Lecture 07: C Basics

C++ vs C

If you have any idea about the origin of C++ you should know that it is based on C. In essence, C++ is often thought of as 'C' with objects. This is true, but only in the simplest sense. If you go into programming in C as if you already know it (because you think you are good at C++… beware this belief) you will waste a lot of your own time. C and C++ are not that similar. This is not really because of syntax or compilation rules or anything like that. Like Bash, programming in C requires a different thought process to programming in C++. You cannot just translate everything you can do in C++ into C blindly.

C can do almost everything C++ can do. This might make you think that the difference is just a few 'conveniences' like libraries, or syntax shortcuts. Again, this is true and at the same time it is not. C is different in some fairly fundamental ways, different enough to require 'fresh eyes'. So, I would recommend that you forget a lot of what you already know. I will point out when something is the same between the two. However, your first thought should be that they are not the same (only to be happily surprised) rather than the opposite (only to waste hours of time).

Booleans

In case you forgot, Booleans are values which contain true or false.

Booleans are a bit more complex in C than C++. Booleans in C do exist… called **_Bool** which I have never seen used. You can include **stdbool.h** which will allow the use of **bool** for Booleans. In general, though, mostly I have seen people use **int** instead of **bool**.

Simply put, any integer other than zero is true. If you think about it, zero is all zeros in byte form. This makes sense for false. For true, every other number has at least one '1' somewhere in the byte representation. If anything -1 is the most 'true' integer because it is all 1's in two's complement.

So:

```
if (27)
{
	// Do stuff
}
```

Is equivalent (mostly) to writing:

```
if (true)
{
	// Do stuff
}
```

Coming straight from Bash, this can be a bit upsetting (where zero means 'it worked' and non-zero means 'it didn't').

Casting

C++ has casting. Easy right?

```
float a = 1.5;
int b = (int)a;
```

Integer ‘b’ has the same value as ‘a’ but as an integer (so rounded down). This should be familiar. So, what is the problem. Well… ask yourself how often you cast in C++? Has it been for the above purpose, some easy maths?

You will find yourself using casting in every non-trivial C program you write. It is just an inevitable fact. We will see why this is the case later, which is why I am mentioning it at all. So, before we go there, let’s just review casting a little.

#1: Casting is dangerous.

The maximum integer is usually 2^{31}. This is a big number but nowhere near the maximum float ~10^{37}. If you cast from one to the other, it can be a big problem. Likewise, sign is not always preserved. Sometimes you can accidentally cast a negative number to a type which does not do negatives.

#2: We can cast pointers.

This is the big takeaway from casting. If you have not done this before, think about it for a moment. What does it even mean to cast an (int *) to a (float *)? What is actually happening? Inside the memory, there is just an address. One address should be as good as another so no real change in how the data is interpreted occurs… does it? The only real difference is when the pointer is de-referenced (i.e., *ptr). This is when type becomes relevant because the compiler needs to know how to interpret the bits in the address.

The other issue is with **size**. Each type can be represented in one or more bytes. Moving a pointer by +1 for an ‘int’ will not be the same for a larger data type. Imagine having an array of integers, with the starting integer defined by a pointer ‘my_ptr’. Let’s say they start at address 96 (in decimal for simplicity). If I ask for ‘my_ptr + 1’, where does this now point? Well, given an integer can be either 2 or 4 bytes (depending on operating system… another headache) we might reasonably expect that ‘my_ptr + 1’ will point at address 98 or 100 (depending on operating system). Likewise, if the array were instead of the ‘unsigned long’ type (8 bytes) we might expect ‘my_ptr + 1’ to point at address 104.

Memory Allocation

Simply put, C does not have **new** and **delete**. Now, this does not mean that C lacks a Stack and Heap (it has both). It simply means that we will need to use a new kind of syntax to achieve our dynamic memory requirements. It is quite ugly to use.

To replace **new** (or arguably what **new** replaced), C has three functions:

- **malloc**
- **calloc**
- **realloc**

Instead of **delete**, we have:

- **free**

So how do these work?

Well… let's start with **malloc**, which is the most common. It has a function signature of:

```
void * malloc (size_t size);
```

Lucky, we have already thought about casting a bit. I mean… what is a **void pointer** meant to be? The simplest explanation goes back to casting. All addresses are addresses. So, the type of pointer is often irrelevant (unless we dereference or use pointer arithmetic). In C, it was decided to create a single function (well three) for allocating memory, leaving the managing of the types to the user. Thus, a very common thing to see is:

```
int * a = (int *)(malloc(10 * sizeof(int)));
```

So, what is going on here (start with the red)? **malloc** isn't super smart. It allocates some bytes and then returns the address where the first of those bytes is allocated. Thus, the user needs to think about the type (such as **int**) and how many of said type they want. In the above code, I have allocated 10 **ints** worth of memory. This is where **sizeof** is super useful because as mentioned above, different operating systems will play differently with **ints**. By using **sizeof** this code will 'do the right thing' whether **int** is 2 bytes or 4 bytes. Obviously, if we were being lazy, we could just write:

```
int * a = (int *)(malloc(40));
```

This assumes that we know that an **int** is 4 bytes (which it mostly is). Now, in this second line (i.e., the red bit) we have a very common bit of syntax. Because **malloc** returns a void pointer (i.e., an address with no specific type) in order to use the memory in any useful way, we need to cast it to the type we care about (in this case an **int***).

Now, with a bit of thought, you can begin to see how C programs can be awful. This syntax is very flexible, which means that C code is very prone to people abusing that flexibility. It would be very easy to stuff-up the calculation of the bytes needed for allocation for example or cast it to the wrong type.

This brings us to **calloc** which has the feeling of being more robust.

```
void * calloc (size_t nitems, size_t size);
```

This is a bit cleaner. Now we can specify the number of items and the size of each item, and the computation is handled invisibly by the system rather than explicitly in our code.

```
int * a = (int *)(calloc(10, sizeof(int)));
```

calloc also has a neat feature whereby the memory you request is all zeros. Oh, had you not thought about that? What do you think **malloc** returns in the allocated memory? You can probably guess… not zeros. In fact, **malloc** does nothing to the memory allocated, so it is just random bits (ones and zeros). Actually, it is not strictly random, it is whatever was there before. For our purposes, it may as well be random (unless we are doing some hacking shenanigans).

So, you might think that **calloc** is nicer is a lot of ways. The syntax is cleaner, and the memory returned is preset to zeros. Unfortunately, this means that the computer will have to do a calculation (i.e., the multiplication) and then wipe the memory clean (change things to zeros). This means that **calloc** is guaranteed to be slower than **malloc**.

This is where the differences between C and C++ really show through. It is not syntax. It is a mindset. Object orientated programming (OOP) is not good because it is fast or saves space. OOP is good because it is scalable. Using objects making writing big programs easier because the data is wrapped up in nice, neat little packages (objects). The interactions of these packages can then be happily discussed (cars drive on roads, users make withdrawals from bank accounts). C… is lower level than this. C programs are not usually that big (there are exceptions). Most people do not choose C for big programs. C is for small programs like hardware drivers or operating systems (admittedly not that small) and other things which are designed to be small and fast. Thus, while the C++ programmer might lean towards **calloc** because it is inherently less buggy, the C programmer would mostly use **malloc** (it is faster).

To that end, it explains the existence of **realloc**. The signature is:

```
void * realloc (void * ptr, size_t size);
```

It takes a **void*** which is a bit curious. The name is a bit of a give-away as to what it does. Unsurprisingly, it stands for **realloc**ate. The pointer you give it will be the memory you wish to reallocate. In essence, the idea of **realloc** is change the size of some memory you already have. If you can imagine that you have an array of size 10 and you decide (perhaps based on something you learned/will learn in ADSA) you now need the array to be size 20. Well, deallocating 10 integers and then reallocating the same 10 integers plus another 20 integers is a complete waste of time. Well, sometimes you do not have to do this. Sometimes, coincidentally, the next 10 bytes of memory will be free for allocation. A simple analogy is if your house is too small, instead of knocking down your house and building a new one somewhere else, you notice that there is room behind your house to build an extension.

realloc does just this. The operating system will check if it can keep the memory in place and simply extend the array. If it is possible, it will do so, saving time. The old data will still be where it was, saving the hassle of copying it. Fortunately, even if there is not room, **realloc** will then copy the existing allocated data to the new location. For example, if I have an array with the numbers: 6, 2 and 4 and then **realloc** it to make it a six element array, I will get a new array: 6, 2, 4, ???, ???, ??? (because random memory). It may not be in the original location, but it may well be.

The last part (and perhaps the easiest) is **free**. It is easy enough to use:

```
free(my_ptr);
```

This will deallocate the memory just as you would with **delete**. Fortunately, it is smart enough to keep the details of how much memory is being deallocated, so the user does not have to keep track and ask it to delete the same number of bytes they allocated in the first place (which would not have surprised me). Of course, this will still have all the headaches of manual memory allocation such as not **free**ing memory twice, forgetting to **free** it at all etc.

Final Thoughts

As I mentioned above, you can probably see that C and C++ are a bit different from each other, in very subtle but fundamental ways. Thus, going forward just remember that your intuitions from C++ might be completely wrong. Just double check what you know.