016 s
Press any key to continue.

```

Great, it works. Now try writing your own example code.

The sorting functions can be incredibly useful, but the comparator functions they need can be tricky to write.

But the more practice you get, the easier they become.

QUESTIONS:

I don't understand the comparator function for the array of strings. What does `char**` mean? Each item in a string array is a char pointer (`char*`). When `qsort()` calls the comparator function, it sends pointers to two elements in the arrays. That means the comparator receives two pointers-to-pointers-to-char. In C notation, each value is a `char**`.

OK, but when I call the `strcmp()` function, why does the code say `strcmp(*a, *b)`? Why not `strcmp(a, b)`? `a` and `b` are of type `char**`. The `strcmp()` function needs values of type `char*`.

Does `qsort()` create a sorted version of an array? A: It doesn't make a copy, it actually modifies the original array. Q: Why does my head hurt? Don't worry about it. Pointers are really difficult to use sometimes. If you don't find them a little confusing, it probably means you aren't thinking hard enough about them.

SUMMARY FOR CHAR**

If a string is a pointer to a char in C, then a pointer to that string would be a pointer to a pointer to char, also known as a `char**`.

In C, a string is just an array of char values, terminated by a null character (`'\0'`).

To create a string, you declare an array of char values and initialize it with a sequence of characters, like this:

```
char my_string[] = "hello";
```

In this example, `my_string` is an array of char values, and because arrays are automatically converted to pointers in certain contexts in C, you can also treat `my_string` as a pointer to a char, like this:

```
char* ptr_to_string = my_string;
```

In this case, `ptr_to_string` is a pointer to the first element of the `my_string`

array, which is a char value representing the letter "h".

To create a pointer to the ptr_to_string pointer, you would use the char** type, like this:

```
char** ptr_to_ptr = &ptr_to_string;
```

In this example, ptr_to_ptr is a pointer to the ptr_to_string pointer. The & operator is used to get the address of ptr_to_string, which is a pointer to the first element of the my_string array.

So, in summary, a pointer to a string in C is a char**, which is a pointer to a pointer to char.

ENUMS SUMMARY

Enums in c are use like this. Let me tell you my understanding. If i have three values inches, yard and centimetres. i want to use them as options. if the user enters 1, they've chosen inches, 2 they've chosen centimeters, 3 they've chosen yards.

Yes, that's a good example of how you can use enums in C. Enums allow you to define a set of named values that can be used as options or choices in a program. In your example, you could define an enum to represent the units of measurement like this:

```
enum Unit {  
    INCHES,  
    CENTIMETERS,  
    YARDS  
};
```

In this example, Unit is the name of the enum, and INCHES, CENTIMETERS, and YARDS are the named values (also known as enumerators) that represent the three units of measurement. By default, the first enumerator has a value of 0, and each subsequent **enumerator** is assigned a value that is one greater than the previous

enumerator.

You can then use this enum in your program to provide options to the user. For example, you could prompt the user to enter a number to choose a unit of measurement, like this:

```
printf("Enter a number to choose a unit of measurement:\n");
printf("1. Inches\n");
printf("2. Centimeters\n");
printf("3. Yards\n");

int choice;
scanf("%d", &choice);

enum Unit chosen_unit;

switch (choice) {
    case 1:
        chosen_unit = INCHES;
        break;
    case 2:
        chosen_unit = CENTIMETERS;
        break;
    case 3:
        chosen_unit = YARDS;
        break;
    default:
        printf("Invalid choice\n");
        return 1;
} // Now you can use the chosen unit enum to perform calculations or make decisions in your program
```

In this example, the user enters a number to choose a unit of measurement, and the program uses a switch statement to assign the appropriate enum value to the `chosen_unit` variable.

You can then use this enum to perform calculations or make decisions based on the user's choice.

AUTOMATING DEAR JOHN LETTERS

Imagine you're writing a mail-merge program to send out different types of messages to different people.

One way of creating the data for each response is with a struct like this:

```
//These are three types of messages that will be sent to people.
enum response_type{DUMP, SECOND_CHANCE, MARRIAGE};
typedef struct
{
    char *name;
    enum response_type;
}response; //you'll record a response type with each piece of response data
```

The enum gives you the names for each of the three types of response you'll be sending out, and that response type can be recorded against each response.

Then you'll be able to use your new response data type by calling one of these three functions for each type of response:

```
void dump(response r)
{
    printf("Dear %s\n", r.name);
    puts("Unfortunately your last date contacted us to");
    puts("Say that they will not be seeing you again");
}

void second_chance(response r)
{
    printf("Dear %s\n", r.name);
    puts("Good news: your last name date asked us to");
    puts("Arrange another meeting. Please call ASAP.");
}

void marriage(response r)
{
    printf("Dear %s\n", r.name);
    puts("Congratulations! Your last date has contacted");
    puts("Us with a proposal of marriage.");
}
```

So, now that you know what the data looks like, and you have the functions to generate the responses, let's see how complex the code is to generate a set of responses from an array of data.

Take code fragments from the pool and place them into the blank lines below.

Your goal is to piece together the main() function so that it can generate a set of letters for the array of response data.

You may not use the same code fragment more than once.

```
int main()
{
    response r[] =
    {
        {"Mike", DUMP}, {"Luis", SECOND_CHANCE}, {"William", MARRIAGE}
    };
    int i;
    for(i = 0; i < 4; i++)
    {
        switch(r[i].type)
        {
            case DUMP:
                dump(r[i]);
                break;
            case SECOND_CHANCE:
                second_chance(r[i]);
                break;
            default:
                marriage(r[i]);
        }
    }
    return 0;
}
```

Well, it's good that it worked, but there is quite a lot of code in there just to call a function for each piece of response data.

Every time you need call a function that matches a response type, it will look like this:

```
switch(r.type)
{
    case DUMP:
        dump(r);
        break;
    case SECOND_CHANCE:
        second_chance(r);
        break;
    default:
        marriage(r);
}
```

And what will happen if you add a fourth response type?

You'll have to change every section of your program that looks like this.

Soon, you will have a lot of code to maintain, and it might go wrong.

Fortunately, there is a trick that you can use in C, and it involves arrays...

CREATE AN ARRAY OF POINTERS

The trick is to create an array of function pointers that match the different response types. Before seeing how that works, let's look at how to create an array of function pointers.

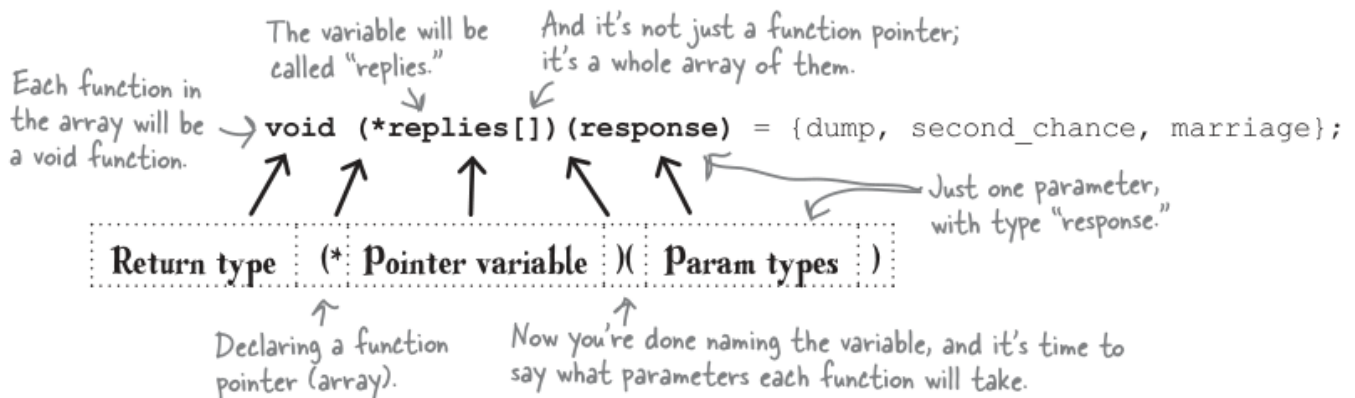
If you had an array variable that could store a whole bunch of function names, you could use it like this:

But that syntax doesn't quite work in C.

You have to tell the compiler exactly what the functions will look like that you're going to store in the array: what their return types will be and what parameters they'll accept.

That means you have to use this much more complex syntax:

```
replies[] = {dump, second_chance, marriage};
```



But how does an array help? Look at that array.

It contains a set of function names that are in exactly the same order as the types in the enum:

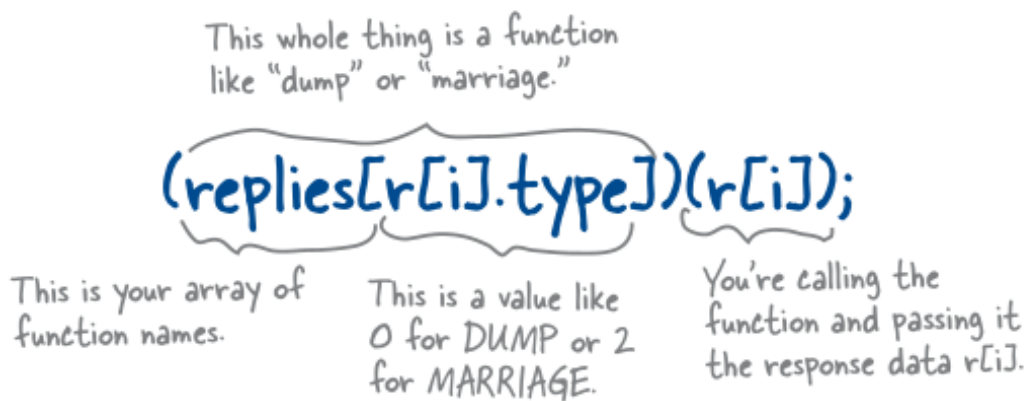
```
replies[] = {dump, second_chance, marriage};  
enum response_type {DUMP, SECOND_CHANCE, MARRIAGE};
```

This is really important, because when C creates an enum, it gives each of the symbols a number starting at 0.

So `DUMP == 0`, `SECOND_CHANCE == 1`, and `MARRIAGE == 2`.

And that's really neat, because it means you can get a pointer to one of your sets of functions using a `response_type`:

Let's break that down.



Now, when you run the new version of the program, you get exactly the same output as before:

```
File Edit Window Help WholsJohn
> ./dear_johns
Dear Mike,
Unfortunately your last date contacted us to
say that they will not be seeing you again
Dear Luis,
Good news: your last date has asked us to
arrange another meeting. Please call ASAP.
Dear Matt,
Good news: your last date has asked us to
arrange another meeting. Please call ASAP.
Dear William,
Congratulations! Your last date has contacted
us with a proposal of marriage.
>
```

The difference? Now, instead of an entire switch statement, you just have this:

```
(replies[r[i].type])(r[i]);
```

The code you provided is an example of how to replace a switch statement with an array of function pointers in C.

In a switch statement, a variable is compared to multiple cases, and the code block associated with the matching case is executed.

However, this can become unwieldy and difficult to maintain when there are many cases.

Instead, the code replaces the switch statement with an array of function pointers that correspond to each case.

The function pointers are then used to call the corresponding function for each element in the response array.

The function pointers are stored in the replies array, which is an array of function pointers that take a response struct as an argument.

The main function initializes an array of response structs and loops through each element, calling the function pointer stored in replies that corresponds to the type field of the response struct.

This approach can be easier to maintain and modify than a long switch statement, as adding or removing cases involves only changing the replies array.

If you have to call the response functions at several places in the program, you won't have to copy a lot of code. And if you decide to add a new type and a new function, you can just add it to the array:

```
//you can add new types and functions like this.  
enum response_type{DUMP, SECOND_CHANCE, MARRIAGE, LAW_SUIT};  
void (*replies[])(response) = {dump, second_chance, marriage, law_suit};
```

Arrays of function pointers can make your code much easier to manage.

They are designed to make your code scalable by making it shorter and easier to extend. Even though they are quite difficult to understand at first, function pointer arrays can really crank up your C programming skills.

SUMMARY

- Function pointers store the addresses of functions.
- The name of each function is actually a function pointer.
- If you have a function `shoot()`, then `shoot` and `&shoot` are both pointers to that function.
- You declare a new function pointer with `return-type(*var-name)(param-types)`.
- If `fp` is a function pointer, you can call it with `fp(params, ...)`.
- Or, you can use `(*fp)(params,...)`. *C* will work the same way.
- The *C* Standard Library has a sorting function called `qsort()`.
- `qsort()` accepts a pointer to a comparator function that can test for (in)equality.
- The comparator function will be passed pointers to two items in the array being sorted.
- If you have an array of data, you can associate functions with each data item using function pointer arrays.

Why is the function pointer array syntax so complex? Because when you declare a function pointer, you need to say what the return and parameter types are. That's why there are so many parentheses.

This looks a little like the sort of object-oriented code in other languages. Is it? It's similar. Object-oriented languages associate a set of functions (called methods) with pieces of data. In the same way, you can use function pointers to associate functions with pieces of data.

Hey, so does that mean that *C* is object oriented? Wow, that's awesome. No. *C* is not object oriented, but other languages that are built on *C*, like Objective-C and C++, create a lot of their object-oriented features by using function pointers under

the covers.

MAKE YOUR FUNCTIONS STRETCHY

Sometimes, you want to write C functions that are really powerful, like your `find()` function that could search using function pointers.

But other times, you just want to write functions that are easy to use.

Take the `printf()` function. The `printf()` function has one really cool feature that you've used: it can take a variable number of arguments:

```
printf("%i bottles of beer on the wall, %i bottles of beer\n", 99, 99);  
printf("Take one down and pass it around, ");  
printf("%i bottles of beer on the wall\n", 98);
```

You can pass the `printf()` as many arguments as you need to print.

So how can YOU do that? And you've got just the problem that needs it.

Down in the Head First Lounge, they're finding it a little difficult to keep track of the drink totals.

One of the guys has tried to make life easier by creating an enum with the list of cocktails available and a function that returns the prices for each one:

```

enum drink
{
    MUDSLIDE, FUZZY_NAVEL, MONKEY_GLAND, ZOMBIE
};

double price(enum drink d)
{
    switch(d)
    {
        case MUDSLIDE:
            return 6.79;
        case FUZZY_NAVEL:
            return 5.31;
        case MONKEY_GLAND:
            return 4.82;
        case ZOMBIE:
            return 5.89;
    }
    return 0;
}

```

And that's pretty cool, if the Head First Lounge crew just wants the price of a drink.

But what they want to do is get the price of a total drinks order.

Easy → `price(ZOMBIE)`

total(3, ZOMBIE, MONKEY_GLAND, FUZZY_NAVEL)

← The number of drinks

← Not so easy

↑ A list of the drinks in the order

They want a function called `total()` that will accept a count of the drinks and then a list of drink names.

They want a function called `total()` that will accept a count of the drinks and then a list of drink names.

A function that takes a variable number of parameters is called a **variadic function**.

The C Standard Library contains a set of macros that can help you create your own variadic functions.

To see how they work, you'll create a **function that can print out series of ints**:

You can think of macros as a special type of function that can modify your source code.

`print_ints(3, 79, 101, 32);`

Number of ints to print The ints that need to be printed

Here's the code:

```
#include <stdarg.h>

void print_ints(int args, ...)
{
    va_list ap;
    va_start(ap, args);

    int i;
    for (i = 0; i < args; i++) {
        printf("argument: %i\n", va_arg(ap, int));
    }

    va_end(ap);
}
```

This is a normal, ordinary argument that will always be passed.

`va_start` says where the variable arguments start.

This will loop through all of the other arguments.

`args` contains a count of how many variables there are.

The variable arguments will follow here.

The variable arguments will start after the `args` parameter.

Let's break it down and take a look at it, step by step.

Include the `stdarg.h` header. All the code to handle variadic functions is in `stdarg.h`, so you need to make sure you include it.

Tell your function there's more to come... Remember those books where the heroine drags the guy through the bedroom and then the chapter ends "..."? Well, that "..." is called an ellipsis, and it tells you that something else is going to follow. In C, an ellipsis after the argument of a function means there are more arguments to come.

No, we don't read those books either. Create a `va_list`. A `va_list` will be used to store the extra arguments that are passed to your function.

Say where the variable arguments start. C needs to be told the name of the last fixed argument. In the case of our function, that'll be the `args` parameter.

Then read off the variable arguments, one at a time. Now your arguments are all stored in the `va_list`, you can read them with `va_arg`. `va_arg` takes two values: the `va_list` and the type of the next argument. In your case, all of the arguments are ints.

Finally...end the list. After you've finished reading all of the arguments, you need to tell C that you're finished. You do that with the `va_end` macro.

Now you can call your function. Once the function is complete, you can call it:

```
#include <stdarg.h>

void print_ints(int args, ...)
{
    va_list ap;
    va_start(ap, args);
    int i;
    for(i = 0; i < args; i++)
    {
        printf("Argument: %i\n", va_arg(ap, int));
    }
    va_end(ap);
}
```

```
print_ints(3, 79, 101, 32);
```

This will print out 79, 101, and 32 values.

A macro is used to rewrite your code before it's compiled. The macros you're using here (`va_start`, `va_arg`, and `va_end`) might look like functions, but they actually hide secret instructions that tell the preprocessor how to generate lots of extra smart code inside your program, just before compiling it.

Wait, why are `va_end` and `va_start` called macros? Aren't they just normal functions? No, they are designed to look like ordinary functions, but they actually are replaced by the preprocessor with other code.

And the preprocessor is? The preprocessor runs just before the compilation step. Among other things, the preprocessor includes the headers into the code.

Can I have a function with just variable arguments, and no fixed arguments at all? No. You need to have at least one fixed argument in order to pass its name to `va_start`.

What happens if I try to read more arguments from `va_arg` than have been passed in? Random errors will occur. That sounds bad. Yep, pretty bad.

What if I try to read an int argument as a double, or something? Random errors will occur.

```
double total(int args, ...)
{
    double total = 0;
    va_list ap;
    va_start(ap, args);
    int i;
    for(i = 0; i < args; i++)
    {
        enum drink d = va_arg(ap, enum drink);
        total = total + price(d);
    }
    va_end(ap);
    return total;
}
```

If you create a little test code to call the function, you can compile it and see what happens:

```
main()
{
    printf("Price is %.2f\n", total(2, MONKEY_GLAND, MUDSLIDE));
    printf("Price is %.2f\n", total(3, MONKEY_GLAND, MUDSLIDE, FUZZY_NAVEL));
    printf("Price is %.2f\n", total(1, ZOMBIE));
    return 0;
}
```

Your code works!

```
File Edit Window Help Cheers
> ./price drinks
Price is 11.61
Price is 16.92
Price is 5.89
>
```

And this is the
output.

Yeah, baby! I could
remember these even
after one too many
Monkey Glands...



~~You will need at least one fixed
parameter.~~

~~Be careful that you don't try to read
more parameters than you've been given.~~

Now you know how to use variable arguments to make your code simpler and more intuitive to use!

Functions that accept a variable number of arguments are called variadic functions.

To create variadic functions, you need to include the `stdarg.h` header file.

The variable arguments will be stored in a `va_list`.

You can control the `va_list` using `va_start()`, `va_arg()`, and `va_end()`.

You will need at least one fixed parameter.

Be careful that you don't try to read more parameters than you've been given.

You will always need to know the data type of every parameter you read.

Function pointers let you pass functions around as if they were data.

Function pointers are the only pointers that don't need the `*` and `&` operators...

...but you can still use them if you want to.

The name of every function is a pointer to the function.

`qsort()` will sort an array.

Each sort function needs a pointer to a comparator function.

Comparator functions decide how to order two pieces of data.

Functions with a variable number of arguments are called "variadic."

Arrays of function pointers can help run different functions for different types of data.

`stdarg.h` lets you create variadic functions.

The name of every function is a pointer to the function.

qsort()
will sort
an array.

Each sort
function needs
a pointer to
a comparator
function.

Comparator
functions decide
how to order
two pieces of
data.

Arrays of function
pointers can help
run different
functions for
different types of
data.

Functions with a
variable number
of arguments are
called "variadic."

stdarg.h lets
you create
variadic
functions.