C/C++
Revision Notes

James Brown

May 3, 2017

Contents

1 Introduction

These are notes I have written in preparation of the 2017 C/C++ exam. This year the module was run by Hayo Thielecke (H.Thielecke@cs.bham.ac.uk). This module focuses on the features of C and C++ that are important in Computer Science in general. It is not a rerun of the first year Software Workshop module.

2 Pointers and Memory Management

2.1 Pointers

Pointers are a fundamental feature of C which have not been encountered in programming languages we have used to this point, and they are used everywhere in C. What is the meaning of `x = x + 1;`? It does not mean $2 = 2 + 1$. The `x` on the left side of the `=` refers to the address (L-value) `x`. The `x` on the right side of the `=` refers to the contents (R-value) of `x`. Here, the L-value is a **pointer**

The view of memory in C is of a graph, as we would like to abstract away from actual hardware addresses. Nodes in the graph are chunks of memory (often this is a struct), and edges between nodes are pointers. Due to this, box-and-arrow diagrams are very useful in representing the state of memory.

Figure 1: A box-and-arrow diagram

Figure 2: The hardware level of a box-and-arrow diagram

2.1.1 The * and & Operators

In C, * is also a unary operator. It is also used for binary multiplication, but the two have nothing to do with each other. If `P` is an expression denoting a pointer, then `*P` is the result of derefencing the pointer (that is getting the value of the thing it points to in memory). If `T` is a type, then `T *p;` declares `p` to be of type 'pointer to `T`'. If `T` is a type, then `T*` is the type of pointers to something of type `T` - this is used in casting. Important to note is that the `*` does not stick to everything in a declaration. For example, `int *p, n;` is like:

```
int *p;
int n;
```

and not

```
int *p;
int *n;
```

Care should be taken when defining pointers in this fashion.

If a variable appears on the right-hand side of an `=`, its R-value is taken. If we want to get the address of a variable rather than its contents, we use the `&` operator - `&x` for example.

In C, two pointers are `==` if they refer to the same address in memory. Pointer equality is different from structural equality that is built into other languages. `p = q` makes `p == q`. `*p = *q` does not make `p == q`.

2.1.2 The Null Pointer

There is a special pointer value, called the **null** pointer. In C, it is called the `NULL` pointer and in C++ it is called the `nullptr`. The null pointer, unsuprisingly, does not point to anything. Derefencing `NULL` will give an undefined behaviour (usually this will be a crash). In C, we have an idiom to test whether a point `p` is equal ot the null pointer:

```
if(p) ...
```

It is important to note that pointers are not always initialised to null.

2.2 Memory Management with `malloc` and `free`

In C, `stdlib.h` provides the functions `malloc` and `free`. The part of memory managed by `malloc` is called the **heap**. `malloc` allows us to borrow some memory from the memory manager, and `free` allows us to give back the memory that we have borrowed. We must promise to not use the memory that we have freed again, although the memory manager cannot force us not to do this. Regardless we still should not touch freed memory has it can lead to undefinied behaviour. Also important to note is that the call to `free` changes the ownership of the memory, not any of the pointer which pointed to that memory. The `free` operator is very important, as our program will leak memory otherwise. Luckily, we have tools such as Valgrind to help us analyse our code and test for memory errors and leaks.

`malloc` and `free` are not part of the C language itself, only its standard library. Due to this, we could implement our own memory allocator in C if we so wish. The allocator would request some memory from the OS (this would be done via `sbrk` in Unix). The available memory would be divided into chunks that are linked together in a 'free list'. A call to `malloc` will then detach a chunk from the free list and return a pointer to it. A call to `free` takes the pointer to the chunk and links it back into the free list. This is not massively efficient and may also result in memory fragmentation, but the fact we can do it is still impressive.

What happens once `free` is called upon some memory? Various things: the same piece of memory may be used again in a later `malloc`, or the memory manager might write its own data structures into the memory (e.g. the free list). Rather than trying to guess exactly what happens, we simply call it undefined behaviour. C, unlike other languages, will not prevent you from doing bad things with freed memory.

Using `malloc` and `free` we could write a piece of example code like such:

```
int *p1, **p2;
p1 = malloc(sizeof(int));
*p1 = 7;
p2 = malloc(sizeof(int*));
*p2 = p1;
free(p1);
```

If we add the line `**p2 = 11;` as the last line of this piece of code, we would be adding an example of use after free, which we should absolutely avoid doing!

2.2.1 The `sizeof` operator

For using `malloc`, we need to tell the function how many bytes to allocate. Usually, we are allocating enough memory to hold a specific type, but sizes of types are implementation dependent. The compiler will tell us how big it makes each type, and this can be found by using `sizeof(T)`. Very commonly we will write a piece of code as follows:

```
T *p = malloc(sizeof(T));
```

2.2.2 Crashes vs Memory Errors

In C, crashes and memory errors are not the same thing. Crashes usually have many names, such as core dumps or segmentation faults. Crashes are errors which have been detected by the hardware or the OS. In C, a memory error *may* lead to a segfault, but it is not guaranteed that it will. A write error which does not lead to a segfault may instead lead to corrupted memory - this could be even worse! A C program with a memory error is always wrong. A memory leak is not the same as a memory error, and not always a bad situation. A memory leak may lead to a crash though when the program eventually runs out of memory.

3 Structures in C

4 Typed Trees in C

5 From C and Java to C++

6 Templates in C++

List of Figures

Index