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.

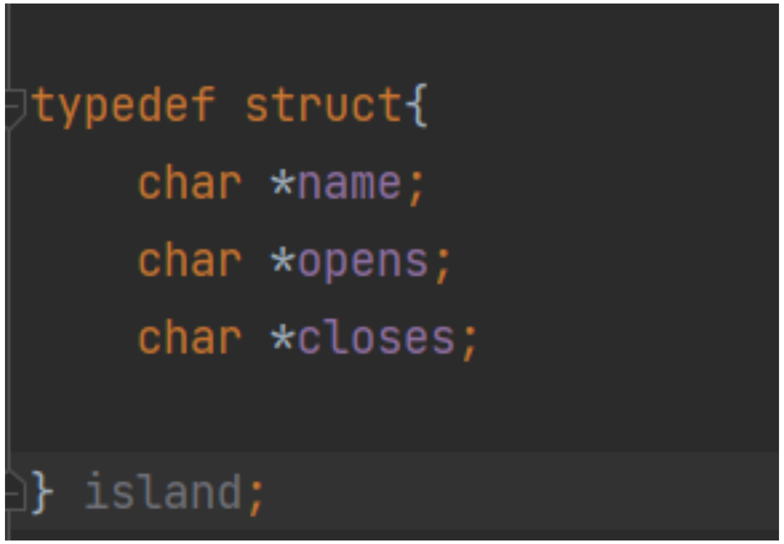
Imagine you're running a travel company that arranges flying tours through the islands.

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];

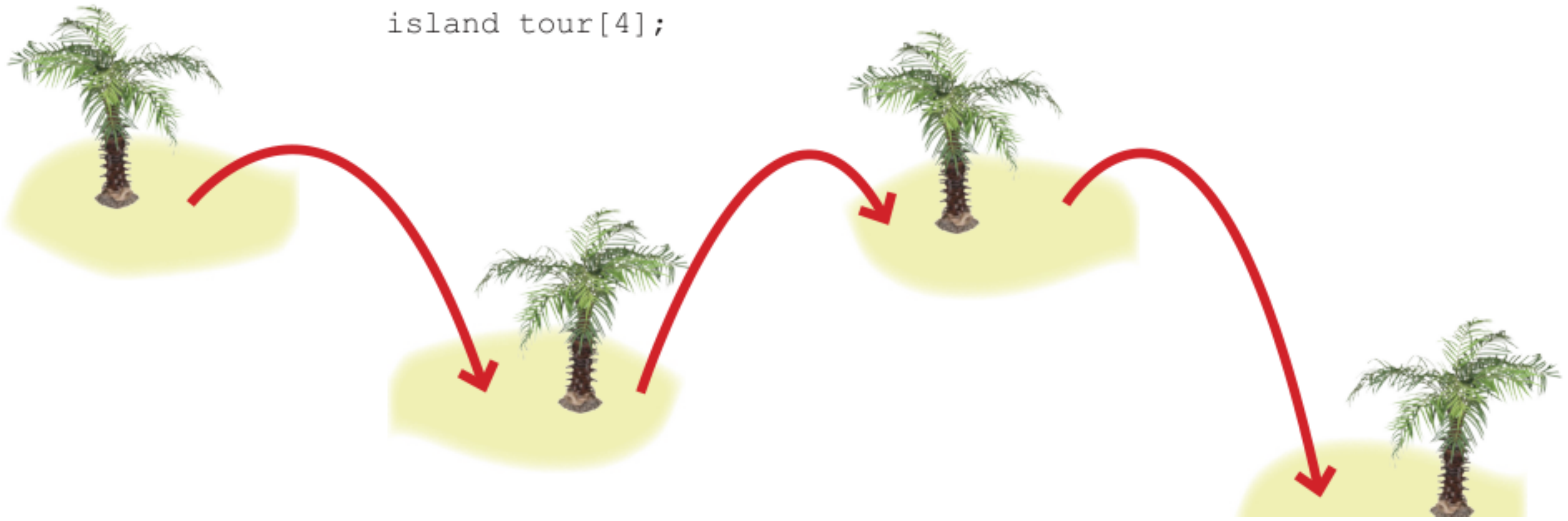
Coconut Airways flies C planes between the islands.

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.**



```
island tour[4];
```

Linked lists are like chains of data.

A linked list is an example of an abstract data structure.

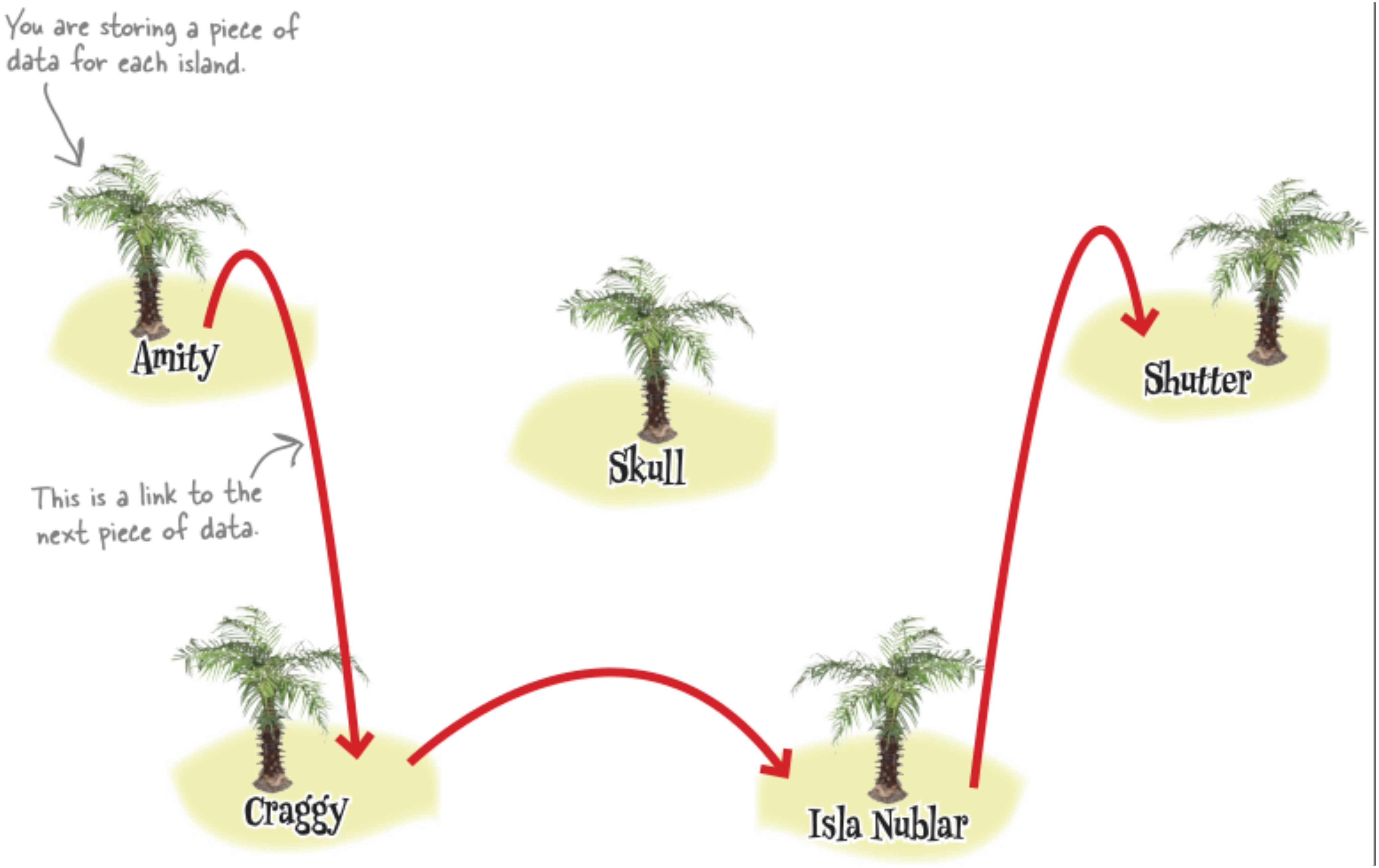
It's called an abstract data structure because a linked list is general: it can be used to store a lot of different kinds of data.

To understand how a linked list works, think back to our tour company.

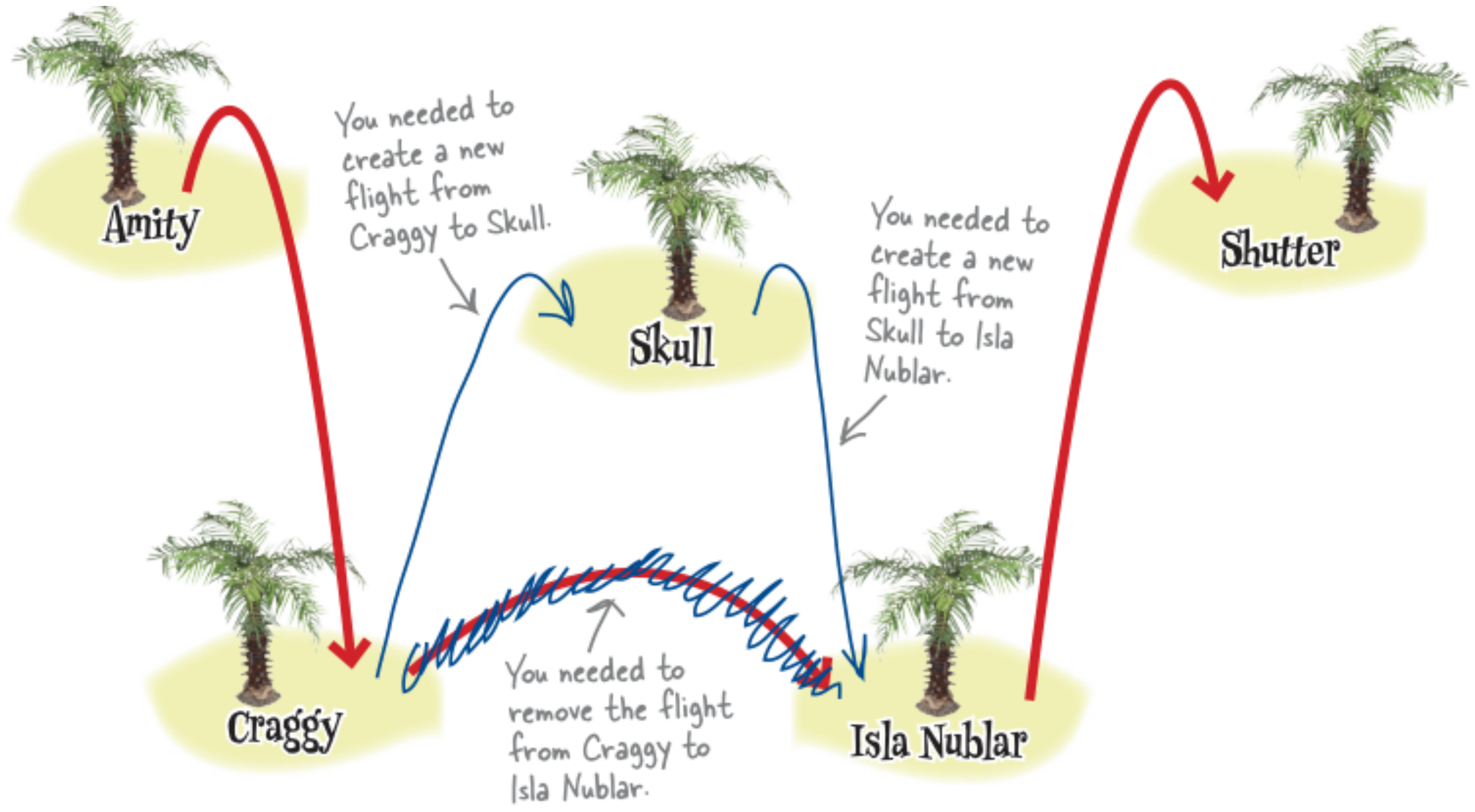
A linked list stores a piece of data, and a link to another piece of data.

In a linked list, as long as you know where the list starts, you can travel along the list of links, from one of 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.
Amity
This is a link to the next piece of data.
Skull
Shutter
Craggy
Isla Nublar

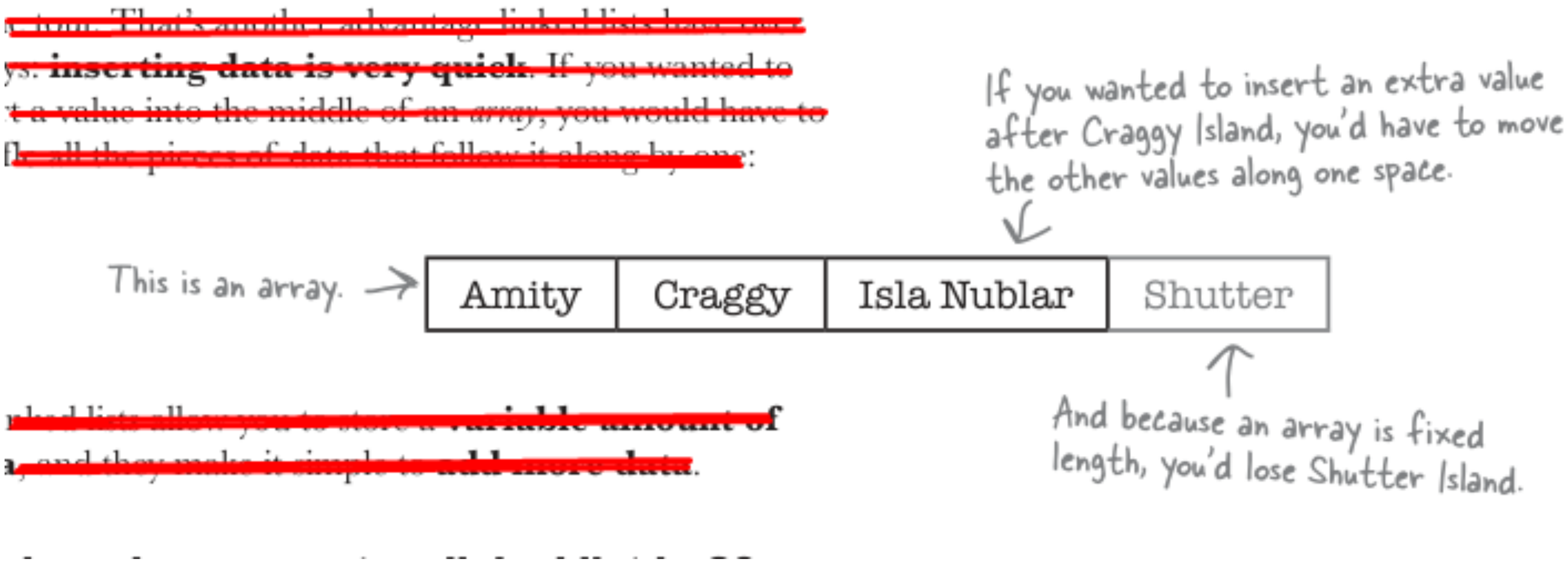
Result:


Amity
You needed to create a new flight from Craggy to Skull.
Skull
You needed to create a new flight from Skull to Isla Nublar.
Shutter
Craggy
You needed to remove the flight from Craggy to Isla Nublar.
Isla Nublar

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:



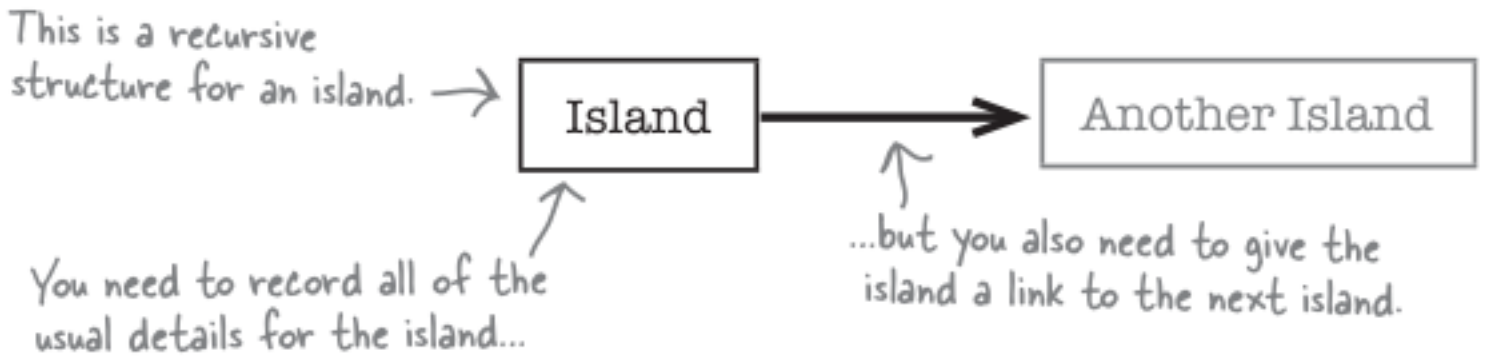
~~tion. That's another advantage linked lists have over~~
~~ys: **inserting data is very quick.** If you wanted to~~
~~t a value into the middle of an *array*, you would have to~~
~~fl all the pieces of data that follow it along by one~~:

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
And because an array is fixed length, you'd lose Shutter Island.

~~nked lists allow you to store a **variable amount of**~~
~~a, and they make it simple to **add more data**~~.

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.**

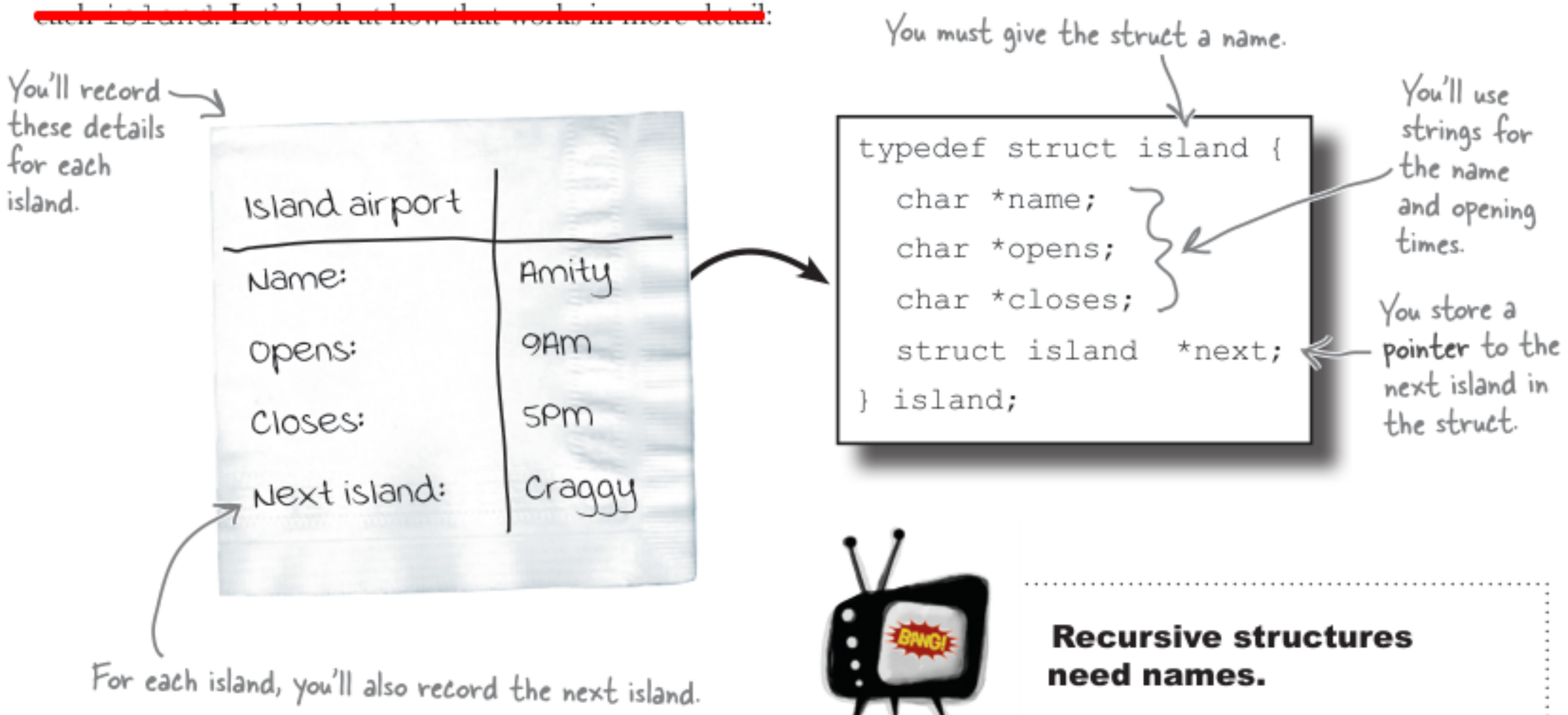

This is a recursive structure for an island.
Island
Another Island
You need to record all of the usual details for the island...
...but you also need to give the island a link to the next island.

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.
You'll use strings for the name and opening times.
You store a pointer to the next island in the struct.

```
typedef struct island {
   char *name;
   char *opens;
   char *closes;
   struct island  *next;
} island;
```

~~Watch it!~~

Recursive structures need names.

~~If you use the typedef~~
~~command, you can normally~~

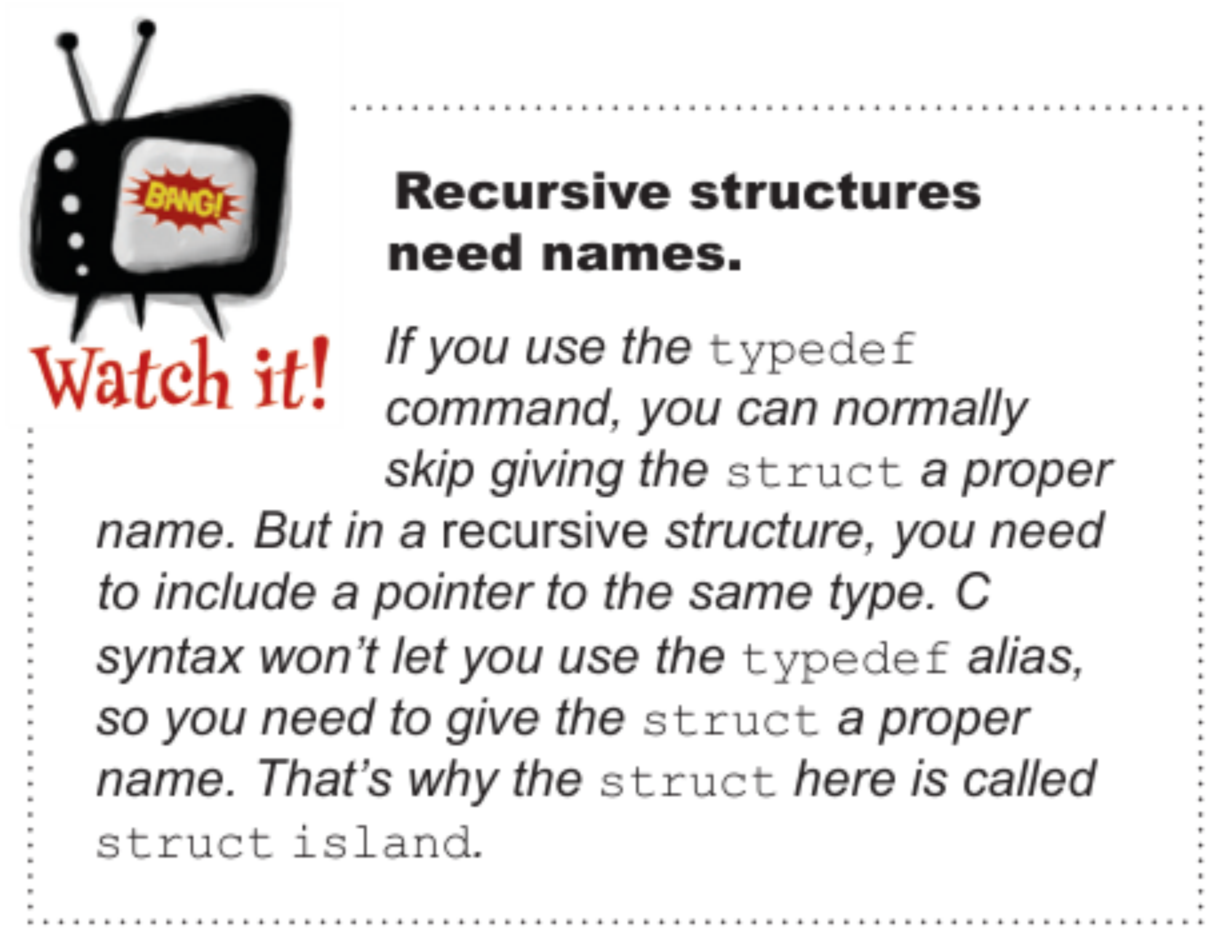
How do you store a link from one struct to the next? With a pointer.

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.

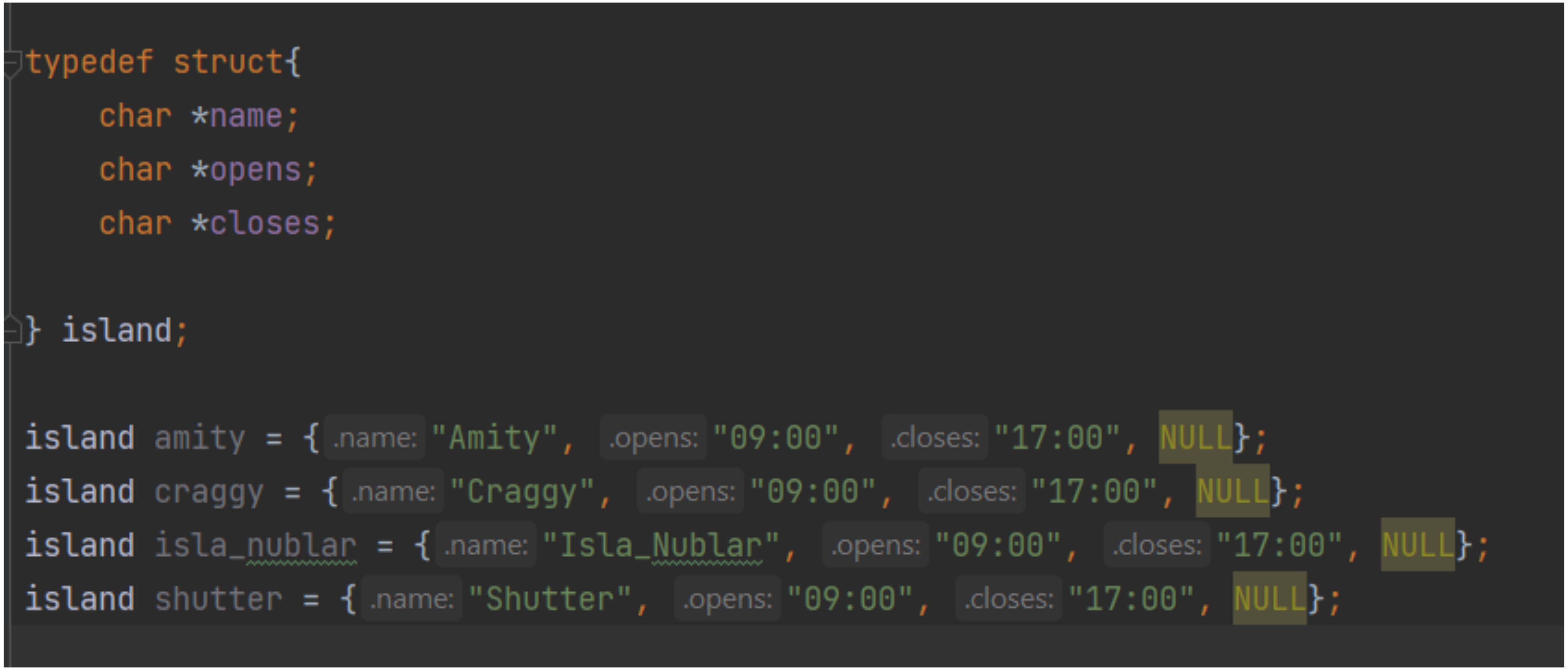
If you use the `typedef` *command, you can normally skip giving the* `struct` *a proper name. But in a recursive structure, you need to include a pointer to the same type. C syntax won't let you use the* `typedef` *alias, so you need to give the* `struct` *a proper name. That's why the* `struct` *here 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;

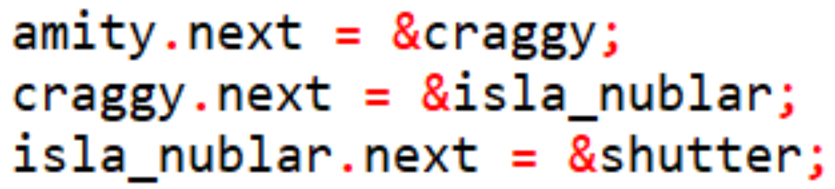
You have to be careful to set the next field in each island to the address of the next island.

You'll use struct variables for each of the islands.

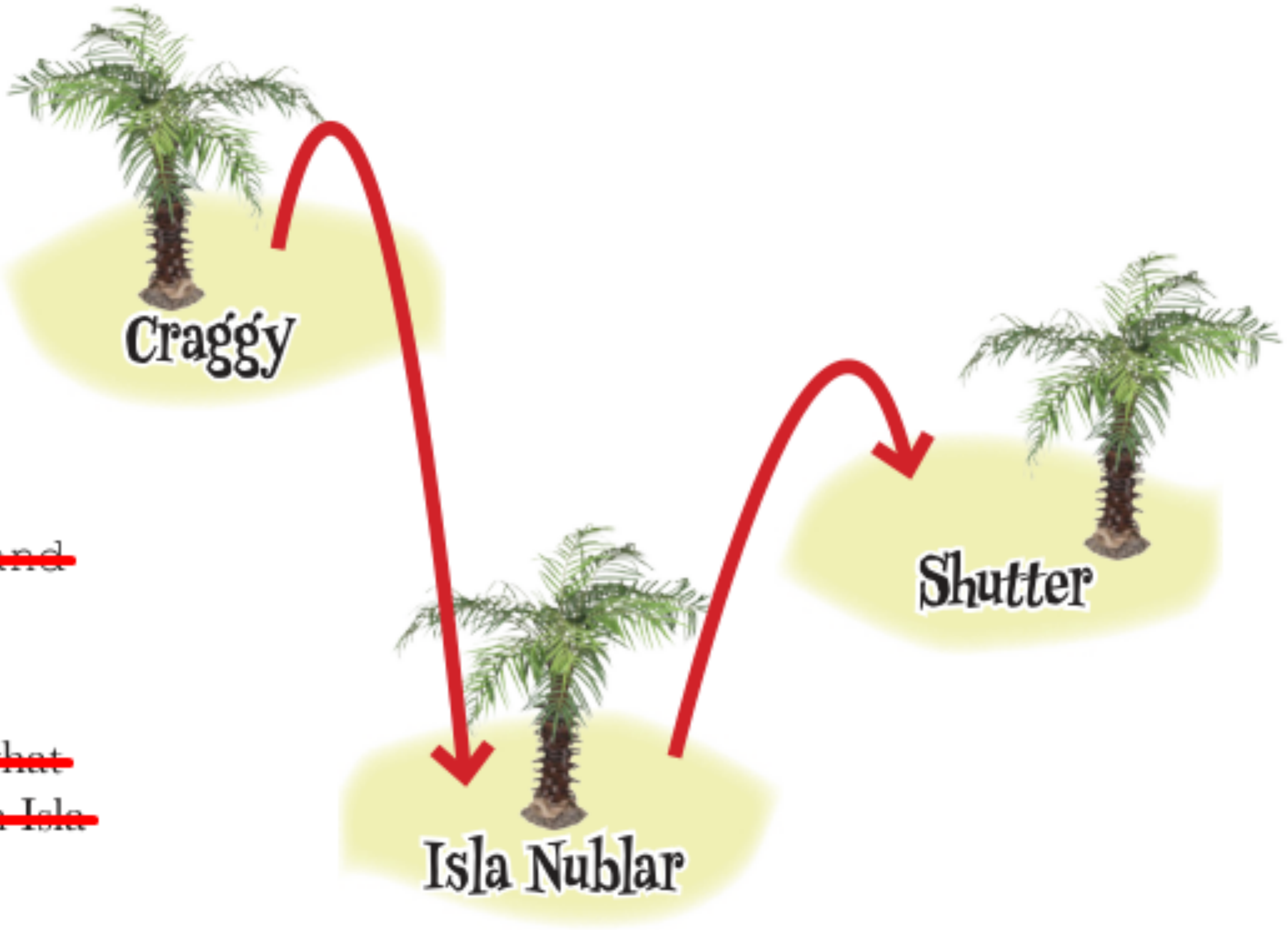
So now you've created a complete island tour in C, but what if you want to insert an excursion to Skull Island between Isla Nublar and Shutter Island?

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;
```



~~and~~

~~what~~
~~on Isla~~

INSERTING INTO THE LIST

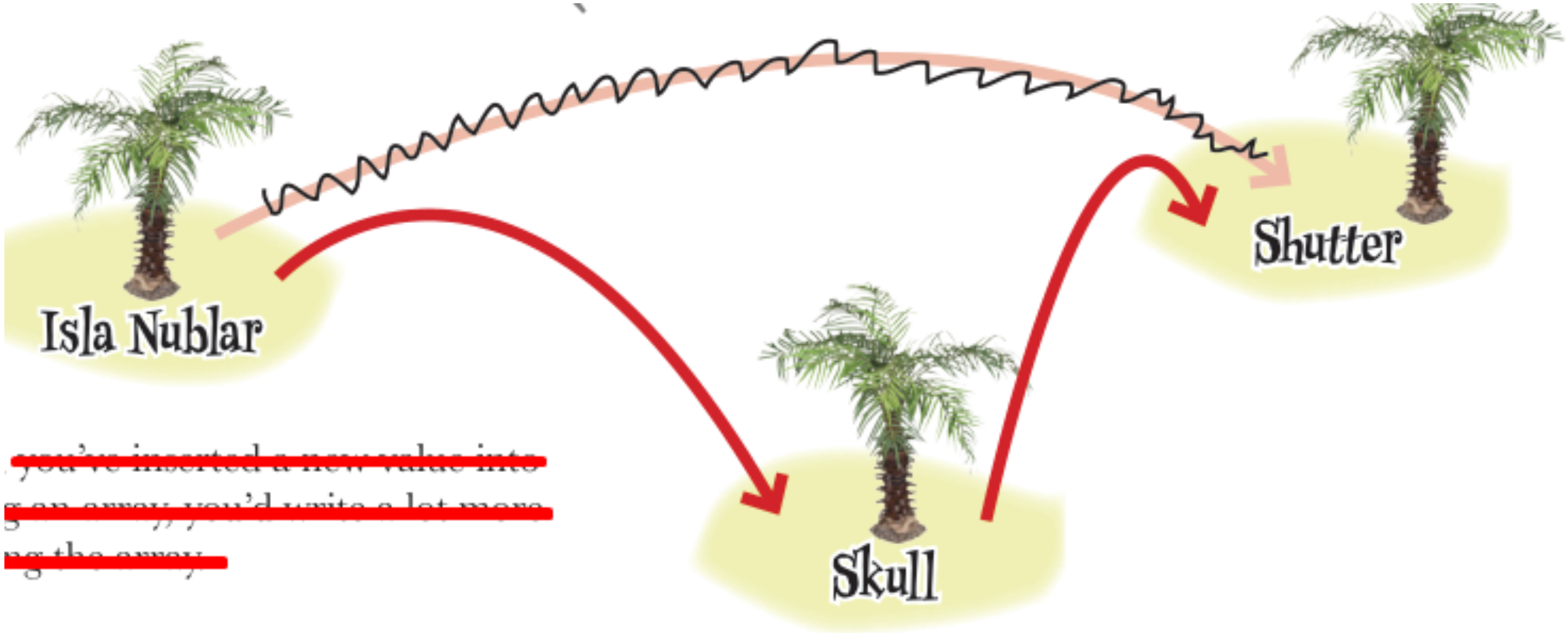
You can insert islands just like you did earlier, by changing the values of the pointers between islands:

```c
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.


Isla Nublar
Skull
Shutter
~~you've inserted a new value into~~
~~g an array, you'd write a lot more~~
~~ng the array.~~

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.

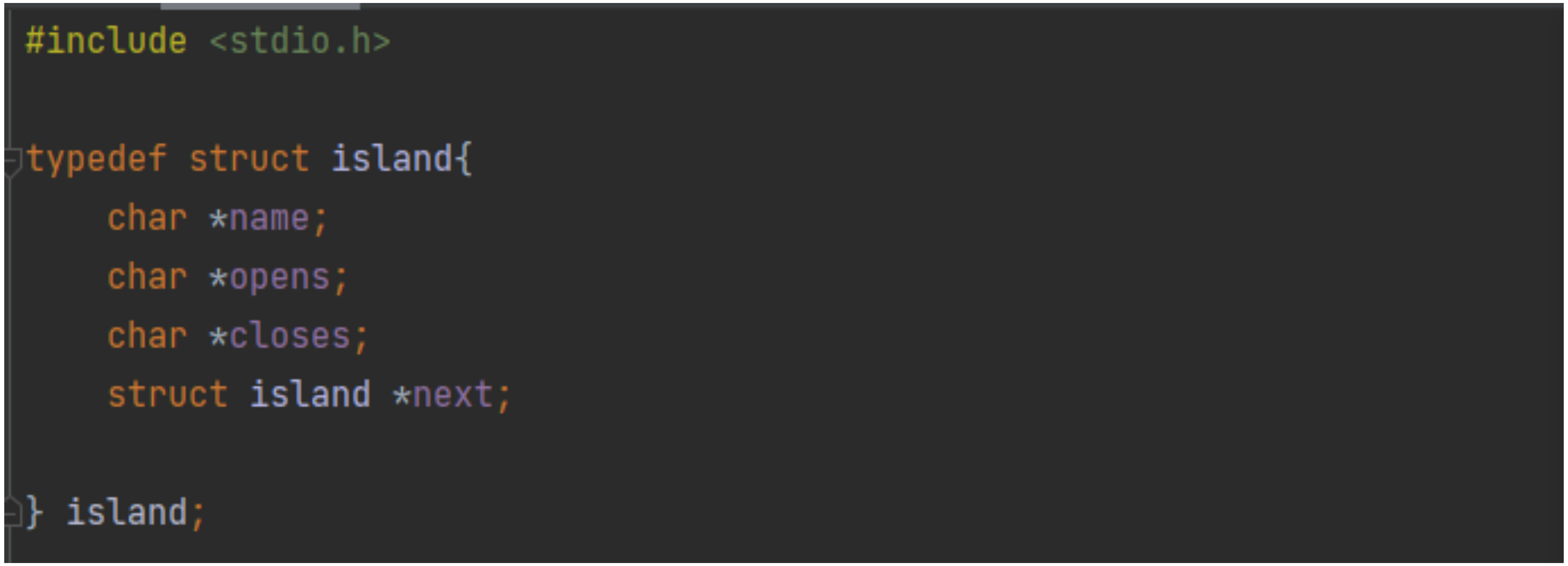
That’s not very good. I thought a linked list was better than an array. You shouldn’t think of data structures as being better or worse. They are either appropriate or inappropriate for what you want to use them for.

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.

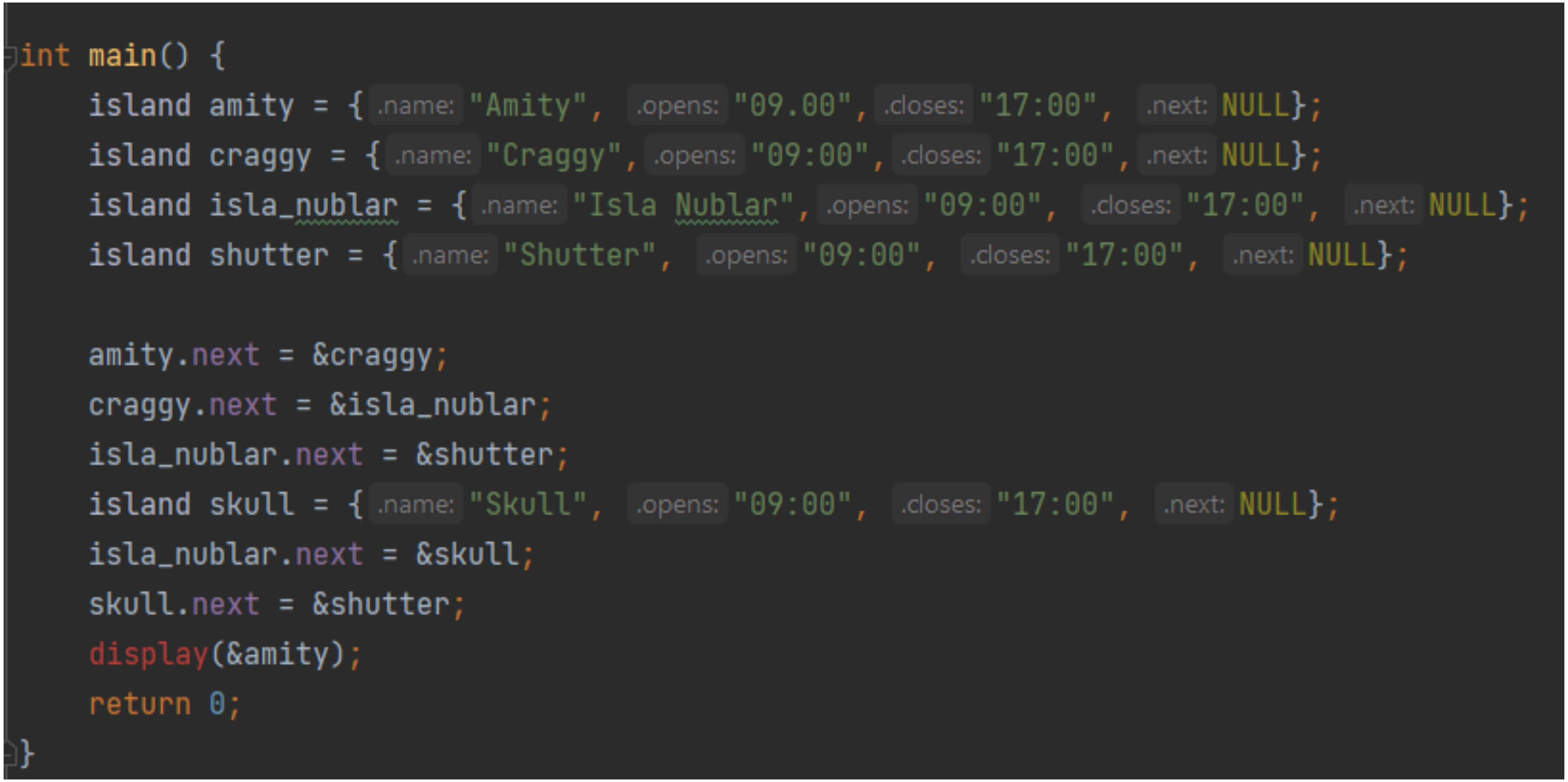
You’ve shown a struct that contains a pointer to another struct. Can a struct contain a whole recursive struct inside itself? Answer: No

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.

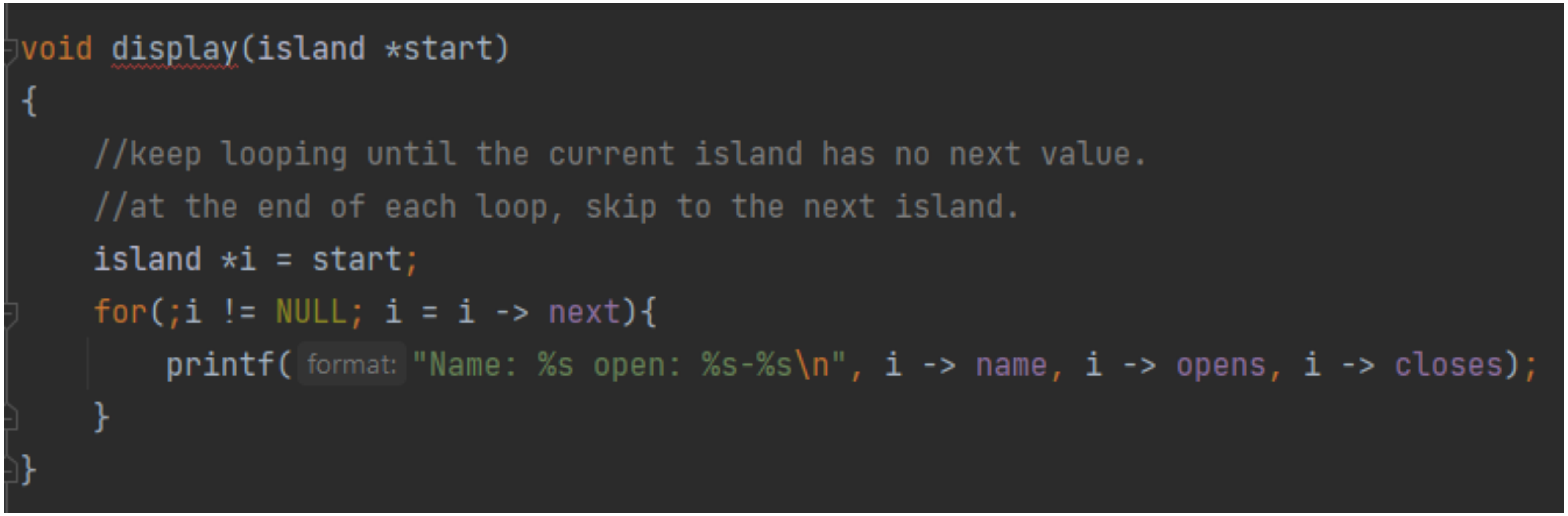
The full linked-list:



```c
#include <stdio.h>

typedef struct island{
    char *name;
    char *opens;
    char *closes;
    struct island *next;

} island;
```

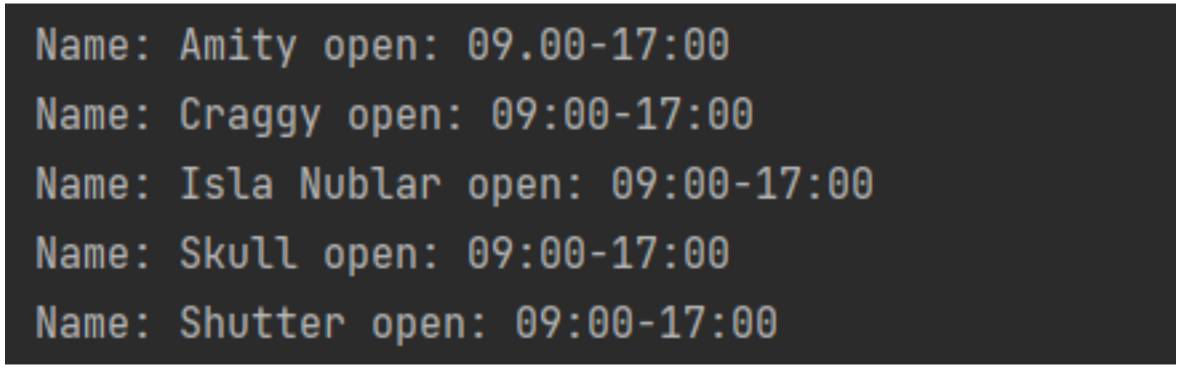


```c
int main() {
    island amity = { .name: "Amity", .opens: "09.00", .closes: "17:00", .next: NULL};
    island craggy = { .name: "Craggy", .opens: "09:00", .closes: "17:00", .next: NULL};
    island isla_nublar = { .name: "Isla Nublar", .opens: "09:00", .closes: "17:00", .next: NULL};
    island 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:



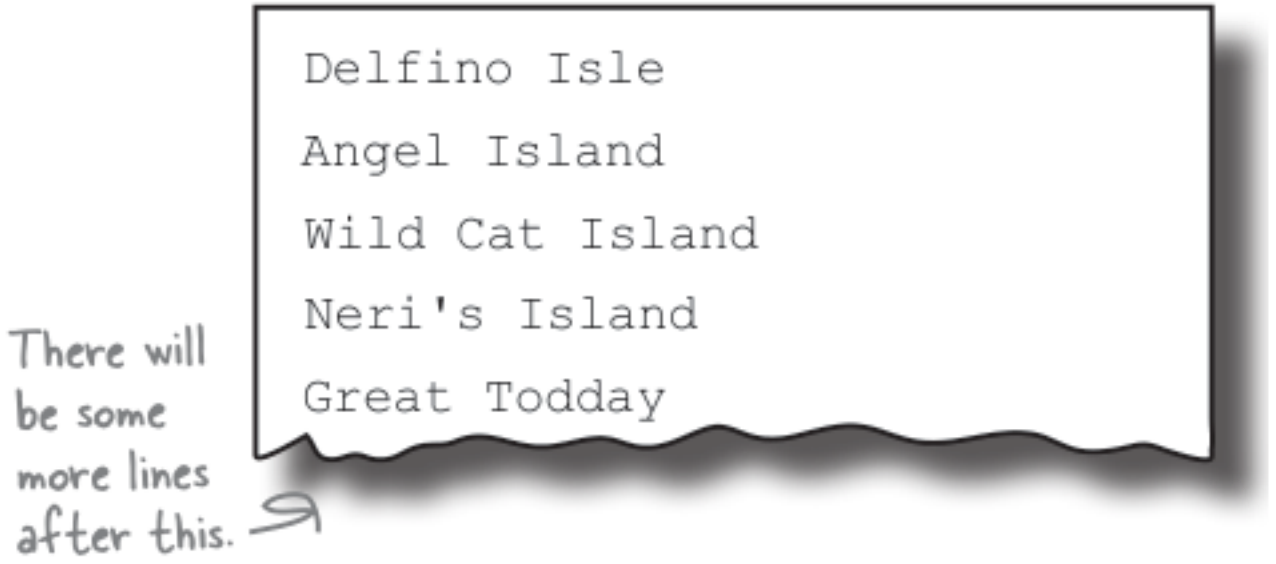
```
Name: Amity open: 09.00-17:00
Name: Craggy open: 09:00-17:00
Name: Isla Nublar open: 09:00-17:00
Name: Skull open: 09:00-17:00
Name: Shutter open: 09:00-17:00
```

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:

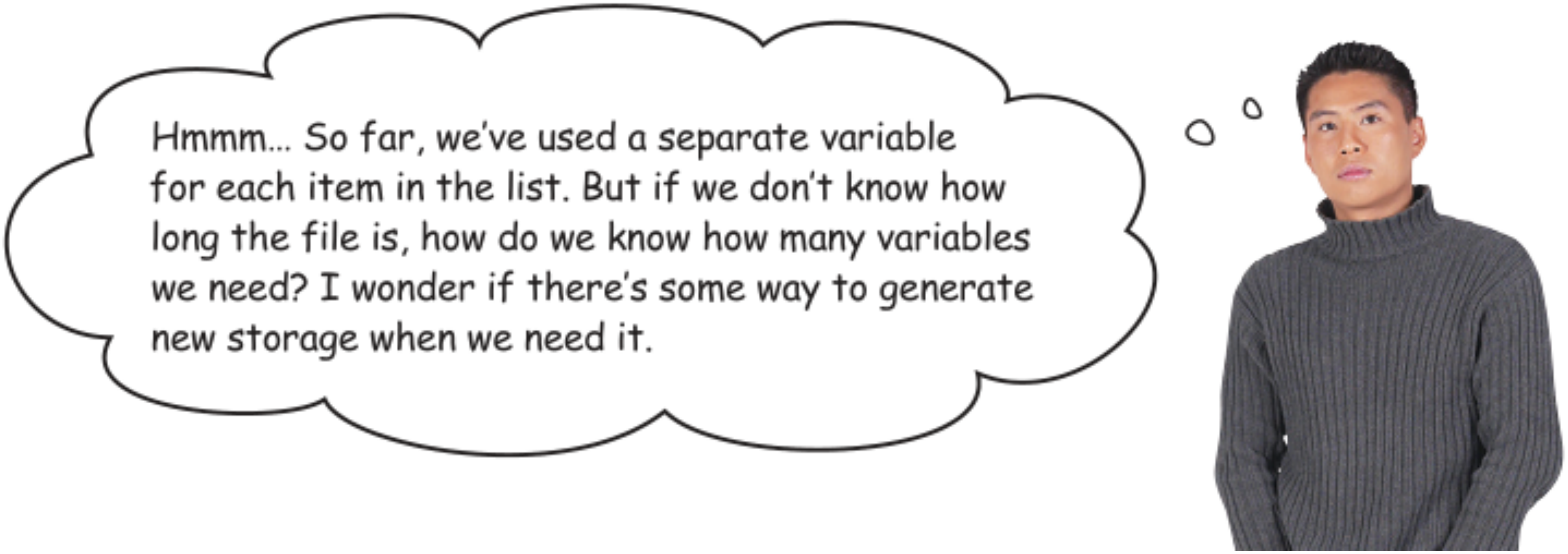


```
Delfino Isle
Angel Island
Wild Cat Island
Neri's Island
Great Todday
```

There will be some more lines after this.

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?


Hmmm... 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.

Hmmm… 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.

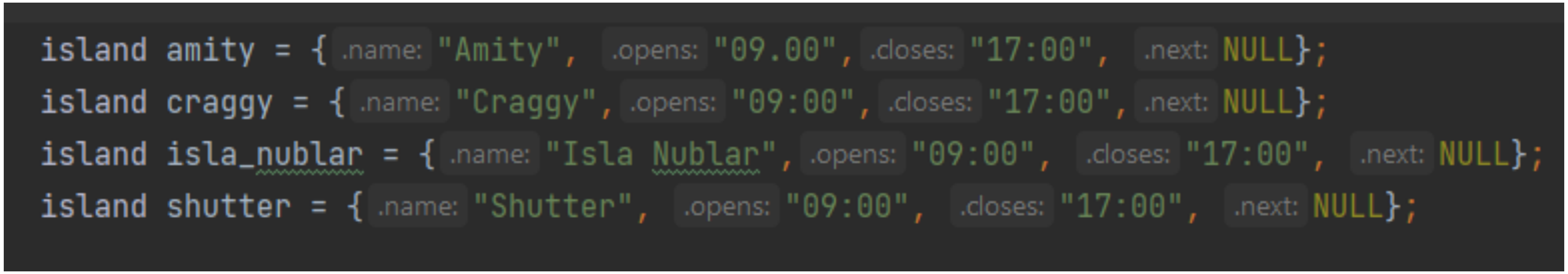
All of the programs you've written so far have used static storage.

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.

Wouldn't it be dreamy if there were a way to allocate as much space as I needed with code at runtime? But I know that's just a fantasy...

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().**

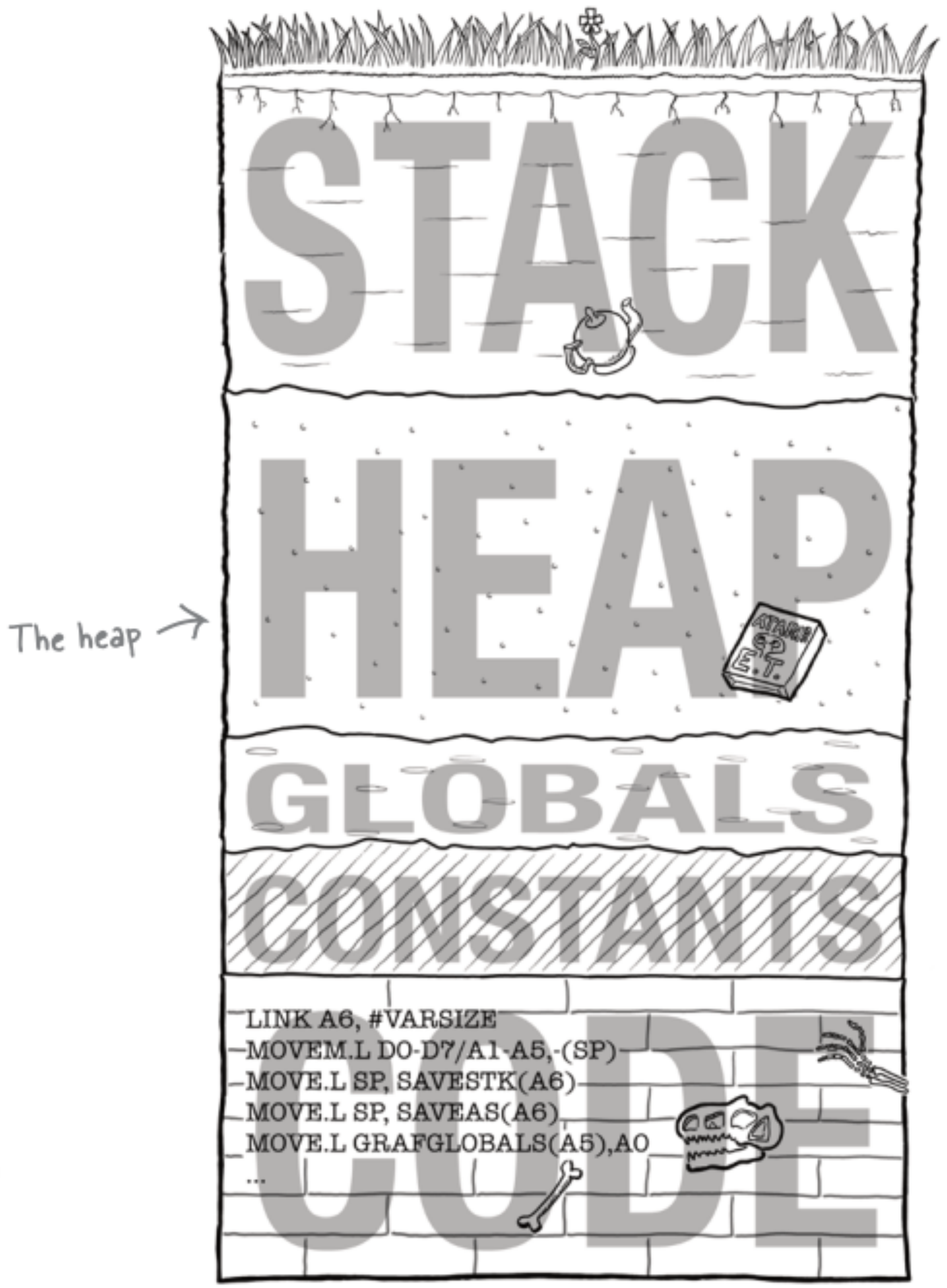
Heap storage is like saving valuables in a locker.

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.

The heap
32 bytes of data at location 4,204,853 on the heap
The malloc() function will give you a pointer to the space in the heap.

It allows you access to the memory, and it can also be used to keep track of the storage locker that’s been allocated.



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

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.

Thanks for the storage. I’m done with it now.
52 bytes of data at location 4,204,853 on the heap

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.

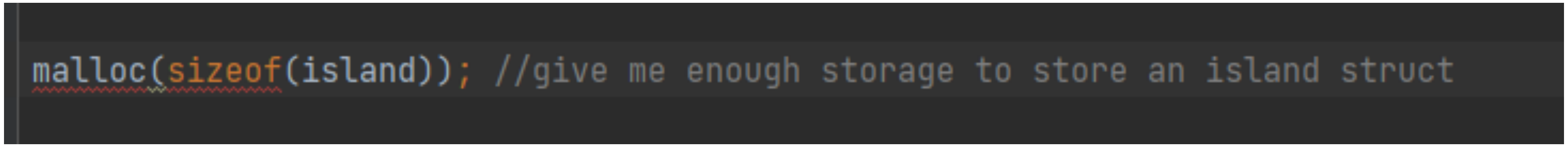
Let's see how **malloc()** and **free()** work.

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:



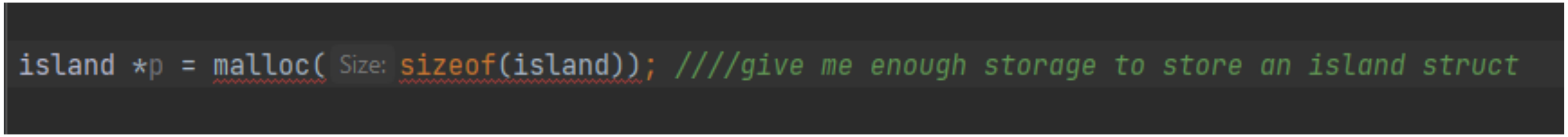
```
malloc(sizeof(island)); //give me enough storage to store an island struct
```

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."

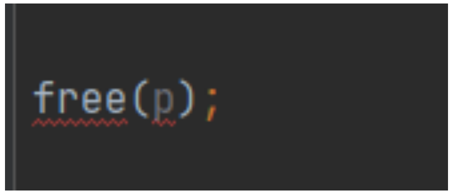
Once you've created the memory on the heap, you can use it for as long as you like.

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:



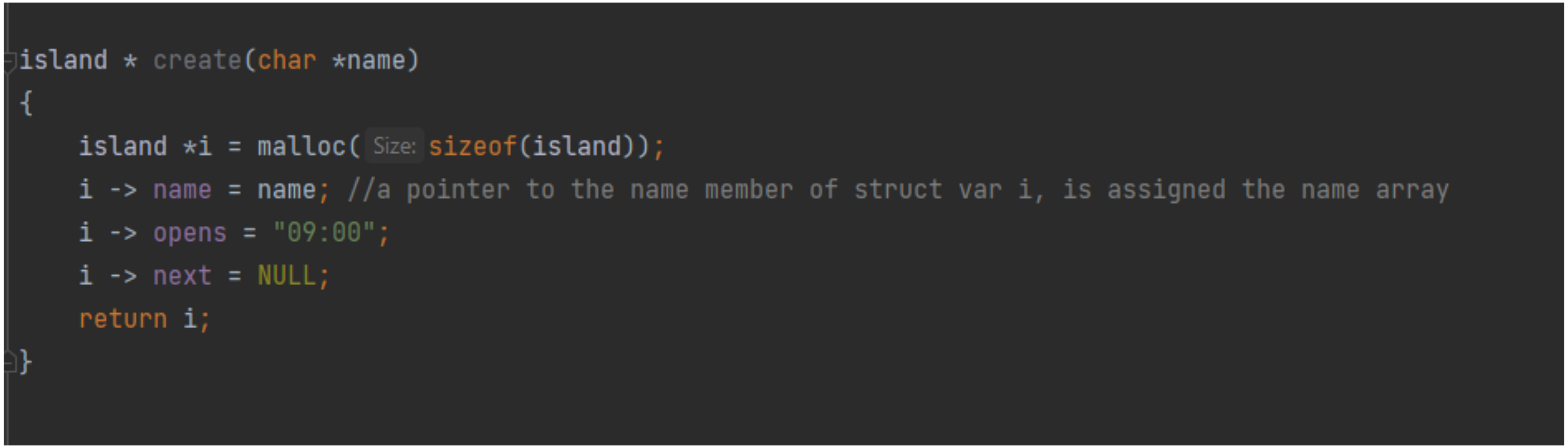
```
free(p);
```

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:



```c
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.

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;
}
```

This will create a new island struct on the heap.

It's using the malloc() function to create space on the heap.

These lines set the fields on the new struct.

The sizeof operator works out how many bytes are needed.

The function returns the address of the new struct.

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.

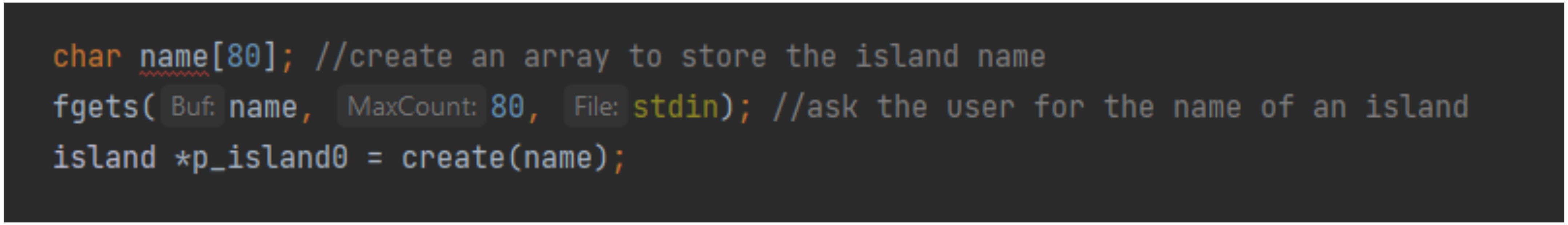
The function returns a pointer to the newly created struct.

Look carefully at the code for the create() function. Do you think there might be any problems with it?

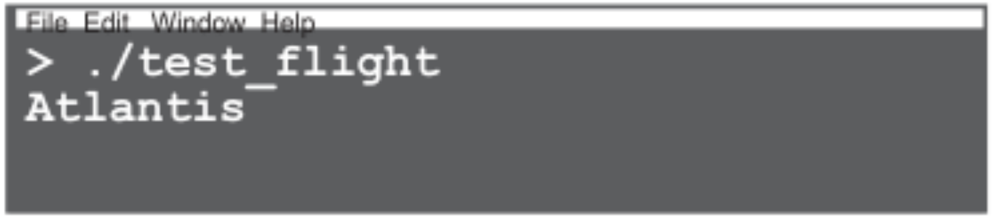
Once you've thought about it good and hard, turn the page to see it in action.

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.

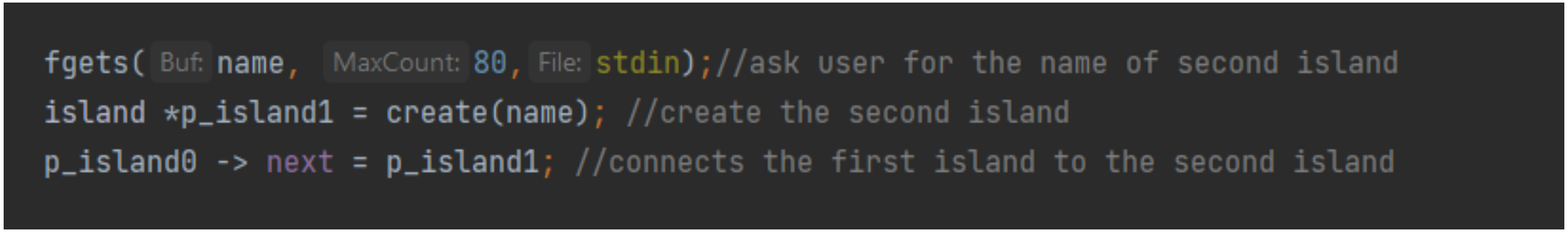


```
char name[80]; //create an array to store the island name
fgets( Buf: name,  MaxCount: 80,  File: stdin); //ask the user for the name of an island
island *p_island0 = create(name);
```

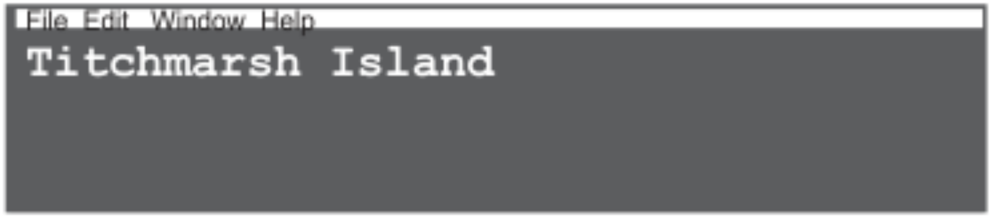


```
> ./test_flight
Atlantis
```

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
```



```
Titchmarsh Island
```

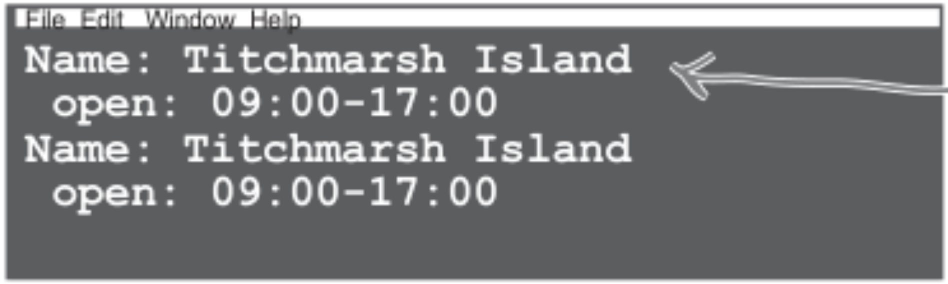
17:50 Back at headquarters tidying up on paperwork. Strange thing. The flight log produced by the test program appears to have a bug.

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. →

```
display(p_island0);
```

What happened to Atlantis???? →



```
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?

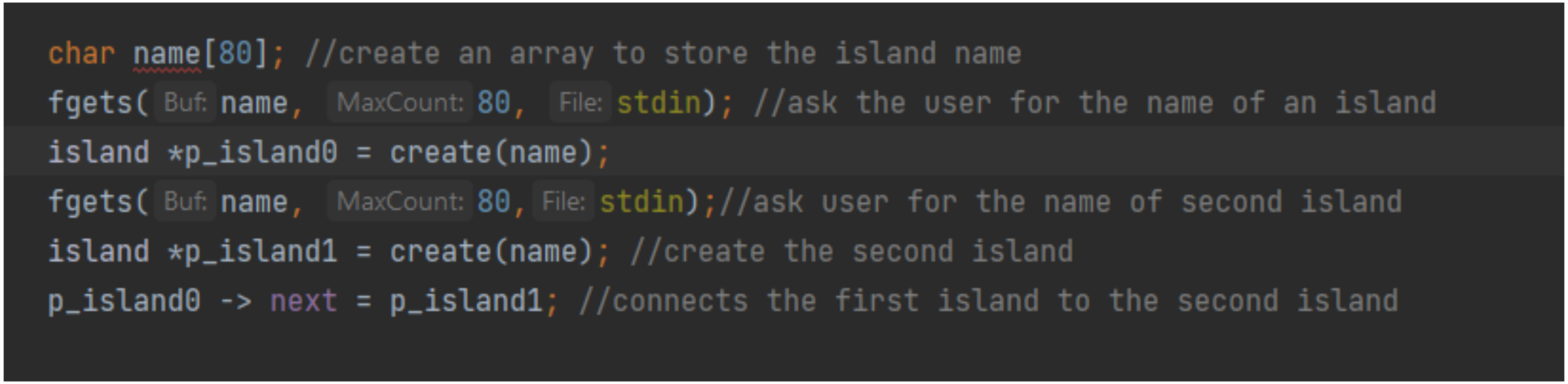
When the code records the name of the island, it doesn't take a copy of the whole name string; it just records the address where the name string lives in memory.

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 name 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
```

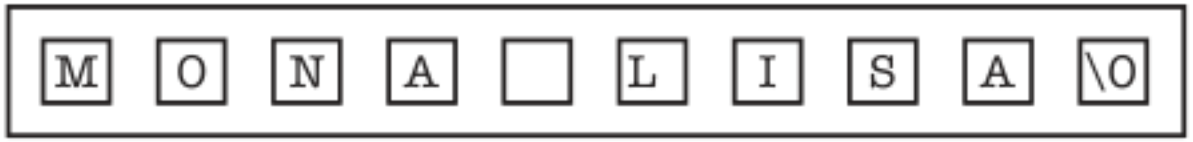
In C, you often need to make **copies of strings.**

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 **strdup().**

Let's say that you have a pointer to a character array that you want to copy:

```
char *s = "MONA LISA";
```



The strdup() function can reproduce a complete copy of the string somewhere on the heap:

The strdup() function can reproduce a complete copy of the string somewhere on the heap:

The strdup() 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.

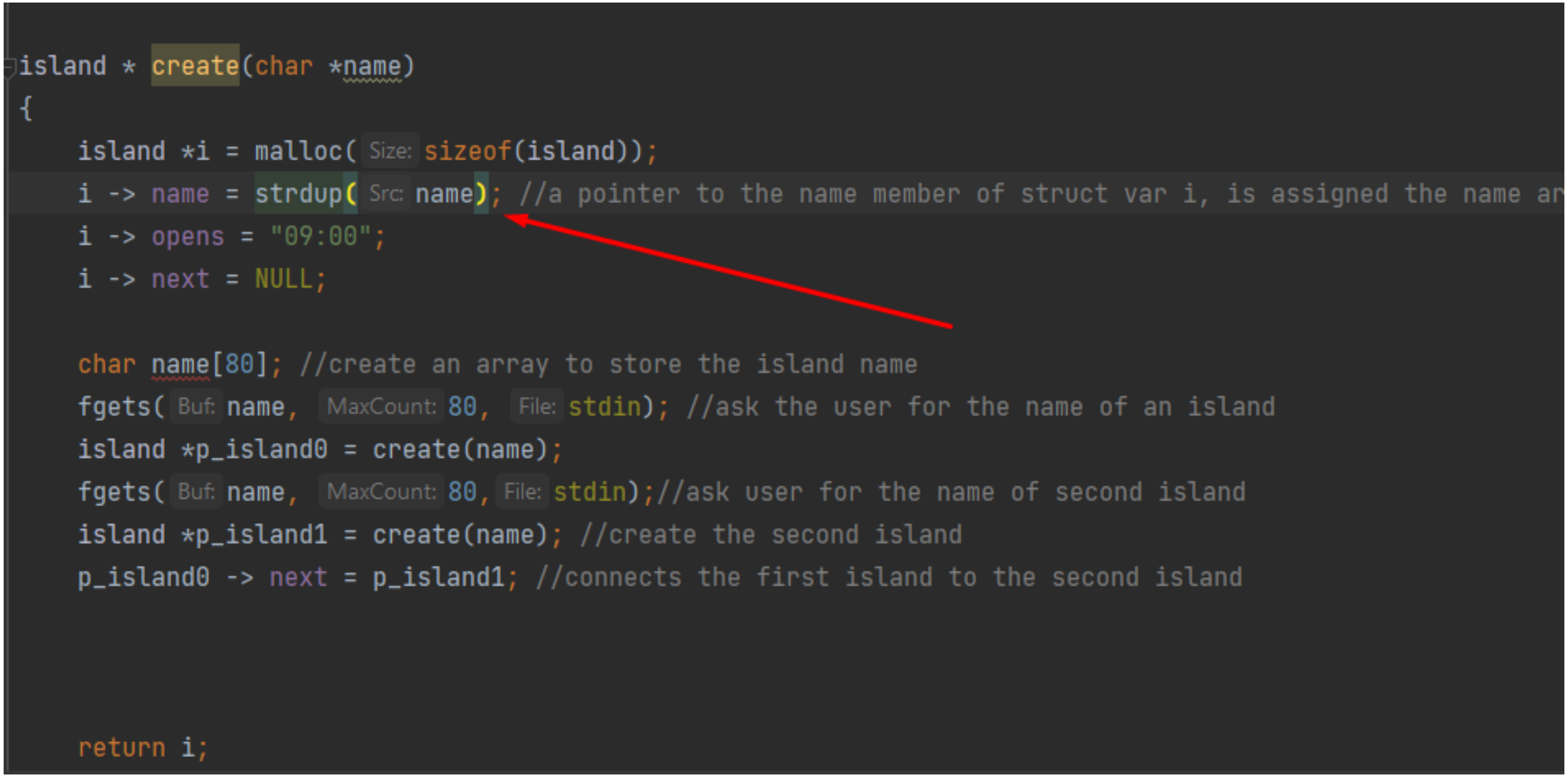
It then copies each of the characters to the new space on the heap.

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 ar
    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.

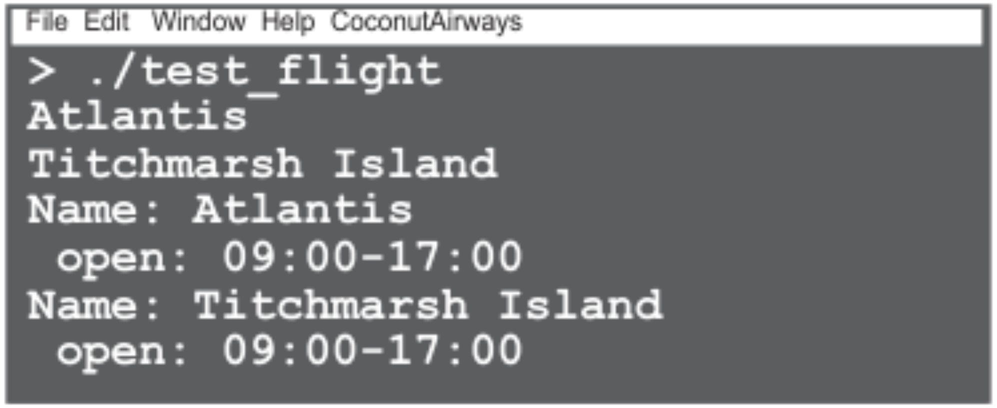
Remember way back when you saw where things were stored in memory?

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:



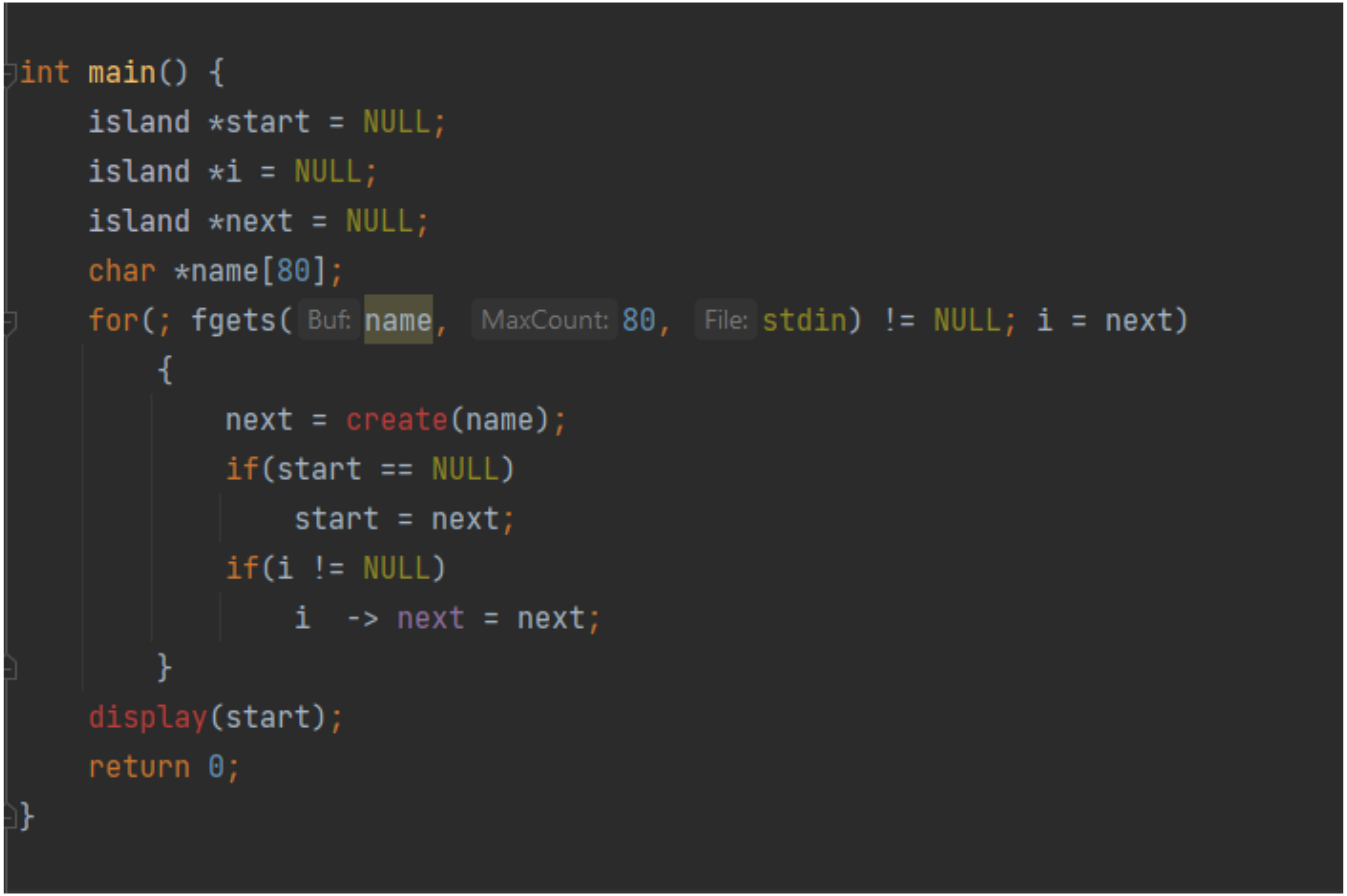
```
> ./test_flight
Atlantis
Titchmarsh Island
Name: Atlantis
 open: 09:00-17:00
Name: Titchmarsh Island
 open: 09:00-17:00
```

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.

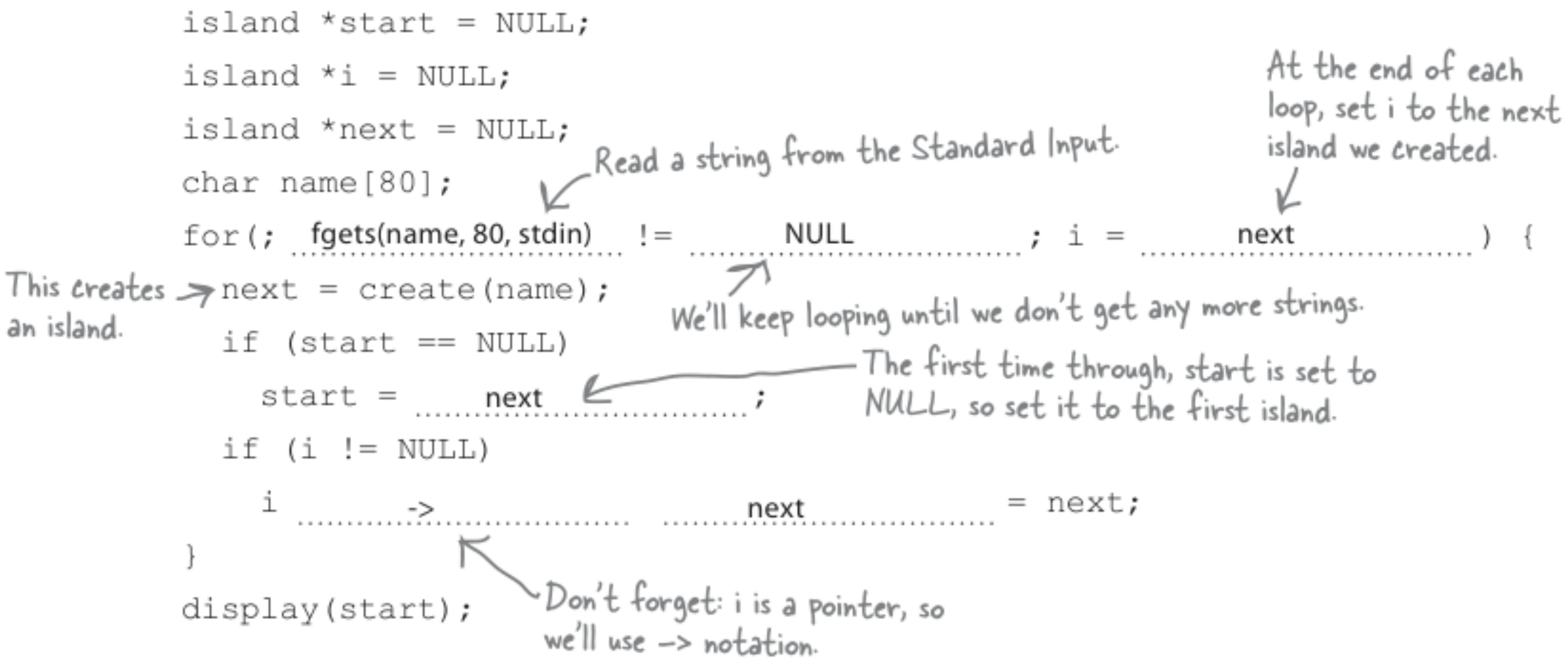
Catastrophe!

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.



```c
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:



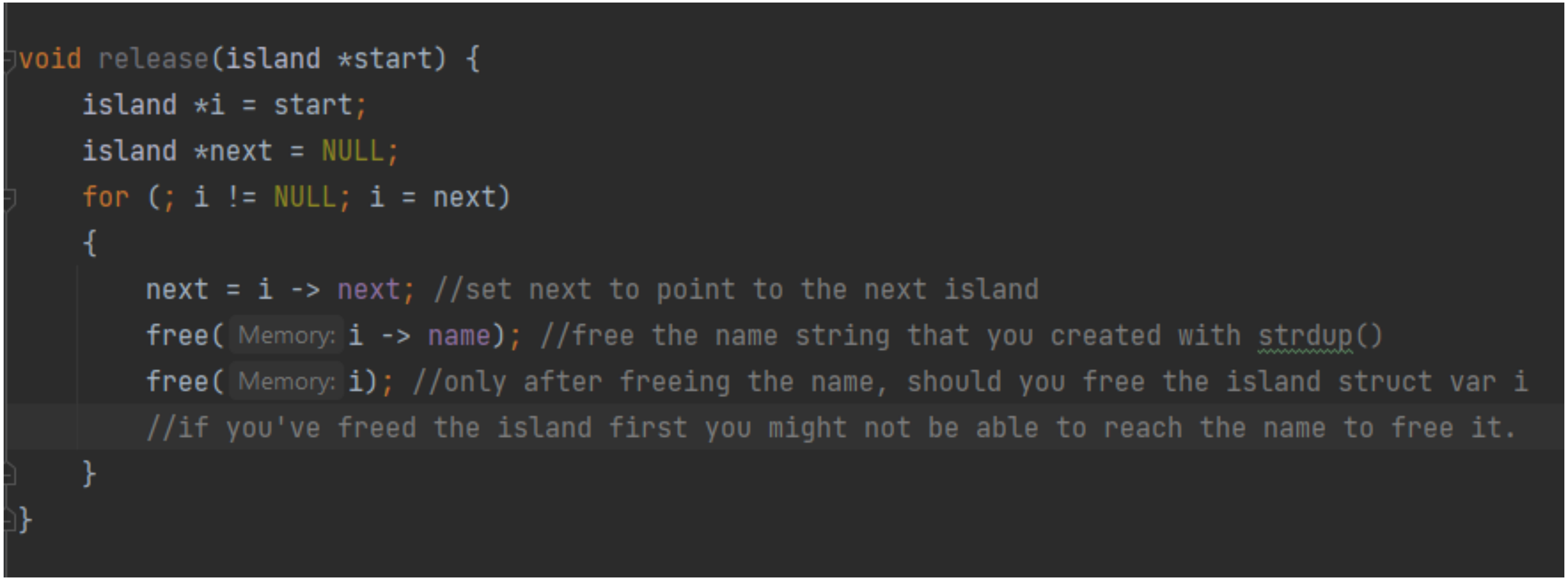
```
island *start = NULL;
island *i = NULL;
island *next = NULL;
char name[80];
for(; fgets(name, 80, stdin) != NULL ; i = next ) {
  next = create(name);
  if (start == NULL)
    start = next ;
  if (i != NULL)
    i -> next = next;
}
display(start);
```

Read a string from the Standard Input.
At the end of each loop, set i to the next island we created.
This creates an island.
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.
    }
}
```

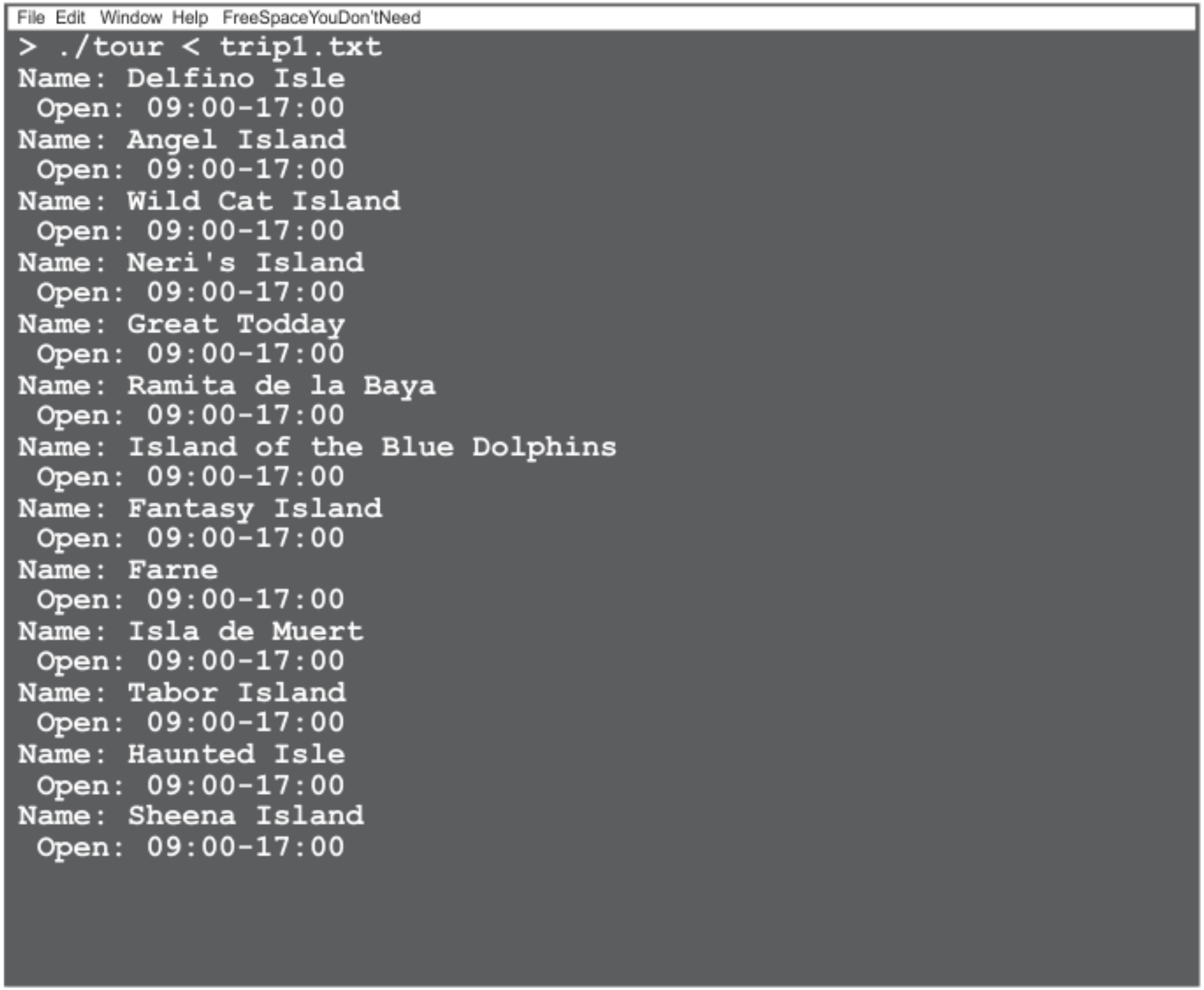
Free the memory when you're done Now that you have a function to free the linked list, you'll need to call it when you've finished with it.

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.



```
> ./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:**
Heap? Are you there? I’m home.	
	Don’t see you too often this time of day. Got a little something going on?
Deep regression. Oops…excuse me… Just tidy that up…	
	What’re you doing?
The code just exited a function. Just need to free up the storage from those local variables.	
	You should take life a little easier. Relax a little…
Perhaps you’re right. Mind if I sit?	
	Beer? Don’t worry about the cap; throw it anywhere.
I…think this is yours?	
	Hey, you found the pizza! That’s great. I’ve been looking for that all week.
You really should consider getting somebody in to take care of this place.	
	Don’t worry about it. That online ordering application left it lying around. It’ll probably be back for it.
How do you know? I mean, how do you know it hasn’t just forgotten about it?	

Hmmm? 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.

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.

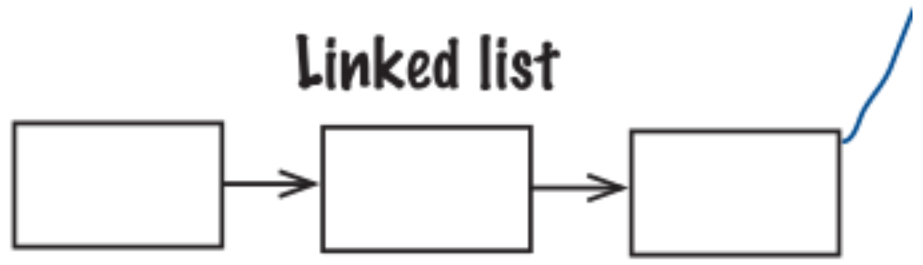
• Dynamic data structures are normally defined in C with recursive structs.

• A **recursive struct** contains one or more pointers to a **similar struct**.

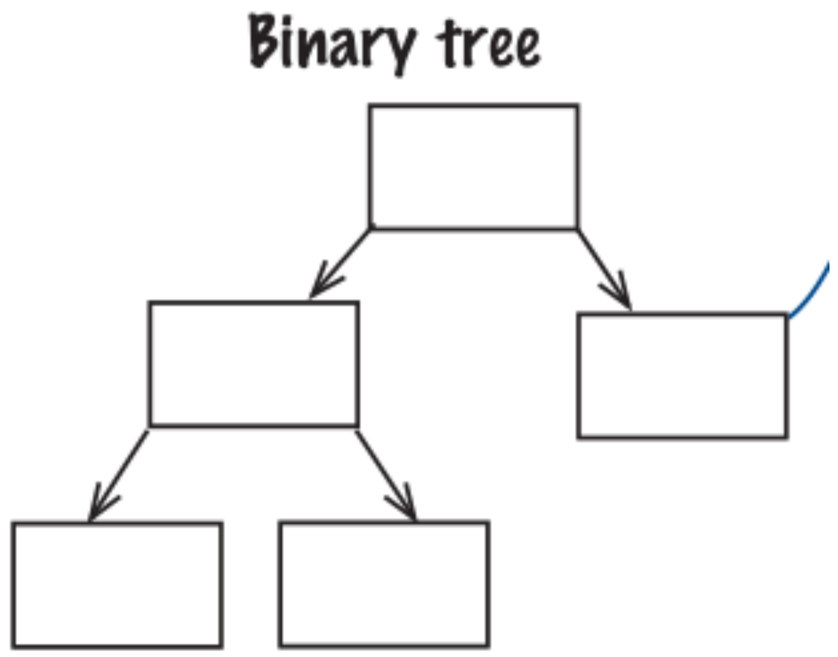
- The stack is used for local variables and is managed by the computer.
- The heap is used for long-term storage. You allocate space with malloc().
- The sizeof operator will tell you how much space a struct needs.
- Data will stay on the heap until you release it with free().

WHAT'S MY DATA STRUCTURES

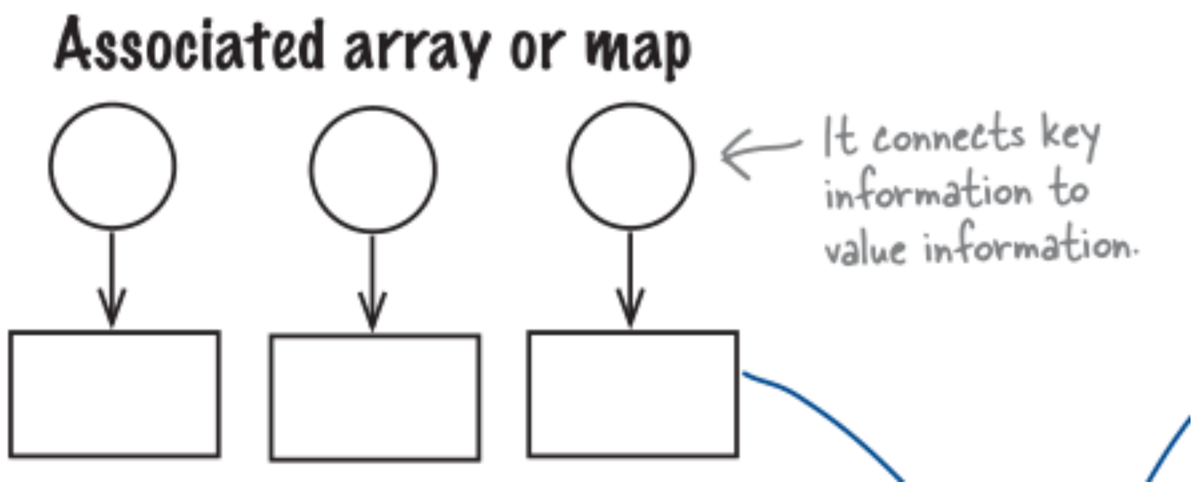
I can be used to store a sequence of items, and I make it easy to insert new items. But you can process me in only one direction.



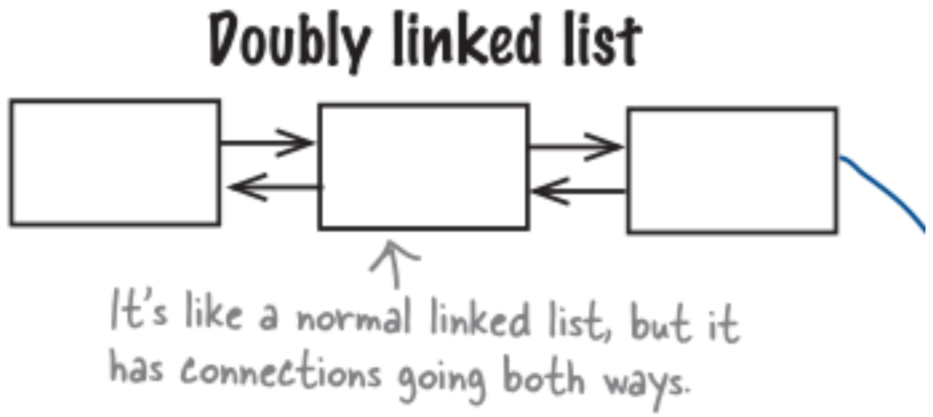
Each item I store can connect to up to two other items. I am useful for storing hierarchical information.



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.



Each item I store connects to up to two other items. You can process me in two directions.


Doubly linked list
It's like a normal linked list, but it has connections going both ways.

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.

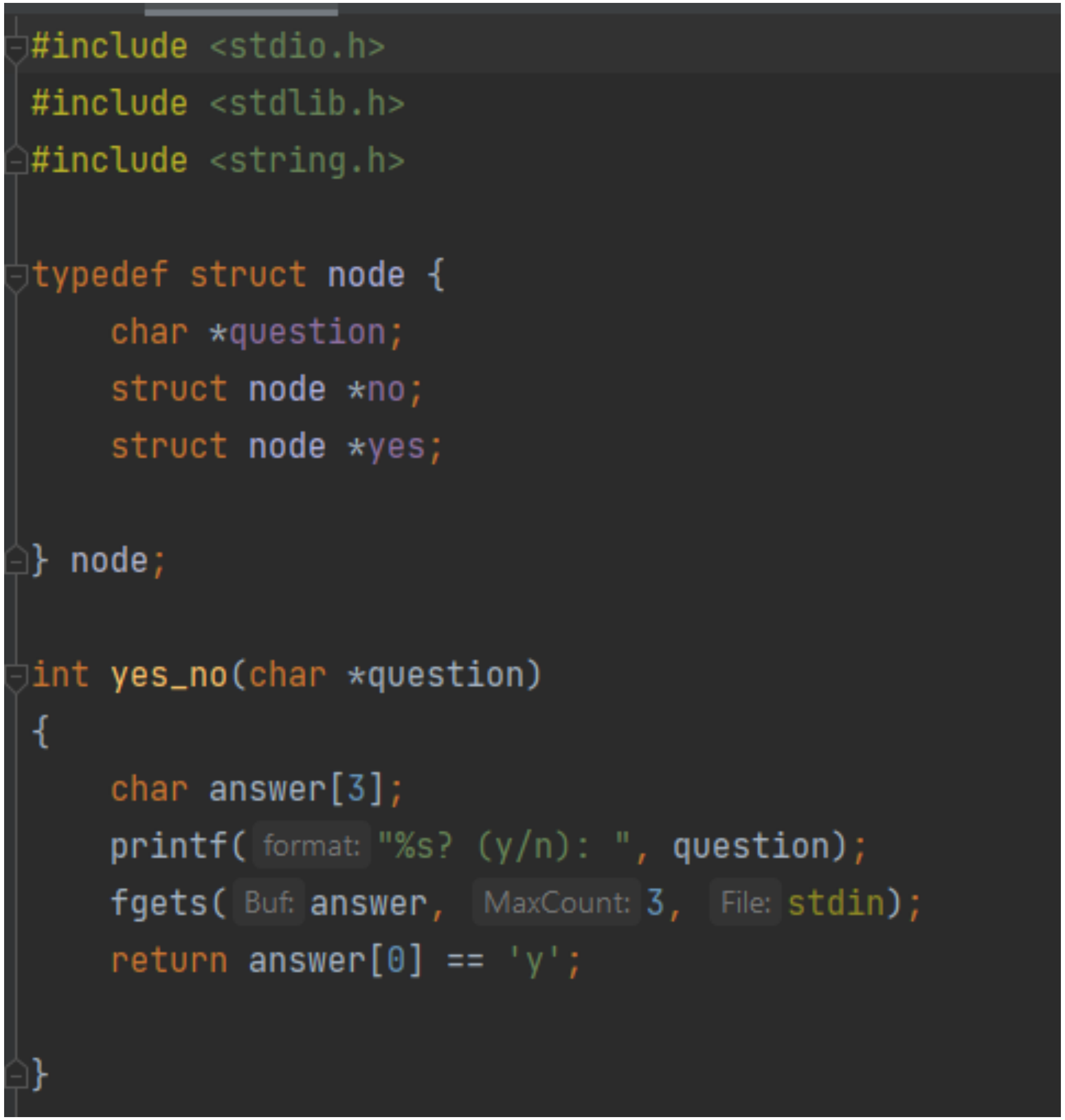
That means it’s really important that you learn to track down and fix memory leaks in your code.

FBI LEAK SOURCE CODE

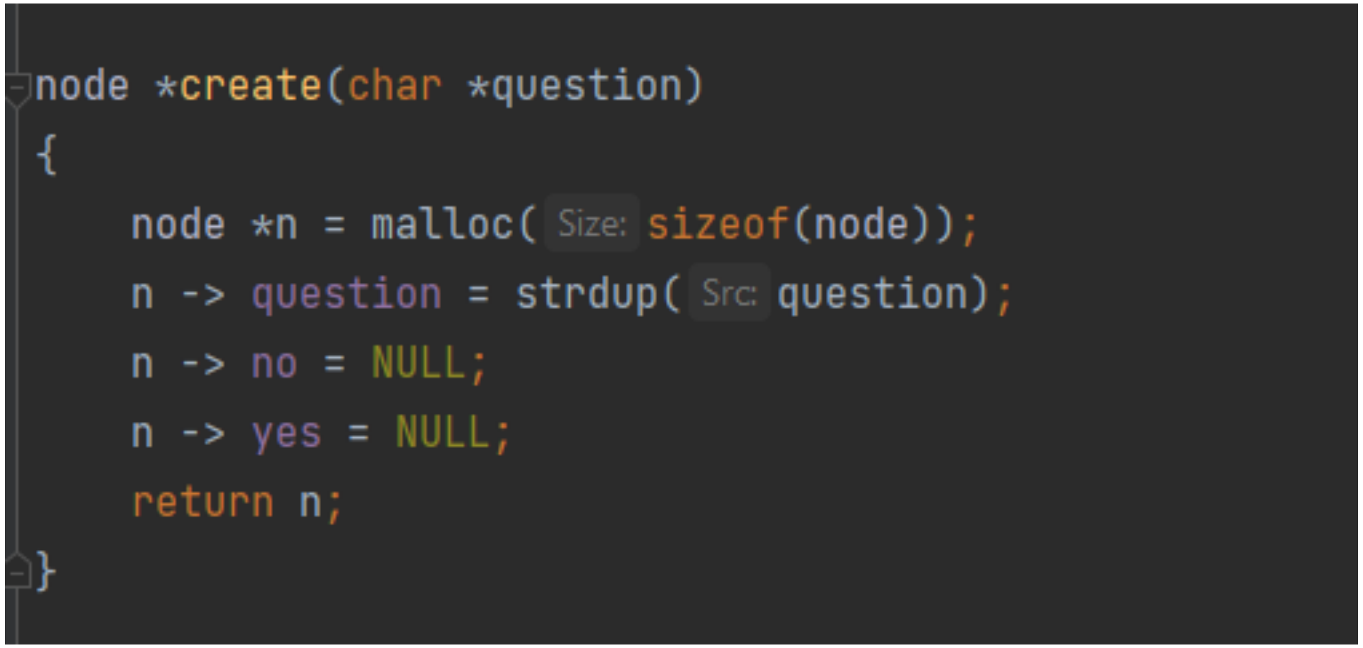
Exhibit A: the source code What follows is the source code for the Suspicious Persons Identification Expert System (SPIES).

This software can be used to record and identify persons of interest.

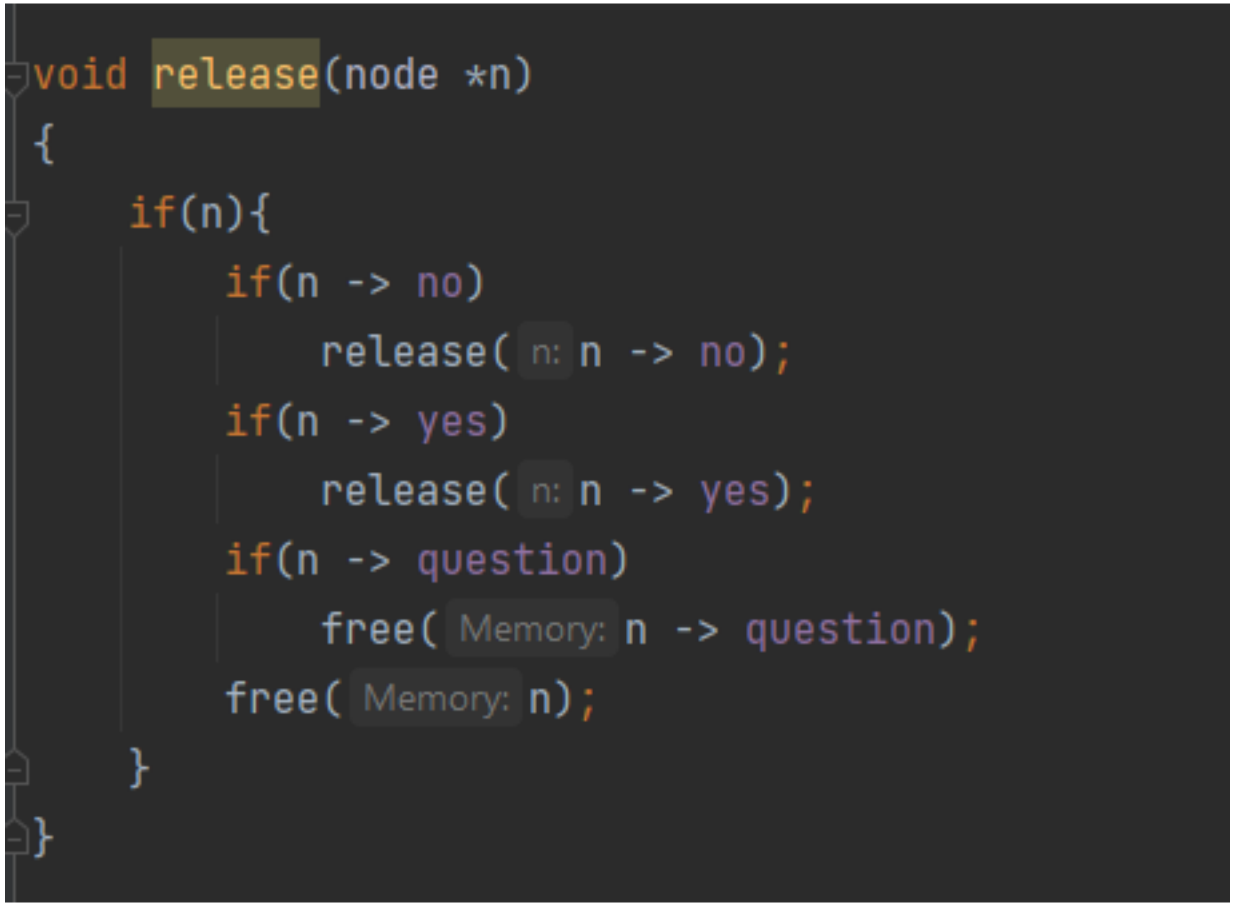
You are not required to read this code in detail now, but please keep a copy in your records so that you may refer to it during the ongoing investigation.



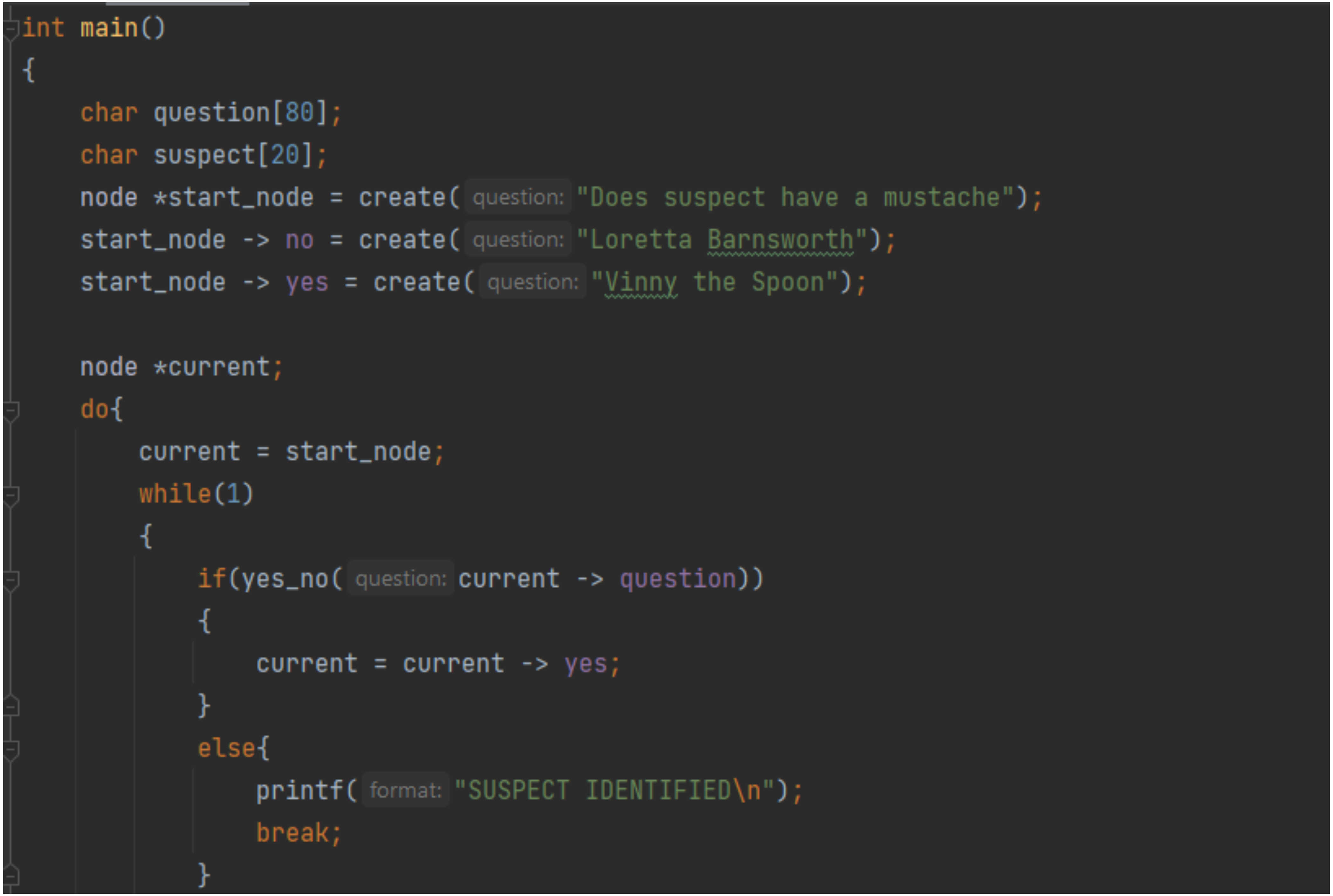
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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';

}
```



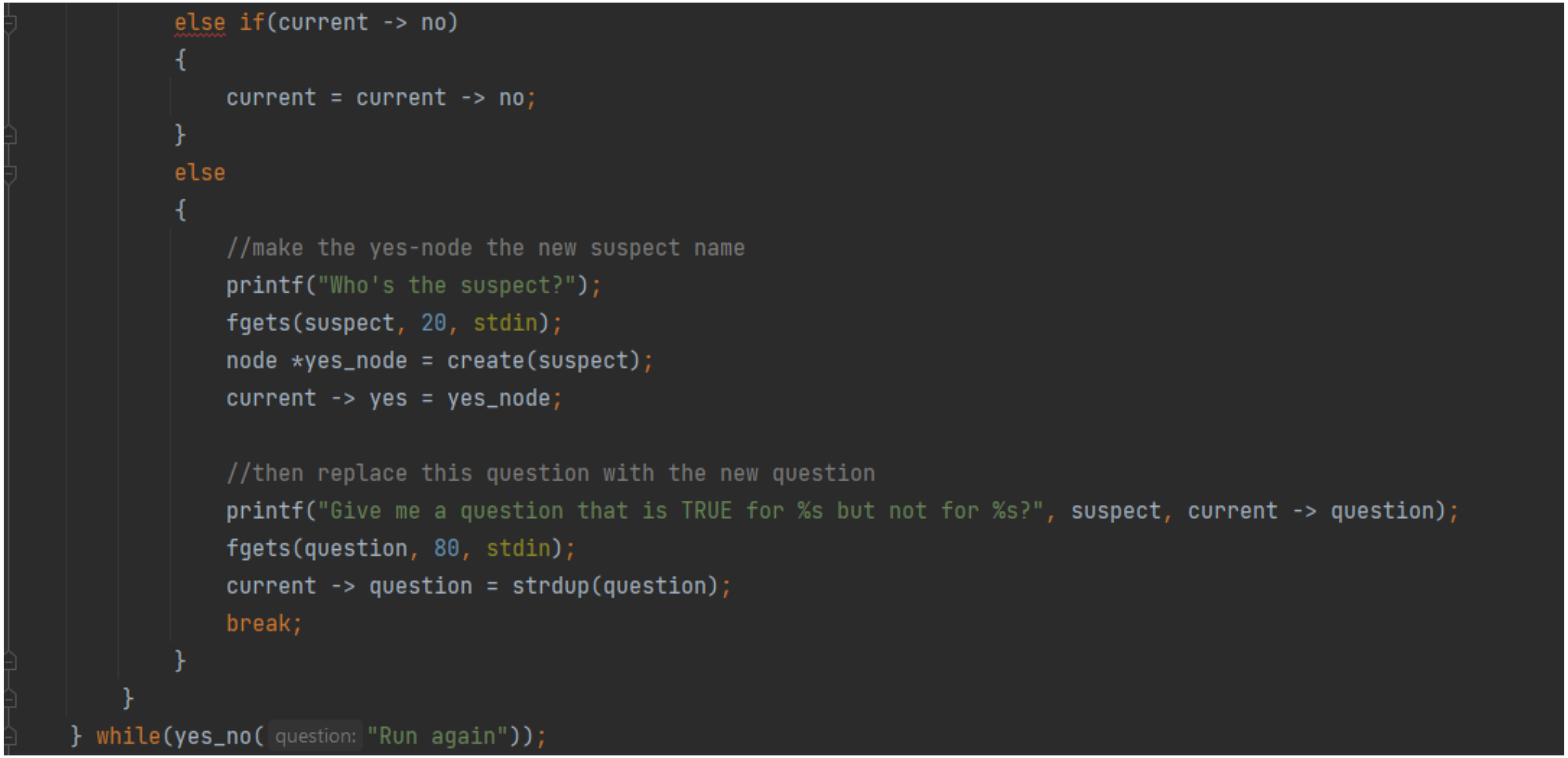
```
node *create(char *question)
{
    node *n = malloc( Size: sizeof(node));
    n -> question = strdup( Src: question);
    n -> no = NULL;
    n -> yes = NULL;
    return n;
}
```



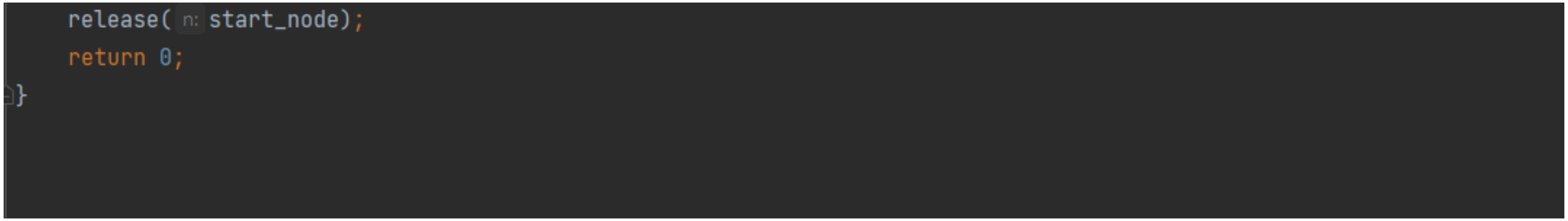
```
void release(node *n)
{
    if(n){
        if(n -> no)
            release( n: n -> no);
        if(n -> yes)
            release( n: n -> yes);
        if(n -> question)
            free( Memory: n -> question);
        free( Memory: n);
    }
}
```



```
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;
            }
```



```c
            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"));
```



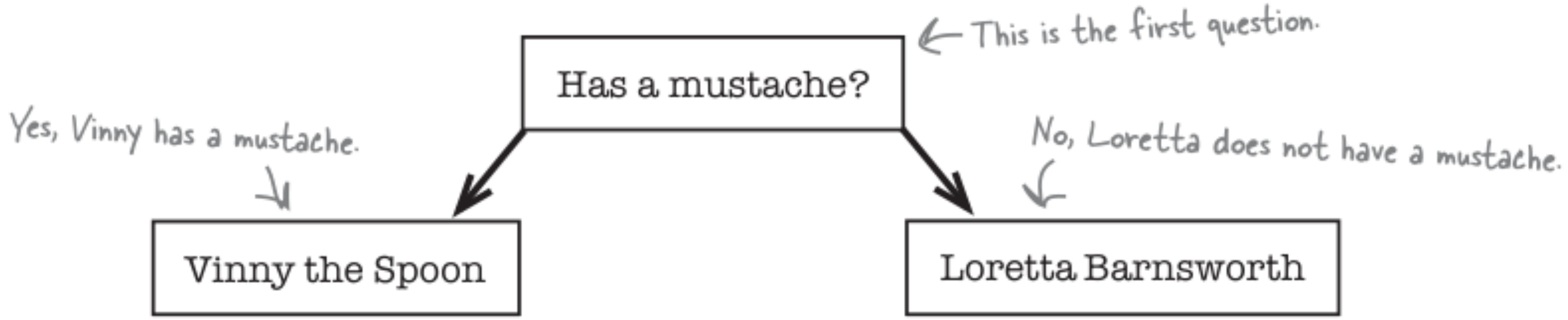
```c
    release( n: start_node);
    return 0;
}
```

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 the first question.
Has a mustache?
Yes, Vinny has a mustache.
No, Loretta does not have a mustache.
Vinny the Spoon
Loretta Barnsworth

This is what the data looks like when the program starts.

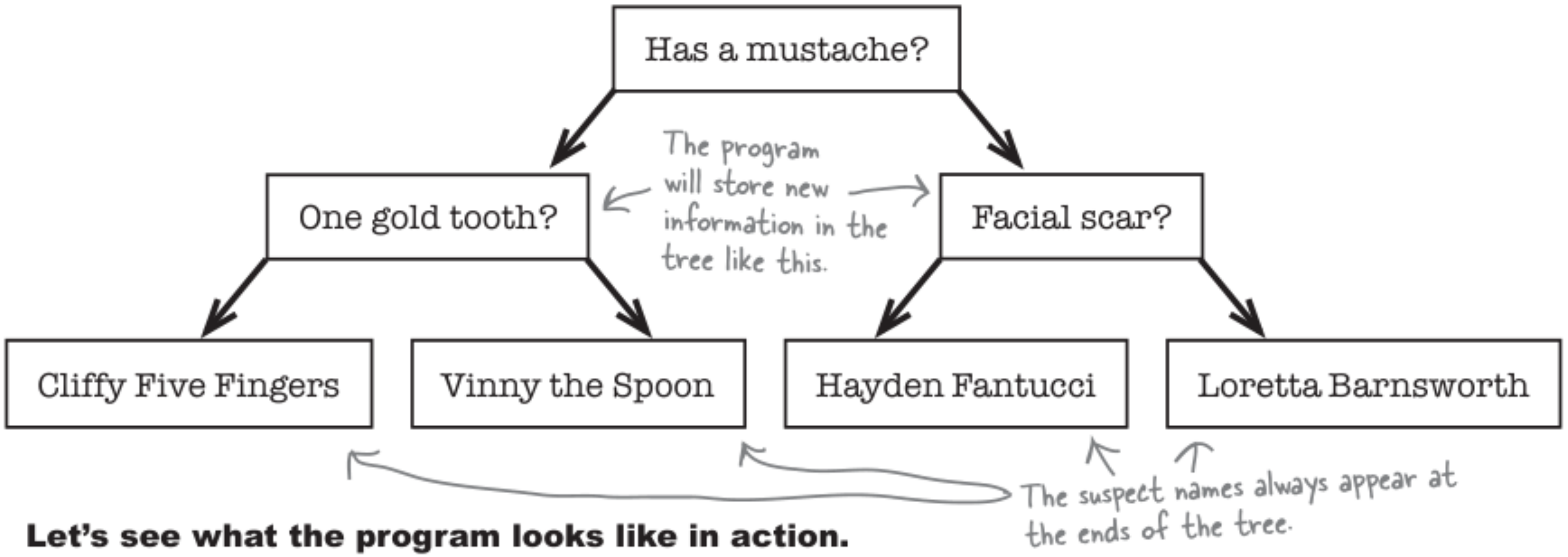
The first item (or node) in the tree stores a question: “Does the suspect have a mustache?”

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.

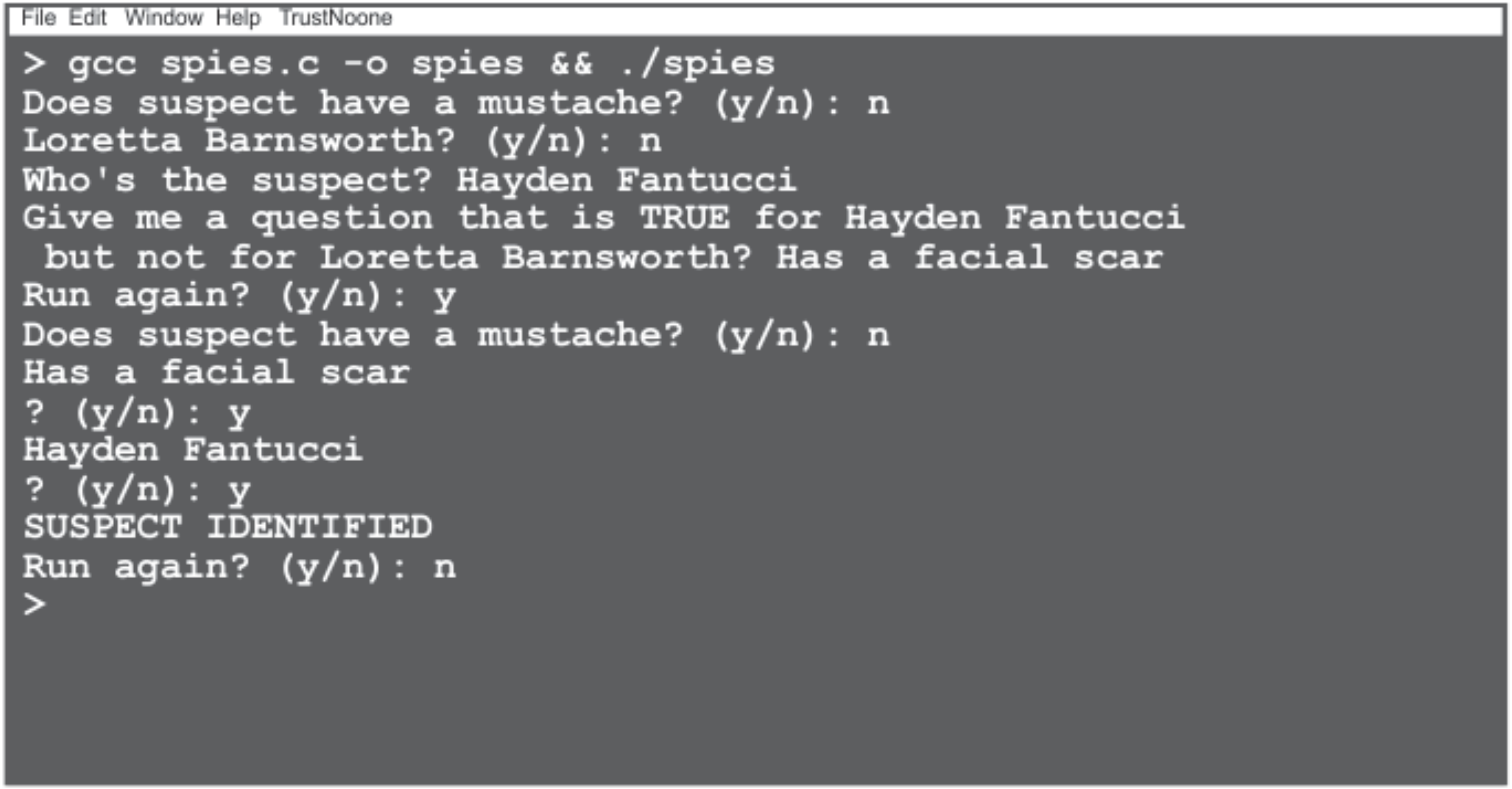
The program will use this tree to ask the user a series of questions to identify a suspect.

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.


Has a mustache?
One gold tooth?
Facial scar?
The program will store new information in the tree like this.
Cliffy Five Fingers
Vinny the Spoon
Hayden Fantucci
Loretta Barnsworth
The suspect names always appear at the ends of the tree.

Let's see what the program looks like in action.



```
> 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.

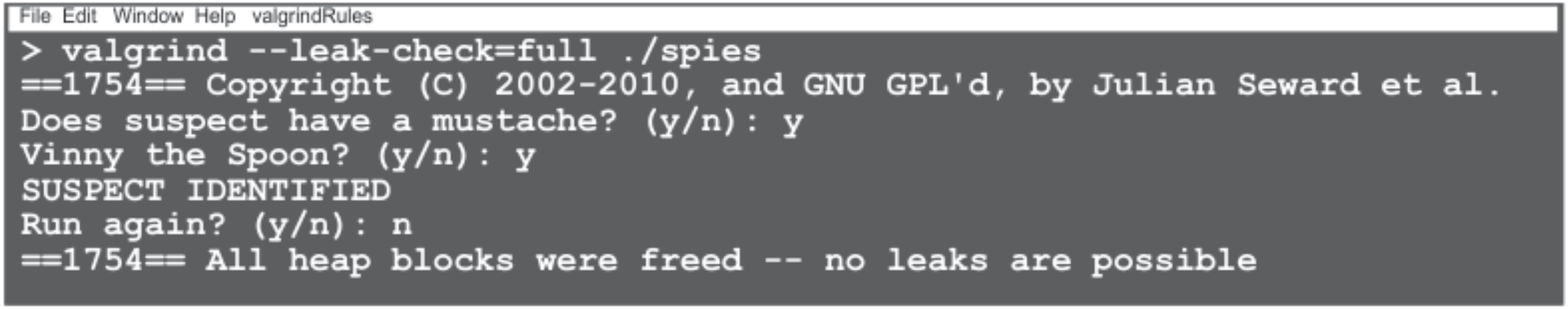
The valgrind version of malloc() will take note of which piece of code is calling it and which piece of memory it allocated.

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.

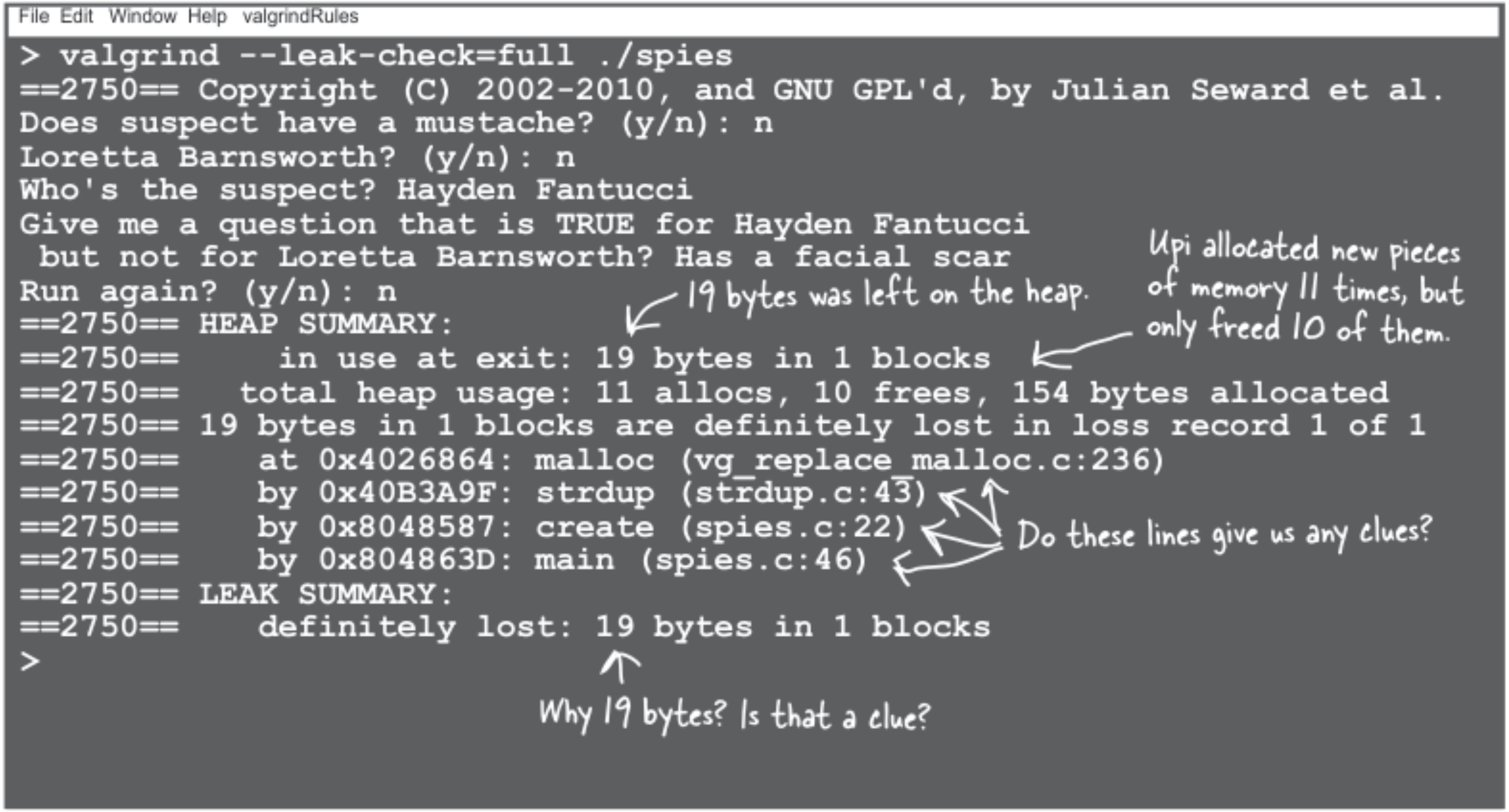
You'll start valgrind on the command line with the **--leak-check=full** option and then pass it the program you want to run:



```
> 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?



```
> 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
>
```

19 bytes was left on the heap.
Upi allocated new pieces of memory 11 times, but only freed 10 of them.
Do these lines give us any clues?
Why 19 bytes? Is that a clue?

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.

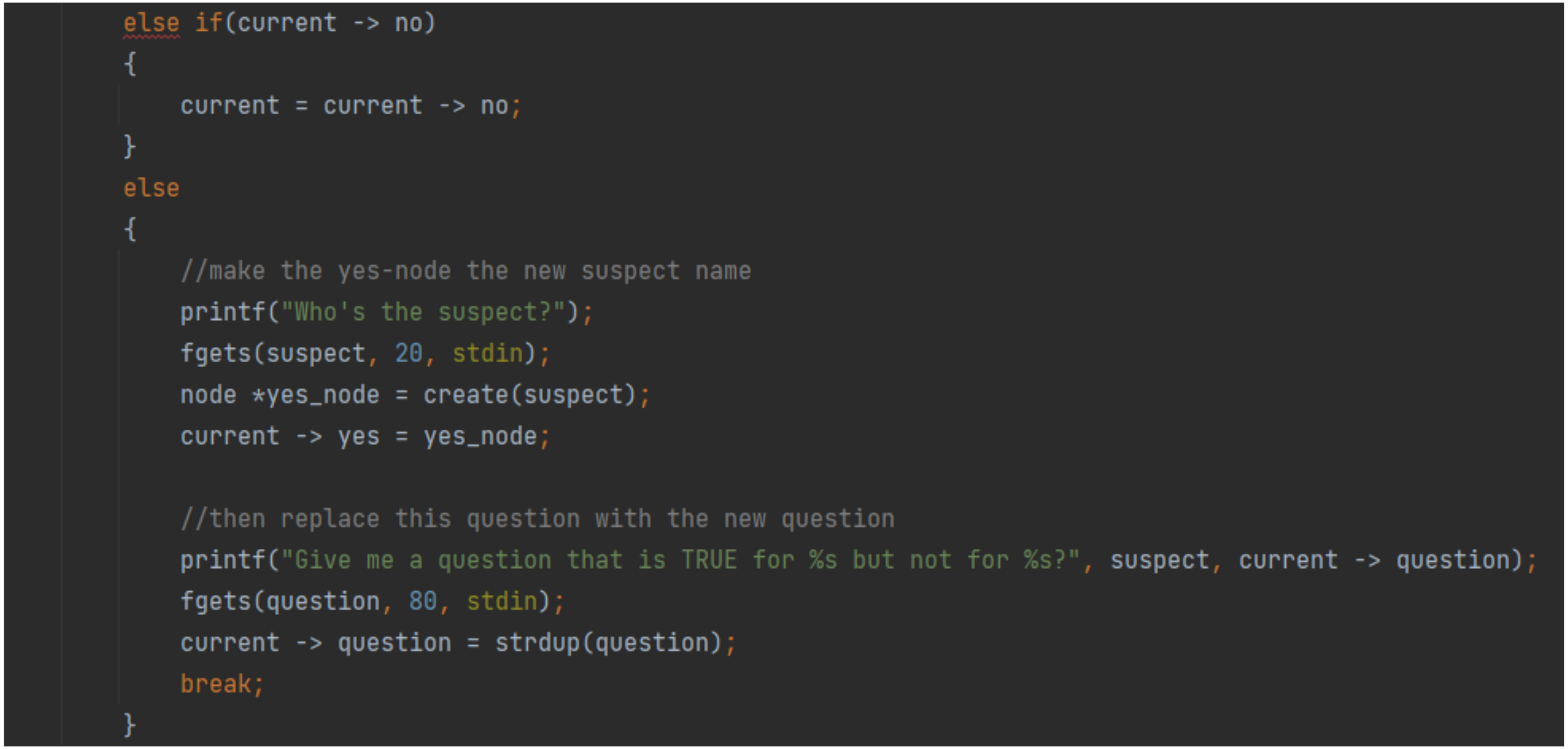
OK, now that you’ve run valgrind, you’ve collected quite a few pieces of evidence. It’s time to analyze that evidence and see if you can draw any conclusions.

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:



```c
        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);
```

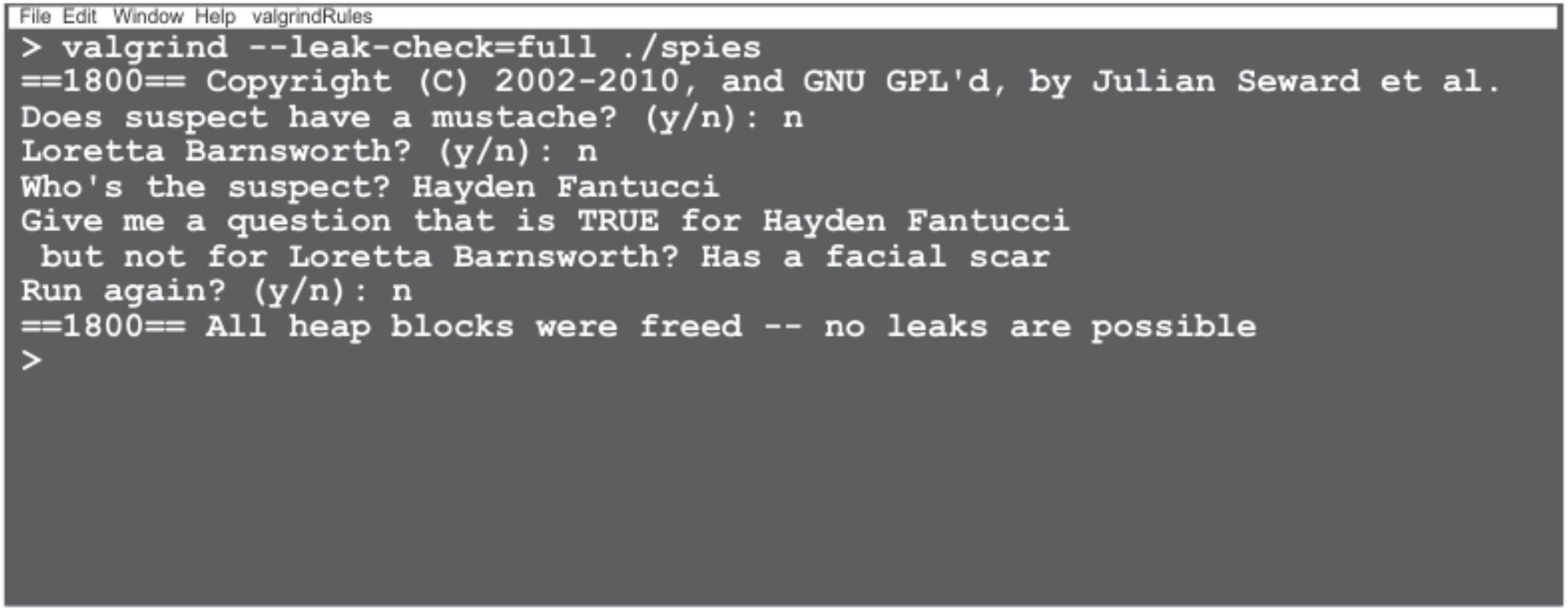
- How do you plug the leak?

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.



```
> 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.

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.

Unlike the stack, heap memory is not automatically released.

The stack is used for local variables.

- 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.