csc2404 Practical

Introduction to Pointers and Arrays

Week 2

Pointers

Reading A :

Chapter 1 of "A Tutorial on Pointers and Arrays in C" by *Ted Jensen*

Chapter 7 of "An Introduction to the C Programming Language and Software Design" by *Tim Bailey* (Section 7.8 can be skipped) or
Chapter 9 of "The GNU C Programming Tutorial" by *Mark Burgess & Ron Hale-Evens*

It is convenient (and essential) in many instances for the programmer to be able to interact directly with memory. Because, the natural way of identifying values in memory to the CPU is via their address—the use of variable names is purely for the convenience of the programmer and as stated before no variable name survives the compiling process—all are replaced by their memory address.

Consider the following program `assembly.c`:

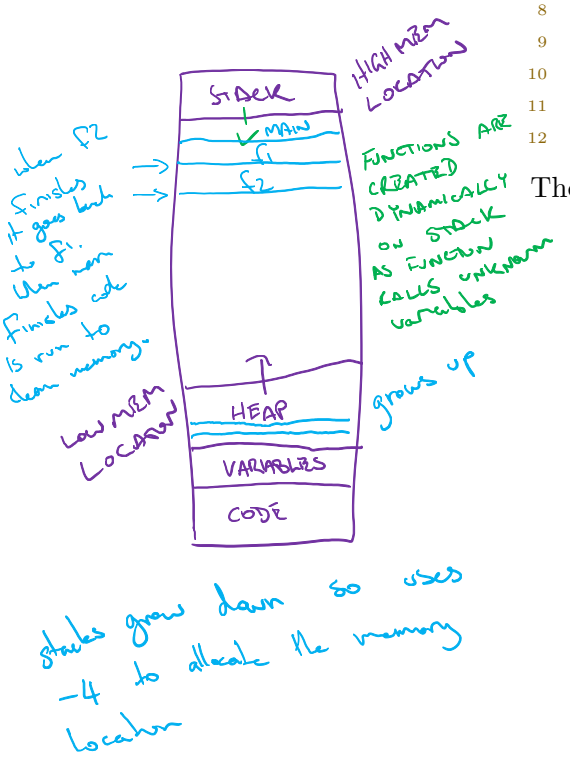
```
1 int main()
2 {
3   int x=10, y=15;
4   return 0;
5 }
```

when this is compiled it is first turned into assembly code. To see the assembly output—compile the above code with the command:

```
gcc -S assembly.c
```

the output will be placed in the file `assembly.s`. The output is (with some initialisation lines removed for clarity)

```
1     .file   "assembly.c"
2     .text
3     .globl  main
4     .type   main, @function
5 main:
```




```
6    pushq   %rbp
7    movq    %rsp, %rbp
8    movl    $10, -4(%rbp)
9    movl    $15, -8(%rbp)
10   movl    $0, %eax
11   popq    %rbp
12   ret
```

The key features of this are:

- The variable names that where used in the code have been removed. Variable values are recognised only by their addresses in memory.

  `movl  $10, -4(%rbp)`
  move the long value `10` to the address `rbp-4` (subtract 4 from the address held in the Base Pointer register) this will be the memory location of what in the code was called `x`.

  `movl  $15, -8(%rbp)`
  move the long value `15` to the address `rbp-8` this will be the memory location of what in the code was called `y`.

- To do any work, the CPU must move values or addresses into registers (held in the CPU)—it can only work on values in registers. There are about 24 registers in a CPU (this varies with architecture) The vast majority of assembly code generated from a high-level language is moving numbers between memory and registers.

- The two special registers rbp (base pointer) and rsp (stack pointer) handle the call and return mechanisms of function calls.

  `pushq  %rbp` save the current position in the calling code (the current frame pointer. The pointer to memory that contains all the information about the current running function)

  `movq  %rsp, %rbp` move the stack pointer (the frame where information for this just called function is stored) to the current frame pointer. All local variables are now accessed as an offset from the base pointer.

  The function return value is returned to the calling program via register `%eax`. When a function is called the information about the function (passed parameters, local variables) are pushed onto the process's "stack". When the function returns, the variables are popped from the stack to restore the calling program status. We will be coming back to process memory structure later in the course.

- All programs are contained in functions. In C the first function to start running is the function `main()`. It is treated like any other function—it is just the first function called.

Exercise 1: Create a file containing the C code above and produce the assembler code above. Modify the code by adding the line:

```
int z = x + y;
```

and reproduce the assembly code.

How is it different. Where is the value for `z` stored? How are the two numbers added?

Code: ptr1.c

Exercise 2: Consider the following code `ptr1.c`:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5  int j = 1;
6  int k = 2;
7  int *ptr = &k;
8
9  printf("j has the value %d and its address is %p\n",
10         j, (void *)&j);
11 printf("k has the value %d and its address is %p\n",
12         k, (void *)&k);
13 printf("ptr has the value %p and its address is %p\n",
14         ptr, (void *)&ptr);
15 printf("The integer pointed to by ptr is %d\n", *ptr);
16 printf("The size of k is %d bytes\n", sizeof(int));
17 printf("The size of ptr is %d bytes\n", sizeof(int *));
18
19 return 0;
20 }
```

Compile and run this program.

What is the size of the memory storage reserved for variable `ptr`? It will be either 4 bytes or 8 bytes—what does this tell you about your machines architecture (or the kernel running on your machine)?

What do the returned addresses tell you about how memory is allocated by the compiler when variables are declared?

Exercise 3: Add a float variable and a float pointer to the code `ptr1.c`. What happens if you try and assign the address of an integer variable to a float pointer?

Exercise 4: Add the following lines at the end of the `ptr1.c` code:

```
1 ptr = ptr + 1;
2 printf("ptr has the value %p and its address is %p\n",
3         ptr, (void *)&ptr);
4 printf("The integer pointed to by ptr is %d\n", *ptr);
```

By what amount was the value of `ptr` incremented? Why?

Why is this exercise stupid and dangerous for the unwary? What would happen if `ptr` was incremented by 10, 100 or 1000?

Pointers and Function Parameters

In C the parameters passed to a function are passed by value—that is, the parameters in the function are given the same values as the passed parameters used in calling the function (that is—the values are copied to different memory locations—in the stack frame for the function). For example:

```
1 float sum(float a, float b)
2 {
3   return a+b;
4 }
5
6 float x,y,z;
7 x=3.1;
8 y=7.149;
9 s=sum(x,y);
```

When the `sum` function runs the parameters `a` and `b` are initialised to the same values as the variables passed to the function when it was called—in this case `3.1` and `7.149` respectively.

This means that passed values are local to the function (stored in the functions stack frame) and the function cannot alter the variables in the calling function—but what if you want to? Then the value passed to the function must be the parameter's address not its value. For example: Consider the following code `ptr1.c`:

Code: function.c

```
1 #include <stdio.h>
2
3 void inc(int *i)
4 {
5   printf("inc: passed parameter value: %p\n", i);
6   *i = *i +1;
7 }
8
9
10 int main()
11 {
12   int k = 4;
13
14   printf("main: k address %p, k value %d\n", &k, k);
15   inc(&k);
16   printf("main: k address %p, k value %d\n", &k, k);
17 }
```

The value passed to the function `inc()` is the address of the variable `k` not its value—by using the address in the function the actual value of `k` can be changed by the function (but of course not its address as that has been passed as a value)!

Exercise 5: Consider the following code:

```
1 int i = 2;
2 int *p = &i;
3
4 *p++;
```

What is the value of `i`? What is happening and how can it be fixed?

To answer these questions place the above lines of code in a program— compile and run it.

Arrays

Reading B : Chapters 2-4 "A Tutorial on Pointers and Arrays in C" by *Ted Jensen*

Chapter 8 of "An Introduction to the C Programming Language and Software Design" by *Tim Bailey* or
Chapter 14 of "The GNU C Programming Tutorial" by *Mark Burgess & Ron Hale-Evens*

It is important to understand that an array is a group of values of the same type that occupy a *contiguous* region of memory. This has a number of implications and consequences:

- the simplest way to reference a value in an array is as an offset from the start of the array. Note: C arrays start with index 0.
- since an array is a contiguous block of memory containing the same type (each element in the array requires the same amount of memory) then a pointer can be used to access any element in an array as long as the address of the first element is known.
- Accessing a block of values by array indices is exactly the same as accessing it by pointers. For example:

```
int val[10];
int *ptr = &val[0];

val[5] = 7;

printf("Val[5]=%d\n",val[5]);
printf("*(ptr+5)=%d\n",*(ptr+5));
```

  will print 7 twice.

- When a pointer is incremented, the pointer's value is increased by the size of the pointer type. For example:

```
int val[10];
int *ptr = val;

ptr = ptr + 2;
```

Means that the variable `ptr` has its value increased by 8, incremented two integer positions in memory. It now points to the value at `val[2]`.

- In C arrays are an example of using pointers! The name of the array also doubles as a pointer to the start of the array. The array index is the address offset from the start of the array. All this is possible because the array is stored in contiguous memory!

Exercise 6: Write a program incorporating the code above. Run the program and show that 7 is printed twice.

Add the following two lines to the end of the code above:

```
printf("*(val+5)=%d\n",*(val+5));
printf("ptr[5]=%d\n",ptr[5]);
```

What is the output?

Exercise 7: One of the problems with arrays is their length is unknown when passed to functions. The programmer must ensure that code does not run off the end of the array storage.

A character string in C is an array of bytes. How do the Standard C library string routines know when a string is finished?

Exercise 8: Write a program to create a new variable that is the concatenation of two strings. This is not straight forward—some thought has to be used.

Start with the following variables:

```
char h[]="Hello";
char w[]="World";
char hw[20];
```

Note: the compiler will allocate the appropriate space for the strings based on the initialisation.

Exercise 9: Repeat the exercise above but start with the following variables:

```
char *h="Hello";
char *w="World";
char *hw;
```

To answer this question first consider compiler initialisation (have a look at the two exercises below). You will also need to study the Standard C library function `malloc()` in detail to complete this exercise.

Exercise 10: In the above two exercise the following two notations where used:

```
char h[]="Hello";
char *h="Hello";
```

Is there any differences between the two variables?

Is there any difference between the following variables?

```
char hw[20];
char *hw;
```

Exercises

Ex. 11: Browse all of "A Tutorial on Pointers and Arrays in C" by *Ted Jensen* except Chapter 10

Ex. 12: Answer the exercises at the end of Chapter 1 of the Study Book.