7 advanced functions

Turn your functions up to 11

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**. And finally, you'll discover how to make your code *super stretchy* with **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 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"
};
```

Let's write some code that uses string functions to filter this array down.

Code Magnets

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.

Beware: you might not need all the fragments to complete the function.

```
void find()
{
  int i;
  puts("Search results:");
  puts("------------------------------------");

  for (i = 0; i ................. ................. ; i++) {

    if (................ ( ................., ................)

        ................. ................. ................( ................., ................)) {

      printf("%s\n", ADS[i]);
    }
  }
  puts("------------------------------------");
}
```

strcmp

strstr

ADS[i]

ADS[i]

"sports"

NUM_ADS

<

strstr

!

&&

||

"bieber"

strcmp

Code Magnets Solution

You were to 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.

```
void find()
{
  int i;
  puts("Search results:");
  puts("-----------------------------------");

  for (i = 0; i <  NUM_ADS ; i++) {
    if ( strstr ( ADS[i] , "sports" )
         && ! strstr ( ADS[i] , "bieber" )) {
      printf("%s\n", ADS[i]);
    }
  }
  puts("-----------------------------------");
}
```

strcmp

strcmp

||

Test Drive

Now, if you take the function and the data, and wrap everything up in a program called `find.c`, you can compile and run it like this:

```
File Edit Window Help FindersKeepers
> gcc find.c -o find && ./find
Search results:
-----------------------------------
William: SBM GSOH likes sports, TV, dining
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.

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**.

Exactly right. If you clone the function, you'll have a lot of duplicated code.

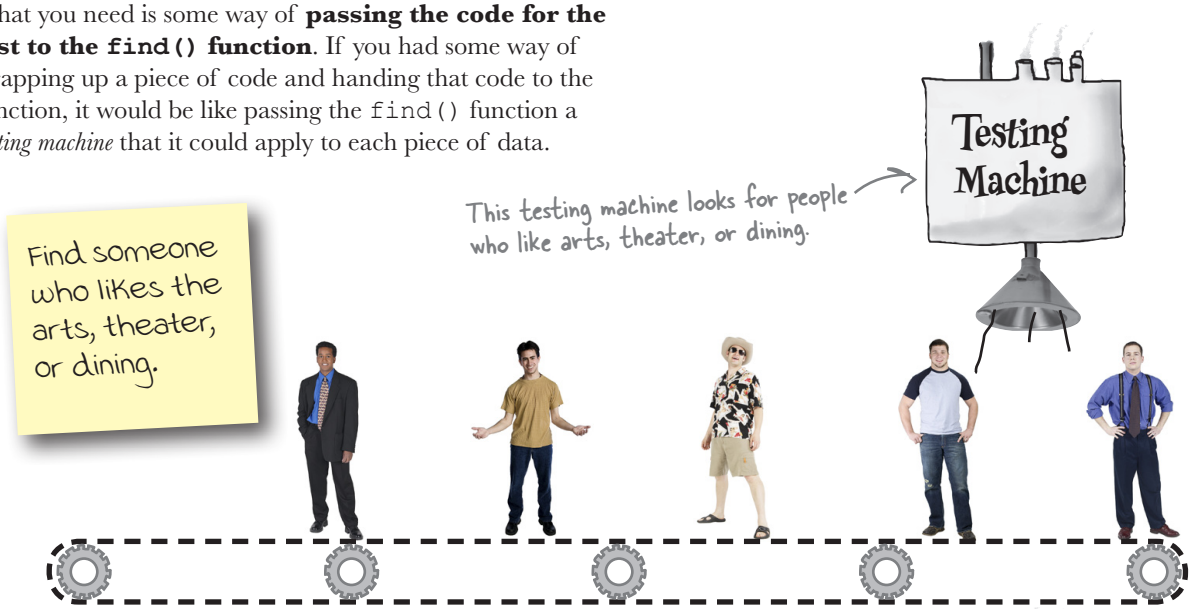
C programs often have to perform tasks that are *almost identical* except for some small detail. At the moment, the `find()` function runs through each element of the array and applies a simple test to each string to look for matches. But the test it makes is **hardwired**. It will always perform the same test.

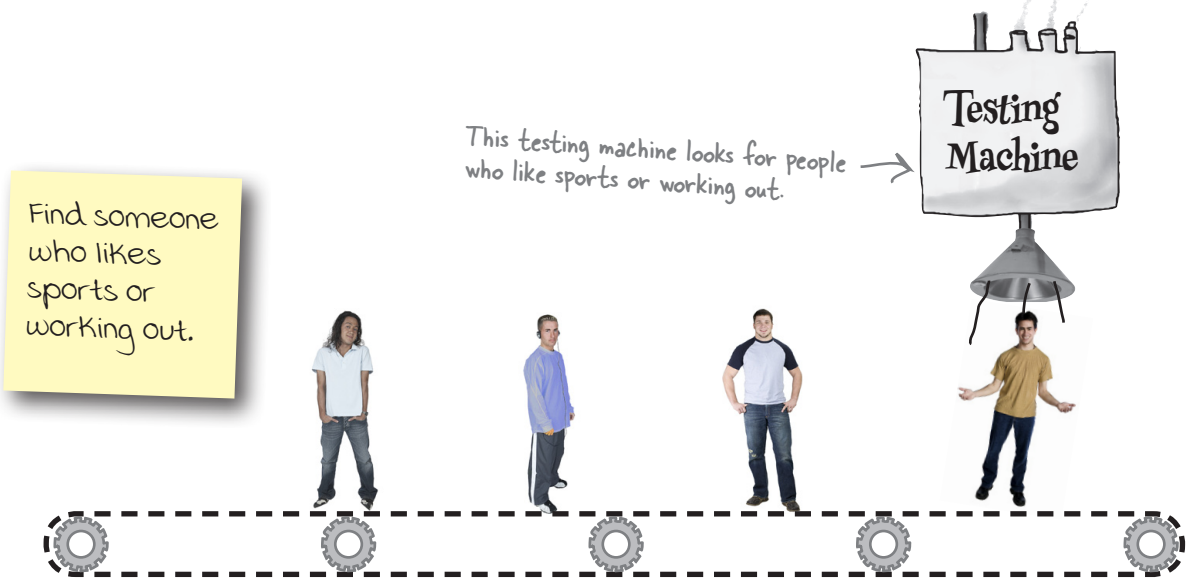
Now, you could pass some strings into the function so that it could search for different substrings. The trouble is, that wouldn't allow `find()` to check for *three* strings, like "arts," "theater," or "dining." And what if you needed something wildly different?

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 sports_no_bieber(char *s)
{
  return strstr(s, "sports") && !strstr(s, "bieber");
}
```

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.

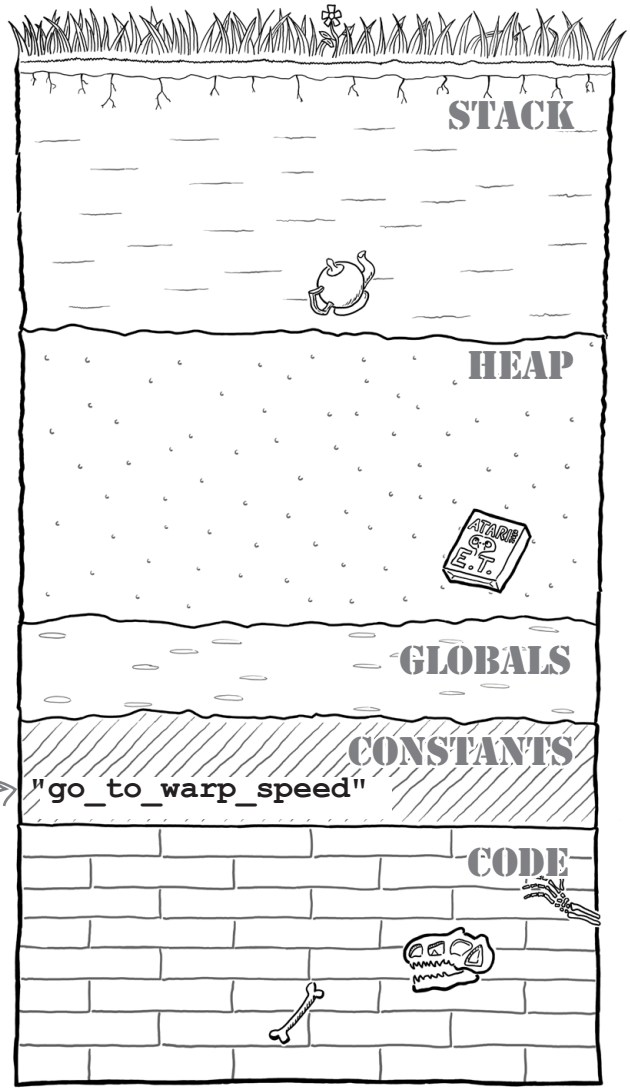
```
int go_to_warp_speed(int speed)
{
  dilithium_crystals(ENGAGE);
  warp = speed;
  reactor_core(c, 125000 * speed, PI);
  clutch(ENGAGE);
  brake(DISENGAGE);
  return 0;
}
```

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.



Let's look at the C syntax you'll need to work with function pointers.

...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…

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…

there are no Dumb Questions

Q: What does `char**` mean? Is it a typing error?

A: `char**` is a pointer normally used to point to an array of strings.

Exercise

Take a look at those other types of searches that people have asked for. See if you can create a function for each type of search. Remember: the first is already written.

Someone who likes sports but not Bieber

```
int sports_no_bieber(char *s)
{
  return strstr(s, "sports") && !strstr(s, "bieber");
}
```

Find someone who likes sports or working out.

```
int sports_or_workout(char *s)
{
  ..............................................................................
}
```

I want a non-smoker who likes the theater.

```
int ns_theater(char *s)
{
  ..............................................................................
}
```

Find someone who likes the arts, theater, or dining.

```
int arts_theater_or_dining(char *s)
{
  ..............................................................................
}
```

Then, see if you can complete the `find()` function:

```
void find( ........................................ )
{
  int i;
  puts("Search results:");
  puts("------------------------------------");
  for (i = 0; i < NUM_ADS; i++) {
    if (match(ADS[i])) {
      printf("%s\n", ADS[i]);
    }
  }
  puts("------------------------------------");
}
```

find() will need a function pointer passing to it called match.

This will call the match() function that was passed in.

Exercise Solution

You were to take a look at those other types of searches that people have asked for and create a function for each type of search.

Someone who likes sports but not Bieber

```
int sports_no_bieber(char *s)
{
  return strstr(s, "sports") && !strstr(s, "bieber");
}
```

Find someone who likes sports or working out.

```
int sports_or_workout(char *s)
{
  return strstr(s, "sports") || strstr(s, "working out");
}
```

I want a non-smoker who likes the theater.

```
int ns_theater(char *s)
{
  return strstr(s, "NS") && strstr(s, "theater");
}
```

Find someone who likes the arts, theater, or dining.

```
int arts_theater_or_dining(char *s)
{
  return strstr(s, "arts") || strstr(s, "theater") || strstr(s, "dining");
}
```

Then, you were to complete the `find()` function:

```
void find(  int (*match)(char*)  )
{
  int i;
  puts("Search results:");
  puts("------------------------------------");
  for (i = 0; i < NUM_ADS; i++) {
    if (match(ADS[i])) {
      printf("%s\n", ADS[i]);
    }
  }
  puts("------------------------------------");
}
```

Test Drive

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(sports_no_bieber);
  find(sports_or_workout);
  find(ns_theater);
  find(arts_theater_or_dining);
  return 0;
}
```

```
File Edit Window Help FindersKeepers
> ./find
Search results:
-----------------------------------
William: SBM GSOH likes sports, TV, dining
Josh: SJM likes sports, movies and theater
-----------------------------------
Search results:
-----------------------------------
William: SBM GSOH likes sports, TV, dining
Mike: DWM DS likes trucks, sports and bieber
Peter: SAM likes chess, working out and art
Josh: SJM likes sports, movies and theater
-----------------------------------
Search results:
-----------------------------------
Matt: SWM NS likes art, movies, theater
-----------------------------------
Search results:
-----------------------------------
William: SBM GSOH 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
-----------------------------------
>
```

This is find(sports_no_bieber).

This is find(sports_or_workout).

This is find(ns_theater).

This is find(arts_theater_or_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**.

The Hunter's Guide to Function Pointers

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.

Return type	(*	Pointer variable	)(	Param types	)

```
char**  (*names_fn)(char*,int)
```

This is the name of the variable you're declaring.

there are no Dumb Questions

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.