C/C++
Revision Notes

James Brown

May 8, 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.

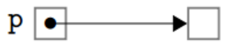

Figure 1: A box-and-arrow diagram

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

A structure (or `struct`) is much like a Java class that contains only data and no methods. When talkinga about structures, we say that they have **members**, not variables. In C, we cannot define functions inside a struct but we can in C++. In C, functions are defined outside of structs and often access them via pointers. Structures and pointers (along with `malloc`) lets us build many classic data structures such as lists, trees and graphs.

The syntax for C structures is very similar to the syntax of Java classes:

```
struct s {
    T1 m1;
    ...
    Tk mk;
};
```

After a structure, `s`, has been declared, `struct s` can be used as a type name:

```
int n;          // declares n as an int
struct s y;     // declares y as a struct s
struct s *p;    // declares p as a pointer to a struct s
```

Access to structure members is done by using the dot operator, e.g. `s.m1`. Structure members are laid out in memory in order, with possibly a few bytes of padding between structure members due to alignment, depending on the hardware. For this reason, `sizeof(struct S)` $\geq$ `sizeof(T1) + ... + sizeof(Tk)`.

3.1 The $->$ operator

`p -> m` is an abbreviation for `(*p).m`. It defrences the pointer and then access the structure member. This operation is very, very common in C and C++ code. This operation can be chained together, for example `p->m1->m2->m3`.

This operator will be commonly used to traverse over a list like so, as C does not have an iterator idiom:

```
while(p) {
    ...
    p = p -> next;
}
```

It would be incorrect to right something like `while(p->next)...)`, as this ignores the fact that a linked list may be `null`, which represents the empty list. We should begin by testing for the `null` pointer.

4 Pointer Arithmetic and Arrays

4.1 Arrays and Strings in C

In C, we define an array in the same way we would define an array in Java: `char b[2];`. The size of the array is also not stored, like it is in Java, and we have to track the array bounds ourselves. When an array is passed into a function, a pointer to the start of the array must be passed - not the contents of the array. In Java, `a[i]` will either refer to the i-th element of the array or throw

```
while(lp) {
    q = lp;
    lp = lp -> next;
    free(q);
}
```

Figure 3: A piece of code to delete all elements of a linked list

```
while(lp) {
    free(lp);
    lp = lp -> next;
}
```

Figure 4: The same code with a use after free error

an index out-of-bounds exception if i is too big. In C, a[i] will either refer to the i-th element of the array or cause undefined behaviour if i is too big - this is completely undesirable behaviour. Not doing bounds checks is faster, for example: if we want to multiply two 100000 x 100000 matricies, do we really want the compiler to do a bounds check for each array access?

In C, a string is an array of char's which is terminated by a zero byte. The zero byte, `\0` is not the same as the character "0" (which in ASCII is 48).

4.1.1 The Arguments to Main

The `main` function takes two arguments: an array of strings and the number of strings there are passed like so:

```
main(int argc, char *argv[]) { ... }
```

Should we not need the arguments, in C (but not C++) we can also write:

```
main() { ... }
```

4.2 Pointer Arithmetic

In C, we can add an integer to a pointer, but we cannot add two pointers. Array access is achieved through pointer arithmetic. When adding to a pointer, `p + 1` does not mean `p` plus one byte. Instead, in `p + n`, `n` is scaled up by the size of the type of what `p` points to.

4.3 Buffer Overflows

All array access in C are potentially dangerous, and as strings in C are arrays they too are dangerous. Arrays can potentially overflow, so we should always check the bounds of the array. An example of a vulnerable function is as follows:

```
void mystrcpy(char *q, char *p) {
    while(*q++ = *p++);
}
...
char a[] = "abc";
char b[2];
mystrcpy(b, a);
```

If we use this function with more characters than array space allocated (as shown in the above example), the function will continue to copy - even if it is overwriting other data - until it hits the zero byte in the string. Some C compilers will do some buffer overflow mitigation, but should not be relied upon. Instead, we should use functions with bounds such as `strncpy` and `fgets`. For more advanced attacks using buffer overflow attacks, we need to understand how C is compiled. If we can overwrite the return address of the function, we can alter the flow of the program. It may even be possible to 'return' back to code we have injected.

4.4 First-class functions in C and C++

C has **pointers** to functions. In C, functions cannot be defined inside of other functions, but functions in C can be passed as a parameter very easily - they are just code pointers. If we have a function pointer `p`, then we can call it via `(*p)(...)`. We may use this in the `fold` function for example, which takes a binary operator function as an argument:

```
int fold(int n, int (*bin)(int, int), struct Linked *p) {
    while(p) {
        n = bin(n, p->data);
        p = p->next;
    }
    return n
}
```

5 Typed Trees in C

Using what we already know, we can build n-ary trees using structures and pointers. Recursion in these trees ends by `null` pointers, hence the `if(p)` and `while(p)` idioms. These trees only have one kind of node, which may be sufficient for some situations. In Computer Science, we have more complex trees with different kinds of nodes with different numbers and kinds of child nodes. These trees need a different type system of different numbers. The most common example of these kinds of trees is **abstract syntax trees**. These are fundamental for compilers.

5.1 `struct`, `union` and `enum`

Using what we have, we cannot represent typed trees such as abstract syntax trees very easily. The addition of `union` and `enum` can make doing this much easier.

5.1.1 `union`

The syntax of `union` is like that of `struct`.

```
union u {
    T1 m1;
    T2 m2;
    ...
    Tk mk;
};
```

We can think of unions as a choice between any of the members. The official C11 draft standard says in section 6.7.2.1 that a union is a type consisting of a sequence of members whose storage overlap. Unions are not tagged, so the memory does not know whether is contains data of type `T1` or type `T2`. If we want a tagged union, we need to enclose the `union` within a `struct` with a `enum`

5.1.2 `enum`

`enum` is the enumeration type, much like as it is in Java.

```
enum dwarf { thorin, oin, gloin, fili, kili };
```

In it's implementation small integers are used, for example `thorin = 0` and so on.

5.1.3 Tagged unions idiom

We will use an `enum` as the tag for our union. We then package the `union` in a `struct` along with the `enum`.

```
enum ABtag { isA, isB };

struct taggedAorB {
    enum ABtag tag;
    union {
        A a;
        B b;
    } AorB;
};
```

Here the `struct` can either be A or B, and we know which it will be by looking at the tag in the `enum`. We can access the tagged unions with `switch`:

```
struct taggedAorB x;
...
switch(x.tag) {
    case isA:
        // use x.AorB.a
    case isB:
        // use x.AorB.b
}
```

Having to invent names for `struct`, `union` and `enum` members can be tedious. We may wish to use a systematic naming scheme for a given situation and stick to it, but this is a matter of taste. Since C11, we can use anonymous `struct`s and `union`s, like so:

```
struct s2 {
    struct t2 { int q; ... } ;
    int n;
}
```

The inner `struct` is anonymous by lacking a member name, but the compiler still knows what `q` is.

5.2 Typed Trees

We have explored how to create tagged `union`s, which can be processed with a `switch` statement. If we add pointers and recursion, we can create **typed trees**. From trees a pointers, we can also get **graphs**

5.2.1 Trees with values only at the leaves

```
enum treetag { isLeaf, isInternal } tag;

struct intbt {
    enum treetag tag;
    union {
        // if tag == isLeaf, use this:
        int Leaf; // no recursion
        // if tag == isInternal, use this:
        struct {
            struct intbt *left;     // recursion
            struct intbt *right;    // recursion
        } Internal;
    } LeafOrInternal;
};
```

This is not the same as a tree using `struct` and pointers. A tree built like such would require `null` pointers to terminate. As our abstract syntax trees are not supposed to contain null pointers at all, we would like to ensure that this is the case. We can use the `assert` macro from `assert.h`. `assert(p)` means that we are not expecting a null pointer in `p`. An error will be thrown if the assertion fails. We can use assertions in debugging, and turn them off in production code.

5.2.2 AST for Lisp-style expresions

Figure 5: A grammar for a Lisp-style language

$E \to n$	(constant)
$E \to x$	(variable)
$E \to (+L)$	(addition)
$E \to (*L)$	(multiplication)
$E \to (= xEE)$	(let binding)
$L \to EL$	(expression list)
$L \to$	(empty list)

Figure 6: The abstract syntax tree of this language

```
enum op {isplus, ismult};
enum exptag {islet, isconstant, isvar, isopapp};

struct exp {
    enum exptag tag;
    union {
        int constant;
        char var[8];
        struct {
            enum op op;
            struct explist *exps;
        }; // anonymous struct
        struct {
            char bvar[8];
            struct exp *bexp;
            struct exp *body;
        }; // anonymous
    }; // anonymous
};

struct explist {
    struct exp *head;
    struct explist *tail;
};
```

6 From C and Java to C++

By now, we have covered all of the most important contructs of the C programming language. We now move to C++. C++ aims to be as efficient as C, but with more structure to the language. It is both a high-level and a low-level language, without garbage collection. C++ also has object orientation, but you are not forced to use it.

6.1 Virtual Functions

- **Non-Virtual**: the compiler determines from the type what function to call at compile time
- **Virtual**: what function to call is determined at run-time from the object and its run time class

At runtime, each object of a class with virtual functions contains an additional pointer to the virtual function table (vtable). Take these two examples:

```
class Animal {
public: void speak() { std::cout << "Noise\n"; }
// not virtual
};

class Pig : public Animal {
public: void speak() { std::cout << "Oink!\n"; }
};

int main(int argc, char *argv[]) {
    Animal *peppa = new Pig();
    peppa->speak();
    // the type of peppa is Animal
}

class Animal {
public: virtual void speak() { std::cout << "Noise\n"; }
};
class Pig : public Animal {
public: void speak() { std::cout << "Oink!\n"; }
};
int main(int argc, char *argv[]) {
    Animal *peppa = new Pig();
    peppa->speak();
    // peppa points to an object of class Pig
}
```

In the both examples, we have defined the type of peppa to be an animal, and then instantiated it as a pig. The first example uses a non-virtual function. At compile time, the compiler checks the type of peppa and sees it is an animal. Therefore, when `speak()` is called, "Noise" is printed. In the second example, `speak()` is a virtual method in `Animal`. When `speak()` is called at run-time, what function to call is determined by the object and its run-time class, which is `Pig`. For this reason, "Oink" is printed.

This is a very simple example, and we could in theory make the compiler deduce the type of peppa as it is only ever instaniated to a pig, and never changed. In this example, the code will be less efficient for no real gain as it has to pass around the vtable and check the correct function to call in the virtual example. However, take the following example:

```
Animal *ap;
if(...) {
    ap = new Baboon();
else
    ap = new Weasel();
ap->speak();
```

Now it would not be so easy to predetermine the type of ap at compile-time, which shows the value of virtual functions. Whether ap points to being a baboon or a weasel will only be known at run-time.

6.1.1 Pure Virtual Functions

A pure virtual function has no implementation in the base class. This has a slightly peculiar notation:

```
class C {
    public:
    virtual T1 f(T2) = 0; // declare f as pure virtual
    ...
}
```

This does not suggest anything is equal to zero, the `= 0` is supposed to suggest that something is missing. Any derived classes of C should implement `f`. For example, we could have said that there is really a noise that any animal can make and we could have defined it as a pure virtual function: `public: virtual void speak() = 0;`

Furthermore, we could use pure virutal functions in our abstract syntax tree example. Each of the nodes in the tree would be an object (instance of a class) with their pointers to any children nodes. Each node would use the evaluation function of its own class, removing the need for a switch statement. This is dynamic 'polymorphism' via virtual functions.

6.2 New and Delete

C has `malloc` and `free` for heap allocation and deallocation. C++ has `new` and `delete` fo heap allocation and deallocation. We can also define **constructors** (not surprising) and **destructors** (somewhat surprising) in C++.

C++ destructors for some class `A` are called `A`. We do not call destructors directly, we let our call to `delete` do it implicitly. The clean-up done by the destructor may involve deleting other objects 'owned' by the object to be deleted. For example, if we delete the root of a tree, we would possbily want to recursively delete all of the child nodes as well. We may also perform other resource management in a destructor such as closing any opened files.

6.3 Strings in C++

C uses 0-terminated character arrays as strings, which is full of pitfalls like buffer overflows and off-by-one bugs. C++ has a dedicated string class which can be used.

7 Templates in C++

There are two kinds of templates in C++: **class templates** and **function templates**. Roughly, these correspond to polymorphic data types and polymorphic functions in functional languages. The syntax is as follows:

```
template<class T>
class Linked {
public:
    T head;
    Linked<T>* tail;
};
```

List of Figures

Index