Systems Programming
LECTURER

Joao Almeida-Domingues[*]

University of Glasgow

September 28th, 2020 – December 10th, 2020

Contents

These lecture notes were collated by me from a mixture of sources , the two main sources being the lecture notes provided by the lecturer and the content presented in-lecture. All other referenced material (if used) can be found in the *Bibliography* and *References* sections.

The primary goal of these notes is to function as a succinct but comprehensive revision aid, hence if you came by them via a search engine , please note that they're not intended to be a reflection of the quality of the materials referenced or the content lectured.

Lastly, with regards to formatting, the pdf doc was typeset in LaTeX, using a modified version of Stefano Maggiolo's class

---

[*]233459oD@student.gla.ac.uk

1 Introduction

1.1 *ILOs - Fundamentals*

The goal of the course will be to deepen your understanding of the fundamental principle and techniques in computer systems, such as:

- Memory and computation as fundamental resources of computing

- Representation of data structures in memory and the role of data types

- Techniques for management of computational resources

- Reasoning about concurrent systems

1.2 *Systems vs Application Software*

We contrast system software with application software. System software can be seen as low-level software, i.e it usually concerns itself with interacting with the machine directly or very close to the metal whilst providing abstractions for application software

There are constraints which come with the nature of system software, such as fast execution time, low memory consumption or low energy usage. Because high-level, managed languages are usually highly abstract, it is almost impossible to build software which abides by these constraints, hence one uses system programming languages like `C` which provide the programmer with more fine grained control over how their program will execute

1.3 *History*

In the 1950s the distinction between these two types of software was nonexisting. Machines were very much purposely built to run certain applications and a single executing application would use the entire machine. *Grace Hopper* was responsible for writing one of the first compilers which turned human readable code into machine code.

Until the 1970s system software was always written in a processor specific assembly language, which meant that for each new processor a systems programmer would need to rewrite the same program in a different assembly language. In the 1970s *Dennis Ritchie* and *Ken Thompson* , while trying to port `UNIX` between two machines, invented `C` as a *portable, imperative* language which supported *structured programming.*

In the 1980s, *Stroustroup* came up with `C++` primarily so that he could automate certain tasks which have to be done manually in C.

Until the 2010s the focus very much shifted towards the development of high-level, managed languages however, the emergency of mobile devices rekindled the interested in new systems languages such as `Swift` and `Rust` which permit the devices to not waste their already scarce resources in tasks such as garbage collection

2   C - Introduction

✎   introduction to C , fundamental features , differences , strong typin , lexical scope , lifetime of variables , call by value , declarations and definitions , compiler errors and warnings , data representation in memory , data types

2.1   *A tale of two languages*

In order to illustrate the advantages of C over managed languages take the following programs

```
                        int main() {
x = 41                      int x = 41;
x = x + 1                   x = x + 1;
                        }
```

2.1 remark. Java syntax was heavily borrowed from C. Programs differ in how they are executed and on how memory is organised

The size (in memory) of x in Python is dependent on the architecture, in this computer for example is about 28B, whilst in C is just 4B. Why? Given the dynamic nature of python, its C implementation uses a descriptor object which stores the alongside the value which significantly increases the memory requirements. In C integers are usually 4B and that's all you need to represent x in memory

What about the number of instructions? That's even harder to know in python, because for example each operation must type-check first, then it must also check that addition is a valid operation and it represents all of that in some structure in the CPU. In C however, we know that only 3 instructions are required, 1 add and 2 mov.

2.2 remark. This level of fine-grained control allows us to confidently reason about the execution behaviour and performance of a program

2.2   *Compiling*

The compiling of code is the act of transforming the source code of a program into an executable file which can be run by the OS. There are two kinds of source files in C:

- Header Files : a file containing C declarations and macro definitions to be shared between several source files. Its contents can be requested in a program by including it, with the C preprocessing *directive* #include. It is conventional to end file names with .h

- Compilation Units : These are .c files whose directives are replaced by the contents of the header files. Each file is compiled separately to keep compilation times short

The compilation process can be split into 3/4 main steps:

- Preprocessor : During the preprocessing stage the preprocessor will look at the source code and replace all directives and macros with the contents of their corresponding files. For examples `#include <stdio.h>` will copy the contents of the I/O library `stdio.h`

  2.3 remark. To see an example of the output run `gcc -E`

- Compiler & Assembler : The compiler transforms the preprocessed code into assembly code which in turn are turned into *object* files `.o` (almost executable) by the assembler

- Linker : The linker will then find, for each name appearing in the object code, the address that was eventually assigned to that name, make the substitution, and produce a true single binary executable in which all names have been replaced by addresses

2.4 remark. You can use constants by using the `#define` compiler directive which will essentially replace every instance of the named variable with the actual value at preprocessing stage

Warnings & Errors

Compiling errors mean that it is impossible to translate the program into an executable. Warnings on the other end indicate that there is something and unusual in the code which will most likely result in a runtime error. Occasionally one might want to keep the piece of code which threw the warning , in that case make sure to make that clear in your code.

For the coursework make sure that to submit code which compiles without warnings

The compiler and the linker will throw different kinds of errors, being able to identify which is which helps with debugging. For example, it is possible to convert a program into object code which has a call to an undefined function this will just throw a warning. However, when running the linker an error will be thrown and an executable won't be created.

2.5 remark. `-Werror` turns all warnings into errors, the `-Wall` flag enables most compiler warnings

TOPICS: introduction to C, fundamental features, differences, strong typing, lexical scope, lifetime of variables, call by value, declarations and definitions, compiler errors and warnings,how data is represented in memory, why data types are fundamental in C

3   Main

The main function is the entry point into the program. By default it returns 0 if the end is reached when executing, any other non-negative return value indicates an unsuccessful execution

There are two valid versions of main, one of them is able to process command line arguments !!INSERT CODE!!

4   PRINTF

printf is a function defined in the stdio.h library, which takes a format string as its first argument, and the following arguments as the values to be printed out

!! INSERT LINK TO COMMON FORMAT STRINGS !!

5   VARIABLES

Variables are composed of a *data type*, and *identifier* and optionally an *initialisation expression*

5.1 remark. C is a *statically typed* language, it is not possible to declare a variable without declaring its type

5.1   *Data Types*

A *bit-pattern* on it's own has no meaning, its meaning is context dependent. Data types give bits meaning. By declaring a variable with a data type one decides what a bit-pattern means when stored in memory

By enforcing operations to respect data types the compiler prevents meaningless computations so that we preserve meaningful representation of the data. If x is a char then x + 1 has a very different meaning than if x was a float. The resulting bit-patterns stored in memory would be totally different

!! INSERT EXAMPLE FROM LECTURE SLIDES !!

5.2   *Representation in Memory*

A variable is allocated a space in memory when it is declared and that location remains unchanged throughout its lifetime, it is the data type of the variable which determine the bits being stored in memory. It follows that for optimal performance one should take into consideration which data type to choose depending on one's needs. For example, if one needs to store an array of small integers then one could save almost 3MB of data by choosing to use char over int. Though in terms of storage/tertiary memory this improvement is marginal, in terms of memory this can lead to a significant improve in performance since primary memory provides faster access speeds at the cost of much lower capacity

A lifetime of a variable depends on how the memory for that variable was allocated. There are 3 possible cases:

Automatic allocation happens for variables which are declared locally within a scope

Static allocation means that the variable will persist throughout the execution of the program. Hence, every time that variable is needed the program will look at the same location

Allocated allocation requires the programmer to manage the lifetime of the variable themselves by explicitly request memory using *dynamic memory function* such as `malloc`

5.3 *Stack-Based Memory Management*

Automatically allocated variables' memory is automatically freed up once they expire. This is achieved by following a LIFO *stack-based memory management* approach. When the compiler sees a start of the block it puts aside a location in memory for every variable in scope, when it reaches the end of the block it frees up all previously assigned memory.

6 Other Data Types

6.1 *Struct*

Structs consist of a sequence of members which need not necessarily be all of the same type. They are comparable to a public Java class without any methods, and similarly to Java a struct's members can be accessed using dot notation. A major advantage of structs is that there is no overhead in memory management since its variables are stored in memory sequentially in the order in which they were declared.

One declares a struct type by using the keyword `struct` before the identifier, but because one might want to initialise multiple structs with different values `typedef` can be used to name that struct. After doing so, the struct keyword can be dropped

6.2 *Arrays*

Arrays consist of multiple elements of the same type. So that the memory is automatically managed the size of an array stored in the stack must have a fixed size

Dynamic arrays are possible if stored in the heap, see next week's lecture notes

6.1 remark. array elements are stored next to each other in memory

6.2 remark. Arrays in C do not carry any information regarding their size. This means that it is not possible to generate arrays dynamically, and one should always make sure to keep track of this information

6.3 *Strings and Characters*

In C strings are just an array of characters, the compiler identifies the end of the array by the
0 character which is appended to its end. This happens automatically if the string is defines as a string literal `""` but must be added manually if the string is defined as an array of chars `''`

6.4 *Function*

Function declarations represent its interface, i.e they let one know what is expected when the function is called; the behaviour of the function can then be *defined* inside a block. In C all identifiers must be declared before they are used. To declare a identifier just means to assign a name and type to that identifier, it allows the compiler to know the properties of that identifier. Functions are no different, except that one must also identify the type of their parameters. When defining a function one must give it a body specifying its behaviour

6.3 remark. the linker is responsible for linking the reference of a function to its implementation

Call-by-value

When an argument is passed to a function its value is passed to the function's parameter, hence changes made to the parameter have no effect on the argument. What happens is that a copy of the argument's value is created and a new local variable whose scope is the function's body is allocated some other space in memory; as it was mentioned above once the function ends execution that location is automatically freed up, hence the changes will not persist unless the value is returned

Arrays are special because when passed to a function they are not copied (expensive), instead the address/pointer of its first element is passed

remember your first attempt at trying to write a function which returned an array's size