# Administration

- Lecturer is `Jonas.Skeppstedt@cs.lth.se` with office E:2190
- Course site is `https://cs.lth.se/edag01`

- Simpler alternative is `https://jonasskeppstedt.net`
  but the Discord server and Tresorit directory are more useful
- You find course files by following the link *here* at the course home page
- You will get an account on an IBM POWER8 Linux server
- You can work on other machines if you wish but performance measurements are to be done on this.

# Lab computer: IBM S882L

- POWER8

- 1 TB RAM

- 20 cores

- 160 hardware threads

- Cost around 1500 EUR from ebay

- `ssh -Y stilid@power8.cs.lth.se`

# Contents of the course

Sedgewick and Flajolet in "An Introduction to the Analysis of Algorithms":

*The quality of the implementation and properties of compilers, machine architecture, and other major facets of the programming environment have dramatic effects on performance.*

*You will learn the C language in detail and a methodology to maximize algorithm performance on a modern computer*

*To write efficient code, you need competence in:*

- Mathematics, algorithms and data structures
- The C programming language and UNIX C programming tools
- Pipelined and superscalar processors
- Cache memories
- What optimizing compilers can do for you — and what you need to fix yourself

# Contents Lecture 1

- The purpose of learning C
- Some simple C programs

# Some views of C

- The *other* language for high-performance, FORTRAN, is mainly focussed on numerical computing and not for writing code eg for embedded systems, operating system kernels, or compilers.

- Very often other languages such as Clojure, Python, Scala, Haskell, Lisp, Prolog, Ada, Java, C++, Mathematica, or Matlab are preferable because they have many convenient features which enable faster program development.

- When performance in terms of memory usage and/or speed is *the* most important aspect, however, the programmer must have complete control over what is happening and then the overhead of many language features can lead to inferior performance.

# Your lecturer's relationship with C

- C is great but not ideal for *everything*.
- It is my favorite language since 1988. Just like Lisp and Prolog, it's nice because it's beautiful, powerful, and is simple.
- I have written the second ISO validated C99 compiler, after `edg.com`.
- If I would manage a large software project with several million lines of code, I would use C.
- I will not try to convince you that C "is best" because there is no such thing as a best language.

# Principles of the C Programming Language

- Trust the programmer

- Don't prevent the programmer from doing what needs to be done

- Keep the language small and simple

- Provide only one way to do an operation

- Make it fast, even if it is not guaranteed to be portable

- Support international programming

# Writing a C program

```c
#include <stdio.h>

int main(int argc, char** argv)
{
        printf("hello, world\n");
        return 0;
}
```

- A Java methods is called a function in C.
- A C program must have a **main** function.
- A C function must be declared before it is used.

# The C preprocessor

- The command **#include <stdio.h>** reads a file with a declaration of printf.

- Commands in a C file which start with a hash, **#**, are performed by the C preprocessor before the compiler starts.

- You can run the preprocessor by typing **cpp**.

- The preprocessor can include files and deal with macros, eg **INT_MAX** is the largest number of type **int**.

- Notice that cpp knows nothing about C syntax.

# Installing the gcc and clang compilers on Windows

- Install Windows subsystem for Linux
- See Tresorit and the file `links.txt` for links to youtube videos (and in the comments part of this video)
- Click on the Ubuntu app and you will get a terminal window.
- Become Ubuntu administrator by typing and press return (or enter)

  `sudo bash`
- Update some files by typing:

  `apt update`
- and then

  `apt upgrade`
- and install

  `apt install gcc clang`
- and to leave administrator mode type

  `exit`

# Installing the clang compiler on a Mac

- Search for and open a terminal window on your Mac

- Then type

  `xcode-select --install`

- Other compilers can be installed using the brew system but you don't need to use them

# Installing Linux on a computer with Apple silicon

- `https://asahilinux.org/`

- Copy the command there and paste into a terminal

- You need to be administrator of your computer

- This way you can use Valgrind on your Mac laptop

- If you make serious mistake, you can destroy your computer

- HDMI does not work yet

- Tresorit app is only for macOS, X86 Linux and Windows

# Compiling a C program

- In this course we will use the GNU C compiler, called **gcc**.

- To compile one or more C files to make an executable program type **gcc hello.c**

- The command gcc will first run cpp, then the C compiler, and then two more programs called an assembler and a link-editor.

- Later in the course you will learn about assembler and the operating system course you can learn about link-editors.

- For this course, gcc takes care of the link-editor and tells it to produce an executable file.

# Running a C program

- By default the executable file (made by typing gcc hello.c) is called **a.out**.

- To execute it in Linux (or MacOS X, or another UNIX), type **./a.out**.

- You can tell gcc that you want a certain name: **gcc hello.c -o hello**.

- Now you type **./hello**.

# Separate compilation

- If you have many big source code files, it is a waste of time to recompile all files every time.

- You can tell gcc to compile a file and produce a so called object file (has nothing to do with object-oriented programming).

- **gcc -c hello.c**

- **gcc hello.o**

- The above two lines are identical to **gcc hello.c** but useful if you have many files. The second line should then contain all .o files.

# Example of I/O: scanf and printf

```c
#include <stdio.h>
int main(int argc, char** argv) // same as String args[]
{
        int      a;
        float    b;
        double   c;

        scanf("%d %f %lf", &a, &b, &c);
        printf("%lf\n", a + b + c);
}
```

- %d for int, %f for float, and %lf for double.
- The program will read three numbers from input and print the sum.

# More about the previous example

- In the call to the function **scanf**, we need & to tell the compiler that the variables should be modified by the called function.

- This does not exist in Java. You cannot ask another method to modify a number passed as a parameter to the method.

- Other useful format-specifiers include:
  - %x for a hexnumber (base 16),
  - %s for a string,
  - %c for a char,

# Writing to files in C

```c
#include <stdio.h>
int main(int argc, char** argv)
{
        int     a = 1;
        float   b = 2;
        double  c = 3;
        FILE*   fp;

        fp = fopen("data.txt", "w"); // open the file for writing.
        fprintf(fp, "%d %f %lf\n", a, b, c);
        fclose(fp);
}
```

- This will create a new file on your hard disk.

# Reading from files in C

```c
#include <stdio.h>
int main(int argc, char** argv)
{
        int     a;
        float   b;
        double  c;
        FILE*   fp;

        fp = fopen("data.txt", "r"); // open the file for reading.
        fscanf(fp, "%d %f %lf", &a, &b, &c);
        fclose(fp);
}
```

- Note again the & since fscanf will modify the variables.

# Three ways to make arrays in C

```c
#include <stdio.h>
#include <stdlib.h>

int size = 10;

int main(int argc, char** argv)
{
        int     a[10], n, i;
        int*    b;
        int     c[size];                // called a variable length array.

        sscanf(argv[1], "%d", &n);      // assumes program is run eg as $ ./a.out 10

        b = calloc(n, sizeof(int));     // like Java's b = new int[n];

        for (i = 0; i < n; i += 1)
                b[i] = i; // use b as if it was an array

        free(b);
}
```

# Explanation of the previous slide

- The **a** and **c** arrays are allocated with other local variables.

- Note that **a** and **c** are "real" arrays.

- On the other hand, b is like an array in Java for which you must allocate memory yourself. Use **new** in Java and eg **calloc** in C.

- Java automatically takes care of deallocating the memory of objects.

- In C you must do it yourself using **free**.

- The variable **b** is not an array — it is a pointer.

# Variable length array in C99 and C11

```
int fun(int m, int n)
{
        int     a[n];
        int     b[m][n];
}
```

- Before C99 the above was illegal due to m and n are not constants.

- In C99 it is OK to write like that but only for local variables.

- Most C compilers still only support C89 and thus it may be wise to stick to that at least sometimes.

- Variable lengths arrays are only optional in C11.

# Class in Java vs Struct in C 1(4)

- C has no classes.
- C has structs which are Java classes with everything public and no methods.

```
struct s {  // this s is a tag.
        int     a;
        int     b;
} s;          // this s is a variable identifier.
```

- Struct names have a so called **tag** which is a different namespace than variables and functions: so the above declares a **struct s** which is a type and a variable **s**.
- If we write **Link p** in Java we declare **p** to be a reference but not the object itself whereas **s** above is the *real* object, or data.

# Class in Java vs Struct in C 2(4)

- In Java we can declare a List class something like this:

```
class List {
        List    next;    // Next is a reference to another object.
        int     a;
        int     b;
}
```

- **next** above only holds the address of another object but **next** *is not a List object itself*. The list does not contain a list.

- Java let's you use pointers conveniently without giving you too much head ache.

- C does not.

# Class in Java vs Struct in C 3(4)

- We cannot write the following in C:

```
struct list_t {
        struct list_t     next;    // Compilation error!!
        int               a;
        int               b;
};
```

- It is impossible to allocate a list within the list!

- We really want to declare **next** to simply hold the address of a list object.

- In C this is done as: **struct list_t\* next;** which makes **next** a pointer.

# Class in Java vs Struct in C 4(4)

- The following is correct in C:

```
struct list_t {
        struct list_t*    next;
        int               a;
        int               b;
};
```

- After going into pointers in more detail we will see how to avoid typing **struct list_t** more than twice using **typedef**.

# Memory

- As you all know, your computer has something called **memory**.

- It is sufficient to view it as a huge array: **char memory[4294967296];**

- It is preferable in the beginning to view it as: **int memory[1073741824];**

- Forget about strings for the moment. Now our world consists only of ints.

- As you know, a compiler translates a computer program into some kind of language which can be understood by a machine.

- That has happened for the software in everybody's mobile phone.

# Instructions

- You will see more details about it later, but the C program which controls your phone is translated to commands which are numbers and can be represented as ints.

- These ints are also put in the memory.

- We can for instance put the instructions at the beginning of the array.

- The instructions will occupy a large number of array elements.

- No problem — our array is huge.

# Global variables in memory

```
int x = 12;

int main()
{
        return x * 2;
}
```

- We also put the variable x in the memory.
- This program will have a few instructions for reading x from memory, multiplying with two, and returning the result.
- It is a good idea to put x after the instructions: next page

# Memory layout

| 0 | READ from 3 into R | read the data in **x** from memory at address 3 |
|---|---|---|
| 1 | MUL 2 | R = R * 2 |
| 2 | RETURN | return R |
| 3 | 12 | **x** lives here |

- The array element where we have put a variable is called its **address**

- The instructions above are not written as integers but rather as commands to make them more readable.

- An instruction is represented in memory as a number however.

- It would be too complicated to demand that the hardware should read text such as **MUL** — it is easier is to build hardware if there simply is a number which means multiplication.

# Function calls and local variables

- When you call a function or method, all the local variables must be stored somewhere.

- It is a convention to put them at the end of the memory array.

- The local variables of the main function are put at the very end of the array.

- When main calls a function, its local variables are put just before main's.

- In general, when a new function starts running, it puts its local variables at the last (highest index) unused memory array elements.

- This works like a stack of plates: main is at the bottom and you put newly called functions on the plate at the top.

# The Stack

```
int main()          int f(int a)          int g(int a)
{                   {                     {
    int x = 12;         int b = a+1;          return a + 3;
    return f(x);        return g(b+2);   }
}                   }
```

| 1073741817 | 15 | a in g lives here. |
|---|---|---|
| 1073741818 | | return address from g is here. |
| 1073741819 | 13 | b in f lives here. |
| 1073741820 | 12 | a in f lives here. |
| 1073741821 | | return address from f is here. |
| 1073741822 | 12 | x in main lives here. |
| 1073741823 | | return address from main is here. |

- When a function returns, it deallocates its memory space.

- This is managed by the compiler which uses a register for holding the current free memory index, called the **stack pointer**.

# Pointers

```c
int x = 12;
int *p;
int main()
{
        p = &x;
        *p = 13;
        return x * 2;
}
```

- A pointer is just a variable and it can hold the address of another variable.

- When **p** points to x, writing **\*p** accesses x.

# Memory layout

|   | instruction/data | Java | comment |
|---|---|---|---|
| 0 | STORE 6 at 7 | MEMORY[7] = 6 | &x is put in element 7, ie p |
| 1 | READ from 7 into R | R = MEMORY[7] | read data in p: R=6 |
| 2 | STORE 13 at R | MEMORY[R] = 13 | *p = 13 |
| 3 | READ 6 into R | R = MEMORY[6] | fetch the value of x |
| 4 | MUL 2 | R = R * 2 | multiply x and R |
| 5 | RETURN | return R |  |
| 6 | 12 |  | x lives here |
| 7 | 0 |  | p lives here |

# More about pointers

- In Java, you have used pointers all the time, but they are called object references.

- Suppose you have **Link p**, then **p** is a pointer.

- In Java, pointers can only point at objects.

- The address of some object is, as you might know, the location in memory where that object lives, ie just an integer number.

- In Java, **new** returns the address of a newly created object.

- In C, **new** does not exist and instead a normal function is used (malloc or calloc).

# More about pointers

- In C, but not in Java, the programmer can ask for the address of almost anything and thus get a pointer to that object (or function).

- To change the value of a variable in a function, you need to pass the address of the variable as a parameter to the function:

```
void f(int* a)
{
        *a = 12;
}
```

```
void g()
{
                int    b;

                f(&b);
}
```

# More about pointers

- If the type of the variable is a pointer, then you will need two stars:

```
void f(int** a)                     void g()
{                                   {
        *a = NULL;                          int*     b;
}
                                            f(&b);
                                    }
```

# More about pointers

- To return multiple values in Java, you create and return an extra object.

- Option 1 in C: use a plain struct which is allocated on the stack.

- Option 2 in C: Pass additional arguments as pointers (preferable).

```
struct s f()                    void g(int* x, int* y, int* u)
{                               {
        struct s a;
        a.x = 1;                        *x = 1;
        a.y = 2;                        *y = 2;
        a.u = 3;                        *u = 3;
        return a;               }
}
```

# Arrays vs Pointers

- Arrays and pointers are not equivalent!

- An array declares storage for a number of elements, except when it is a function parameter:

```
int fun(int a[], int b[12], int c[3][4]);
int fun(int *a,  int *b,    int (*c)[4]);
int main()
{
        int     x, y[12], z[4];
        fun(&x, y, &z); // valid.
}
```

- The compiler changes the first [ ] to * for array parameters.

- Array parameters are not arrays. They are pointers.

- Doing so avoids copying large arrays in function calls.

# C has row-major matrix memory layout

```
int c[3][4] = { { 1, 2, 3, 4}, { 5, 6 }, { 7 } };
int i, j;
for (i = 0; i < 3; i++)
      for (j = 0; j < 4; j++)
            x += c[i][j];
```

- In a two-dimensional array, one row is layed out in memory at a time, ie row-major.

- Could also be called "rightmost index varies fastest".

# Arrays as parameters

```
int fun(int c[3][4])
{
        printf("%zu %zu\n", sizeof c, sizeof c[0]);
}
```

- If the output is "8 16", what conclusions can we draw about the size of a pointer and the size of an int?

- Answer: an pointer is eight bytes and an int is four bytes.

- The variable **c** in the function is simply a pointer: **int (*c)[4]**.

# Representation of array references

- a[i] is represtented as **\*(a+i)** internally in the compiler.

```
int main()
{
        int     a[10], *p, i = 3;

        /* the following are equivalent: */

        i[a];
        a[i];
        p = a; p[i]; i[p];
        p = a+i; 0[p]; p[0]; *p;
}
```

# Memory allocation in C

1. Variables with static storage duration (globals, static).

2. Stack variables.

3. `alloca(size_t size)` takes memory from the stack.

4. `malloc/calloc/realloc` take memory from the heap.

# Use tools to find memory errors

- Memory errors:
  - Use pointer which does not point to anything
  - Index out of bounds
  - Forget to free — called a memory leak
  - Free twice
- Two tools you will use in Lab 3
  - Valgrind
  - Google Sanitizer

# Global variables and functions

- Visible from others source files.

- Automatically set to zero unless there is an initializer:

```
int x;
int y = 1;
int f() { return x * y; }
```

- Often it is best to avoid global variables due to:
  - Compilers are not good at using them efficiently
  - They sometimes make it more difficult to understand the program

# Static variables and functions

- Similar to global variables and functions

  ```
  static int x;
  static int y = 1;
  static int f() { return x * y;}
  ```

- Only visible in the scope it is defined

- Functions can only be defined at file scope — no nested functions!

- Always use static instead of global unless the symbol is "exported" to other files

- There is no syntax in C to export symbols — use a header file with declarations

# Stack variables

- Easy for compilers to use efficiently

- Don't use huge arrays since the stack may be too small

- You can use a struct as a parameter and return value — but not array

- As we saw arrays are converted to pointers in the declaration

- There is no syntax to return an array — only a pointer:

```
int a[10];
int* f()                        int* g()
{                               {
    return a; // ok                 int a[10];
}                                   return a;     // bad idea
                                }
```

- The pointer returned from g becomes invalid immediately

# Stack variable

- No automatic initial value — just garbage
- We can initialize a struct or array:

```
int main()
{
    int     a[10] = { 1, 2, 3 };

}
```

- Zero is used for the "missing" expressions

# alloca

- Takes memory from the stack

- Automatically deallocated at function return

- Problem 1: **alloca** is not standard.

- Problem 2: if no memory is available, **NULL** is not returned (as for malloc/calloc).

- Somewhat bad reputation, but nevertheless used.

- Much more efficient than **malloc/calloc**.

# Heap memory

- `void* malloc(size_t s);`

- `void* calloc(size_t n, size_t s);`

- `void* realloc(void* p, size_t s);`

- `void free(void* p);`

# Free-list

- Using Java new or malloc/free takes time
- Sometimes a free-list is useful
- Instead of calling free, put it aside for future use
- Instead of calling malloc, check if there already is something put aside
- With "put aside" is meant putting it in a list — but don't allocate memory for the list!
- Use the object type itself somehow

# sizeof

- Use the **sizeof** operator when requesting memory.
- The **sizeof** operator either takes a type or an expression as operand:

```
int* p; /* lots of code... */ p = calloc(n, sizeof(int));
int* q; /* lots of code... */ q = calloc(n, sizeof *q);
```

- The latter is safer: what happens if somebody changes from `int` to `long` and forgets the sizeof-operand?