I explain shortly how C is compiled into LLVM.

Purpose is to make you understand that

1. C needs static typing.
2. C does not like variables of unpredictable size (arrays, strings, big integers).

static typing: All checks are made at compile time. At run time, no checks are necessary.

C-Execution Model

C was designed to be a portable assembly language. This means that mapping to machine instructions should be easy.

A computer consists of the following parts:

Memory stores words (usually bytes) at certain addresses. The CPU can ask the memory for the value at a certain address (read), and it can ask the memory to store a value at a certain address (write). Such action usually takes 7.5 nanoseconds.

Registers store data inside the CPU. The CPU has not many registers, (16?), and they have designated functions: There may be **int** registers (4 bytes), **float** registers (6 bytes), **double** registers (8 bytes), and **address** registers (4-8 bytes).

Reading and writing into CPU registers is very fast (350 picoseconds) and the CPU can access multiple registers at the same time, while performing an instruction.

Distribution of the Memory

During execution, three areas of memory are assigned to the program.

1. Space for the program itself. The CPU has one register, the program counter, PC that indicates the current point (address) where the program is. After excuting the instruction, the PC is increased to the next instruction. (Unless it was a goto or a branch. In that case, the new address is loaded into PC.)

2. A stack of local variables. This stack usually grows downward. Its current end is indicated by an address pointer called stack pointer, SP. Local variables in a function or procedure are allocated on this stack. Whenever a function or procedure is called, the current PC is stored on this stack as well.

Heap Memory

In addition to the three areas mentioned before, the program can ask the Operating System for blocks of memory. It is the responsibility of the program (and its author or the compiler) to return all memory that is not needed anymore, back to the system.

Forgetting to do this results in a situation where a program has reserved much more memory than it is actually using. If a program is not well-written, it is quite possible that it reserves all memory in the computer, without really needing it.

Such situation is called memory leak. It happens quite often. (Also with disk space.) We will hear about this problem later in the course.

Lots of professionally written software has memory leaks. Spirit Mars Rover had a memory leak which almost led to its loss.

Intermediate Representation

If you want to see intermediate representation, then use **clang**. Clang is C/C^{++}-compiler that compiles into LLVM.

Use option:

```
clang++ -emit-llvm -c -S prog.cpp -o prog.llvm
```

```
double fact( unsigned int i )
{
   double res = 1.0;
   while( i != 0 )
   {
      res = res * i;
      i = i - 1;
   }
   return res;
}
```

```
; Function Attrs: nounwind uwtable
define double @_Z4factj(i32 %i) #3 {
  %1 = alloca i32, align 4
  %res = alloca double, align 8
  store i32 %i, i32* %1, align 4
  store double 1.000000e+00, double* %res, align 8
  br label %2

; <label>:2                                     // ( i != 0 )?
  %3 = load i32, i32* %1, align 4
  %4 = icmp ne i32 %3, 0
  br i1 %4, label %5, label %12
```

```
; <label>:5                                 // res = res * i
  %6 = load double, double* %res, align 8
  %7 = load i32, i32* %1, align 4
  %8 = uitofp i32 %7 to double
  %9 = fmul double %6, %8
  store double %9, double* %res, align 8
                                  // i = i - 1
  %10 = load i32, i32* %1, align 4
  %11 = sub i32 %10, 1
  store i32 %11, i32* %1, align 4
  br label %2

                  // return res
; <label>:12
  %13 = load double, double* %res, align 8
  ret double %13 }
```

What did the compiler do for us

- Reserve memory space for `i` and `res`. (Namely `%1` and `%res`.)
- Decide that `int` will be `i32`.
- Decide that `-` and `!=` work on `i32`, and that `*` works on `double`.
- Insert a conversion from `i32` to `float` in `res = res * i`

Next

The compiler will optimize the LLVM (mostly avoid unecessary `loads` and `stores`, use registers as much as possible).

Translate the LLVM into machine instructions. This implies replacing the `allocas` by stack allocations and deallocations.

It will be something like:

```
%1 = SP; SP = SP - 4;
%res = SP; SP = SP - 8;
```

The *C* model assumes that variable sizes are known at compile time. At initialization time would also work, but it would be difficult. Reassignment is impossible. Therefore, *C* cannot handle variable size objects in local variables.

Observations

So now you understand why you have to declare variables in *C*. The compiler needs this information for selecting the proper LLVM instructions (which later will be machine instructions).

You also understand why local variables of unpredictable length are problematic (Strings, Inheritance, Arrays of Variable Length)

Because of this, people write such ugly code:

```
#define MAXNAME 200
char name [ MAXNAME ];
print( "Hi, what's your name?" );
scanf( "%s", name );
   // Lot of space being wasted, and possibly
   // still not enough.
```