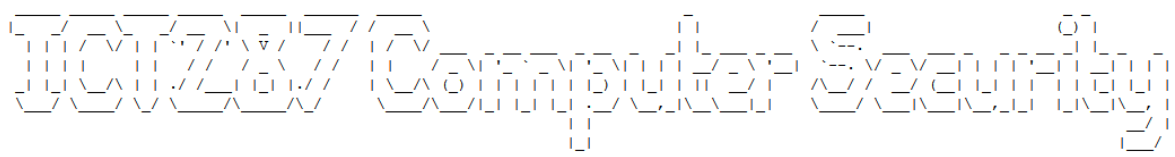

Tutorial Three: Introduction to C Pointers and Memory Allocation

This is a mini lab / crash course to introduce you to pointers and memory allocation in C. For more in-depth tutorials on the subject see the readings on LMS.

It is assumed that you are familiar with the basics of C as this is one of the prerequisites of this unit.

Hello World

Firstly, check that the C compiler (gcc) is working fine with a hello world program. In this program we have one integer variable and one character array.

```
#include <stdio.h>
#include <string.h>

int main(void)

{

   int var = 1;

   char buf[64];

   sprintf(buf, "Hello World, var=%i\n", var);

   printf("%s", buf);

   return 0;

}
```

Make sure you understand the above program. Save the code with a name like hello_world.c and compile it with gcc.

```
$ gcc hello_world.c -o hello_world
```

When you run the resulting executable (./hello_world), the results should be self-explanatory.

Addresses of Variables

Every variable in C is stored at some memory location (address). For characters, integers and floats we can get the memory location by simply putting an & (ampersand) in front of the variable name. For arrays it is a little more complicated. For arrays the variable name without square brackets is the pointer to the array (and also the pointer to the first element of the array). You can also use the ampersand to point to the locations of specific elements in the array. For example, buf is the address of the array (and first character) and &buf[1] is the address of the second character.

Let's print out the memory locations for a few things.

Task 1: Add the following lines to you program at a sensible location. Compile and run the program.

```
printf("Address of var: %p\n", &var);

printf("Address of buf: %p\n", buf);

printf("Address of 2nd char in buf: %p\n", &buf[1]);
```

You will see that unsurprisingly the address of the second character is the address of the first character in buf + 1.

Task 2: Modify the program to also print out the address of the sixth character of the array. Is it at the address you would expect?

Pointers

Pointers are simply variables that store addresses to other variables. Pointers are declared by defining the type of the variable the pointer points to, followed by a * (asterisk) and then the name of the pointer variable.

A pointer that doesn't point to anything yet is initialised with the special value NULL.

To access the content of a variable a pointer points to, use the variable name with an asterisk in front of it. For example, if iptr is a pointer to var, then *iptr is the value of var.

The following program illustrates these concepts with two pointers, one for var and one for buf.

Task 3: Compile and run the following program.

```
#include <stdio.h>
#include <string.h>

int main(void)

{

   int var = 1;

   int *iptr = NULL;

   char buf[64];

   char *cptr = NULL;

   iptr = &var;

   cptr = buf;

   sprintf(cptr, "Hello World, var=%i\n", *iptr);

   printf("%s", cptr);

   return 0;

}
```

Note that the output of the program is exactly the same as the output of the first program, but now we used the pointers to construct the string rather than using the variables var and buf directly.

Task 4: Add the printf lines to print the pointers you used for program 1 to program 2, but this time do not refer to var and buf directly but use the pointer variables in the printf statements, like this.

```
   printf("Address of var: %p\n", iptr);

   …
```

Task 5: Before printing var (before the sprintf), increment the value of var by one, but without using var directly. You are only allowed to use the iptr variable.

Pointer Arithmetic

You can do arithmetic with pointers but this is out of scope of this lab (research this in your own time).

Why Pointers?

Why are pointers needed? Especially given that some other programming languages don't have pointers.

C (and C++) uses call-by-value, which means that when you call a function with a variable as parameter the value of that variable is copied into the value of the local variable that exists only inside the function. Code inside the function cannot change the value of the original variable.

By passing a pointer into a function we can access the value of the original variable and also change it.

Further, it is much more efficient to pass a pointer to a large data structure into a function rather than copy the large data structure when passing it.

Memory Allocation

Now we want to rewrite the second program to allocate memory for the array dynamically. Allocating memory dynamically means the memory is allocated during the run time of the program rather than statically during compile time.

Dynamically allocated memory is on the heap whereas statically allocated memory is on the stack (see lecture).

Instead of an array we declare a pointer to a character variable at the start of the program. Then the memory is allocated with the malloc() function and assigned to the pointer. The allocated memory is freed at the end of the program with free().

Task 6: Compile and run the program below.

The output should be unchanged, but now the memory is allocated during compile time and buf is stored on the heap and not the stack.

Task 7: What would happen if the malloc() line in the following program is omitted? Compile and run the program without that line and see whether you get the expected result.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)

{

   int var = 1;

   int *iptr = &var;

   char *buf = NULL;

   buf = malloc(64);

   sprintf(buf, "Hello World, var=%i\n", *iptr);

   printf("%s", buf);

   free(buf);

   return 0;

}
```