Lecture 09: Parameters, Memory, INodes

Arrays as Function Parameters

As previously mentioned, C programs do not treat **return** values like you would in C++. Instead, functions normally take pointers/references as parameters and then modify the memory from those pointers. Let us consider an example:

In C++:

```
string domain = "adelaide.edu.au";
string url = "www." + domain;
```

What an odd example to use. Let us unpack it a bit. Did you know that '+' is a function? In C++ it is possible to redefine what '+' for different types. We call this **operator overloading**. The idea is that if you were to make a class that was say, an imaginary number (with a real part and an imaginary part). You could make '+' work for them the way you want (i.e. to add the real to real and the imaginary to imaginary). If you were a troll, you could make '+' do subtraction and '-' set everything to 13. I mean… why not.

So here, we are implicitly calling the '+' function (when we do the www + domain bit). This could be better read as (where concatenate is a function I made up that does the same thing):

```
string url = concatenate("www.", domain);
```

Now to the point. What you see here is that the function concatenate **returns** the nominal new string "www.adelaide.edu.au".

The equivalent C code would look like this:

```
char url[100] = "www.";
char domain[100] = "adelaide.edu.au";
strcat(url, domain);
```

There are a few immediate problems here. It is not as immediately obvious where the result is going to end up. In this case, **url** is the one to be updated and will contain "www.adelaide.edu.au". **domain** is added on the end. What does it return? In the case of **strcat**, it does actually return a pointer to the concatenated string. The point is, you already have said concatenated string in **url**. Moreover, the memory is not new. It is old.

Function Calling

If you have done a reasonable amount of programming, you are probably familiar with the idea of functions. Moreover, you should be able to think about functions calling other functions. For example, main can call another function (let us call it **f1**) and the **f1** might call **printf** or **scanf** or something else. So, when **f1** finishes… what happens? The simple answer is that it returns its return value to **main**… somehow.

What is happening?

The Stack

When you call a function (in this case a C function), the system needs to be able to freeze some information that it can come back to later. This information includes things like… what was in the registers and a location telling us where in the code we are. Otherwise when the function completed, the system would not know what to do next. When we call a function we also need some stereotypical locations for where the parameters should be stored and where the return value should be put when the function returns.

This is what we refer to as the stack. Now the locations of these fundamental function 'bits' differ from language to language and system to system. Most of the time it is not important unless you are writing a compiler, or a virtual machine, or an instruction set simulator, or trying to exploit a security flaw, or debug an array out of bounds problem…

Persistence

This stack-based function paradigm does have its drawbacks. Everything assigned to the stack during a function call will cease to exist when the function exits (except those **static** references). To get around this there are a few options.

1. Input and Output to files etc
2. Use the **return** value (well duh)
3. By using pointers and references as parameters and changing their value
4. **static**..
5. Dynamic memory

We have covered 1-4 already. Dynamic memory in functions is actually a lot like 2 and 3 put together.

The Heap

We can use the heap (i.e. #4, Dynamic Memory). As previously mentioned, instead of **new** and **delete**, we use **malloc**, **calloc**, **realloc** and **free()**. To take advantage of this in a function:

```
int * newint(int value)
{
      int * result = (int *) malloc(sizeof(int));
      * result = value;
      return result;
}
```

Here we have a function that basically just wraps around malloc and makes an equivalent to " = **new** int()" for C. Hooray! Of course, the general risk of using the heap is memory leaks.

A word on Memory Leaks

So, I tend to find a lot of people have a bad understanding of what a memory leak is. At its most fundamental it can be described thusly.

- I allocated memory on the heap. I have no pointers pointing at that memory.

Here is an example:

```
int *x;           // A pointer
int y = 4;        // Boring
x = newint(3);    // We make new memory
x = &y;           // Oh dear…
```

The issue here is that when we redirect **x** to **&y**, the memory we made using **newint** now has nothing pointing to it. It is over there in the heap without a way of finding it. If you cannot find it, you cannot **free** it.

Thus, you must watch your memory. This seems like it is an easy task. Never redirect a pointer from a bit of memory on the heap unless there is something else pointing at it…

But how do you know nothing is pointing at it? In programming, big bits of programs (like **main**) usually have some idea about where small parts of a program are. **main** calls functions, which calls functions which call functions. So, it is sort of at the top of the pile. Small parts of a program do not have the reverse relationship. A function really should not have any idea what it is being used for.

This leads to a problem when it comes to memory management. Can a low-level function **free** memory with no idea if the memory is being used by some 'higher level' program? Does the higher-level function then need to assume responsibility for deletion the moment it gains access to the memory? If so, how does it know it is the highest? How does it tell the lower-level function?

If you follow this train of thought, the concept of modularity (i.e. breaking things into parts) completely collapses. One solution to this is a paradigm of ownership. This way it is known who owns what and when it should be deleted.

Another option is garbage collection (in other languages). This basically checks to see if a bit of memory is being referenced and when the references disappear, it deletes the memory automatically.

String Functions

Now we are in C, we need to think a bit more carefully. Let us start with the basics. What is a string?

- A **null** terminated character array…

A **null character** (accessed using "\0") is a character that is simply used to tell computers to stop reading more bytes. Think of it a bit like how a full-stop ends a sentence. Like a full-stop, a **null character** is a character, so the string "hello" really has six characters, the normal five plus the **null character** ("hello\0").

With that in mind, here are some basic string functions:

`strcpy``(dest, src)`
- Copies the **src** into the **dest** (**dest** must be sufficiently big)

`strncpy``(dest, src, n)`
- As above but at most n bytes. At most 'n' bytes means that if a '\0' is encountered, it will stop there even if there are other characters after it. You cannot use strncpy to cheat '\0'.

`strdup``(const char * s)`
- Basically a malloc & strcpy (returns char*)

`strcat``(dest, src)`
- Concatenates, src to the end of dest (remember space)

INodes

Well, it had to be covered somewhere. Sometimes I think some of these lectures would be better served as mini 10-minute videos. </rant>

So, we have talked a bit about the **stack** and the **heap** and briefly mentioned **data** (where **static** and global variables may be found). There is also **code** (where the code is stored).

INodes is not a C topic, however. It is a Unix topic, so do not get that confused. Of course, we are running C on Unix… nevermind.

What is an INode (sounds like an exam question)

The Unix system is based on **INodes**. A single **INode** is a data structure which stores information about a file. Every file therefore has an **INode** associated with it. You might therefore think that the number of **INodes** is a dynamic value which is basically like a global variable called TOTAL_INODES which goes up and down whenever you create or delete a file.

Nope.

Let me explain. It turns out that the number of **INodes** on a file system is finite. This means that the file system is making a big statement about how many files it will ever be able to support. Normally this is not a big issue because files are reasonably big, and most people do not have that many. Yet, in the event that some strange person has decided to make a bajillion tiny files, it can begin to cause problems. It is possible to still have space available but be unable to allocate it because you have run out of **INodes**. A simple analogy would be if you had a giant block of land (like Adelaide) which had houses with addresses, but you had a finite number of street signs and house numbers. If you built your houses too small, you would run out of streets and house numbers before you ran out of land (normally it is the other way around).

So how do we inquire about **INodes**?

>> **df**

This unix command tells the user about the available disk space including the number of **INodes**. You can ask specifically for Megabytes/Gigabytes using “-h”, the number of **INodes** using “-i" and the total free space using “--total".

We can also use:

>> **stat**

This gives detailed information about the device, mode, **INODE!**, access modifiers etc. Actually it gives you nearly everything about a file.

INodes and the File System

So, it is all well and good that INodes are associated with files, but like… how? Well, let us talk briefly about the file system. In Unix it is divided into three parts.

- The **super block** (containing a bunch of general information). If I installed Unix on five different computers, the **super block** would look pretty similar.
- The **INode table**. Unsurprisingly, a table of INodes.
- **Data**. The stuff. For a movie, it would be the data associated with the file (think Gigabytes).

Here is the last piece of the puzzle. We have the **INode** table (which the file system knows about), and we know a file is associated with an **INode**. How is an **INode** associate with **data**?

The answer to this question is also the answer to: what exactly is in an INode?

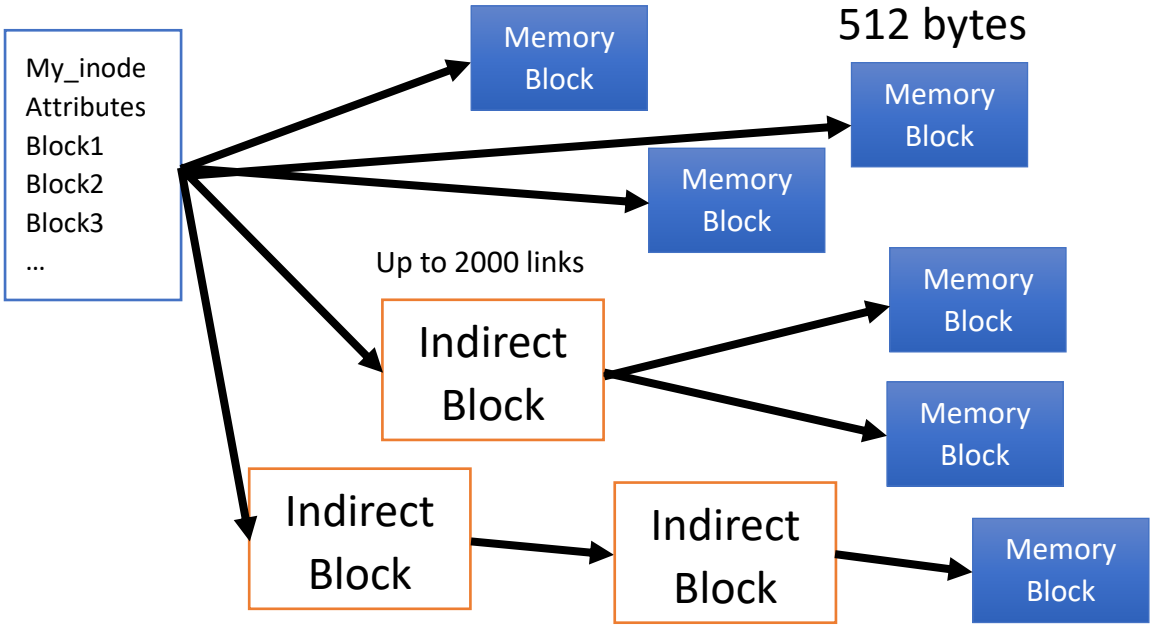
INodes have attributes within them (some usual stuff about themselves) and then a series of references to **Blocks**. A **Block** is a bit of data… in a block. Let us assume that a **Block** is finite and about 512 bytes. The **INode** can reference a bunch of **Blocks** (it has a little table of **Blocks** it can point at). So if I know the **INode**, I can find the associate data… except. How many **Blocks** can an INode reference? Is that a finite number? Yes. Then… if I have a 20GB blu-ray rip… how does that file get an INode? More importantly, how does the file system know where the 20GB of blu-ray is?

The answer is **Indirect Blocks**. So instead of pointing at **data** directly, you can point at an **Indirect Block** that points at **data**. If an INode can point at many Indirect Blocks and Indirect Blocks can point at many **Memory Blocks** (or even more Indirect Blocks) the amount of storage we can point at grows exponentially.

As an example a Direct-only INode can reference: 80KB

Via single indirect blocks: 16MB

Via double indirect blocks: 32GB



One natural consequence of using indirect blocks is that when the file system is accessing a big file, it is slower because it is making multiple hops (first here, then there).

Access Modifiers

Let us quickly talk a bit about **stat** and access modifiers. You will see when you run **stat** that it pops up with three timestamps: Access, Modify, Change. To explain what these words actually mean (because it is totally obvious how **modify** and **change** mean different things) let us use some examples.

If I open a file, but do not modify it (i.e. make something different to what it was before – ha ha, I didn't say change) then it would be true to say that the file has been **accessed** (use **cat** when in doubt). If I use **chmod**, it is true to say that the file has been **changed**.