C Interview Questions

Get ready for your C programming interview

Don't get blindsided in your C coding interview. String manipulation, null pointers, static, and diagnosing inconsistent buggy behavior...it's a lot. Try these practice questions and see how ready you are for the interview.

Icing on the Cake

Our freshly baked cake just came out of the oven, and while the cake is cooling, it's time to make the frosting. We've written some C code to help us figure out how much we'll need:

```c
#define PI 3.14
#define AREA(r) PI * r * r

float showIcingArea()
{
    // initial guess at cake size, in feet
    float radius = 4.0;  // that's a big cake!
    printf("Initial cake radius: %.2f Area: %.2f\n", radius, AREA(radius));

    // make a little extra ... just to be sure
    printf("Slightly bigger cake radius: %.2f Area: %.2f\n", radius + 1, AREA(radius + 1));

    return 0;
}
```

But when we run this code, we see:

```c
Initial cake radius: 4.00 Area 50.24
Slightly bigger cake radius: 5.00 Area 17.56
```

That's unexpected... what's going on here?

Answer

Our issue is with the macro expansion for `AREA()`. When the C preprocessor expands macros, it drops the arguments *directly* into the macro definition. The first time, this is fine:

```c
PI * radius * radius
```

But the second time, we get:

```c
PI * radius + 1 * radius + 1
```

And since multiplication has higher precedence than addition, we actually calculated:

```c
(PI * radius) + (1 * radius) + 1
```

Even though we meant to calculate:

```c
PI * (radius + 1) * (radius + 1)
```

The most surefire way to make sure each argument sticks together when the macro is expanded is to wrap it in parentheses.

So, let's add them to our macro. Like this:

```c
#define AREA(r) ((PI) * (r) * (r))
```

In addition to wrapping *each argument* in parentheses, we've also wrapped the *entire macro definition*. This ensures our definition sticks together, regardless of other operations *around it*.

> Okay, but do we *really* need parentheses around `PI`?
>
> In this code, we *could* get away with skipping them since we can see `PI` is a simple constant. But it's a good practice because now even if we change how or where we define `PI`, we don't have to worry about *that* macro.
>
> For example another definition, correct to 3 decimal places, is:

```
#include <math.h>
#define PI sqrtf(2.0) + sqrtf(3.0)  // woops! forget parentheses
```

C

What happens if we call our macro like this: `AREA(radius++)`?

Our macro expands to:

```
((PI) * (radius++) * (radius++))
```

C

When run, the increment operation will happen twice, leading to unexpected values for `radius` and area:

```
float radius = 4.0;
float a = AREA(radius++);
// expands to:  ((PI) * (4.0) * (5.0))

printf("Slightly bigger cake radius: %.2f Area: %.2f\n", radius, a);
// prints "Slightly bigger cake radius: 6.00 Area: 62.80"
```

C

One common strategy to sidestep this headache is to avoid macros entirely and use `inline` functions instead:

```
inline float area(float radius)
{
    return (PI) * radius * radius;
}
```

C

Language Transfer

Write the body of a function `strReplace(src, old, new)` that returns a copy of `src` but with the first occurrence of `old` replaced with `new`. For example:

```
char *dst = strReplace("My favorite language is Python", "Python", "C");
```

C

should result in `dst` pointing to a new string that contains `"My favorite language is C"`.

Answer

String manipulation is notoriously tricky in C, so we'll want to build up an answer piece by piece. The question says we should return a copy of `src`, so we'll need to allocate memory.

Even just making sure we have the right amount of space can be tricky. Does `strlen()` include the `NUL` terminator in its count?

Let's check the man pages:

```
$ man strlen
...
RETURN VALUES
    The strlen() function returns the number of characters the precede the
    terminating NUL character.
```

So we need to add 1 to make sure we can `NUL`-terminate `dst`:

C

```c
char *dst = malloc(sizeof(char) * (strlen(src) + 1));  // +1 for NUL terminator
```

One way to get our desired output is to copy all of `src` into `dst` and then overwrite `old` with `new`. Instead of rolling this code by hand ourselves, there are some useful string functions: `strcpy()` for copying strings over and `strstr()` for finding a substring in a string. Let's add those in:

```c
char * strReplace(const char *src, const char *old, const char *new)
{
    char *oldInDst = NULL;

    // allocate space for the new string
    char *dst = malloc(sizeof(char) * (strlen(src) + 1));  // +1 for NUL terminator
    assert(dst != NULL);

    // copy all of src into dst
    strcpy(dst, src);

    // find old inside dst
    oldInDst = strstr(dst, old);

    // overwrite old with new inside dst
    strcpy(oldInDst, new);

    return dst;
}
```

If we run this on our example we get what we wanted:

```c
char *dst = strReplace("My favorite language is Python", "Python", "C");
printf("%s\n", dst);  // prints "My favorite language is C"
```

Awesome. So, we're done?

Let's try another example:

```c
char *dst = strReplace("Python strings are easy to use", "Python", "C");
```

If we run this function and try to print what's in `dst` we get:

```c
"C"
```

All we got was `"C"` instead of `"C's strings are easy to use"`. What happened?

Let's take another look at the man pages:

```
$ man strcpy
...
DESCRIPTION
    The stpcpy() and strcpy() functions copy the string src to dst (including
    the terminating `\0` character.)
```

When we copied `new` into `dst`, it copied the `NUL` terminator which ended our string early.

How can we get around this? If we keep reading the man pages:

```
$ man strcpy
...
    The stpncpy() and strncpy() functions copy at most len characters from
    src into dst. [If src is more than len characters long] dst is not terminated.
```

We can use this to make sure we don't copy the `NUL` terminator when copying over `new`:

```C
char * strReplace(const char *src, const char *old, const char *new)
{
    char *oldInDst = NULL;

    // allocate space for the new string
    char *dst = malloc(sizeof(char) * (strlen(src) + 1));  // +1 for NUL terminator
    assert(dst != NULL);

    // copy all of src into dst
    strcpy(dst, src);

    // find old inside dst
    oldInDst = strstr(dst, old);

    // overwrite old with new inside dst, don't include NUL terminator
    strncpy(oldInDst, new, strlen(new));

    return dst;
}
```

Now if we run the example above, we get:

```C
"Cython's strings are easy to use"
```

Oops. That's not what we wanted!

> Also we broke our original test: `strReplace("My favorite language is Python", "Python", "C")` now returns `"My favorite language is Cython"`.
>
> This is a good reminder to rerun **all** tests after making changes!

The issue is that `new` isn't the same length as `old`. This causes a couple problems:

- We need to `malloc()` more (or less) than `src` since we need more (or fewer) characters in `dst` .
- We can't copy all of `src` directly into `dst` because everything after `new` will be shifted.

Solving the first issue isn't too bad since we know how much we're taking out—`strlen(old)`—and adding in—`strlen(new)`. Instead of:

```c
char *dst = malloc(sizeof(char) * (strlen(src) + 1));  // +1 for NUL terminator
```

we would need:

```c
size_t dstLen = strlen(src) - strlen(old) + strlen(new);
char *dst = malloc(sizeof(char) * (dstLen + 1));  // +1 for NUL terminator
```

Solving the second issue is gonna take a bit more thought, so let's break it down.

- Copy everything in `src` *before* `old`—let's call this `prefix`
- Copy `new` at the end of `prefix`
- Copy everything in `src` *after* `old`—let's call this `suffix`

We're gonna need to be really careful with our pointer arithmetic here—off by one errors can creep in just about anywhere. And it's easy to get lost in which string is going where, so we should add some helpful comments.

```
char * strReplace(const char *src, const char *old, const char *new)
{
    size_t newLen = 0;
    char *srcSuffix = NULL;

    // get memory to hold dst
    size_t dstLen = strlen(src) - strlen(old) + strlen(new);
    char *dst = malloc(sizeof(char) * (dstLen + 1));  // +1 for NUL terminator
    char *dstOffset = dst;
    assert(dst != NULL);

    // copy src up to old (prefix) into dst
    char *oldInSrc = strstr(src, old);
    size_t prefixLen = oldInSrc - src;

    strncpy(dstOffset, src, prefixLen);
    dstOffset += prefixLen;

    // copy new into dst
    newLen = strlen(new);
    strncpy(dstOffset, new, newLen);
    dstOffset += newLen;

    // find suffix start in src
    srcSuffix = oldInSrc + strlen(old);
    // copy suffix into dst
    strcpy(dstOffset, srcSuffix);

    return dst;
}
```

Now, if we call:

```
char *dst = strReplace("Python strings are easy to work with", "Python", "C");
```

and print `dst` we see:

```
"C strings are easy to work with"
```

Let's keep thinking about edge cases. What else might go wrong?

What happens if `old` isn't even in `src`?

We SEGFAULT! Why? Back to the man pages:

```
$ man strstr
...
RETURN VALUES
    [...] if needle occurs nowhere in haystack, NULL is returned;
```

We never checked if we could write to `oldInDst`, we just blindly called `strncpy()` on it.

Let's try to handle that a little more gracefully. Now might be a good time to ask the interviewer how they want it handled. You could:

- Return a copy of `src` since nothing changed
- Return `NULL` to let the caller know something unexpected might have happened

To keep things simple, let's just return `NULL` if `old` isn't in `src`:

```
char * strReplace(const char *src, const char *old, const char *new)
{
    size_t dstLen = 0;
    size_t prefixLen = 0;
    size_t srcLen = strlen(src);
    size_t oldLen = strlen(old);
    size_t newLen = strlen(new);
    char *dst = NULL;
    char *dstOffset = NULL;
    char *srcSuffix = NULL;

    // see if old inside src
    char *oldInSrc = strstr(src, old);
    if (oldInSrc == NULL) {
        return NULL;
    }

    // get memory to hold dst
    dstLen = srcLen - oldLen + newLen;
    dst = malloc(sizeof(char) * (dstLen + 1));  // +1 for NUL terminator
    assert(dst != NULL);
    dstOffset = dst;

    // copy src up to old (prefix) into dst
    prefixLen = oldInSrc - src;
    strncpy(dstOffset, src, prefixLen);
    dstOffset += prefixLen;

    // copy new into dst
    strncpy(dstOffset, new, newLen);
    dstOffset += newLen;

    // find suffix start in src
    srcSuffix = oldInSrc + oldLen;
    // write suffix into dst
    strcpy(dstOffset, srcSuffix);

    return dst;
}
```

Why move the check before the call to `malloc()`? Not only would it waste time to `malloc()` a bunch of space we don't use, but also we'd need to remember to `free()` it so we don't leak memory.

Catchin' some static

What's `static`? When should you use it?

Answer

Your immediate response should be "Which `static`—global or local?"

In C, the keyword `static` gets used in two different ways, so you want to be sure you're talking about the right one! To cover our bases, let's explain both.

- ## Global `static`

  At the global level, a variable or function can be declared `static`. **Global `static` restricts the scope of that symbol to only that file**. The main reason this is helpful is for hiding internal variables and functions that shouldn't be accessed by other parts of your code base.

  Let's look at some code to see how `static` helps. Say we're building up a library of C data structures and want to add `LinkedListNode` and `DoublyLinkedListNode`.

  In `linkedList.h`:

  C

  ```c
  struct LinkedListNode {
      struct LinkedListNode *next;
      int value;
  };

  // Keep track of the start of the list
  struct LinkedListNode *root = NULL;
  ```

  And in `doublyLinkedList.h`:

```c
struct DoublyLinkedListNode {
    struct DoublyLinkedListNode *next;
    struct DoublyLinkedListNode *previous;
    int value;
};

// Keep track of the start of the list
struct DoublyLinkedListNode *root = NULL;
```

Now we can try to use both these headers in the same file:

```c
#include "linkedList.h"
#include "doublyLinkedList.h"

int main(void)
{
    return 0;
}
```

but when we try to compile, we get an error:

```
error: redefinition of 'root' with a different type:
'struct DoublyLinkedListNode *' vs 'struct LinkedListNode *'
```

Ok, that makes some sense—we defined `root` in both headers and then included both of them in main. What if we hide these away in .c files, shouldn't that keep them out of the way?

In `linkedList.h`:

```c
struct LinkedListNode {
    struct LinkedListNode *next;
    int value;
};
```

and `linkedList.c`:

```c
#include "linkedList.h"

struct LinkedListNode *root = NULL;
```

Then in `doublyLinkedList.h`:

```c
struct DoublyLinkedListNode {
    struct DoublyLinkedListNode *next;
    struct DoublyLinkedListNode *previous;
    int value;
};
```

and `doublyLinkedList.c`:

```c
#include "doublyLinkedList.h"

struct DoublyLinkedListNode *root = NULL;
```

Finally, `main.c`:

```c
#include "linkedList.h"
#include "doublyLinkedList.h"

int main(void)
{
    return 0;
}
```

Now when we compile, we get a *different* error:

```
duplicate symbol root in:
    /foo/linkedList.o
    /foo/doublyLinkedList.o
ld: 1 duplicate symbol
error: linker command failed
```

Even when they're hidden away inside .c files, all variable names get passed to the linker to be resolved. So, the linker got two different symbols with the same name: `root`.

Here's where `static` comes in. Variables declared `static` can only be used inside their file. They can't clash with other variables in different files, even if those variables have the same name.

All we have to do to get our code to work is add the `static` keyword to our declarations in the .c files. Like this:

`linkedList.c`

```c
static struct LinkedListNode *root = NULL;
```

and `doublyLinkedList.c`

```c
static struct DoublyLinkedListNode *root = NULL;
```

Adding `static` at the file level hid our `root` variable from other files.

With `static`, we can hide internal variables and functions, forcing other parts of our code base to interface with our file through exposed APIs. As long as our interfaces stay the same, we are free to change our implementation details (like what we called our `root` variable) without worrying about breaking other parts of our code base.

- ## Local `static`

Using `static` at the local level, inside a function, serves a completely different purpose than using it at the global level. **Variables declared `static` inside functions are only initialized once and persist across calls.**

Normally when we declare a variable inside a function, it gets created each time the function is called. It gets added to the stack, used throughout the function, and discarded when the function ends.

Just like regular local variables, local variables declared `static` can only be used inside that function. But unlike regular local variables, `static` local variables keep their value between function calls.

Let's see how this works:

```c
void printCount()
{
    static int count = 0;
    count++;
    printf("printCount() calls: %d\n", count);
}

int main(void)
{
    printCount();
    printCount();
}
```

will print:

```c
"printCount() calls: 1"
"printCount() calls: 2"
```

Static variables aren't that common in functions ... most programmers usually use a global variable instead. If your global variable is only referenced in one function though, consider using a `static` local variable instead.

What's in a cake?

A cake by any other name would taste as sweet...

Now that we've gotten our icing calculations straightened out, our cake business is doing better than ever.

To keep track of all the cake orders we're getting, we've come up with our very own `CakeTable` data structure which tracks a cake's identification number and any other information (like who ordered it, any text in the icing).

Here's the interface we have so far:

```c
/* Supply orderNumber and any data, which can be NULL */
int addCakeToTable(CakeTable *cakeTable, int orderNumber, void *cakeData);

/* Supply orderNumber, get pointer to cakeData, which can be NULL  */
void * lookupCakeByOrderNumber(CakeTable *cakeTable, int orderNumber);
```

Next, we'd like to add a way to iterate through all of the cakes in our database. The design team specified that we need to get both the `orderNumber` and `cakeData` for each of the cakes as we're iterating through.

You've patiently explained that in C you can only return one value from a function, but they keep sending you sample code that looks like this.

```c
while (getNextCake(cakeTable, /* TODO */ ???)) {
    // whatever we want to do with each cake
}
```

What should our function *declaration* (not function *definition*) look like for `getNextCake()`?

Answer

To get started, what can we fill in already from the sample code we got?

We definitely need to know which `CakeTable` we're working on. That's fine ... it looks like the design team has already filled that in as the first argument.

```c
getNextCake(CakeTable *cakeTable);
```

What about the return type?

Looking back at the sample, the design team is using the return value to signal whether we're done or not. This makes sense—we don't know how many cakes we're iterating over, so we need to call `getNextCake()` over and over until we've seen all of them. We can return `1` if there are more cakes to process (which continues the loop), and `0` when we're done (which breaks out of the loop).

```c
int getNextCake(CakeTable *cakeTable);
```

Okay. What about the order number and cake data?

Here's where it starts to get a little tricky. Since we're in C we can't return more than one variable. But the bakeries want to know the `orderNumber` and the `cakeData`. How can we give them both pieces of information from just one function call?

One way to solve this is to use an *input* parameter that can be changed by `getNextCake()`. To make the change "stick", we need to pass in a *pointer* to whatever variable we want filled in. That way, the function can set the value and whoever called the function will have the pointer to it. Here's what that would look like if we wanted to use `orderNumber` as an input/output argument:

```c
int getNextCake(CakeTable *cakeTable, int *orderNumber);
```

And, let's do the same thing for `cakeData`.

```c
int getNextCake(CakeTable *cakeTable, int *orderNumber, void **cakeData);
```

Whoa what's with `void **`? Why do we need two '*'s?

It's the same reason we needed `int *` and not `int` for `orderNumber`—we need to pass in a *pointer* to an `int` so `getNextCake()` can change that `int`'s value. With `cakeData` we need to pass in a *pointer* to a `void *` so we can *change that `void *`* to point to a new set of cake data.

Now we can return the `orderNumber` and the `cakeData` while using the actual return value to indicate when we're done looking at all the cakes. Score!

Littered Linked Lists

We've got some code that uses `LinkedLists`. And we just added a helper function to remove an item at a specified index. Problem is, we're getting weird behavior—sometimes the code works and sometimes it doesn't. Even worse, when we put in some debugging code, the problem seems to disappear.

Can you figure out what's going on and how to fix it?

We've got a pretty standard `LinkedListNode` implementation:

```c
struct LinkedListNode {
    struct LinkedListNode *next;
    int value;
};
```

and here's our code to delete nodes:

```
LinkedListNode * deleteNode(LinkedListNode *head, size_t index)
{
    size_t i = 0;
    LinkedListNode *current = NULL;
    LinkedListNode *nodeToDelete = NULL;

    // special case if there's no linked list to begin with
    if (head == NULL) {
        return head;
    }

    // special case for removing head - need to return new head of linked list
    if (index == 0) {
        free(head);
        return head->next;
    }

    // walk the list until we get to node *before* the one we want to delete
    current = head;
    for (i = 0; i < (index - 1); i++) {
        // move to next node
        current = current->next;

        // if we hit the end of the list before index, just return head
        if (current->next == NULL) {
            return head;
        }
    }

    // make current->next point to the node after the one to delete
    nodeToDelete = current->next;
    current->next = nodeToDelete->next;

    // free memory we no longer need
    free(nodeToDelete);

    // head of the list didn't change
    return head;
}
```

Answer

Code that works sometimes and breaks other times can be really nasty to debug because it's never clear if the issue got fixed or if it just happened to work correctly this time around.

Usually these sorts of bugs come up when we have race conditions between threads or we're accessing the wrong memory. Since we don't have any threads, let's look more closely at the memory.

There are a few common ways to access incorrect memory:

- Uninitialized variables
- Out of bounds
- Freed memory

Let's look at each of these and see if they apply.

Uninitialized variables

There are only a few variables in `deleteNode()`, so we should be able to tell pretty quickly if something is being used without being initialized:

- **`head`**: The *first thing* the function does is check if `head` is `NULL`. So we're good there.
- **`current`**: This gets set to `head`, which we know isn't `NULL`. And we only advance it after making sure `next` isn't `NULL`. So that's fine.
- **`nodeToDelete`**: This gets set to `current->next`, which we know isn't `NULL`. So this is OK too.

Out of Bounds

We need to be careful whenever we have some sort of container like an `array` or `LinkedListNode` and we use an `index` to access elements. If our container isn't big enough for `index`, we need to make sure we don't use it.

Since `index` is an `unsigned int` we don't have to worry about negative numbers.

Let's look at the piece of code where we use `index`:

```c
size_t i = 0;
for (i = 0; i < (index - i); i++) {
    // move to next node
    current = current->next;

    // if we hit the end of the list before index, just return head
    if (current->next == NULL) {
        return head;
    }
}
```

We always check to make sure `current->next` isn't `NULL`, since that would mean we hit the end of the list. The early `return` makes sure we never try to use `current` where it could be `NULL`. So we're OK there.

Freed memory

In C, we need to make sure never to use memory after we've called `free()` on it since we can't make any guarantees about what's in those memory blocks anymore.

We use `free()` twice in `deleteNode()` so let's check both cases to see if we've used memory after we've freed it.

- In the main block of code:

  ```c
  LinkedListNode *nodeToDelete = current->next;
  current->next = nodeToDelete->next;
  free(nodeToDelete);
  ```

  This is fine because we know `nodeToDelete` won't be `NULL`. And we use `nodeToDelete->next` before we `free()` it. And we never use `nodeToDelete` again. So everything here looks good.

- In the special case where we need to return a new `head`:

  ```c
  free(head);
  return head->next;
  ```

  Ah, there's an issue. We try to return `head->next` right *after* we `free()` that memory.

Now that we've found the error, we should:

- Understand why it caused buggy behavior
- Fix it

Understanding it

Why did our code work sometimes but not other times? Why did it sometimes give incorrect values and sometimes `SEGFAULT`? Because **we broke our promise to never use freed memory**. Whenever we use `free()`, we're saying "even though we still have this pointer, we promise not to use it anymore ever again".

> It's considered good practice to set pointers to `NULL` right after freeing them (e.g.: `free(head); head = NULL;`). This ensures that if we try to dereference a pointer that's already been freed, we'll get a `SEGFAULT`, preventing the weird behavior where our code sometimes works and sometimes crashes.

> The C standard (http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1570.pdf) warns that using a pointer after its been freed causes "undefined behavior." That means all bets are off about what will happen!

Once we've called `free()`, the C Runtime Library (which manages dynamically-allocated memory) knows it can use that block of memory however it wants. But there are no guarantees for what happens or even when.

- Maybe the memory referenced by `free(head)` won't be touched between the time we call `free()` and our access of `head->next`. In that case, everything works fine and we get a pointer to the new `head`.
- Maybe the memory referenced by `free(head)` will be overwritten with zeros. Then, `head->next` would be `NULL`, and we'd lose track of the remaining nodes in the list.
- Maybe the memory referenced by `free(head)` will be overwritten with data that makes `head->next` an invalid memory address. When we try to use it in the next iteration, we'll get a `SEGFAULT`.
- Maybe the memory referenced by `free(head)` will be overwritten with data that makes `head->next` some random memory address. Now, we'll be corrupting our program's memory, which could cause a crash later on, in a completely different section of our code.

Which of those is going to happen is anybody's guess. It might depend on how many threads of the program are running or how much debug code is in there or what happened the last time the program was run.

Fixing it

Lucky for us, fixing this is simple. We did things right when we freed `nodeToDelete`. We just need to do that same process here: grab the data we need *first*, and *then* call `free()`:

```c
LinkedListNode *newHead = head->next;  // grab new LinkedList head
free(head);  // free memory we don't need
head = NULL;  // make sure we don't accidentally use head later on
return newHead;
```

Double the Fun

We've got an array of `doubles`—prices for all the different cakes around town. Instead of always having to pass around another variable that tells us the length of the array, we've used a `0.0` terminator to indicate the end of the array. We figured this would make things easier but we're running into some issues with one of our functions.

It should be a simple function that adds up the price of all the cakes:

```c
double buyAllTheCakes(const double *prices)
{
    double totalCost = 0.0;

    while (*prices) {
        totalCost += *prices;
        prices += sizeof(double);
    }

    return totalCost;
}
```

But we aren't always getting the right answer when we call this function. The `totalCost` usually ends up way too low and we run out of money on our cake-buying spree. And sometimes we get an absurdly high `totalCost`. Like, *impossibly* high, because there's only 25 cakes in there and they shouldn't cost millions of dollars.

Can you figure out what's going wrong in `buyAllTheCakes()` and how to fix it?

Answer

This is another case where we get different results depending on when and how the code is run. We talked about some of those issues in the Littered Linked Lists question above. Let's look at the same ones here:

- Freed memory
- Uninitialized variables
- Out of bounds array indexing

We don't need to worry about freed memory because we aren't using `free()` anywhere. And we don't need to worry about uninitialized variables because the only variable we have is `totalCost` and it's set to `0.0` when we declare it.

So maybe we're ending up outside of our array somehow. Or not looking at everything in the array? Is there any pattern to when it's too low or too high?

Let's add in some debugging code to print the cost of each cake as we add it to our total. Then we can run it on a few small examples and see if anything pops out.

Our updated function:

```c
double buyAllTheCakes(const double *prices)
{
    double totalCost = 0.0;

    while (*prices) {
        printf("Price: %f\n", *prices);
        totalCost += *prices;
        prices += sizeof(double);
    }

    return totalCost;
}
```

and our test cases:

```c
const double oneCake[]      = { 1.0, 0.0 };
const double twoCakes[]     = { 1.0, 2.0, 0.0 };
const double fourCakes[]    = { 1.0, 2.0, 3.0, 4.0, 0.0 };
const double eightCakes[]   = {
    1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 0.0
};
const double sixteenCakes[] = {
    1.0,  2.0,  3.0,  4.0,  5.0,  6.0,  7.0,  8.0,
    9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0,
    0.0
};
```

Now let's call `buyAllTheCakes()` on all our test cases and see what we get:

```
oneCake:
    Price: 1.0
twoCakes:
    Price: 1.0
fourCakes:
    Price: 1.0
eightCakes:
    Price: 1.0
sixteenCakes:
    Price: 1.0
    Price: 9.0
```

> You may not get these exact same results if you try to run the code above. If you want to force those results, try padding out all the cake arrays with `0.0` to 10 or 20 and that *should* do the trick. To understand why, keep reading!

At first it seemed like we were skipping from the first element in the array to the end because we only saw `1.0`. But that last test made things a bit more confusing because we saw two cakes ... one from the beginning and one from the middle.

If we keep expanding our tests, we'll start to see that it skips the same number of elements every time—8, from what we saw above. And depending on the size of our array, after printing every 8th item, it might continue past the end and that's where we start getting crazy values.

How are we jumping 8 spots in the array instead of moving just 1? All we do is call `prices += sizeof(double);`, what could go wrong?

Turns out, C is smarter than we gave it credit for. When we have a pointer to the start of an array, C does arithmetic on that pointer based on the type of the elements it points to and doesn't treat them like plain-old numbers.

So, if `prices` points to the first array element, then `prices + 1` points to the second array element, `prices + 2` points to the third element and so on. Think about how we would index into this array. If we wanted the first element, we'd use `prices[0]` and to get the next element we'd use `prices[1]` and not `prices[ sizeof(double) ]`.

That explains how we kept skipping 8 elements at a time! When we wrote `prices += sizeof(double);` we moved forward `sizeof(double)` elements. And a `double` takes up 8 bytes!

All we need to change is how we advance our pointer—we want to move forward 1 element, not 8:

```c
double buyAllTheCakes(const double *prices)
{
    double totalCost = 0.0;

    while (*prices) {
        totalCost += *prices;
        // instead of: prices += sizeof(double);
        prices++;
    }

    return totalCost;
}
```

Apple Stocks »

Figure out the optimal buy and sell time for a given stock, given its prices yesterday. keep reading »

(https://www.interviewcake.com/question/c/stock-price)

Implement A Queue With Two Stacks »

Implement a queue with two stacks. Assume you already have a stack implementation. keep reading »

(https://www.interviewcake.com/question/c/queue-two-stacks)

Find Repeat, Space Edition »

Figure out which number is repeated. But here's the catch: optimize for space. keep reading »

(https://www.interviewcake.com/question/c/find-duplicate-optimize-for-space)

Find Repeat, Space Edition BEAST MODE »

Figure out which number is repeated. But here's the catch: do it in linear time and constant space! keep reading »

(https://www.interviewcake.com/question/c/find-duplicate-optimize-for-space-beast-mode)

Find Duplicate Files »

Your friend copied a bunch of your files and put them in random places around your hard drive. Write a function to undo the damage. keep reading »

(https://www.interviewcake.com/question/c/find-duplicate-files)

Top Scores »

Efficiently sort numbers in an array, where each number is below a certain maximum. keep reading »

(https://www.interviewcake.com/question/c/top-scores)

Bracket Validator »

Write a super-simple JavaScript parser that can find bugs in your intern's code. keep reading »

(https://www.interviewcake.com/question/c/bracket-validator)

Recursive String Permutations »

Write a recursive function of generating all permutations of an input string. keep reading »

(https://www.interviewcake.com/question/c/recursive-string-permutations)

In-Place Shuffle »

Do an in-place shuffle on an array of numbers. It's trickier than you might think! keep reading »

(https://www.interviewcake.com/question/c/shuffle)

Which Appears Twice »

Find the repeat number in an array of numbers. Optimize for runtime. keep reading »

(https://www.interviewcake.com/question/c/which-appears-twice)

Find in Ordered Set »

Given an array of numbers in sorted order, how quickly could we check if a given number is present in the array? keep reading »

(https://www.interviewcake.com/question/c/find-in-ordered-set)

Find Rotation Point »

I wanted to learn some big words to make people think I'm smart, but I messed up. Write a function to help untangle the mess I made. keep reading »

(https://www.interviewcake.com/question/c/find-rotation-point)

Cafe Order Checker »

Write a function to tell us if cafe customer orders are served in the same order they're paid for. keep reading »

(https://www.interviewcake.com/question/c/cafe-order-checker)

Simulate 5-sided die »

Given a 7-sided die, make a 5-sided die. keep reading »

(https://www.interviewcake.com/question/c/simulate-5-sided-die)

Inflight Entertainment »

Writing a simple recommendation algorithm that helps people choose which movies to watch during flights keep reading »

(https://www.interviewcake.com/question/c/inflight-entertainment)

Product of All Other Numbers »

For each number in an array, find the product of all the other numbers. You can do it faster than you'd think! keep reading »

(https://www.interviewcake.com/question/c/product-of-other-numbers)

Highest Product of 3 »

Find the highest possible product that you can get by multiplying any 3 numbers from an input array. keep reading »

(https://www.interviewcake.com/question/c/highest-product-of-3)

Merging Meeting Times »

Write a function for merging meeting times given everyone's schedules. It's an enterprise end-to-end scheduling solution, dog. keep reading »

(https://www.interviewcake.com/question/c/merging-ranges)

Compute nth Fibonacci Number »

Computer the nth Fibonacci number. Careful--the recursion can quickly spin out of control! keep reading »

(https://www.interviewcake.com/question/c/nth-fibonacci)

Rectangular Love »

Find the area of overlap between two rectangles. In the name of love.
keep reading »

(https://www.interviewcake.com/question/c/rectangula love)

Balanced Binary Tree »

Write a function to see if a binary tree is 'superbalanced'--a new tree property we just made up. keep reading »

(https://www.interviewcake.com/question/c/balanced-binary-tree)

Parenthesis Matching »

Write a function that finds the corresponding closing parenthesis given the position of an opening parenthesis in a string. keep reading »

(https://www.interviewcake.com/question/c/matching-parens)

Temperature Tracker »

Write code to continually track the max, min, mean, and mode as new numbers are inserted into a tracker class. keep reading »

(https://www.interviewcake.com/question/c/temperatu tracker)

Simulate 7-sided die »

Given a 5-sided die, make a 7-sided die. keep reading »

(https://www.interviewcake.com/question/c/simulate-7-sided-die)

Word Cloud Data »

You're building a word cloud. Write a function to figure out how many times each word appears so we know how big to make each word in the cloud. keep reading »

(https://www.interviewcake.com/question/c/word-cloud)

Two Egg Problem »

A building has 100 floors. Figure out the highest floor an egg can be dropped from without breaking. keep reading »

(https://www.interviewcake.com/question/c/two-egg-problem)

Binary Search Tree Checker »

Write a function to check that a binary tree is a valid binary search tree. keep reading »

(https://www.interviewcake.com/question/c/bst-checker)

Reverse Words »

Write a function to reverse the word order of a string, in place. It's to decipher a supersecret message and head off a heist. keep reading »

(https://www.interviewcake.com/question/c/reverse-words)

Permutation Palindrome »

Check if any permutation of an input string is a palindrome. keep reading »

(https://www.interviewcake.com/question/c/permutatic palindrome)

Kth to Last Node in a Singly-Linked List »

Find the kth to last node in a singly-linked list. We'll start with a simple solution and move on to some clever tricks. keep reading »

(https://www.interviewcake.com/question/c/kth-to-last-node-in-singly-linked-list)

Reverse A Linked List »

Write a function to reverse a linked list in place. keep reading »

(https://www.interviewcake.com/question/c/reverse-linked-list)

The Stolen Breakfast Drone »

In a beautiful Amazon utopia where breakfast is delivered by drones, one drone has gone missing. Write a function to figure out which one is missing. keep reading »

(https://www.interviewcake.com/question/c/find-unique-int-among-duplicates)

MillionGazillion »

I'm making a new search engine called MillionGazillion(tm), and I need help figuring out what data structures to use. keep reading »

(https://www.interviewcake.com/question/c/compress-url-list)

The Cake Thief »

You've hit the mother lode: the cake vault of the Queen of England. Figure out how much of each cake to carry out to maximize profit. keep reading »

(https://www.interviewcake.com/question/c/cake-thief)

2nd Largest Item in a Binary Search Tree »

Find the second largest element in a binary search tree. keep reading »

(https://www.interviewcake.com/question/c/second-largest-item-in-bst)

Largest Stack »

You've implemented a Stack class, but you want to access the largest element in your stack from time to time. Write an augmented LargestStack class. keep reading »

(https://www.interviewcake.com/question/c/largest-stack)

Delete Node »

Write a function to delete a node from a linked list. Turns out you can do it in constant time! keep reading »

(https://www.interviewcake.com/question/c/delete-node)

Making Change »

Write a function that will replace your role as a cashier and make everyone rich or something. keep reading »

(https://www.interviewcake.com/question/c/coin)

Reverse String in Place »

Write a function to reverse a string in place. keep reading »

(https://www.interviewcake.com/question/c/reverse-string-in-place)

Ticket Sales Site »

Design a ticket sales site, like Ticketmaster keep reading »

(https://www.interviewcake.com/question/c/ticket-sales-site)

What's next?

Check out our mock coding interview questions (https://www.interviewcake.com/next).

They mimic a real interview by offering hints when you're stuck or you're missing an optimization.

Try some questions now ➡