C fundamentals


Grégory Mounié (CC-BY-SA 4.0)
2023-2024
The C language
Computer hardware basics in few slides
C basics
Function parameters: local copy (stack) of the arguments
Preprocessor
Binary operators
Advanced C
Programming in C
Static analysis
Advanced concepts for later time this year
And now
Content of the lecture and the practical
This presentation: to understand the C design and philosophy to master it.

Practical labs in following sessions: small code to fill in, with unit testing for instant validation. If finishing early, you may skip the following sessions, or continue with new ideas and ask for new questions.
Two tracks (you have roughly 2 weeks before OS lecture praticals)

If you have no previous experience of C

- **Write code** (not copy-pasting): quickly at least 1000 lines (3h-6h)
- **Books**: *Modern C*, Jens Gustedt (level by level book),
- [https://exercism.io](https://exercism.io) has many (80+) C exercises

If you are fluent in C: our dedicated small set of exercises and tutorials

The C language
PDP-11 was the first computer with an unified memory and byte addressing.

Ken Thompson and Dennis Ritchie writing UNIX around 1970 on a PDP11

"Something as complex as an operating system, which must deal with time-critical events, had to be written exclusively in assembly language”.

Then, in 1973, ...

... Ken Thompson and Dennis Ritchie create the C language with the second version of UNIX, and the 4th UNIX core was rewritten in C.
Fairy tale 🧜‍♀️: major principles of C design

C is designed to write an operating system. Thus, C core can not use any operating systems services!

C does nothing, but it does it really well

**Simplicity** a simple C compiler is small and easy to write, with a mostly direct translation to the assembly code.

**Portability** compiling on probably every architecture from 1973 to now (eg. GCC officially includes 51 architectures, including the PDP-11, 30 more exist separately)

**Explicit** no hidden computational complexity in the code. Never!

**Crystal clear memory semantic** and memory management using **pointers**.
Modern processor design includes many cores, many tricks, speculative pipeline, hierarchical cache, cache line, prefetch, etc.

Modern C compilers interpret the programmer intends to change the code. GCC and Clang are neither small, simple nor fully explicit anymore (portability remains strong). They guess the programmer intends to improve performance on modern hardware. Indeed, Rust is becoming the contender of C, as being stricter (predictable) but Riust is neither simple nor small too.

No atomic memory behavior in the hardware explicit requirement if needed.

```c
1 int *a = ...; // some of the 4 bytes may cross the boundary
2 *a = 123; // of two successive cache lines
```
C uses only **simple concepts** and **simple rules**. Programming is not simple, thus the language usage is **difficult to master**.

1. **All the burden is on the programmer shoulders.** He has to write explicitly almost everything, including strings, arrays, list, hashes and memory management.

2. **Memory management has to be perfect** to avoid pointer usage pitfall (Infamous **SEGFAULT**):
   - Freeing memory, once and on time, is especially difficult
     - Java and Go use Garbage Collector: slower and difficult to control
     - Objective-C use reference counter, slower and memory leak are still possible,
     - Rust uses 18 different kinds of pointer for most basic cases, and **unsafe**.

**Memory checkers (Valgrind, bound checkers, static analysis, etc.),**
They save your days. Use them!
The goals of this lecture and the practical

• Learn C, one of the two most popular languages for decades!

But why studying C and not D, Rust, Go, ...?

• C language is small, fewer things to explain
• C programming exposes the programmer to the subtleties of computers and operating systems. Other languages hide many of these subtleties to ease the programmer task.
• Mastering C required to understand these subtleties: no shortcut or misunderstanding.
This lecture is not about C syntax and grammar

Hopefully, you probably already know quite a lot about C syntax, even without ever writing a single line of C, as its syntax is the part of the syntax of numerous languages, old and new, such as Java, C++, Objective-C, Swift, Kotlin, Perl, Raku, Go, Rust, etc. (eg. curly brackets for blocks)

```c
float function_name(unsigned int a, int b) {
    if (a == b) {
        for(int i=0; i < 10; i++)
            break;
    } else
        while( --a ) continue;
    return 0.0;
}
```
Computer hardware basics in few slides
A processor, basically, do only few things:

- **Program counter** contains the address in memory of the current instruction (irreducible part).

- **Binary arithmetic operations** (+, -, ×, and /), integer and floating point. Arguments and results are read and write, from and to, registers. Logical tests may be done on register values and the results may influence the program counter evolution.

- Using the bus, the processor ask the memory for values at one particular address, and loads the values in its registers, or the processor stores the values of the registers into the memory.
• The memory store **binary data** (0 or 1, one bit)
• The data storage unit is the **byte** (8 bits)
• The memory is an array of bytes. The index of the byte in the array is the **address** of the byte.
• the **instructions** of the program in execution are **in the memory** too, with other data, as it is the only large storage device readable fast enough by the processor.
Fundamentals of hardware history

- Since *Multics*, ancestor of UNIX, the addresses in the processor registers are virtual addresses and are translated before reaching the memory bus (pagination).
- Up to the 1990, the memory was faster than the processor. Then the processor got caches.
- Since 2000, the frequency of the processors is stable around few Ghz. To increase the performances, the processors became *multi-core*. 
Fundamentals of programming language

Few disturbing facts often forgotten by most programmers

- General purpose registers have no type! A register store just a small sequence of bytes (a word)
- Data in memory have no type! Again, the memory is just a big array of bytes.
- The data types exist only in the instruction sequence of the running program. The instructions define the way to manipulate the data in the memory.
  - A character (letter) is just an integer
  - A string is just a sequence of integer

The pointer concept: data type view of the memory

In C, the pointers are the basic tool to explain to the compiler the data types used, their sizes, and where to read and write them.
At each function call, a new frame is stacked and it contains the local variables of this call of the function. Thus, after the end of the call, the local variable should not be used anymore (through pointer on the variable), as the next call will write at the same place. (The bug of the C test :) (In fact, there are 4 bugs in the C test :-))
```c
struct Record {
    char name[128];
    int age;
};

struct Record *read_record(FILE *f) {
    struct Record r; // allocation in the STACK
    int res = fscanf(f, " %s %d", r.name, &r.age);
    if (res == 2)
        return &r; // &r should not be used after the end
    else
        return NULL;
}
```
Correct code with dynamic allocation

```c
struct Record {
    char name[128];
    int age;
};

struct Record *read_record(FILE *f) {
    struct Record *r = calloc(1, sizeof(struct Record)); // HEAP + init
    int res = fscanf(f, " %127s %d", r->name, &(r->age));
    if (res == 2)
        return r; // you have to free(r) once later
    else
        return NULL;
}
```
Alternative: correction with copy

```c
struct Record { // fixed size type, thus C can copy it.
    bool error;
    char name[128]; // it is not equivalent to char* name
    int age;
};

struct Record read_record(FILE *f) { // return a copy
    struct Record r = {};
    int res = fscanf(f, "%127s %d", r.name, &r.age);
    if (res == 2)
        return r; // copy
    else {
        r.error = true; // and test failure with it outside
        return r; // copy
    }
}
```
C basics
Integer of various sizes
char/short/int/long [int]/long long [int], signed (default), or unsigned and with bit size int8_t/uint8_t [#include <stdint.h>]
(bitsize 8, 16, 32, 64)

Floating point, scalar and complex, of various sizes
float/double/long double and float/double/long double complex [#include <complex.h>]

Char (1 byte) and wide char (multi-bytes encoding of char)
char, wchar_t [#include <wchar.h>]

Boolean
bool [#include <stdbool.h>]

Query the byte size of any type
sizeof(type) or sizeof(variable)
Suffix modifier (infix usage for simple types)

```c
const int a = 10; // really: int const a = 10;
const int *pa = &pa; // really: int const *pa
int b = 10;
int *const pb = &b; // pointer const
*pb = 12; // b = 12
```
Type variable_name[size]

- int t0[20]; single dimension array from 0 to 19. (20 int × (sizeof(int) == 4) thus 80 contiguous bytes in memory)
  t0[4]: the fifth element

- char t1[10][20]; 2 dimensions (10 blocs of 20 char). Indice 0 à 9 et 0 à 19 (200 contiguous bytes).

Question
memory position of t1[2][1] from the array beginning? (C is row order)
Array initialization

```
int t0[30] = {2, 1, 4};
```

Everything else is initialized at 0.

```
int t0[30] = {}; everything is initialized at 0
int t0[30] nothing is initialized
```

Beware: nothing is free! Initialization after-effect

- First touch allocation of memory (multi-thread programming)
- cache pollution
- RAM filling with unused memory full of 0
- useless computation
Again, a datatype is not in memory, neither in the processor register.
A datatype exists only in the sequence of instructions!
A pointer is the C tool to explain to the compiler the datatype of a range of bytes.
Everything in C is a pointer.
Pointers explain to the compiler how to handle a bunch of bytes starting at one particular address (types and size of the bunch).

The bunch of bytes is **contiguous**.

The pointer will be used to read or write the data. Partial access are, sometimes, possible.
Pointer declaration

1

```c
TYPE *NAME;
```

2

```c
int *p0; // pointer to int
```

3

```c
void *p; // universal pointer (no type)
```

25
Pointer, access to the value: *ptr

```c
int n = 20;
int * ad_n; // declaration
ad_n = & n; // address of n
*ad_n = 30; // n=30
(*ad_n)++; // n=31
*ad_n++; // !!! ad_n == ((& n) + 1)
```
```c
int b;
int n = 0b00010011; // binary notation
int * ad_n = &n;
*(ad_n + 1) = 32; // BUG ! write b or ad_n

// ad_n is a pointer to an int.
// ad_n+1 = @ of the next integer !!!
// if ad_n == 0x00000000 then
// ad_n+1 == 0x00000004 (as sizeof(int) == 4)
```
Array as a pointer

```c
int t[20];
```

t is (roughly) equivalent to &t[0]

```c
t + 1 // &t[1]
t + i // &t[i]
t[i] // *(t+i)
```
Array: commutativity of addition versus square bracket

C square brackets are roughly just syntactic sugar (but real operator in C++)

```
t[2] = 3;
*(t+2) = 3; // t[2] = 3;
*(2+t) = 3; // t[2] = 3
2[t] = 3; // t[2] = 3
```
Dynamic array as a pointer

Recall Static array (global) or local array (stack)

```c
int t0[20]; // fix size
```

Dynamic array (in the heap): only allocation differs

```c
int *t0;
t0 = malloc( sizeof(int[20]) ); // 20 * sizeof(int)

*t0 = 3; // t0[0] = 3
*(t0+1) = 4; // t0[1] = 4
t0[2] = 5; // *(t0+2) = 5
free(t0);
```
Function parameters: local copy (stack) of the arguments
Function arguments are always copies of values of the parameters

This code does not swap values of a and b of main

```c
void swap(int c, int d) {
    int sv;
    sv = d; c = d; d = sv;
    printf("swap c: %d, d: %d\n", c,d);
}

int main(int argc, char **argv) {
    int a=5; int b = 10;
    printf("a: %d, b: %d\n", a,b);  // a: 5, b: 10
    swap (a,b);  // c: 10, d: 5
    printf("a: %d, b: %d\n", a,b);  // a: 5, b: 10
}
```
Thus copy the **addresses** of values to exchange a and b

This code swaps values of a and b of main

```c
void swap(int *c, int *d) { // COPY of a/b addresses
    int sv;
    sv = *c; *c = *d; *d = sv;
    printf("swap *c: %d, *d: %d", *c,*d);
}

int main(int argc, char **argv) {
    int a=5; int b = 10;
    printf("a: %d, b: %d", a,b); // a: 5, b: 10
    swap (&a,&b); // *c: 10, *d: 5
    printf("a: %d, b: %d", a,b); // a: 10, b: 5
}
```
void swap(int c[static 1], int d[static 1]) {
    int sv;
    sv = *c; *c = *d; *d = sv;
    printf("swap *c: %d, *d: %d", *c, *d);
}

int main(int argc, char *argv[argc + 1]) {
    int a=5; int b = 10;
    printf("a: %d, b: %d", a,b); // a: 5, b: 10
    swap (&a,&b); // *c: 10, *d: 5
    printf("a: %d, b: %d", a,b); // a: 10, b: 5
}
**Structure: compound type**

Example: a "point" type grouping 2 integers

```c
struct point {
    int x;
    int y;
};
struct point p0 = { 1, 2 }; // init
typedef struct point Point;
Point p1 = { .y = 10 }; // init
...
p0.x = 2; p0.y = 3; // edit of p0
p1 = p0; // copy of p0 in p1
p0 = (struct point){ 1, 2 }; // copy anonymous to p0
```
A set of integer constant

```c
enum VAL { VAL0=0, VAL3=3, VAL4, VAL5 };  

enum VAL v = VAL4;  
```
Union

Byte sharing. Difficult to handle correctly!

```c
union Data {
    int error_code;
    long double value;
};

struct Result {
    bool ok;
    union Data value;
} r;

r = compute_first();
if (!r.ok)
    printf("error %d\n", r.error_code);
else
    compute_next(r.value);
```
A list of points (basics)

```c
struct liste_point { 
    int x; int y;  
    struct liste_point *next;  
} ;  
struct liste_point *p; 
... 
p->x = 2; p = p->next; 
```
A list of points (v1.0)

```c
typedef struct liste_point {
    struct point p0;
    struct liste_point *next;
} ListePoint;
ListePoint *p;
...
(p->p0).x = 2; p = p->next;
```

**Question**
What the difference in memory placement and usage of both version?
Beware of pointer cast priority

1

```c
struct liste_point *p ;
```

**Pointer arithmetic**

What the difference between

- `p+4`
- `(int) p + 4`
- `((int)p) + 4`
- `(int)(p + 4)`
Preprocessor
Compilation process

1. **preprocessor** to produce the full C file (**include**, **define** stuff)
2. **compilation** to produce assembly file
3. **assembly** to produce object file (machine language)
4. **link** to merge object file in a single application
Macro processor

Often used for constant definition.

```c
// Examples from math.h

#define M_E 2.7182818284590452354 /* e */
#define M_LOG2E 1.4426950408889634074 /* log_2 e */
#define M_LOG10E 0.43429448190325182765 /* log_10 e */
#define M_LN2 0.69314718055994530942 /* log_e 2 */
#define M_LN10 2.30258509299404568402 /* log_e 10 */
#define M_PI 3.14159265358979323846 /* pi */
#define M_PI_2 1.57079632679489661923 /* pi/2 */
#define M_PI_4 0.78539816339744830962 /* pi/4 */
#define M_1_PI 0.31830988618379067154 /* 1/pi */
#define M_2_PI 0.63661977236758134308 /* 2/pi */
```
#include may be done multiple times with the same header or even recurse infinitely. Thus #ifndef... is used to avoid multiple definitions and infinite loop.

```
#ifndef __MEM_ALLOC_H
#define __MEM_ALLOC_H
...
// Definitions
...
#endif
```
Global variables are often declared in a header file. It’s a good practice.

Variables should be initialized. It’s a good practice.

But initializing a global variable in the header file, will declare different variables at every `#include`.

Use `extern` keyword to declare the variable in the header file and initialize the variable in a single C file.
How to define and initialize global variables 2/2

```c
/// in toto.h
extern int toto_globale;
// AND NOT int toto_globale = 10

/// in toto.c
#include "toto.h"
int toto_globale = 10;
```
Do not underestimate the C preprocessor capabilities 1/2

Your linux kernel and the uthash/utlist implement generic data structures fully with macro.

```c
#include "utlist.h"

typedef struct lp { int x;
               struct lp *next; } LP; // NB: link has 'next' name

LP *head=NULL;
LP *elem = malloc( sizeof(LP) );
*elem = (LP){42, NULL};
LL_APPEND(head, elem);
LL_FOREACH(head, elem) {
   printf("%d\n", elem->x);
}
```
Genericity support of C is used in conjunction with the macroprocessor.

```c
// in tgmath.h
#define __tgmath_complex_type2(expr1, expr2) \n  __tgmath_type_if( \n    _Float32x, \n    __tgmath_type_if( \n      _Complex _Float32x, __tgmath_complex_type2_base(expr1, \n        expr2), \n      _Generic((expr1) + (expr2), _Complex _Float32x: 1, \n        default: 0)), \n    _Generic((expr1) + (expr2), _Float32x: 1, default: 0))
```
Binary operators
Classical binary operators

AND &
OR |
NOT ~
XOR ^
SHIFL <<
SHIFTR >>

They should be used on **unsigned** integer.
Exercise

Bit operation
Do the following operations on a single unsigned int a:

1. set the 4th least significant bit to 1
2. set the 3rd least significant bit to 0
3. read the value of the 5th significant bit
4. invert the 6th significant bit

What the difference in your C code between little endian ou big endian processor?
Advanced C
A function name is a pointer. Indeed, ( ) operator can also be used on function pointer.

```c
int fibo(int n) {
    return (n < 2)?1:fibo(n-1)+fibo(n-2);
}
int (*g)(int) = fibo;   // or &fibo, it is the same
struct pf { int (*g)(int); } dummy = { &fibo };

void h( struct pf a, int (*b)(int)) {
    a.g(10) == b(10);
}
h( dummy, fibo );
```
float complex tab[1000];

#include <stdlib.h> // definition of qsort
// void qsort(void *base, size_t nmemb, size_t size,
//     int (*compar)(const void *, const void *));
// compar: return an integer <=> 0 if arg1 <=> arg2
C language changes (slightly) with the time:

- **C K&R**, original from (1972).
- **C ANSI** or **C89**: introduce the modern syntax for function arguments
- **C99** variable array, restrict, complex, single line comment
- **C11** unicode, multi-threads, generic expression, (bound checking)
- **C17** addressed defects and clarification of C11
- **C23** const with array, attributes (fallthrough, nodiscard, deprecated), binary literal
Anonymous variables: beware of the lifetime of your variables.

Automatic variable (stack) have a short lifetime.

```c
struct point { int x; int y; };  // Anonymous variable

int f(struct point p) {...};  // f and its children must not copy and keep the reference
int g(struct point *p) {...};

f( (struct point){ 1, 1 } ); // copy of the anonymous: no problem

// f and its children must not copy and keep the reference

g( & (struct point){ 2, 2 } ); // g and its children must not copy and keep the pointer
```

After the execution of `g`, the argument pointer must not be used anymore!
Designated initializers

Partial initialization

```c
struct point { int x; int y; };

struct point p = { .y = 10 };

int a[100] = { [42] = 10 };

```
Small function may be inlined. They must be written in header file, and they must be instancied once and only once in a .c!

```c
// in MYADD.h
inline int add1( int a) { return a + 1;}

// in a SINGLE .c
#include <MYADD.h>
int add1(int);
```
Unicode/Utf8 support in the strings

Warning: as C does nothing, and thus has no type 'string', a lot of Unicode or Utf8 difficulties are still there.

```c
#include <stdio.h>
#include <wchar.h>
#include <locale.h>

int main() {
    wchar_t string[100];
    setlocale(LC_ALL, "");
    printf(u8"éàï\n"); // utf8 string [C11]
    scanf("%ls", string);
    printf("%ls string of %d glyphs using %d Bytes\n", 
        string, wcslen(string), strlen(string));
}
```
Mathematics and genericity

C has (limited) genericity for standard mathematical function. In the following example: float and long double complex sqrt.

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

int main()
{
    float f = 1; long double complex fc = -1.0 - I;
    printf("%e\n", sqrt(f));
    printf("%Lf + %Lf i = %Lf e^{i %Lf}\n", 
            creal(sqrt(fc)), cimag(sqrt(fc)), 
            fabs(sqrt(fc)), carg(sqrt(fc)));
}```
Annex K: bound and pointer checking at runtime

Library function extension to catch non null pointer and bound checking. The idea and its implementations are controversial and thus never land in major compiler like gcc ou clang. It may be deprecated and removed in the future.

```c
// Example from annex K of C2X draft
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
/*...*/
int n, i; float x; char name[50];
int n = fscanf_s(stdin, "%d%f%s", &i, &x, name, (rsize_t) 50);
```
Programming in C
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
    SSLBuffer signedParams, uint8_t *signature, UInt16 signatureLen)
{
    OSStatus err;
    ...
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...
    fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
Separate compilation allows to build your program in small files. Then the linker will take every object file and build the final program.

This process allows quick, yet correct, compilation of large program constantly modified by its developers.

To check the code, the compiler needs to know the function used but, with their code implemented in other files. The *header* files provide the declaration of the functions.
Detailed example for pedagogical reason. Automatic rules could make it much smaller.

```bash
# classical compilation options
# Warning and debuggers are your best friends
CFLAGS= -Wall -g -Werror -Wextras

# the result program need two objects file
# and link them together with some libraries
memshell: alloc.o memshell.o
   $(CC) $(LDFLAGS) -o memshell alloc.o memshell.o
 → $(LDLIBS)
```
# to produce one object, the compiler needs code
# and the related header file
alloc.o: alloc.c alloc.h

$(CC) $(CFLAGS) -c alloc.c
CFLAGS = -Wall -g -Werror -Wextras

memshell: alloc.o memshell.o
Autoconf/automake, cmake, qmake, scons, ... generate makefile from a small
description.

Small CMakeList.txt (cmake)

1. `projet(Memshell)`
2. `set(CMAKE_BUILD_TYPE Debug)`
3. `add_executable(memshell alloc.c memshell.c)`
Cmake: compile in a subdirectory to avoid garbage in your code

$ ls
CMakeList.txt build/
$ cd build
$ cmake .. # create the makefile
$ make

To keep your source tree clean.
Valgrind is a debugging tool, with many capacities. Its main tool is to follow the usage of pointer to detect errors such as:

- buffer overflow
- usage after free
- double free

But it does not do miracles and, legal memory access, even done strangely, are legal memory access.
Valgrind example: this code has a bug

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

int main(int argc, char **argv) {
    char buffer[256]={}; char *copy;
    scanf("%255s", buffer);
    copy = malloc( strnlen(buffer, 256) );
    strncpy(copy, buffer, 256); return 0; }
```

Even with the carefull usage of `scanf` and the initialization of buffer, this code has a bug.
Valgrind explain the bug at runtime. Read the explanations!

==20137== Memcheck, a memory error detector
==20137== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==20137== Using Valgrind-3.5.0-Debian and LibVEX; rerun with -h for copyright info
==20137== Command: ./a.out
==20137== Invalid write of size 1
==20137== at 0x4025DB0: strncpy (mc_replace_strmem.c:329)
==20137== by 0x80484DE: main (exemple_valgrind.c:10)
==20137== Address 0x41a202c is 0 bytes after a block of size 4 alloc'd
==20137== at 0x4024C4C: malloc (vg_replace_malloc.c:195)
==20137== by 0x80484B8: main (exemple_valgrind.c:9)

==20137== Invalid write of size 1
==20137== at 0x4025DBD: strncpy (mc_replace_strmem.c:329)
==20137== by 0x80484DE: main (exemple_valgrind.c:10)
==20137== Address 0x41a202e is 2 bytes after a block of size 4 alloc'd
==20137== at 0x4024C4C: malloc (vg_replace_malloc.c:195)
==20137== by 0x80484B8: main (exemple_valgrind.c:9)
Valgrind explain the bug at runtime. Read the explanations!

---

HEAP SUMMARY:
- in use at exit: 4 bytes in 1 blocks
- total heap usage: 1 allocs, 0 frees, 4 bytes allocated

LEAK SUMMARY:
- definitely lost: 4 bytes in 1 blocks
- indirectly lost: 0 bytes in 0 blocks
- possibly lost: 0 bytes in 0 blocks
- still reachable: 0 bytes in 0 blocks
- suppressed: 0 bytes in 0 blocks

Rerun with --leak-check=full to see details of leaked memory

For counts of detected and suppressed errors, rerun with: -v

ERROR SUMMARY: 252 errors from 2 contexts (suppressed: 11 from 6)
Dynamic assertions

C provides simple assert. Macro may also be useful.

Macro for dynamic assertion (checked at runtime)

```
#include <assert.h>
#define handle_error(msg) do { perror(msg); exit(-1); } while (0)

int f(int *a)
{
    assert( a != NULL);
    if (most_system_calls(...)) // == 0: OK, false
        // != 0: problem
        handle_error("fail, due to:");
}
```
C provides also compile time assertion.

```c
#include <assert.h>

int f(int *a)
{
    // macro defined in assert.h
    static_assert(sizeof(int) == 4, "32 bits integer required");
    // the real C keyword
    _Static_assert(sizeof(int) == 4, "32 bits integer required");
}
```
Static analysis
Both CLANG and GCC (GCC-10+) propose static analyzers. They try to find common pitfalls in pointer and library usage (double free for example).

In the two following examples, I use them on the code on the file `elempools.c` from the exercises. I remove warnings related to the fact, that the solution is not written yet.

**GCC analyzer**  It finds that the code do not check the result of the malloc, thus it may be NULL, and thus `memset` may failed. It is a bug!

**Clang-analyzer**  It finds that the code allocate an array of one type but use a different pointer type to store the address. It is commonly a bug, but not for this particular case (writing a memory allocator).
GCC-10+, into the classical warning flow

```c
$ gcc -fanalyzer -Wall -Wextra elempool.c
elempool.c: In function ‘init_elems’:
   49 |   memset(memoire_elem_pool, 0, 1000 * sizeof(struct elem));
   ^~~~~~~~~~
‘init_elems’: events 1-3
   / | 46 | if (memoire_elem_pool == NULL) {
   / | | ^
   / | | | 
   / | | | (1) following ‘true’ branch...
   / | 47 |   memoire_elem_pool = malloc(1000 * sizeof(struct
   / | | ~~~~~~~~~~~~~~~~~~~~~~~~~~~
   / | | | 
   / | | | (2) ...to here
   / | 48 | }
   / 49 |   memset(memoire_elem_pool, 0, 1000 * sizeof(struct el
   / | ~~~~~~~~~~~~~~~~~~~~~~~~~~~
   / | | 
   / | | (3) argument 1 (‘<unknown>’) could be NULL where non
```
Clang provides interactive output (HTML)

```bash
$ scan-build-12 clang-12 -c -Wall -Wextra elempool.c

elempool.c:47:29: warning: Result of 'malloc' is converted to a pointer of type 'unsigned char', which is incompatible with sizeof operand type 'struct elem'
↪ [unix.MallocSizeof]
    memoire_elem_pool = malloc(1000 * sizeof(struct elem));
      ^~~~~~ ~~~~~~~~~~~~~~~~~~~
1 warning generated.
scan-build: Analysis run complete.
scan-build: 1 bug found.
scan-build: Run 'scan-view /tmp/scan-build-2020-08-29-183802-19055-1' to examine bug reports.
$ scan-view-12 /tmp/scan-build-2020-08-29-183802-19055-1
```

```c
void init_els() {
    btkReset();
    if (memoire_elem_pool == NULL) {
        memoire_elem_pool = malloc(1000 * sizeof(struct elem));
    }
    memset(memoire_elem_pool, 0, 1000 * sizeof(struct elem));
}
```
Advanced concepts for later time this year
Multi-core memory model of consistency

Copy-paste of the core part of POSIX threads, C proposes threads and atomic operations for multi-core. The goal is to minimize the usage of assembly code. And the C compiler will be able to do more optimization of the code.

#include<stdatomic.h> and #include<threads.h>

- atomic variable
- acquire/release memory consistency model
- several memory consistency models (none to sequential)
- threads
- monitor
And now
Theses slides are an almost full summary of the C language and the C programming.

Now, you have to practice C programming and fall in its pitfalls. Understanding these pitfalls is the real goal of this lecture as these pitfall understanding is valuable in any other language and many other parts of CS.