Lecturer is Jonas.Skeppstedt@cs.lth.se with office E:2190
Office hours during LP1 and LP4: every lunch at 12.30 - 13.00.
Lectures at 8 on Mondays and Thursdays at 10, both in MA:6
Course web page http://cs.lth.se/edaf15
You will get accounts on the following machines:
  - power.cs.lth.se — a 4 processor Power/Linux machine 2.5 GHz
You can work on other machines if you wish but performance measurements are to be done on these.
You can access it with ssh -Y user@power.cs.lth.se...
Contents of the course

F1 Introduction, the project, and motivation
F2 More C examples
F3 The ISO C Standard
F4 The Oprofile and Valgrind tools demonstration
F5 Introducing the Fast code project: Fourier-Motzkin Elimination
F6 Instruction set architectures. The Power architecture
F7 Pipelined processors
F7 Superscalar processors
F8 Cache memories
F9 Optimising compilers
F10 Writing fast and/or small code
F11 Code tuning
F12 Example exam
You will learn a methodology to maximise algorithm performance on a specific real machine (we will use PowerPC G5)

To write efficient code, you need competence in:

- Mathematics, algorithms and data structures (not the focus in this course)
- The C programming language and UNIX C programming tools
- Pipelined and superscalar processors
- Cache memories
- What optimising compilers can do for you — and what you need to fix yourself
Contents Lecture 1

- Reflexions on the use of C
- Some simple C programs
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 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.
Citation of Dr. Mayan Moudgill, Chief Software Architect at Sandbridgetech (in the DSP business) on the choice of implementation language for compilers (from comp.compilers 2004):

Optimizations have to be done *fast*; even when doing whole program compilation of very large programs (100kloc+), users expect compile times of at best a few minutes. And for the case of compiling single large files (say 10kloc) people expect compile times to be considerably under a minute, even with all optimizations turned on.

With the more aggressive optimizations, memory usage will be quite large. If you’re doing whole program analysis, you could easily exceed 0.5 GB.

Compare that to compiling C++
Both of these suggest that you need to worry about both performance and memory usage. Of course, if you’re dealing with small compile units, and simple optimizations, then this may not be an issue.

...

Summary: if you’re worrying about the language in which to write a compiler, you’ve probably got more serious problems. I’ve written about 7 serious compiler/interpreters/translators in C, LISP and C++, and IMO C is the best language for writing compilers.

If I had to pick a second-best, it would probably be Ada-95.
C is great but not ideal for *everything*. 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).

If I would manage a large software project with several million lines of code, I would still use C.

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

*Update for the new ISO C version: Don’t trust the programmer.*
In 1827 Fourier published a method for solving linear inequalities in the real case. This method is known as Fourier-Motzkin elimination and is used in compilers as an approximation.

If Fourier-Motzkin elimination finds that there is no real solution, then there certainly is no integer either. But if there is a real solution, there may or may not be an integer solution.

Fourier-Motzkin elimination is regarded as a time-consuming algorithm and to apply it so perhaps thousands of data dependence tests may make the compiler too slow. Therefore, it is used as a backup tests when other faster tests fail to prove independence.
An interesting question is how frequently Fourier-Motzkin elimination finds a real solution when there is no integer solution. Some special cases can be exploited.

For instance, if a variable $x_i$ must satisfy $2.2 \leq x_i \leq 2.8$ then there is no integer solution. Otherwise, if we find eg that $2.2 \leq x_i \leq 4.8$ then we may try the two cases of setting $x_i = 3$ and $x_i = 4$, and see if there still is a real solution.

It is easiest to understand Fourier-Motzkin elimination if we first look at an example.
Fourier-Motzkin Elimination

Assume we wish to solve the following system of linear inequalities.

\[
\begin{align*}
2x_1 & - 11x_2 & \leq & 3 \\
-3x_1 & + 2x_2 & \leq & -5 \\
x_1 & + 3x_2 & \leq & 4 \\
-2x_1 & & \leq & -3
\end{align*}
\] (1)

We will first eliminate \(x_2\) from the system, and then check whether the remaining inequalities can be satisfied. To eliminate \(x_2\), we start out with sorting the rows with respect to the coefficients of \(x_2\):

\[
\begin{align*}
-3x_1 & + 2x_2 & \leq & -5 \\
x_1 & + 3x_2 & \leq & 4 \\
2x_1 & - 11x_2 & \leq & 3 \\
-2x_1 & & \leq & -3
\end{align*}
\] (2)
First we want to have rows with positive coefficients of $x_2$, then negative, and lastly zero coefficients.

Next we divide each row by its coefficient (if it is nonzero) of $x_2$:

\[
\begin{align*}
-\frac{3}{2} x_1 + x_2 & \leq -\frac{5}{2} \\
\frac{1}{3} x_1 + x_2 & \leq \frac{4}{3} \\
\frac{2}{11} x_1 - x_2 & \geq \frac{3}{11}
\end{align*}
\]  

Of course, the $\leq$ becomes $\geq$ when dividing with a negative coefficient.

We can now rearrange the system to isolate $x_2$:

\[
\begin{align*}
x_2 & \leq \frac{3}{2} x_1 - \frac{5}{2} \\
-\frac{1}{3} x_1 + \frac{4}{3} \\
\frac{2}{11} x_1 - \frac{3}{11} & \leq x_2
\end{align*}
\]
At this point, we make a record of the minimum and maximum values that $x_2$ can have, expressed as functions of $x_1$. We have:

$$b_2(x_1) \leq x_2 \leq B_2(x_1)$$

(5)

where

$$b_2(x_1) = 2x_1$$

$$B_2(x_1) = \min\left(\frac{3}{2}x_1 - \frac{5}{2}, -\frac{1}{3}x_1 + \frac{4}{3}\right)$$

(6)
To eliminate $x_2$ from the system, we simply combine the inequalities which had positive coefficients of $x_2$ with those which had negative coefficients (ie, one with positive coefficient is combined with one with negative coefficient):

$$ \begin{align*}
\frac{2}{11} x_1 - \frac{3}{11} & \leq \frac{3}{2} x_1 - \frac{5}{2} \\
\frac{2}{11} x_1 - \frac{3}{11} & \leq -\frac{1}{3} x_1 + \frac{4}{3}
\end{align*} $$

These are simplified and the inequality with the zero coefficient of $x_2$ is brought back:

$$ \begin{align*}
-\frac{29}{22} x_1 & \leq -\frac{49}{22} \\
-\frac{17}{33} x_1 & \leq \frac{53}{33} \\
-2x_1 & \leq -3
\end{align*} $$
We can now repeat parts of the procedure above:

\[
\begin{align*}
    x_1 &\leq \frac{53}{17} \\
    x_1 &\geq \frac{49}{29} \\
    x_1 &\geq \frac{3}{2}
\end{align*}
\]  

(9)

We find that

\[
\begin{align*}
    b_1() &= \max(49/29, 3/2) = 49/29 \\
    B_1() &= 53/17
\end{align*}
\]  

(10)

The solution to the system is \(\frac{49}{29} \leq x_1 \leq \frac{53}{17}\) and \(b_2(x_1) \leq B_2(x_1)\) for each value of \(x_1\).
procedure fourier_motzkin_elimination(x, A, c)
    r ← m, s ← n, T ← A, q ← c
    while (1) {
        n₁ ← number of inequalities with positive \( t_{rj} \)
        n₂ ← n₁ + number of inequalities with negative \( t_{rj} \)
        Sort the inequalities so that the \( n₁ \) with \( t_{rj} > 0 \) come first,
        then the \( n₂ - n₁ \) with \( t_{rj} < 0 \) come next,
        and the ones with \( t_{rj} = 0 \) come last.
        for (i = 1; i ≤ r - 1; i ← i + 1)
            for (j = 1; i ≤ n₂; j ← j + 1)
                \( t_{ij} ← t_{ij} / t_{rj} \)
        for (j = 1; i ≤ n₂; j ← j + 1)
            \( q_j ← q_j / t_{rj} \)
        if (\( n₂ > n₁ \))
            \( b_r(x_1, x_2, \ldots, x_{r-1}) = \max_{n₁+1 \leq j \leq n₂} (-\sum_{i=1}^{r-1} t_{ij} x_i + q_i) \)
        else
            \( b_r ← -\infty \)
        if (\( n₁ > 0 \))
            \( j_r(x_1, x_2, \ldots, x_{r-1}) = \min_{n₁+1 \leq j \leq n₂} (-\sum_{i=1}^{r-1} t_{ij} x_i + q_i) \)
        else
            \( B_r ← \infty \)
        if (r = 1)
            return make_solution()
/* We will now eliminate $x_r$. */
$s' \leftarrow s - n_2 + n_1(n_2 - n_1)$
if ($s' = 0$) {
    /* We have not discovered any inconsistency and */
    /* we have no more inequalities to check. */
    /* The system has a solution. */
    The solution set consists of all real vectors $(x_1, x_2, \ldots, x_m)$,
    where $x_{r-1}, x_{r-2}, \ldots, x_1$ are chosen arbitrarily, and
    $x_m, x_{m-1}, \ldots, x_r$ must satisfy
    $b_i(x_1, x_2, \ldots, x_{i-1}) \leq x_i \leq B_i(x_1, x_2, \ldots, x_{i-1})$ for $r \leq i \leq m.$
    return solution set.
}
/* There are now $s'$ inequalities in $r-1$ variables. */
The new system of inequalities is made of two parts:
$\sum_{i=1}^{r-1} (t_{ik} - t_{il})x_i \leq q_k - q_j$ for $1 \leq k \leq n_1, n_1 + 1 \leq j \leq n_2$
$\sum_{i=1}^{r-1} t_{ij}x_i \leq q_j$ for $n_2 + 1 \leq j \leq s$
and becomes by setting $r = r \leftarrow 1$ and $s \leftarrow s'$:
$\sum_{i=1}^{r} t_{ij}x_i \leq q_j$ for $1 \leq j \leq s$
}

function make_solution()
/* We have come to the last variable $x_1$. */
if ($b_1 > B_1$ or (there is a $q_j < 0$ for $n_2 + 1 \leq j \leq s$))
    return there is no solution
The solution set consists of all real vectors $(x_1, x_2, \ldots, x_m)$,
such that $b_i(x_1, x_2, \ldots, x_m) \leq x_i \leq B_i(x_1, x_2, \ldots, x_m)$ for $1 \leq i \leq m.$
return solution set.
end
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.
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.
#include <stdio.h>
int main(int argc, char** argv)
{
    int    a;
    float  b;
    double c;

    scanf("%d %f %lf\n", &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.
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,
#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.
#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\n", &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 run eg as $ a.out 10
    b = calloc(n, sizeof(int));    // like to Java’s b = new int[n];
    for (i = 0; i < n; i += 1)
        b[i] = i;                // play around with b as if it was an array
    free(b);
}
```
The \textbf{a} and \textbf{c} arrays are allocated with other local variables.

Note that \textbf{a} and \textbf{c} are "real" arrays.

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

Java automatically takes care of deallocating the memory of objects.

In C you must do it yourself using \textbf{free}.

The variable \textbf{b} is not an array — it is a pointer.
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.
C has no classes!

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

```c
struct s {
    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.
In Java we can declare a List class something like this:

```java
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.
We cannot write the following in C:

```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.
The following is correct in C:

```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`.
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.
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
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.
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); }

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.
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

<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>MEMORY[7] = 6</td>
</tr>
<tr>
<td></td>
<td>&amp;x is put in element 7, ie p</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>READ from 7 into R</td>
<td>R = MEMORY[7]</td>
</tr>
<tr>
<td></td>
<td>read data in p: R=6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>STORE 13 at R</td>
<td>MEMORY[R] = 13</td>
</tr>
<tr>
<td></td>
<td>*p = 13</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>READ 6 into R</td>
<td>R = MEMORY[6]</td>
</tr>
<tr>
<td></td>
<td>fetch the value of x</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>MUL 2</td>
<td>R = R * 2</td>
</tr>
<tr>
<td></td>
<td>multiply x and R</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>RETURN</td>
<td>return R</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>x lives here</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>p lives here</td>
</tr>
</tbody>
</table>
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` is a normal function and is called `malloc`.
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:

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

void g()
{
    int b;
    f(&b);
}
```
If the type of the variable is a pointer, then you will need two stars:

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

void g() {
    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).

```c
struct s f()
{
    struct s a;
    a.x = ...; *x = ...;
    a.y = ...; *y = ...;
    a.u = ...; *u = ...;
    return a;
}

void g(int* x, int* y, int* u)
{
    *x = ...;
    *y = ...;
    *u = ...;
}
```
The **typedef** command creates another name for the specified type:

```c
typedef int integer;
integer a,b;

typedef struct list_t list_t; // list_t is not a variable!!
struct list_t {
    list_t* next;
    int a;
    int b;
};
```
Typedef names, eg `list_t` declared above live in the same namespace as variables.

This makes parsing C a bit complicated:

```c
typedef int x;
x f(x, x (*)(x, x (*)(int x)));
```

All `x` above except the last is a type. Don't worry about reading this.

However, parsing C is trivial compared to parsing C++.
typedef struct _list_t list_t; // initial underscore is bad
struct _list_t {
    list_t* next;
    int    a;
    int    b;
};

- Two errors: starting a tag (or identifier) with an underscore is permitted **only for compiler and library implementors**. Your compiler may do anything (the technical term is **undefined behaviour**).
- There is no need to invent two identifiers here: call both the typedef name and the tag the same thing!
Arrays vs Pointers

- Despite common belief, arrays and pointers are not equivalent.
- An array declares storage for a number of elements, except when it is a function parameter:

  ```c
  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

```c
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 laid out in memory at a time, i.e., row-major.
- Could also be called "rightmost index varies fastest".
int fun(int c[3][4])
{
    printf("%zu %zu\n", sizeof c, sizeof c[0]);
}

- If the output is "4 16", what conclusions can we draw about the size of a pointer and the size of an int?
- Answer: both are four bytes.
- The variable c in the function is simply a pointer: int (*c)[4].
• a[i] is represented 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;
    }
1. Variables with static storage duration (globals, static).
2. Stack variables.
3. `alloca(size_t size);` normally adjusts the stack-pointer (no need for free).
4. Allocate from the heap with `malloc/calloc/realloc`. Must `free` explicitly.
5. Write your own malloc-replacement. Can take advantage of your application.
6. Write your own arena-based memory allocator.

   Java has (1) for some static data, (2) for scalars and (4) for objects.
Memory allocation is extremely important to master.
Doing it right will lead to a much faster application.
Doing it wrong will certainly introduce painful bugs.
Memory allocation bugs can show up after you think your code works!
However, it is actually quite simple.
As you all know, Java uses garbage collection and this can make life easier but sometimes you need the exact control.
A variable declared outside all functions is global and is visible from others source files.

Global variables are automatically initialised with zeros.

Explicit initialisers can also specify values other than zero.

Think of global variables as static variables in Java (ie static in a class).

Often it is best to avoid global variables due to:
  - Compilers are not good at putting them in registers.
  - They sometimes make it more difficult to understand the program.
Precede the type specifier with the **static** storage-class specifier.

For example: `static int a;`

Static variables are initialised with zeros (unless initialisers override that).

A static variable is only visible in the scope where it was declared, i.e., in a file, in a function, or in a particular block in a function.

A function can also be declared static making its visibility is limited to the file.

Unless a variable or function must be "exported", you **should** declare it static!
Stack allocated variables are usually allocated registers by the compiler, hence accessing them becomes more efficient.

Allocating large variables on the stack can cause stack overflow.

Returning and using the address, or contents, of a stack-allocated variable is an error.

Parameters are stack variables and at function calls they are given values, as if by assignment (except for arrays; which really are no arrays as parameters).

Passing a huge struct as a parameter will on some platforms result in copying the struct and on others result in passing the address (and letting the called function do the copying only if needed, to preserve the as if behaviour).
void f()
{
    struct { int a, char b, int c[100]; } s = { 1 };  
    if (s.b != 0) abort();
}

- Initialising a part of a stack variable ensures that the rest become initialised to zero (NB: only variables, ie not "padding" between variables in eg a struct).
- Above only s.a is initialised but b and c also becomes initialised, with zero.
- There is typically padding space between b and c and it is not defined whether that is initialised to anything.
To allocate extra memory on the stack, use `alloca`.

No need to free such memory explicitly.

Use only if function returns ”sufficiently often”.

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`. 
To allocate memory for lists, trees, etc, use `malloc` or `calloc` (or `alloca`).

- `malloc` takes one parameter: number of bytes to allocate.
- `calloc` takes two parameters: number of items and size of each item.
- `calloc` returns memory initialised to zero; `malloc` leaves it undefined.

Potential problem with using `malloc` (avoided with `calloc`):

```c
typedef struct { int a, c, d; } info_t;
info_t* info = malloc(sizeof *info);
info->a = ...; info->c = ...; info->d = ...;
```

Then add `b` to `info_t` but forget to initialise it.
Allocating memory with **malloc/calloc** (or **new** in Java) takes time.

Try to avoid allocation eg by reusing data of old objects, using a freelist:

- When allocating an object, see if there is something on the freelist first.
- When deallocating an object, put it first in the freelist instead of returning the memory to **free**.
- Sometimes the freelist grows too much should be deallocated.

There are numerous implementations of **malloc/calloc/free** and you might want to try different ones, or write your own — compulsory project in EDAF35 Operating Systems which starts tomorrow.

Using data already freed is illegal — actually, it is illegal to even look at the address! (although nothing happens on most systems).
Use the `sizeof` operator when requesting memory.

The `sizeof` operator either takes a type or an expression as operand:

- `int* p; ..... p = calloc(n, sizeof(int));` (type in parenthesis)
- `int* p; ..... p = calloc(n, sizeof *p);`

The latter is better because if somebody changes the type, the compiler will automatically calculate the correct value for `sizeof`, otherwise if the type is changed from `int` to `long long` and the operand of `sizeof` is not also changed to `long long` someone (eg you) will suffer from a painful bug at some point in the future.
This is a very interesting topic which you can explore in EDAF01 (Operating Systems with Project)

The basic idea is that all memory allocators are "optimised" for the "average" application.

There cannot exist an allocator which is best for all applications.

Writing the application and the allocator together, both can be adapted to each other.
Memory allocation in C 11(11): an arena allocator

- Somewhat similar to alloca, in that (almost) no free is required.
- An arena is simply an abstract data type which contains a list of buffers.
- A pointer keeps track of the next position in a buffer to use.
- When some data has been allocated from the buffer, the pointer is incremented by the same amount.
- We will return to arenas later in the course.
Programming in C on a non-Linux, non-Mac OS X computer

- Download VirtualBox from virtualbox.org
- Download Ubuntu Linux from ubuntu.com
- Start VirtualBox, and assign at least 10 GB to Linux