Lecture 9: Introduction to Pointers

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/09-pointers-introduction.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

Here we introduce a fundamental concept in C, pointers

Program Memory Layout

Before explaining what a pointer is, it's important to understand how the memory that is accessible to a program looks like. All the data manipulated by a program, as well as all the code it executes, is present somewhere in memory. The area of memory that is accessible for the program to read, write and execute is called the **address space**.

One can see the address space as a very large array of contiguous bytes. Each byte has an index into this array, named an address. Addresses range from 0 to the maximum addressable byte. On x86-64 the kernel configures virtual memory from address 0 to 128 TB to be accessible by each program. Note that this is virtual memory, so each program has his own copy of that array. Its size is also completely independent of the actual physical memory in the machine.

The address space is generally very sparse and only the bytes that are read/written are mapped to physical memory. Finally, note that we generally use hexadecimal numbers, prefixed with `0x`, to indicate addresses as they can be pretty big.

Addresses

Recall that all code executed and data manipulated by the program reside in memory. Each variable has an address which is the first byte of location in memory. We can get the address of a variable in C with the `&` operator. In this example we use that operator to retrieve and print the address of an `int` variable and also a data structure:

```c
int glob = 12;

typedef struct {
    int member1;
    float member2;
} mystruct;

int main(int argc, char **argv) {
    mystruct ms = {1, 2.0};
    // With modern processors an address is a 64 bits value, so we need the
    // right format specifier: `%lu` or `%lx` if we want to print in hexadecimal
    printf("ms address is: 0x%lx, glob address is 0x%lx\n", &ms, &glob);
```

```
    return 0;
}
```

Note the use of the marker `%lx`: an address is 64 bits on modern processors, so it's a `long`, and we use `x` to specify printing in hexadecimal.

Pointers

A pointer is simply a variable that holds an address, possibly the address of another variable. We can declare a pointer by indicating the type of the pointed variable, followed by `*`, and then the name of the pointer. For example here we create a pointer named ptr that holds the address of `v`:

```
int v = 42;
int *ptr = &v;  // ptr holds the address of v, i.e. v's location in memory
```

A pointer is itself a variable, and because it holds an address its size is equal to that of the address bus: with modern 64 bits architectures, the size of pointers is 64 bits.

Dereferencing a Pointer

An important feature of pointers is that one can access the pointed value through the pointer, with an operation called **dereferencing** the pointer. We can read or write this value it with the `*` operator:

```
int v = 42;
int *ptr = &v;
printf("pointer value:   0x%lx\n", ptr);
printf("pointed value:   %d\n", *ptr);
```

Through `ptr` one can read or write the value of `v` In the example, in the second `printf` we read `v`'s value with `*ptr`, which evaluates to the value of `v`. It is an `int` so we use `%d` to print it. Here is a further example where we have 3 pointers to `int`, `double`, and a data structure variable:

```
int glob = 12;
double glob2 = 4.4;

typedef struct { int member1; double member2; } mystruct;

int main(int argc, char **argv) {
    mystruct ms = {55, 2.23};

    int *ptr1 = &glob;
    double *ptr2 = &glob2;
    mystruct *ptr3 = &ms;

    /* Print each pointer's value (i.e pointed address), and pointed value */
    printf("ptr1 = %p, *ptr1 = %d\n", ptr1, *ptr1);
    printf("ptr2 = %p, *ptr2 = %f\n", ptr2, *ptr2);
    printf("ptr3 = %p, *(ptr3).member1 = %d, *(ptr3).member2 = %d\n",
            ptr3, *(ptr3).member1, *(ptr3).member2);
}
```

We print the values of each pointer, i.e. the addresses of the pointed elements, and we also print the value of the pointed elements by dereferencing the pointers. Note that it also works for data structures: `*(ptr3).member1*` evaluates to `55`, the value of the first member of `ms`. Note the use of the parentheses here: without them the `.` operator takes precedence on the `*`. Try to reason about what would happen if one forgot the parentheses (hint: nothing is mapped at address `55` and accessing it results in a crash.)