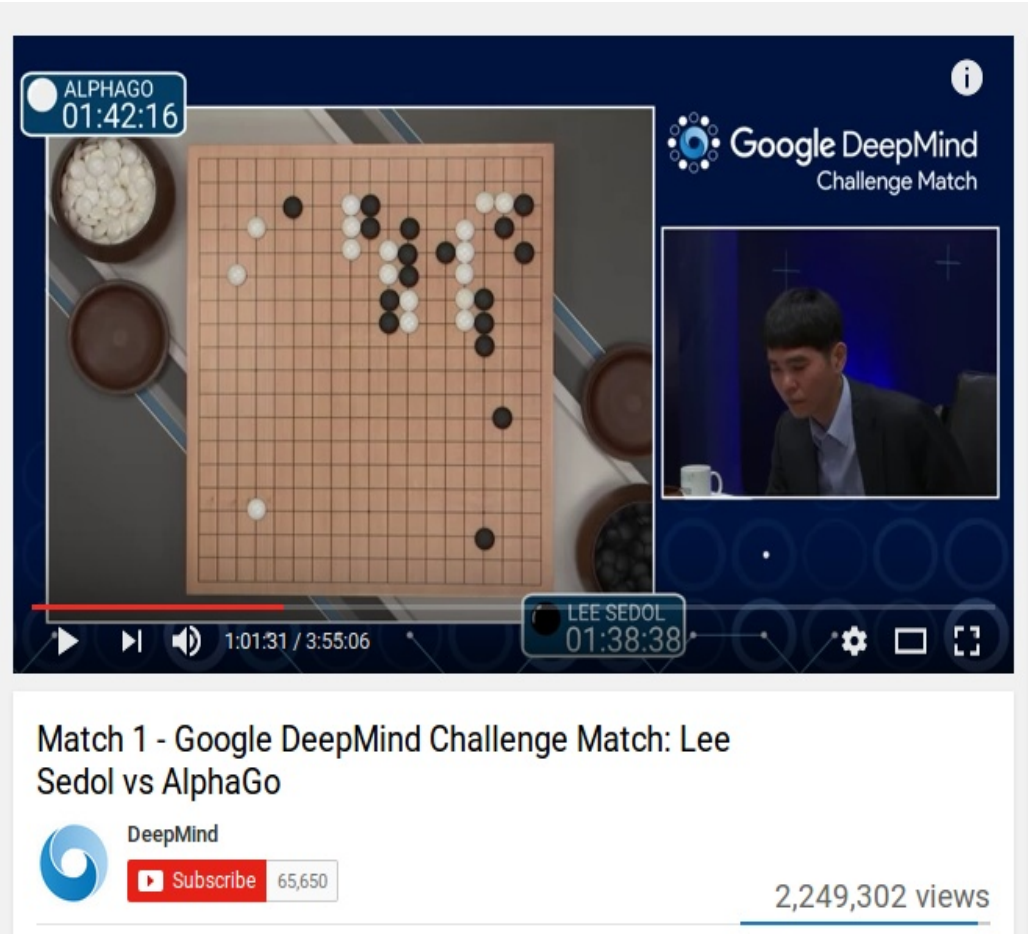
# COMP281
# Principles of C and memory management

## lecture 1

**Dr. Frans Oliehoek**
**Department of Computer Science,**
**University of Liverpool.**

# Seen this?




- Advance due to new algorithms…
- …but not possible without massive computational sources ("The distributed version in October 2015 was using 1,202 CPUs and 176 GPUs." [1])
- **and using them efficiently!**

[1] https://en.wikipedia.org/wiki/AlphaGo, retrieved January 23, 2017.

# Today

- Admin and module info
- Introduction to C
  - Why C?
  - first bit of C syntax

# Module Delivery

Lecturer: Dr. Frans Oliehoek
Room 222, Ashton Building (2nd floor)
Email: frans.oliehoek@liverpool.ac.uk

Module info page: see http://vital.liv.ac.uk

Slides and sample code will be available on vital as the course progresses

# Module Delivery

- **Lectures** times and locations:
  - Tuesday 9:00am, Maths, Proudman Lecture Theatre
  - Thursday 9:00a, Chemistry, Chemistry Gossage Lecture Theatre

- Also two hours of **practicals** (labs) every week:
  - Each student will be assigned to a practical class ('lab section') – check 'Liverpool life' (also 'Spider') for details of your allocation
  - There will be one demonstrator **'responsible' for your lab section**: see Vital->Module staff.

  - No labs first week

- The practicals are even more important than the lectures
  - **Good programming takes practice!**

# Suggested Reading

- There is no **required** text book
  - Most of the information is in the slides
  - There is a wealth of information about C programming online!
- Still, a text book can be useful. I would **recommend** getting one
  - Some may be accessible online via the UoL library

See "learning resources" on vital

# Module Aims and Objectives

- The aims of the module are:
  - To introduce the **issues of memory and memory management** within the context of a system-level procedural programming language
  - To familiarise students with the **C programming language**.
  - To demonstrate principles, provide indicative examples, develop problem-solving abilities and provide students with experience and confidence in the **use of algorithms with consideration and management of memory usage**

# Learning Outcomes

- At the end of the module the student should be able to:
  - **analyse and explain the use of memory resources** within software applications, including memory usage on the stack during function calls and heap-based dynamic memory management;
  - use **debugging tools** (e.g. gdb, Valgrind) to inspect memory usage, and to assist in the development of software;
  - **develop applications with the C programming language**, including use of command-line driven C development tools;
  - deal with underlying memory-based issues in using dynamic data-structures through the **implementation and management of data structures** using the C programming language.

# Module Syllabus (Approximate)

- Various elements of the C programming language
- Exploration of the use of dynamic memory allocation in C through the use of arrays, pointers, and strings
- Dynamic data structures in C:
  - structs and the dynamic creation and destruction of structs
- More advanced issues, including function pointers and the C pre-processor.
- We do not cover any GUI programming

# Assessment

- Assessment is by continuous assessment – i.e., three programming assignments, each consisting of a number of programming tasks.
  - 30%, 30% and 40% of your grade
  - Each assignment will be accompanied by a brief report.

- If you can successfully write the programs in C, you will pass the module
  - If you don't……

- Submissions are automatically tested by anti-plagiarism software

- The **Golden Rule**
  - if you didn't write some part entirely by yourself, then declare this in your report.

# More Assessment

- You will be able to check online that your program works correctly before submitting it.

- If it works correctly (and you wrote it yourself) you will pass!
  - Although, of course,  better marks are available for better programs
  - More details will be given during the module

- If it doesn't work, **submit anyway**.
  - Some marks will be awarded for correct concepts.

- See "Marking descriptors" (on vital) for more detailed information on grading

# Assessment: Deadlines

- 3 Assignments, due
  - Wed, February 15, 4:30pm
  - Wed, March 1, 4:30pm
  - Wed, March 15, 4:30pm

- Submission via departmental submission system:
  - http://www.csc.liv.ac.uk/cgi-bin/submit.pl
  - usual late penalties apply
  - can submit as often as you like before the deadline
  - do **not** submit both before and after the deadline:
    - in order to provide you with feedback, we need to start grading right away, so the version you submitted before the deadline will be graded
  - please make sure you submit correctly...!

# Feedback

- All the assignments are graded and you will receive feedback on each of the problems in each assignment.

- If you don't understand an assignment: best time to come with questions is:
  - **before** the deadline
  - and **during** the labs

- If you have questions about a past assignment, please ask the demonstrators at the lab

# Questions outside lab sessions?

- Lab sessions are **the** place to ask questions…!
- Outside of lab, follow the following steps:
  - first post on the **discussion board**
  - if no help from fellow students within reasonable time
    → email the demonstrator responsible for you lab section.
    - include the link to your post!
- Academic practice is also about sharing your knowledge:
  - I expect that you will **help each other** on the discussion board!

# Module organization – Summary

- Each week: 2 lectures, 2 labs

- Questions? use the discussion board!
  - Primary contact: the responsible demonstrator for your lab section

- Continual assessment determines the grade
- You will receive feedback on these assignments

- The vital page contains all information

# Why learn C?

- It can produce efficient programs
  - Where performance is critical it is often a good choice.
  - Efficiency is also important in terms of battery life
  - but still is portable (if you stick to the standards)
- It does not require the support of a large operating system or virtual machine
- It is a very commonly used language
  - C compilers exist for many different systems
  - There is a lot of existing code written in C
  - Typical language of choice for systems-level programming, operating systems, embedded systems etc.
- It is the base for many other languages
  - like C++, Objective C, Java

# Why learn C?

- It can produce efficient programs
  - Where performance is critical it is often a good choice.
  - Efficiency is also important in terms of battery life
  - but still is portable (if you stick to the standards)
- It does not require the support of a large operating system or virtual machine
- It is a very commonly used language
  - C compilers for virtually all systems
  - There is a lot of existing C code in use
  - Typical language of choice for systems-level programming, operating systems, embedded systems etc.
- It is the base for many other languages
  - like C++, Objective C, Java

Standards?

- Yes, different standards (C89, C90, C95, etc.)
- see https://en.wikipedia.org/wiki/ANSI_C

In general, we will follow the C89 standard.

- newer standards include a number of features that are less frequently used

# The C Language

Think of C as a multi-purpose, portable, but 'lightweight' adaptation of assembly language programming

- It is not object-oriented
  - That concept came later
  - Although similar styles can be used
- In general, if you can do something in assembly language (i.e. everything that the processor can do), you can also do it in C
  - This does not apply to Java – try using the 'address' of an object, for example
- Fast: C was written for, and on, computers that were not very powerful by modern standards.
- If you expect something to happen 'automatically', then it probably won't.

# Watch out....

## C != Java

- The *syntax* for Java **is** based on C
  - Methods (functions) loops, arrays etc. often look the same
- But some of the underlying concepts are quite different
- This can lead to some quite subtle, but important, differences
- Understanding ***why*** they are different will help you to understand ***when*** they are different

# Example

- Print a 12 times table

```
0           12          0
1           12          12
2           12          24
3           12          36
4           12          48
5           12          60
6           12          72
7           12          84
8           12          96
9           12          108
10          12          120
11          12          132
12          12          144
```

# Example

```
#include<stdio.h>
main()
{
    int start, end, step;
    int x;
    start = 0;
    end = 12;
    step = 1;
    x = start;
    while(x <= end)
    {
        int answer = 12 * x;
        printf("%d \t 12 \t %d\n",x,answer);
        x = x + step;
    }
}
```

file:e1.c

# Example

```
#include<stdio.h>
main()
{
    int start, end, step;
    int x;
    start = 0;
    end = 12;
    step = 1;
    x = start;
    while(x <= end)
    {
        int answer = 12 * x;
        printf("%d \t 12 \t %d\n",x,answer);
        x = x + step;
    }
}
```

**declare** variables before use

file:e1.c

# Example

```
#include<stdio.h>
main()
{
    int start, end, step;
    int x;
    start = 0;
    end = 12;
    step = 1;
    x = start;
    while(x <= end)
    {
        int answer = 12 * x;
        printf("%d \t 12 \t %d\n",x,answer);
        x = x + step;
    }
}
```

**initialize** variables

file:e1.c

# Example

```
#include<stdio.h>
main()
{
    int start, end, step;
    int x;
    start = 0;
    end = 12;
    step = 1;
    x = start;
    while(x <= end)
    {
        int answer = 12 * x;
        printf("%d \t 12 \t %d\n",x,answer);
        x = x + step;
    }
}
```

write output by using 'printf'

file:e1.c

# Example

```
#include<stdio.h>
main()
{
    int start, end, step;
    int x;
    start = 0;
    end = 12;
    step = 1;
    x = start;
    while(x <= end)
    {
        int answer = 12 * x;
        printf("%d \t 12 \t %d\n",x,answer);
        x = x + step;
    }
}
```

which is defined in the 'stdio.h' standard header

write output by using 'printf'

file:e1.c

# Example

```
#include<stdio.h>
int main(void)
{
    int start, end, step;
    int x;
    start = 0;
    end = 12;
    step = 1;
    x = start;
    while(x <= end)
    {
        int answer = 12 * x;
        printf("%d \t 12 \t %d\n",x,answer);
        x = x + step;
    }
    return 0;
}
```

let's be (a bit more...) correct about 'main'

0 indicates all is well...

# Example

```
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
    int start, end, step;
    int x;
    start = 0;
    end = 12;
    step = 1;
    x = start;
    while(x <= end)
    {
        int answer = 12 * x;
        printf("%d \t 12 \t %d\n",x,answer);
        x = x + step;
    }
    return EXIT_SUCCESS;
}
```

defined in 'stdlib.h'`

but using 'EXIT_SUCCESS' is even a bit neater.

# Standard Headers & 'return'

- The *standard libraries* are included like this (similar to an *import* statement in Java)
- The <> brackets are important – they tell the compiler to look in the correct folder

```
#include<stdio.h>
int main()
{
  printf("Hello, COMP281\n");
  return 0;
}
```

- Note: you can get away without having a return value by just using main() and not having a return statement
  - To save space, the header files often won't be shown in the slides

# Declaring variables

A variable is **declared** in the same way as Java

e.g.,

```
int number;
double some_fraction;
```

This simply tells the compiler to declare some space for the variable with the given name.

You may also **initialise** the variable;

```
int number = 0;
```

Question: **If you don't initialise the variable, what value does it take?**

# Declaring variables

A variable is **declared** in the same way as Java

e.g.,

```
int number;
double some_fraction;
```

This simply tells the compiler to declare some space for the variable with the given name.

You may also **initialise** the variable;

```
int number = 0;
```

Undefined!

(i.e. "garbage")

Question: **If you don't initialise the variable, what value does it take?**

# Primitive Data types

- There are several different sizes of integer
  - `char`
  - `short`
  - `int`
  - `long`
  - can also be ‘unsigned’
- The length of these can vary depending on the system you are using!
  - http://en.wikipedia.org/wiki/C_data_types
- Floating point types:
  - `float`
  - `double`
  - `long double`
- A Boolean type ‘bool’...
  - exists in the C99 and C11 standards, but not in C89.
  - The standard gnu compiler (gcc) will compile it, but e.g., Visual Studio will not.
  - workaround using 'typedef', put this at begin of your file:

```
typedef enum { false, true } bool;
```

# Input and Output

- C has various functions for
  - reading from stdin
  - writing to stdout
- For instance:
  - formatted out and input: printf, scanf ← today
  - getchar(), putchar()
  - gets(), puts()

- also have 'f-versions' (e.g., fprintf )
  - for reading writing to other places

- More info: next lecture, 'man' pages, or look online
  - e.g.:
    - "man getchar"
    - http://www.tutorialspoint.com/cprogramming/c_input_output.htm

# Output with `printf`

- `printf` takes
  - a format string
    - includes 'placeholders' for variables to be printed
  - the variables are passed to printf
    - in the same order as in the format string
  - e.g.

```
printf(" x = %d   y = %d  ", x, y);
```

- You should also explicitly print new line characters with "`\n`";

- About **strings**
  - C does not have classes – hence **there are no String objects**!
  - Strings are represented as a sequence (or array) of chars
  - The printing stops when it reaches a char containing the numerical value 0 (which can be written as '`\0`')

# Printing with `printf`

- **`%d` print as decimal integer**
- `%6d` print as decimal integer, at least 6 characters wide (with leading spaces – use %06d for leading 0s)
- **`%f` print as floating point**
  (**`%lf` print as double-precision floating point**)
- `%6f` print as floating point, at least 6 characters wide
- `%.2f` print as floating point, 2 characters after decimal point
- `%6.2f` print as floating point, at least 6 wide and 2 after decimal point (i.e., 3 digits before the point, 2 after)
- `%o` for octal
- `%x` for hexadecimal
- **`%c` for character**
- **`%s` for character string**

# Example (using `for` loop)

```
#include<stdio.h>
main()
{
     int start,end, step;
     int x;
     start = 0;
     end = 12;
     step = 1;
     for (x=start; x<= end; x+= step)
     {
          int answer = 12 * x;
          printf("%d \t 12 \t %d\n",x,answer);
     }
}
```

file:e2.c

- The (initialise, condition, increment) format is the same as Java
  - `while` and `do...while` formats are also the same
- Note that you *declare* the variable `x` **before** you use it in the loop.
  - In ANSI C, **variables must always be declared at the top of the block of code**, before any other statements.

# Input: `scanf`

- `printf` prints output, with a *format string*
- reading input from the keyboard can be done similarly with `scanf`
- Example reading a character string

```
#include <stdio.h>
main()
{
    char input[10];
    scanf("%s", input);
    printf(input);
}
```

file:scan_string.c

# Input: `scanf`

- `printf` prints output, with a *format string*
- reading input from the keyboard can be done similarly with `scanf`
- Example reading a character string

```
#include <stdio.h>
main()
{
    char input[10];
    scanf("%s", input);
    printf(input);
}
```

file:scan_string.c

C has no 'string' type.
Instead it uses array of 'char'

Question: what happens if the input is more than 10 (or really 9! one for the '\0' !) characters?

# Input: `scanf`

- `printf` prints output, with a format string
- reading input from the keyboard can be done similarly with `scanf`
- Example reading a character string

```
#include <stdio.h>
main()
{
    char input[10];
    scanf("%s", input);
    printf(input);
}
```

```
$ ./a.out
123456789
string is: 123456789

$ ./a.out
12345678901
string is: 12345678901

$ ./a.out
123456789012345678901234 5
Segmentation fault
```

Question: what happens if the input is more than 10 (or really 9! one for the '\0' !) characters?

# Input: scanf

- printf prints output, with a format string
- reading input from the keyboard can be done similarly with scanf
- Example reading a character string

```
#include <stdio.h>
main()
{
    char input[10];
    scanf("%s", input);
    printf(input);
}
```

```
$ ./a.out
123456789
string is: 123456789
$ ./a.out
12345678901
string is: 12345678901
$ ./a.out
123456789012345678901234 5
Segmentation fault
```

C has no string type. Instead it uses array of 'char'

Question: what happens if the input is more than 10 (or really 9! one for the '\0' !) characters?

**When executing the program tried to access memory which is not assigned to the program!**

**'runtime error'**

# Input: `scanf`

- `printf` prints output, with

actually, here we are **also overwriting some memory that we should not...!**

'undefined behavior'

that just happens to not give an error in this case

```
$ ./a.out
123456789
string is: 123456789
$ ./a.out
12345678901
string is: 12345678901
$ ./a.out
123456789012345678901234 5
Segmentation fault
```

**When executing the program tried to access memory which is not assigned to the program!**

**'runtime error'**

# Input: read decimals

- You can also read numbers – **decimals** or floating point

```
#include <stdio.h>
main()
{
     int n=0;
     int sum = 0;
    while (n > -1)
    {
    scanf("%d", &n);
    sum += n;
    printf("sum=%d\n", sum);
    }
}
```

file:e3.c

# Input: read decimals

- You can also read numbers – **decimals** or floating point

```
#include <stdio.h>
main()
{
     int n=0;
     int sum = 0;
     while (n > -1)
     {
     scanf("%d", &n);
     sum += n;
     printf("sum=%d\n", sum);
     }
}
```

file:e3.c

"`&n`" is a so-called pointer...

In order for scanf to store the value in `n`, it needs to know where in memory it is stored

much more in this later!

# Input: read floating point

- You can also read numbers – decimals or **floating point**

```
#include <stdio.h>
main()
{
    double value=0;
    double sum = 0;
    while (n > -1)
    {
    scanf("%lf", &value);
    sum += value;
    printf("sum=%lf\n", sum);
    }
}
```

file:e6.c

- Be careful with mixing %f and doubles;
  - `printf` will usually work, but `scanf` will not!
    - and no, the compiler will not warn you!

# Summary of C elements covered

- for, while
- variables
  - declaring (before using!)
  - initialising
- including standard headers.
  - '#' for pre-processor directives
- input/output with scanf/printf
- your first 'segfault'

# Start Practicing!

- Getting started?
  - a few pointers to get you started on vital ("Getting Started")
- My advice:
  - get access to a linux box (or use department machines)
  - write code in a text editor with syntax support
    - E.g., gvim, emacs
  - compile on the command line:
    - $ gcc -Wall INPUT_FILE.c
- Suggested reading (see "resources" on vital)
  - Kernighan&Ritchie chapter 1
  - Bradley chapters 1 & 2

# Review

- Summary:
  - general module info: assessment, feedback, etc.
  - first bits of C syntax
    - using headers
    - declaring and initializing variables
    - functions for in- and output

- You know how to:
  - Compile and run a simple program
  - Print string or numerical output
  - Read string or numerical input
  - Do conditional loops (while, do…while, for)
  - Get started with C