Lecture 06: Functions and Subshells

Functions

You should already be familiar with the concept of a function from C++. Unfortunately, this is unhelpful when talking about Bash because functions in Bash have several key distinctions when compared to C++.

Let us consider a basic function in C++.

```
int sum(int a, int b)
{
      return a + b;
}
```

The idea of this function is that it computes something (in this case an integer) from some inputs. This differs from a procedure (a function that does not return anything) in a fundamental way. Yes, procedures and functions are technically different. A function is something which computes a result. It can meaningfully be assigned to the right side of an equation (i.e., a = function(x, y)). Integral to this operation is the **return** value which is the bit of memory ready to store the result of the function and is returned to the location where the function was called.

So, what happens in Bash? Is it the same? Consider the following Bash function.

```
return1()
{
      return 1;
}
```

If this were C++ the behaviour of this function would be obvious (and trivial). It would evaluate as one so if I were to write:

```
a=return1();
```

I could reasonably expect ‘a’ to contain 1. Except Bash is… different. In more Bash-centric syntax, would write:

```
a=$(return1)
echo $a
```

We might expect the terminal to print out a friendly ‘1’ for us. Alas no. What is going on here? Let us think about this again. What happens when we write:

```
a=$(ls)
echo $a
```

We would expect to get a list of directories printed to the screen. But does ‘ls’ ‘return’ all of this information? Not really. It prints it all out to stdout. It is little more than a lot of **echo** statements. So, when we assign ‘a’, we are not assigning it to the return value, but instead to the stdout output of the function. We can test this easily by making an equivalent Bash function.

```
return2()
{
      echo 1;
}
a=$(return2)
echo $a
```

This code behaves far more like what we would want… so is **echo** the equivalent of **return**? Not really. We can **echo** all day if we want, and **echo** is not the only Bash function which prints to stdout. So, a function is really like an encapsulated Bash Script in many ways. Firstly, it takes parameters the same way and secondly, it seems to deal with output in a very similar way. That being the case… what is **return** for? Like **exit**, **return** is used to return the error state of the function (this is nothing like C++). In general

- return 0: Means everything went well
- return 1+: Means something went poorly

There are a few places where this can be used to great effect, such as in **if** statements or **while loops**. Consider, what if we have a bit of code that is reliant on some other code having been run. This is especially true of hardware. For example, we could have a function usePrinter, but also checkPrinterOnline. We definitely don't want to call usePrinter if the printer is down, so we can use:

```
if checkPrinterOnline
then
      usePrinter
fi
```

This code is now dependent on the return value of checkPrinterOnline (whether it is a function that **returns** 0/1 or a script that **exits** 0/1). What this means in practice is that you will need to think differently when programming in Bash compared to C++. They are not the same thing with different syntaxes, they are tools designed for entirely different ways of conceiving problems.

So, what if you really want that **return** value. Clearly just knowing a bit of code failed is not always helpful. There may be one or more ways it could fail. Similarly, there could be dozens of different **exit** codes or **return** values. One way to retrieve an **exit code** is to use the '$?' operator. The '$?' operator contains the exit status of the last executed instruction.

```
a=$(return1)
echo $?
```

This would now print out '1', but do not get confused. The point is not to try to turn Bash into C++, the point is that it is useful to print out error codes sometimes. In theory then, almost every bit of Bash code you write should probably end in **exit 0**, just like your C++ main functions should end in **return** 0. Yes, that is what that weird bit of code is for, did your C++ program end gracefully or did it crash midway through.

Now this '$?' is a bit finicky. Mostly because it changes a lot. Consider:

```
a=$(return1)
echo $?
echo $?
```

Instead of printing ‘1’, ‘1’, this will instead print ‘1’, ‘0’. While ‘return1’ happily returns ‘1’ as its output, after calling ‘echo $?’, the ‘$?’ value will be replaced with the **exit code** of **echo** which in this case is a success (echo did not crash) and hence returns ‘0’.

This also leads to a fundamental difference between C++ and Bash which completely broke my brain the first time I thought about it. So… in C++, zero is false and anything but zero is true… but in Bash, zero means success and anything but zero means failure which means **if statements** kind of work backwards to each other. Within the arithmetic (()) **if statements** zero is still false, but if you just have:

```
if ./bash_script
then
      # stuff
fi
```

Then **# stuff** will only execute if ‘bash_script’ returns zero. Quirky.

Local Variables

So, if this were not enough, Bash also has a very lazy conception of scope. Normally, one might expect variables defined within a function to be scoped (in their own private universe of the function). In Bash (depressingly) this is not the case.

```
function normalBash()
{
      a=1
}
b=$(normalBash)
echo $a
```

Intuitively, I would expect this to print… nothing (I am used to Bash not crashing when variables are unassigned). Counterintuitively, it does not. It will print ‘1’ just as if I had assigned ‘a’ outside the function. In this way, the default behaviour of Bash is to make everything global. This is… ugly.

<rant> So scope has a point. The idea is that there are a limited number of useful names for variables in the world and it would be nice to be able to re-use them whenever you need. Things like ‘x’, ‘y’, ‘time’, ‘sum’, ‘speed’ are all variable names that give a good indication about what they contain. If functions overwrite global scope, then these names can only be used once rather than when needed within the scope of a function. Bash seems insane to me. </rant>

Fortunately, it is not the end of the world. Bash has a work-around using the **local** keyword. A variable defined as **local** within a Bash function works like any normal variable defined in a function in most other languages. It is its own thing and has no effect on the world outside the function.

Final Word of Functions

As I mentioned previously, Bash functions work a lot like scripts. You can add arguments at will and access them using $1, $@ etc.

Given all the strangeness that functions in Bash entail, I have found very little use for them in practice. They occasionally pop up in cases where you need to repeat a small piece of code repetitively in a larger program but if it has not already sunk in. Bash is not really designed with large programs in mind. The default of global variables in functions is evidence of this. Bash is effectively a hack built upon another hack built upon another hack. People have tried to straighten out some of the hackiness of the language but that is what it is.

Subshells

Stop. Do not read this if you are tired. Understanding Subshells is not for tired people. We are about to embark on Matrix within a Matrix territory (or Shell-ception). You have been warned.

We have actually already been using Subshells a lot. We have just used them in their most trivial form, to do one operation.

```
a=$(ls)
```

This code invokes a subshell. What does this mean? First let us take a step back and think about processes again.

Processes and Process IDs

There are two environment variables which are useful for understanding subshells. These are:

- $PPID The current process ID is (each process has a unique ID)
- $BASH_SUBSHELL The current bash subshell level (???)

Testing Subshells

So ,what about the following code, what might we expect it to do?

```
echo "test1"
a=$(echo "test2")
echo $a
```

The simple answer is print 'test1' then print 'test2'. So far so good, but again, what is the point. Let us try looking at $PPIDs.

```
echo "test1 $PPID"
a=$(echo "test2 $PPID")
echo $a
```

There should be no difference (other than the extra process ID appended to each line). The PPIDs will be the same because it is the same process.

```
echo "test1 $BASH_SUBSHELL"
a=$(echo "test2 $BASH_SUBSHELL")
echo $a
```

Here there will be a difference. The first one will be "test1 0" and the second "test2 1". By invoking a subshell using '$(??)', we have changed the subshell level from 0 to 1. As a further experiment we can be even sillier.

```
echo "test1 $BASH_SUBSHELL"
a=$(echo "test3 $BASH_SUBSHELL;
    "b=$(echo "test3 $BASH_SUBSHELL");
    echo $b)
echo $a
```

Here we will get down to subshell level 2 (or up I guess as the numbers are ascending?). The use of a single subshell is pretty obvious. We have been using it a fair bit up until now. Except we still have not drilled down into what a subshell really is. Well, let us consider a few extra aspects of subshells. Unlike functions, subshells do have their own scope. Variables defined within a subshell are not accessible by the parent. Consider the following:

```
a=$(b=1)
echo "$a, $b"
```

This prints nothing! The variable ‘b’ is defined within the subshell and thus is not defined outside it. Likewise, the statement ‘b=1’ doesn’t send anything to stdout so ‘a’ is empty.

A Final Word on Subshells

There are a lot of things you can do with subshells. They are incredibly useful. However, they are again, something to use with caution. While it might be tempting to put subshells within subshells within subshells, it leads to code which is difficult to read and understand.