Welcome to this course!

Three assignments — they must be correct before you are allowed to write the exam.

No grades on the assignments and you can try as many times as you wish.

Hand in source code through email to edaa25@cs.lth.se.

Write both your name and your social security number on your assignments.

Exam with no help (i.e. no C book either).

Literature: Skeppstedt/Söderberg: ”Writing Efficient C Code: A Thorough Introduction”
Seven lectures

F1 Today: introduction to C
F2 malloc and free, strings, lists
F3 Types, conversions, linkage
F4 Declarations
F5 Expressions and statements
F6 The C preprocessor
F7 The C library
Hints for passing the course

- Do the programming assignments with the help of GDB and Valgrind.
- Of course you can discuss things with friends or me if you want to.
- Learn at least the meaning of each keyword.
- Study the book and foremost the examples — based on which grade you aim at. See the reading advice.
- Ask questions to the lecturer at his office hours — 12.30 – 13.00 every week-day.
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 if you know what you are doing
- Provide only one way to do an operation
- Make it fast, even if it is not guaranteed to be portable
- Support international programming
C is great but not ideal for *everything*. C is my default language since 1988. Just like Lisp and Prolog, it’s beautiful because it’s powerful *and* has few language features.

I have written the second ISO validated C99 compiler (EDG was first).

I will not try to convince you that C ”is best” because there is no such thing as a best language — see next slide.

I’m certain C will remain as popular and important as it is now well beyond the next 50 years — the popularity is increasing of this 40-year old language.
Some thoughts on how to select the language for a project

- External requirements.
- Availability of *good* compilers and their price.
- Availability of competent programmers in that language.
- Availability of required third party libraries.
- Interoperability with other languages.
- If your software intended to survive the death of language X, don’t use X.
Writing a C program

```c
#include <stdio.h>

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

- A Java method is called a function in C.
- A C program must have a `main` function.
- A function must be declared before it is used.
- All functions are at file scope, i.e. not declared in a class as in Java.
The C Preprocessor

- The `#include <stdio.h>` includes a file with a declaration of `printf`.
- `#` directives in a C file 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.
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 fixes 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 files, it is a waste of time to recompile all files every time.
- You can tell gcc to compile a file and save it in 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.
Primitive types

- Types such as int, float etc are sometimes called primitive types.
- In Java the size of each primitive type is specified which is necessary for making Java portable.
- In C the sizes are specified only by their minimum sizes.
- A char is at least 8 bits.
- An int is at least 16 bits.
- An long is at least 32 bits.
- An long long is at least 64 bits.
- By including <stdint.h> we can use types with specified widths — if supported by the compiler.
Ranges and not widths

- Actually, except for char the other primitive types are not specified by their widths but by their ranges.
- By including <limits.h> we can find the number of bits in a char in CHAR_BIT.
- The minimum ranges for some types are:
  - signed char: $-127 \ldots 127$.
  - unsigned char: $0 \ldots 255$.
  - signed short: $-32767 \ldots 32767$.
  - unsigned short: $0 \ldots 65536$.
  - signed int: $-32767 \ldots 32767$.
  - unsigned int: $0 \ldots 65536$.
- The reason the minimum value for example for a signed char is not $-128$ is that some machines don’t use that range.
- The actual ranges are also specified in <limits.h>.
- In C we also have unsigned integer types — in Java only char is an unsigned type.
Example of I/O: scanf and printf

```c
#include <stdio.h>
int main(int argc, char** argv)
{
    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 hex number (base 16),
  - `%s` for a string,
  - `%c` for a char,
Writing to files in 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");
    fprintf(fp, "%d %f %lf\n", a, b, c);
    fclose(fp);
    return 0;
}

- 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");
fscanf(fp, "%d %f %lf", &a, &b, &c);
fclose(fp);
    return 0;
}
```

- Note again the & since fscanf will modify the variables.
The size of an object

- When we allocate memory for an array in Java, we can say:
  \[
  b = \text{new int}[n];
  \]
- The Java compiler knows the size of an int.
- That knowledge has also the C compiler, but the C compiler is not involved in allocating memory on the heap — where all Java objects are stored.
- That is done using library functions as we will see.
- Therefore there is an operator in C to ask for the size of a type: sizeof.
  \[
  \text{int} \quad a;
  \]
  \[
  \text{sizeof} \quad a;
  \]
  \[
  \text{sizeof(int)}
  \]
- The type of a size is some unsigned integer type, called size_t.
I/O with the type size_t

- size_t n;

- Should we use %d with printf to print n?
  - No, %d is wrong since size_t is an unsigned type.
- Should we use %u ?
  - No, that may be too small.
- Can we use %llu like this:
  ```c
  printf("n = %llu\n", (unsigned long long)n);
  ```
  - Yes, but that is often a waste.
- We should use %zu like this:
  ```c
  printf("n = %zu\n", n);
  ```
```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    int a[10], i;
    size_t n;
    int* b;

    sscanf(argv[1], "%zu", &n); // run as $ a.out 10
    b = calloc(n, sizeof(int)); // b = new int[n];

    for (i = 0; i < n; i += 1)
        b[i] = i;
    free(b);
    return 0;
}
```
The a array is allocated with other local variables.

Note that a is a "real" array.

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

```c
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.
- In C11, variable length arrays are optional — but supported by GCC.
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 of our study of C 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.
You will see more details about it in other courses, but the C program written for 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

```c
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

<table>
<thead>
<tr>
<th></th>
<th>READ from 3 into R</th>
<th>read the data in x from memory at address 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>MUL 2</td>
<td>R = R * 2</td>
</tr>
<tr>
<td>2</td>
<td>RETURN</td>
<td>return R</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>x lives here</td>
</tr>
</tbody>
</table>

- 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.
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, typing *p the machine will access x.
- The access will be a write or a read depending on the context.
- *p = 0; /* x is written */
- y = *p; /* x is read */
- a[*p] = 0; /* x is read and selects where to write */
<table>
<thead>
<tr>
<th>instruction/data</th>
<th>Java</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STORE 6 at 7</td>
<td>MEM[7] = 6</td>
</tr>
<tr>
<td>1</td>
<td>READ from 7 into R</td>
<td>R = MEM[7]</td>
</tr>
<tr>
<td>2</td>
<td>STORE 13 at R</td>
<td>MEM[R] = 13</td>
</tr>
<tr>
<td>3</td>
<td>READ 6 into R</td>
<td>R = MEM[6]</td>
</tr>
<tr>
<td>4</td>
<td>MUL 2</td>
<td>R = R * 2</td>
</tr>
<tr>
<td>5</td>
<td>RETURN</td>
<td>return R</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>