The Descent to C

by Simon Tatham

1. Introduction

This article attempts to give a sort of ‘orientation tour’ for people whose previous programming background is in high (ish) level languages such as Java or Python, and who now find that they need or want to learn C.

C is quite different, at a fundamental level, from languages like Java and Python. However, well-known books on C (such as the venerable Kernighan & Ritchie) tend to have been written before Java and Python changed everyone's expectations of a programming language, so they might well not stop to explain the fundamental differences in outlook before getting into the nitty-gritty language details. Someone with experience of higher-level languages might therefore suffer a certain amount of culture shock when picking up such a book. My aim is to help prevent that, by warning about the culture shocks in advance.

This article will not actually *teach* C: I'll show the occasional code snippet for illustration and explain as much as I need to make my points, but I won't explain the language syntax or semantics in any complete or organised way. Instead, my aim is to give an idea of how you should expect C to differ from languages you previously knew about, so that when you do pick up an actual C book, you won't be distracted from the details by the fundamental weirdness.

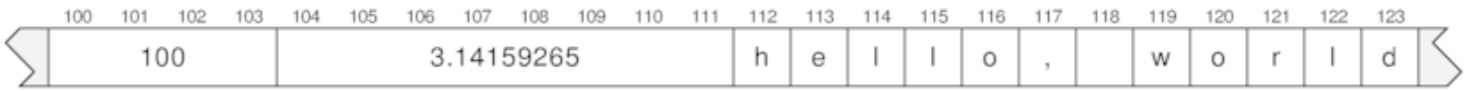
I'm mostly aiming this article at people who are learning C in order to work with existing C programs. So I'll discuss ways in which things are commonly done, and things you're likely to encounter in real-world code, but not things that are theoretically possible but rare. (I do have other articles describing some of those.)

2. Memory layout

Modern high-level languages generally try to arrange that you don't need to think – or even know – about how the memory in a computer is actually organised, or how data of the kinds you care about is stored in it. Indeed, they actively try to hide this information from you: if you do know about that sort of thing, and you find yourself wondering (say) how the fields of a Java class are laid out in memory or where a Python lambda stores the variables captured from the scope it was defined in, the languages will provide no means for you to even ask it those questions. The implication is that that's not your business: you just write the semantics, and don't bother your head with implementation details.

By contrast, C thinks that these implementation details *are* your business. In fact, C will *expect* you to have a basic understanding that memory consists of a sequence of bytes each identified by a numeric address...

that most items of data are stored in one or more bytes with consecutive addresses...



... and that the language's methods of aggregating multiple data items into one larger one (arrays and structures) work by placing those data items adjacent to each other in contiguous pieces of memory (sometimes with padding to make the addresses nice round multiples of numbers like 4). For example, defining this array...

```
struct foo { char c; int i; };
struct foo my_array[3] = {
    { 'x', 123 },
    { 'y', 456 },
    { 'z', 789 },
};
```

... might give you a memory layout looking something like this:

C will expect you to not only *understand* this concept, but to *use* it in your work. For example, the C standard library provides a function called ‘`memcpy`’, which copies a certain number of ‘`char`’ values (i.e. bytes) from one contiguous memory area to another. If you wrote that function in Java, it would work fine, but you'd only be able to use it with actual arrays of `char`; there would be no way to use it to copy the contents of an array of `int`, or the contents of a Java class. But in C, `memcpy` can copy anything at all – as long as you know the address in memory where your object starts and how many bytes it occupies, you can copy it to a different address one byte at a time using `memcpy`, and all the larger data items such as `int`s will arrive at their destination unharmed by taking them apart and putting them back together like that. This isn't just a curiosity: sometimes you'll *have* to do this sort of thing, because it will be the only convenient way to get the data you need into the place you need it. That's what I mean by saying that C will expect you to have and use a basic understanding of memory organisation.

3. Pointers

Naturally, if you're going to refer to objects in memory by their addresses, the language you do it in must provide some kind of data type suitable for storing an address. In C, this data type is called a *pointer*. In fact, there's a whole family of them: for every data type, there is a corresponding pointer type (pointer to `int`, pointer to `char`, pointer to some structure you defined yourself) which stores the address of an object of the original type. (Yes, you can have pointers to pointers too.)

Higher-level languages generally have some kind of mechanism which is essentially pointer-like in implementation, in that it lets you have more than one variable through which you can access the same actual data. However, C pointers are a bit different in several ways.

3.1. Pointers are explicit

If you've used Java or Python, you'll probably be familiar with the idea that some types of data behave differently from others when you assign them from one variable to another. If you write an assignment such as ‘`a = b`’ where `a` and `b` are integers, then you get two *independent* copies of the same integer: after the assignment, modifying `a` does not also cause `b` to change its value. But if `a` and `b` are both variables of the same Java class type, or Python lists, then after the assignment they refer to the same underlying object, so that if you make a change to `a` (e.g. by calling a class method on it, or appending an item to the list) then you see the same difference when you look at `b`.

In Java and Python, you don't get much of a choice about that. You can't make a class type automatically copy itself properly on assignment, or make multiple ‘copies’ of an integer really refer to the same single data item. It's implicit in the type system that some types have ‘value semantics’ (copies are independent) and some have ‘reference semantics’ (copies are still really the same thing underneath).

In C, this distinction is explicit, not implicit. Every data type has value semantics by default, even big structures containing lots of fields (analogous to a Java class): assigning one structure variable to another causes a large amount of memory to be physically copied, and then you end up with two independent instances of your large structure type. On the other hand, if you want reference semantics, you can *take the address* of any ordinary variable, creating a value of the appropriate pointer type which points at the original variable.

For example, suppose you write this snippet of code:

```
void function(int x)
{
    x = x + 1;
    other_function(x);
}
```

Just like in Java or Python, the parameter variable `x` is treated as a local variable of the function: you're allowed to modify it, but the modifications only apply to your own copy. If someone calls this function by writing

```
    function(my_important_int_variable);
```

then the value of their important `int` variable will be unchanged afterwards.

But if you *want* to write a function that modifies an `int` variable specified by the caller, you can do it using a pointer. You'd change the function so that it looked like this:

```
void function(int *x)
{
    *x = *x + 1;
    other_function(*x);
}
```

and then the caller would have to write this:

```
    function(&my_int_variable);
```

The syntax '`int *x`' declares `x` to be a pointer to `int`; the syntax '`*x`' inside the function is called a *dereference*, and means 'the `int` value stored at the address given by `x`'. The '`&`' sign is the opposite of '`*`', in that it starts from an `int` (or anything else) and returns a pointer value giving the address of that `int`.

For these purposes, there's no difference between simple types like `int` and complicated user-defined structure types analogous to Java classes. If I'd written the entire example above using a structure type in place of `int`, it would have looked exactly the same – passing a structure argument to a function in the most obvious way causes an independent copy to be made, whereas if you want the called function to be able to modify the structure then you have to pass a pointer instead, and there will be an '`&`' visible at every call site to remind you of that.

3.2. Pointers can go stale

In the previous section I showed an example where you call a function and pass it the address of a variable you want it to write into. Suppose the calling function had done something like this:

```
int caller(void)
{
    int my_int_variable = 3;
    function(&my_int_variable);
    return my_int_variable;
}
```

This function initialises a variable to 3; then it calls the example function from the previous section to change the value of the variable; finally it returns the updated value.

Now suppose that, instead of doing what it did in the previous section, `function()` had *kept* a copy of the pointer it was passed, by stashing it in a global variable:

```
int *stashed_pointer;
void function(int *x)
{
    stashed_pointer = x;   /* save a copy of the pointer */
}
```

Then it returns to `caller()`, which returns in turn. When `caller()` returns, its local variable `my_int_variable` goes 'out of scope' – that is to say, it stops existing.

So now what's the status of the value in '`stashed_pointer`'?

The answer is that it's a *stale pointer*. That is, it's the address of a piece of memory which *used* to hold a currently active variable, but doesn't hold one any more. So if any part of the program uses that pointer value in future, bad things will happen. Some kinds of stale pointer access can cause the program to crash, but the one described here is unlikely to do that; more probably, it will have the much worse effect of overwriting a piece of memory that's been reused to store something completely different, so that you don't immediately notice the problem, and your program starts behaving oddly at a later time when it's too late to easily work out why.

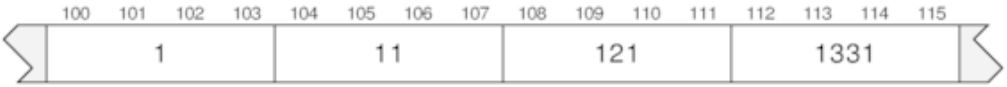
(Java's implicit approach to reference semantics avoids this risk, because local variables in functions are either references *already* – pointing to objects whose lifetime is not tied to the lifetime of the function – or else cannot be converted into references. The risk arises because C lets you take the address of any variable you like.)

3.3. You can do arithmetic on pointers

Objects in C are often stored at addresses which differ by a fixed amount. Successive elements of an array are laid end-to-end in memory, for example, and so are fields of a structure. So it's often useful to be able to construct a new pointer value by doing arithmetic on an old one.

C arranges this by letting you add an integer to a pointer value, e.g. if `p` is a pointer then you can write `p+1` or `p-100`. The number you add is implicitly multiplied by the size of the type that `p` points to. In other words, if `p` is a pointer to `int`, then writing `p+1` returns a pointer whose address is not one *byte* after it in memory, but one `int` after it in memory. This means, in particular, that if `p` originally pointed to one of the elements of an array, then adding 1 to it causes it to point at the *next* element in the array. For example, in this snippet:

```
int array[4] = { 1, 11, 121, 1331 };
int *ptr = &array[1];  /* ptr points to the second element (11) */
```



if we suppose that our array starts at address 100 as shown in the above diagram, then the pointer variable `ptr` holds the address 104 (pointing at `array[1]`). If you compute `ptr+1`, the resulting pointer will hold the value 108 (not 105), so that it points at the next array element `array[2]`; if you compute `ptr-1`, you'll get a pointer holding the value 100, pointing at `array[0]`.

Put another way: in C, a pointer to type `Foo` isn't just an address where you can expect to find an object of type `Foo`. It's always possible that what you can find there is one `Foo` in a long sequence stuck end-to-end, so the language provides you with a convenient way to step back and forth along that sequence.

Idiomatic C code will use this facility frequently. String handling code in particular (which we'll examine in more detail in section 7) has a strong tendency to use pointer arithmetic in preference to array indices. That is, instead of having an integer variable `i` starting at zero, examining a character of the string by referring to `string[i]`, and incrementing `i` when you want to move on to the next character, it's more common to assign a pointer value `p` to point at the start of the string, examine a character by referring to `*p`, and increment `p` to step on to the next character.

3.4. Pointers are not type-safe

Another way in which pointers in C differ from high-level languages' references is that you're not constrained to point them at objects of the right type.

A reference in a high-level language can be relied on to either point at a valid object of the right type, or at `NULL` or `nil` or `None` or some such. We've already seen (section 3.2) that C pointers don't obey the same constraint, because they can point at memory that *used* to contain a valid object but doesn't any more.

But that's not the only way in which a pointer can point at ‘wrong’ things. Pointers are just addresses in memory, and C doesn't assume it knows better than you about what you keep where in memory. So C will typically let you just construct any pointer value you like by casting an integer to a pointer type, or by taking an existing pointer to one type and casting it so that it becomes a pointer to an entirely different type.

For example, in section 2 I already mentioned that you can treat any C object of any type as a plain sequence of bytes, just by taking the address of the object and casting it to a pointer to `char`. It's not generally a good idea to rely on the exact byte values you'll read out of an object that way; the exact details of objects' memory layout will vary between platforms, so you won't get the same answers on every kind of system if you do this. But you can depend on a few things, such as that if you copy a whole object byte by byte from one place to another then you'll end up with a valid copy of the same object at the destination.

4. Arrays are not bounds-checked

In higher-level languages, you're probably used to the idea that looking up an element of an array will either successfully return an element if the index is in the correct range, or else crash with an exception of some kind. In order to do this, arrays have to come with some implicit metadata indicating their length, and the language generally lets you access that metadata directly as well (`len(array)` in Python, `array.length` in Java).

In C, arrays are a much simpler and more primitive concept. An array is just some number of pieces of data of the same type, all laid out in memory end-to-end. You have a pointer to the start of the array, and you access `array[i]` by first adding `i` to the value of that pointer, and then looking up whatever is in memory at that location. (In fact, that's what `array[i]` *means* – the language defines it to be a synonym for `*(array+i)`.)

So the length of the array isn't stored in memory anywhere, and therefore C arrays have no way to look up the length at run time like Java and Python do. This also means that *you are responsible* for making sure you don't overrun the bounds of the array: the language

cannot detect it if you do, so you won't be notified of your mistake by a nice clean exception. Instead, overrunning an array will have effects similar to those of dereferencing a stale pointer (see section 3.2 above): if you're lucky the operating system will manage to crash your program, but if you're unlucky, you'll overwrite memory that's being used by something else, and cause an unpredictable change to your program's behaviour which might not show up for a long time.

Therefore, it's important to make sure you keep track of array lengths yourself. This is often done by keeping the array pointer alongside a variable giving the array's length; alternatively, you might arrange that the array has some kind of recognisable terminating element at the end, and then any function to which you pass a pointer to the array must make sure to stop when it encounters that terminating element. (Strings are often handled this way; see section 7.)

A compensatory advantage to C's very primitive concept of arrays is that you can pretend that they're a different size or that they start in a different place. For example, if you have a function which expects to be given a pointer to an array of ten `Whatsits`, and in fact you have an array of fifty `Whatsits` and you want the function to operate on elements 20,…,29 of that array, that's just fine – pass the value `array+20` to the function and it'll never know or care that the pointer it received wasn't 'really' the start of the array from your point of view.

5. Allocated memory must be manually freed

Perhaps the best-known difference between C and high-level languages is the manual memory management.

In a language like Java, you're used to being able to say '`new Foo`' to create a new object of type `Foo`. In C, things are not too different: you say '`malloc(n)`' to ask for a piece of memory `n` bytes long, so if you want to create a new object of type `Foo` then you say '`malloc(sizeof(Foo))`', which will allocate just the right amount of memory to keep a `Foo` in.

(Though note that the small difference between the apparently synonymous '`new Foo`' and '`malloc(sizeof(Foo))`' does have one noticeable consequence, which is that the return value of `malloc` has a generic pointer type rather than being a pointer to `Foo`. So if you pass `sizeof(`*the wrong thing*`)` to `malloc`, nothing at compile time will warn you that you've allocated the wrong amount of memory. This is another way in which you can accidentally overwrite important data when programming in C.)

The difference in C is that when you've finished with that piece of memory, you have to dispose of it by hand, by calling the function `free()` and passing it the same address you got from `malloc`.

This has two complementary consequences. Firstly, of course, it means you have to *remember* to free your memory, and in particular, to free it at the right moment – after it's not needed any more, but before you lose all your copies of the pointer (so that you wouldn't be able to free it anyway). And secondly, this gives you another way in which pointers can become stale: if you have several copies of a pointer to allocated memory in different parts of your code, and then one part of the code frees the memory, the other parts now have a stale pointer which they must avoid using for anything.

(In particular, you mustn't *free* a pointer that's already been freed. Freeing twice is an error and will typically corrupt the memory allocator's state.)

A typical technique for getting this right is to imagine a concept of 'ownership' of allocated memory. The idea is, if several parts of the code have copies of the same allocated pointer, one of them is considered to 'own' it. The owner of the memory is the only one who can free it. So the piece of code which owns the pointer must make sure to free it at some appropriate moment (e.g. if the pointer is kept in another allocated structure, then when the latter structure is disposed of that's often a good moment to free the pointer too); on the other hand, any code which *doesn't* own the pointer has a potential risk of finding the pointer has become stale, if the owner frees it and doesn't let them know. So you need to think about how that's going to work, and find a way for non-owners of the pointer to avoid making that mistake. (There are lots of ways to do that. For example, you might be able to arrange that the owner is freed last, so that all the non-owners have gone away already before the owner calls `free()`. Alternatively, you might keep a list of non-owners, and notify them all when you free the memory. And so on.)

6. Undefined behaviour

In previous sections I've described a number of things you can get wrong in a C program – accessing stale pointers, allocating the wrong size of memory, overrunning the bounds of an array, freeing the same pointer twice – which will cause your program to overwrite the wrong piece of memory. In a 'safe' high-level language, you would expect all of these to be either impossible in the first place (because the language syntax doesn't permit you to even express the idea) or else be checked at run time and give rise to an exception, so that the program deliberately crashes and tells you something about what you did wrong.

In C, you are not protected in either of those ways, so making mistakes of this kind can lead to your program corrupting pieces of

memory that other parts of the program were depending on. If you're *lucky*, this will lead to a reasonably prompt crash of some kind; but if you're unlucky, the effects won't be immediately noticeable, and your program will appear to work sensibly for the moment and then fail much later for no obvious reason. Worse still, if your program is on a security boundary and any of these errors can occur in response to input from untrusted users, then a malicious attacker may be able to manipulate errors like this to arrange to overwrite particular pieces of memory *on purpose*, and perhaps take control of your program.

The technical term for this kind of situation, in the C standard, is 'undefined behaviour'. The C standard is intentionally only a partial specification of how programs must behave: there are a lot of things you can do in C for which the standard specifies no behaviour, so that if you do any of those things then *anything can happen* without the compiler or runtime being in violation of the standard.

Overwriting the wrong piece of memory is a major and important example of undefined behaviour, but it's not the only one. There are other things which you should avoid doing in C, but which the language implementation won't give you any help with, and if you do get them wrong then *unpredictable weirdness* can result.

An example of this is integer overflow: trying to store a too-large or too-small value in a C signed integer type such as '`int`'. You could imagine several plausible things that might happen if you try this: wraparound (adding one to the largest possible value gets you the smallest possible value), saturation (adding to the largest possible value leaves it unchanged), or a crash (the run-time environment detects the error and terminates the program). In fact, C says that the behaviour if you do this is completely undefined: that is, any of those reasonably sensible things could happen, *but anything else could happen instead, no matter how silly*. For example, consider this code:

```
int f(int n)
{
    if (n < 0)
        return 0;
    n = n + 100;
    if (n < 0)
        return 0;
    return n;
}
```

Looking at this function, you'd think there was no way it could return a negative number. We first make sure `n` is positive; then even *after* we add 100 to it, we check again that it's positive, in case wraparound due to integer overflow caused it to become negative. You could imagine the function crashing, if it were compiled and run on a platform where that's the response to overflow, but if it returns *at all* then surely it must return either zero, or a positive integer. Right?

Actually, no. The GNU C compiler (`gcc`) generates code for this function which *can* return a negative integer, if you pass in (for example) the maximum representable '`int`' value. Because the compiler knows after the first `if` statement that `n` is positive, and then it *assumes that integer overflow does not occur* and uses that assumption to conclude that the value of `n` after the addition must still be positive, so it completely removes the second `if` statement and returns the result of the addition unchecked.

Put another way, you could imagine that the compiler is thinking to itself: 'If the user passed in an integer so large that the addition overflows, that would be undefined behaviour, so we can assume the user didn't do it. So they must be implicitly promising to always call this function with input values that don't cause overflow – in which case the function doesn't need the second test, and will run faster with it removed.' And, in a sense, it's right: as long as you *don't* cause integer overflow by passing a too-large value as input to the above function, it will work correctly like that and run faster. It's just that the test you deliberately put in as a safety check hasn't worked, because undefined behaviour is *too weird* for any safety check to be able to reliably spot it after it's happened.

And even that is still only one possible example of how a program might misbehave if you cause undefined behaviour – another time, it might be some other totally different thing that you couldn't have predicted in advance. So beware! The only safe thing is not to allow undefined behaviour to happen in the first place: for example, in the above code, the right thing would have been to check whether `n` is close to overflow *before* trying to add anything to it, and not do the addition at all if so. (Another option in some situations is to rewrite the code using the type '`unsigned int`', which the standard defines to be somewhat better behaved.)

7. There is no convenient string type

String handling in C is very, very primitive by the standards of almost any other language. In Java or Python, you expect that most of the time you can treat string variables just like any other kind of variable. In C, that's not true at all.

In C, a string is just an array of '`char`' values (or '`wchar_t`', if you're using Unicode, but that doesn't really affect the upcoming discussion). That is to say, you just have a lot of `char`s laid end-to-end in memory. The standard convention (although you can choose to do it differently if you need to) is that strings are terminated with a zero byte; so if you pass a string to a function, for example, you

typically just pass a `char` pointer that points to the first character of the string, and the function receiving it has to search along the string for the zero byte if it wants to know how long the string is.

The interesting effect of this is: suppose you want to construct a *new* string, for example by concatenating two existing strings. Where do you put it? You need to find an appropriately sized chunk of memory to keep your new string in.

The most obvious approach is to use `malloc` to allocate a chunk of memory the right size. In order to do that, first you have to find out the lengths of your two input strings, by counting along each one looking for the terminating zero byte. Then you have to call `malloc` and ask for a number of bytes equal to the sum of those two lengths *plus one* (leaving room for the terminating zero byte), and then you have to copy the actual string data from the two input strings into the newly allocated memory. And *then*, of course, you have to keep track of when you've finished with the string, and remember to free it once it's no longer needed.

That all adds up to a lot of effort compared to the typical high-level language just letting you write something like '`newstring = string1 + string2`'. And it's a typical example, unfortunately: *most* string handling operations in C are about as annoying as that.

Of course, the comparatively easy string handling in high-level languages will *really* be having to do all of this same work; it's just that the language hides it from you, and takes care of the tedious details automatically. So in a high-level language, you can easily write some pretty slow string-handling code without really noticing – you can concatenate two strings with a single plus sign, and in some situations that one character can cause the language to have to move megabytes of data bodily around the computer's memory. In C, you're much more likely to *notice* that the string handling you asked for is likely to be long and tedious and slow – and sometimes, you can find ways to avoid some of the pain.

For example, suppose you've got a long string, and you want to break it up at the space characters so that you end up with lots of smaller strings representing individual words. If you do that in a high-level language with the built-in string handling, you'll almost certainly end up with a series of string variables containing *copies* of the parts of the original string that contain each word.

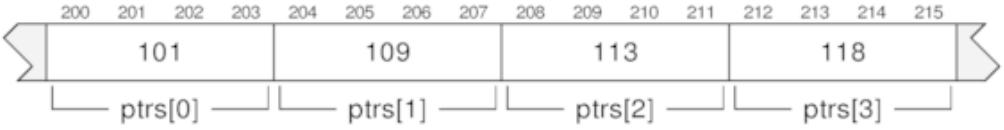
In C, a common approach to this problem (at least if you didn't need the original string itself afterwards, which you often don't) avoids having to actually copy *anything*, so it will run much faster. What you do is to modify the original array of `char`s, by writing zero bytes just beyond the ends of words. For example, suppose you have an input string like this:

```
char input_string[] = " string  of  four words";
```

(The `\0` in the last place is C's notation for the zero byte value used to terminate the string.)

Then you might transform it by writing zero bytes into these locations:

Now we can construct an array of `char`-pointer values (let's call it `ptrs`) pointing at the beginning of each word. For example, our pointer array might look like this in memory ...



... so that the values in it point to the start of each word, avoiding the spaces before them:

So each pointer points to a zero-terminated string containing one of the original words. Hence, we've effectively constructed a separate string value per word, without ever having to move the string data from one place to another or allocate new memory to contain copies of it. (Well, we probably did have to allocate memory for our `ptrs` array, but that's typically a lot smaller.)

In C, this kind of trickery is done as often as possible, to save the considerable pain of allocating memory and copying stuff. When it's possible, it can make C's string handling a lot *faster* than doing the same thing in a high-level language – so there at least is some kind of advantage in return for all the extra programmer effort.

7.1. The standard library is not always your friend

Something worth remembering about string handling in C is that although the standard library provides you with a set of functions to do typical things (e.g. copying a string, finding its length, comparing two strings), it's not always sensible to do a particular job by using the library function for that job. It can often be worth thinking about *how* the library function will work, and noticing that in some situations there's a way to avoid doing unnecessary work.

For example, the standard library provides a function called '`strcat`', which concatenates one string on to the end of another. This doesn't work as I describe above (allocating new memory to put the combined string in), but instead it expects you to have spare space after the end of the first string. For example, if you already had room for 100 characters of data, and what's actually stored in those 100 characters is the string 'hello' followed by a terminating zero byte, then there's plenty of room to add more stuff on the end without overflowing your buffer, so you could safely use `strcat` to add the word 'world' on to the end.

So, suppose you have an enormous buffer already allocated, and you want to concatenate hundreds of little strings together. You might naturally suppose that the best way to do that is to start with an empty string and to repeatedly call `strcat` to add another little bit on to the end of it. But in fact, that's a slow way to do the job: each time you call `strcat`, it has to start at the *beginning* of the combined string (because that's the only thing you gave it a pointer to), count all the way along it to find the zero byte at the end, and then when it finds it, write the new string into the buffer. So if you combine lots of tiny strings this way, you'll spend most of your time walking repeatedly along the finished part.

A better approach is to keep a pointer to the *end* of the string, i.e. always pointing at the terminating zero byte. Every time you add an extra little bit to the string, you just copy it to where your pointer currently points, and then you advance the pointer until it finds the new terminating zero. Then you don't have to keep retracing your steps. So the string-concatenation function provided by the standard library is not always the best way to concatenate strings.

(Of course, in this scenario, you *also* have to make sure you don't overrun your buffer, either by tracking how much space you have left and stopping if you're about to go over, or by counting up the total length of all the strings in the first place and allocating the right amount beforehand.)

8. No object orientation

Object orientation is pretty standard these days: most modern languages have it in some form. C does not, because it predates that being true.

The closest thing in C to a class type is a structure, which is effectively a class without any methods – just a collection of data items, clustered together in memory so they can be conveniently treated as a unit, but there's no support provided for defining functions that go along with that data, and also there's no direct language support for data-hiding ('`private`' or '`protected`').

So, how do you solve problems in C that users of other languages would do with classes?

One very common pattern in C is just to do the same thing *unofficially*. Define a structure type containing your data fields, and then write a set of C functions which take a pointer to your structure type as one argument, and can read and write the data fields in order to implement the operations provided by the 'class'. A simple class in Java is really no different from that; it's just that where a Java programmer writes '`myclass.doThing(1, 2)`', a C programmer writes something more like '`do_thing(&myclass, 1, 2)`'.

Although there's no hard guarantee of data hiding, you can do *something* about it: if your program is big enough to be divided into multiple source files, then you can put the definition of your structure type and the implementations of all the 'methods' in one source file, so that no other file can see the structure definition. That will prevent other parts of the program from reading the fields of the structure in practice, because they don't have access to the information about what the fields all are and where they live in the structure. Of course, this being C, it's *possible* to do uncontrolled writes to the structure regardless (just convert the structure pointer into some other kind of pointer and then write into the memory it points at), but sensible programmers won't do that, because the results will almost certainly not be useful.

All of that works fine as long as you don't want to use inheritance, or Java-style interfaces, or polymorphism. In that situation, you would typically do something in C which mimics the underlying mechanism by which high-level languages implement that kind of feature: you'd define your structure type to contain some *pointers to functions*, and implement certain class operations by extracting one of those pointers from the structure and calling the function it points to. That way, different instances of the same structure type could

behave differently, by having their function pointer fields point at different functions.

In other words, you can do anything in C that a higher-level language's object orientation system provides. You just have to do it by hand, rather than having it done automatically for you.

More importantly, you don't get the error checking. In a high-level language, you can typically take a reference to a derived class and treat it as if it was a reference to the base class, but if you try to do the same with a class that *isn't* a derived class of that base, you'll get a compiler error to let you know you've made a mistake. In C, there's no selective type checking like that: you can convert any structure pointer to a different kind of structure pointer if you write an explicit cast operator to indicate to the compiler 'don't worry, I know what I'm doing', but if you do that, the compiler will *never* complain, no matter what two pointer types you convert between. So you can't make it diagnose only the conversions that are semantically wrong; instead, you have to use a lot of self-discipline to make sure you don't make those mistakes in the first place.

9. The preprocessor

An unusual feature of C, not found in more modern languages (unless you count C++), is its preprocessor. Before being compiled, every C source file is put through a conceptually separate textual translator, which responds to special directives beginning with '`#`' on lines by themselves and uses them to transform the source code. The output of the preprocessor still looks like C source code, but with pieces removed or added and some words replaced by other things; then that output goes to the 'real' compiler, which turns it into machine code.

The main preprocessor directives are: '`#include`', '`#define`', and a set of conditional directives '`#if`', '`#else`', '`#elif`' and '`#endif`'.

9.1. `#include`

`#include`'s job is to cause another file of C code to be copied into the main one. This is usually used as part of C's system for breaking a program up into modules: in order to refer to a function or variable in another module, the compiler needs to know what type the variable is, or what types of arguments and return values the function expects. Typically you don't `#include` a file containing the actual *definitions* of functions; instead, you include a file that just gives that information on its own. For example, if you had two source files '`main.c`' and '`foo.c`', and `main.c` needs to call a function `foo()` which is defined in `foo.c`, you might write a third file '`foo.h`' saying something like this:

```
int foo(int a, char *b);
```

That *declares* the function, meaning that it lets the compiler know that a function with that name exists and that it takes two arguments of particular types and returns an `int`, but it does not *define* the function (meaning to provide the actual code showing what the function does). Then, in `foo.c`, you'd repeat basically the same line of code, but replace the trailing '`;`' with the body of the function:

```
int foo(int a, char *b)
{
    return a * strlen(b) + 1;
}
```

And finally, in `main.c`, you'd include `foo.h` in order to be able to call the funtion:

```
#include "foo.h"

int main(int argc, char **argv)
{
    printf("The answer is %d\n", foo(argc, argv[0]));
}
```

(In fact, it's usually a good idea to include `foo.h` in `foo.c` itself as well as in other modules that need to use the functions. That way, if you make a mistake and write different argument types in the function definition in `foo.c` and the declaration in `foo.c`, the compiler will warn you that they don't match.)

9.2. `#define`

`#define` allows you to define 'macros', which are pieces of text that the preprocessor substitutes for other text whenever it sees them in the source code. Macros can look like individual words, or they can look like function calls. For example:

```
#define ARRAYSIZE 1000
#define TRIANGLE(n) (n*(n+1)/2)

int array[ARRAYSIZE];
int red_snooker_balls = TRIANGLE(5);
```

After the preprocessor has done its job, the `#define` statements themselves have been removed, and everywhere the macros appear elsewhere they will have been substituted for whatever they were defined to be. So this source file will turn into:

```
int array[1000];
int red_snooker_balls = (5*(5+1)/2);
```

and then the compiler in turn will actually do the arithmetic to turn the second definition into the number 15.

It's important to note that all of this substitution is done *textually*, with no understanding of how C works. So, for example, one of my definitions above contains a deliberate mistake: `TRIANGLE` will go wrong if you write the following.

```
    a = TRIANGLE(b+c);
```

To work out what would happen to this, go back to the definition of the macro `TRIANGLE`, and substitute the text 'b+c' everywhere the definition used the name of the macro parameter '`n`'. The preprocessor will transform it into this:

```
    a = (b+c*(b+c+1)/2);
```

But now this doesn't mean what you might have expected it to mean, because of operator precedence: as in most languages, C interprets '`b + c * stuff`' to mean multiplying `c` by `stuff` and then adding `b` to the result, whereas the macro definition clearly wanted '`(b+c) * stuff`'. So macros can be dangerous if not used carefully, because they don't have to respect the syntactic integrity of their parameters – as here, where what looked like an unambiguous command to add `b` to `c` turned into something entirely different by the time the code was generated. This is one reason why it's conventional to write macro names in capitals – to warn the programmer that they might do strange things.

This unchecked nature can sometimes be used to deliberate effect. If you do it right, it's possible to write C macros containing partial statements and unbalanced brackets in such a way as to extend the capabilities of the language and let you pretend it has all sorts of useful features that the C designers didn't think to put in. However, I won't go into that here; the *usual* uses for the preprocessor are the sorts of thing you see above. Just be aware that if you're reading someone else's C code and it uses a control construction that you don't recognise, or it looks as if it's violating the structure of the language in some way, it's worth checking to see if there's a macro somewhere which makes the strange-looking code turn into legal code by the time the main compiler sees it.

You might be wondering why it wasn't more sensible to define '`ARRAYSIZE`' as a constant and '`TRIANGLE`' as a function, something like this:

```
const int ARRAYSIZE = 1000;
int TRIANGLE(int n) { return n*(n+1)/2; }
```

The answer is that if you do that, then those definitions can't be used in some contexts. In the example above, I used '`ARRAYSIZE`' to set the number of elements in an array variable, and '`TRIANGLE`' to set the initial value of a global variable. In both cases, the compiler needs to be able to know the right number at compile time rather than waiting until the compiled code is run, and it can't do that if you define them like this: `TRIANGLE` is a function call, which means running the compiled code, and `ARRAYSIZE` requires looking in the piece of memory where the variable is stored. (Yes, even if it's declared as '`const`', for annoying technical reasons.)

For this sort of reason, a lot of constants and simple functions of this kind tend to be written using the preprocessor rather than in the main C language.

9.3. `#if`

Finally, `#if` and its friends allow you to 'conditionally compile' code: that is, to decide at preprocessing time whether to include a piece of code in the output program, depending on any criterion the preprocessor has the ability to judge. A typical use for this would be to write a program which *mostly* looks the same on several operating systems, but in one particular place where things have to be done differently on (say) Windows and Linux, there's a `#if` segment which tells the preprocessor to choose between the Windows and the Linux implementations of a function depending on what OS it's compiling for. You might write something like this, for example:

```
void clean_up_temp_file(void)
{
#if defined _WINDOWS
```

```
    DeleteFile("C:\\TEMP\\TEMPFILE.DAT");
#elif defined linux
    unlink("/tmp/tempfile.dat");
#else
#error Unrecognised platform
#endif
}
```

This would cause the call to the Windows operating system function '`DeleteFile`' to be included in the program when a Windows compiler compiles it, because the Windows C compiler's preprocessor defines the special macro '`_WINDOWS`' to allow this kind of decision-making in the preprocessor. On Linux, on the other hand, the macro '`linux`' is defined and so the function call to the Linux function '`unlink`' would be used instead. And if you try to compile the same program on a platform which defines *neither* of those macros, then the '`#error`' directive (another feature of the preprocessor) ensures that compilation fails, so that you don't get as far as running the program before realising you left something out.

10. So why is C like this, anyway?

You're probably thinking, by now, that C sounds like a horrible language to work in. It forces you to do by hand a lot of things you're used to having done for you automatically; it constantly threatens you with unrecoverably weird behaviour, hard-to-find bugs, and dangerous security holes if you put one foot across any of a large number of completely invisible lines that neither the compiler nor the runtime will help you to avoid; and, for goodness' sake, it can't even handle *strings* properly. How could anyone have designed a language that bad?

To a large extent, the answer is: C is that way because *reality* is that way. C is a low-level language, which means that the way things are done in C is very similar to the way they're done by the computer itself. If you were writing machine code, you'd find that most of the discussion above was just as true as it is in C: strings really *are* very difficult to handle efficiently (and high-level languages only hide that difficulty, they don't remove it), pointer dereferences *are* always prone to that kind of problem if you don't either code defensively or avoid making any mistakes, and so on.

(That doesn't explain *all* of C's curious design. The undefined-behaviour problem with integer overflow wouldn't happen in machine code; that's a consequence of C needing to run fast on lots of very different kinds of computer, which is a problem machine code doesn't even *try* to solve. And there's no simple excuse for the preprocessor; I don't know exactly why that exists, but my guess is that back in the 1970s it was an easy way to get at least an approximation to several desirable language features without having to complicate the actual compiler. These days compilers don't have to fit into such small computers, so we don't mind complicating them a lot more.)

A key feature of C is that it needs very little 'run-time support', by which I mean not just library functions your program can call if it wants to (C does have those) but library code which the program can't run at all without. Higher-level language features like garbage collection, bounds checking and exceptions all need complicated library code to support them, and all that library code has to be written in *some* language – and you can't write Java's garbage collector in Java, because you need to have the garbage collector already before you can even run Java code. So one niche in which C is still important is that it's a language in which it's possible to write the supporting code that high-level languages need to run in the first place. The Python interpreter is written in C, for example.

C can also be extremely fast. As a direct result of leaving out all the safety checks that other languages include, C code can run faster, because you can judge for yourself which of the safety checks are actually necessary, and not bother with the rest. Of course, make one mistake and you've had it – but that's just like the rest of C…

But those aren't the reasons why *most* C code is in C. Mostly, C is important simply because lots of code was written in it before safer languages gained momentum, and now lock-in and network effects mean that C (or C++) can still be the path of least resistance – if you need to work with existing libraries of code that have C interfaces, or reuse and adapt existing programs that were written in C, then naturally you'll have to write in C too, and so the cycle continues.

OPENCONTENT

Home | Open Content License v1.0 | Open Publication License v1.0

OpenContent License (OPL)

Version 1.0, July 14, 1998.

This document outlines the principles underlying the OpenContent (OC) movement and may be redistributed provided it remains unaltered. For legal purposes, this document is the license under which OpenContent is made available for use.

The original version of this document may be found at http://opencontent.org/opl.shtml

LICENSE

Terms and Conditions for Copying, Distributing, and Modifying

Items other than copying, distributing, and modifying the Content with which this license was distributed (such as using, etc.) are outside the scope of this license.

1. You may copy and distribute exact replicas of the OpenContent (OC) as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the OC a copy of this License along with the OC. You may at your option charge a fee for the media and/or handling involved in creating a unique copy of the OC for use offline, you may at your option offer instructional support for the OC in exchange for a fee, or you may at your option offer warranty in exchange for a fee. You may not charge a fee for the OC itself. You may not charge a fee for the sole service of providing access to and/or use of the OC via a network (e.g. the Internet), whether it be via the world wide web, FTP, or any other method.

2. You may modify your copy or copies of the OpenContent or any portion of it, thus forming works based on the Content, and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified content to carry prominent notices stating that you changed it, the exact nature and content of the changes, and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the OC or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License, unless otherwise permitted under applicable Fair Use law.

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the OC, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the OC, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it. Exceptions are made to this requirement to release modified works free of charge under this license only in compliance with Fair Use law where applicable.

3. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to copy, distribute or modify the OC. These actions are prohibited by law if you do not accept this License. Therefore, by distributing or translating the OC, or by deriving works herefrom, you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or translating the OC.

NO WARRANTY

4. BECAUSE THE OPENCONTENT (OC) IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE OC, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE OC "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK OF USE OF THE OC IS WITH YOU. SHOULD THE OC PROVE FAULTY, INACCURATE, OR OTHERWISE UNACCEPTABLE YOU ASSUME THE COST OF ALL NECESSARY REPAIR OR CORRECTION.

5. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MIRROR AND/OR REDISTRIBUTE THE OC AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE OC, EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Home | Open Content License v1.0 | Open Publication License v1.0

OPENCONTENT