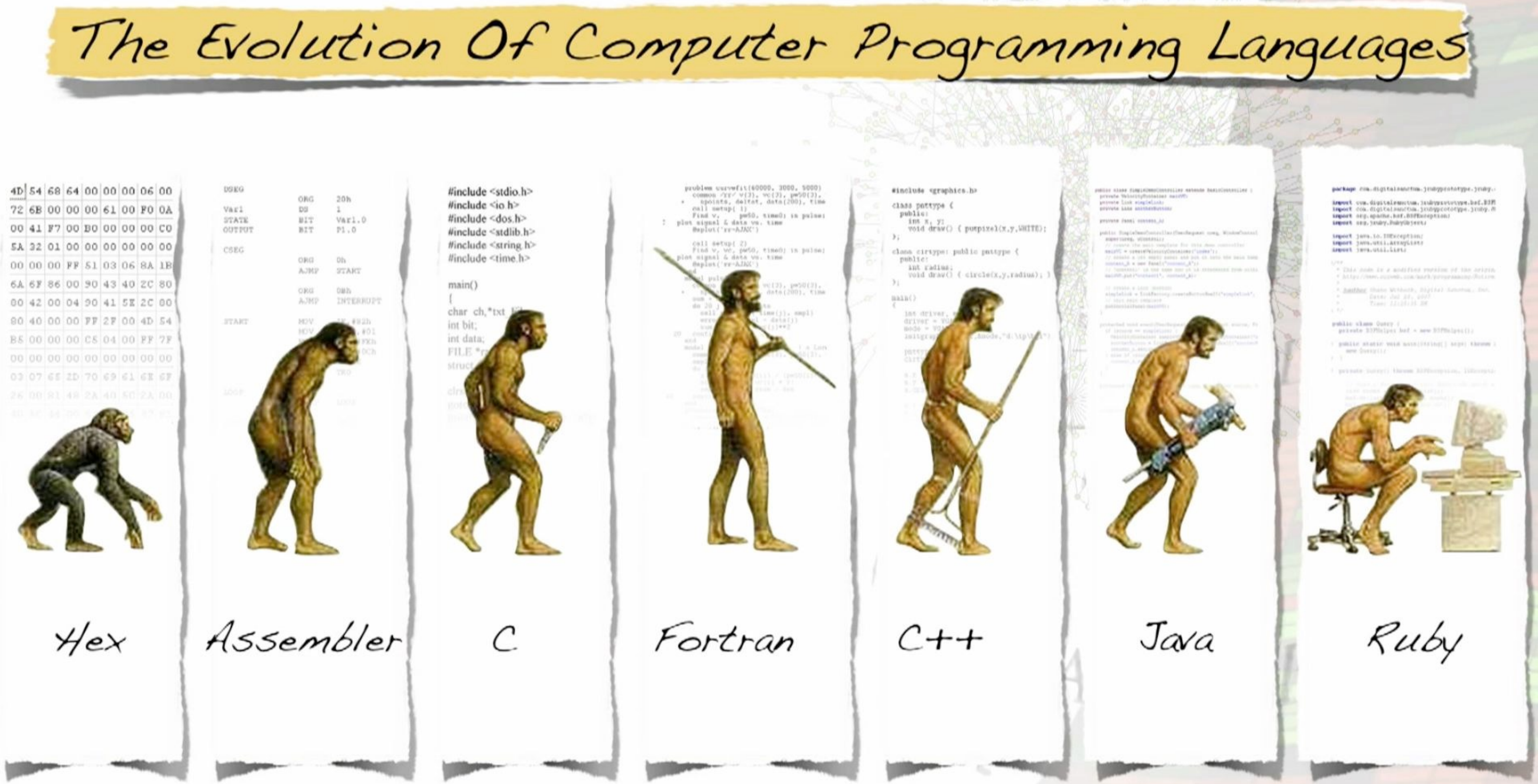
Introduction aux Systèmes d’Exploitation

Unit 9: The C Language

(a very short introduction)

François Taïani



http://www.flickr.com/photos/dullhunk/4833512699/

The C Language

- The ancestor of C++ and Java
- The 'DNA' of most operating systems (incl. Linux)
  - Including many tools and commands (bash, ssh, httpd)
- Pros (= advantages)
  - Close to the hardware
  - Fast
  - Many, many libraries and tools
- Cons (= disadvantages)
  - Difficult to code in (pointers, and mem. management)
- Recent competitors: Go (Google) & Rust (Firefox)

Hello World

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  printf("Hello world!");
} // EndMain
```

- Procedural language (functions, but no objects!)
- Entry point: `main`
- General form of a function:

```
ret_type func(type1 arg1, type2 arg2, ..) {
  <body>
  return <something> // optional
}
```

- Special case: `void` (no return value)

General Syntax

- Very close to Java:
  - Variable declaration & definition
  - Control structure (`for`, `if`, `while`, `do/while`, `switch case`,…)

- But some big differences
  - Obviously no keywords related to OO (`class`, etc.)
  - Some constructions do not exist in C
    - `for(int t: my_collection) {..}`
    - lambdas, monitors, String type, …
  - Some concepts do not exist in Java
    - Pointers (more on this), structs

Basic C Data Types

- Everying is a number: integer or float
- Integer types:
  - `char`, `short`, `int`, `long`, `long long`
  - signed by default (except for char, where it depends)
  - modifiers: `unsigned` (+`signed`)
  - growing sizes (`char` to `long long`)
  - size not standardized, but `char` usually = 1 byte
- Floating point number types:
  - `float`, `double`, `long double`
  - size not standardized (usually 32, 64, and 128 bits)

Char Type

- Dual role : character *and* 8 bit integer

```
#include <stdio.h>

int main( int argc, char** argv) {

  char x = 'A';

  printf("%c\n", x ); // as an ascii character
  printf("%i\n", x ); // as a number in decimal notation
  printf("%x\n", x ); // as a number in hexidecimal notation

  x++;

  printf("%c\n", x );
  printf("%i\n", x );
  printf("%x\n", x );

} // EndMain
```

What is the result of this program?

Note on printf

Printing with **`printf(..)`**

- Part of stdio library: `#include <stdio.h>`
- Basic form: `printf("<some string>")`
- General form:
  `printf("format_s",arg1,arg2,..)`
  - ➔ `"format_s"` contains placeholders for args that follow
  - ➔ `%c` one `char`
  - ➔ `%i` one `int` (decimal output) (`%x`: hexadecimal)
  - ➔ `%s` one string (more on this)
  - ➔ `%p` one pointer (memory address, more on this)
  - ➔ Many, many more

C Arrays

- All basic types can be used to declare arrays
  - ➔ Same syntax as Java: `type var_name[size];`
- Example :
  - ➔ `int   array1[5];`
  - ➔ `short array2[5] = {1,2,3,4,5} ;`
  - ➔ `char  array3[5] = {'a','b','c','d'} ;`
  - ➔ `short array4[]  = {1,2,3,4,5} ;`
  - ➔ `char  array5[]  = "abcd" ;`
- Remember
  - ➔ As in Java, index of 1st elem is 0
  - ➔ Note the implicit sizes of `array4` and `array5`

Pointers

- A pointer = an address + info on what is being pointed
- Compiler remembers what is being pointed to
  - ➔ Denoted by a * after a type
  - ➔ `int` * : a pointer to an int (address that contains an int)
  - ➔ `char` * : a pointer to a char (address that contains a char)
  - ➔ `void` * : a pointer to something unknown
- Manipulating the content of a pointer (dereferencing) : *
- Obtaining the address of a variable : & (referencing)

```
int * ptr_i = &i ;
```

Remembering how * works

$$\boxed{\texttt{int *}}\ \texttt{ptr\_i ;}$$

- `int*` means "pointer to an `int`"
  - ➔ `prt_i` is a pointer to an `int`

$$\texttt{int}\ \boxed{\texttt{* ptr\_i}}\ \texttt{;}$$

- `*prt_i` means "the content of pointer `prt_i`"
  - ➔ `*prt_i` is an `int`

Casting

- Compiler will check that types are respected
  - ➔ E.g.

```
int i = 10;
char* ptr_i = &i;
```

```
$ gcc -g playing_with_pointers.c
playing_with_pointers.c:5: warning: initialization
from incompatible pointer type
```

- Solution: explicit cast

```
char* ptr_i = (char*)&i;
```

Pointers Arithmetic

- Pointers are addresses = numbers
  - → They can be manipulated as numbers
- For instance
  - → `int * ptr_j = ptr_i + 10 ;`
- BUT increment on address <u>multiplied</u> by size of base type
  - → `int` usually on 32 bits (4 bytes)
  - → `ptr_i + 1` → adds 4 to `ptr_i`
  - → `ptr_i + 2` → adds 8 to `ptr_i`
  - → `ptr_i + 10` → adds 40 to `ptr_i`

Example

What is the result of this program?

```
#include <stdio.h>
#include <stdint.h>

int16_t i = 10;
char* ptr_i = (char*)&i;

int main( int argc, char** argv) {

  printf("%p : 0x%02x\n", ptr_i  , * ptr_i    );
  printf("%p : 0x%02x\n", ptr_i+1, *(ptr_i+1));

  *(ptr_i+1) = 1 ;

  printf("0x%04x\n", (int)i );
  printf("%i\n"    , (int)i );

} // EndMain
```

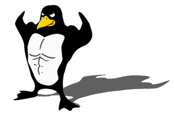
Pointers & Parameter Passing: By Value & By Reference

```
void foo1(int i) {
  i=i+2;
}

void foo2(int* prt_i) {
  (*prt_i)=(*prt_i)+2;
}

int main(int argc, char* argv[]) {
  int i=10;
  foo1(i);
  printf("%i\n",i);
  foo2(&i);
  printf("%i\n",i);
} // EndMain
```

- What does this program print? Why?

Pointers and Arrays

- Very close in C
  - ➔ except for `sizeof()`, and allocation in functions



- If `my_var` is a pointer or an array

$$\texttt{my\_var[k]} \equiv \texttt{*(my\_var+k)}$$

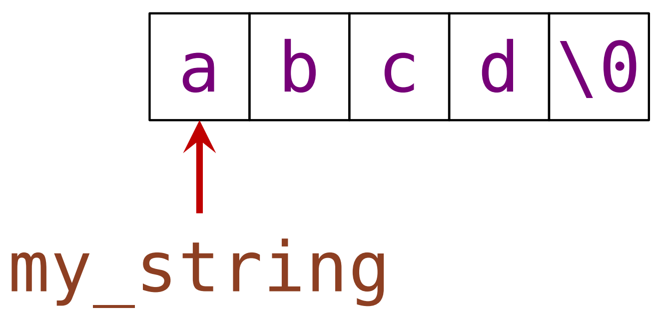
  - ➔ Example:

```
int my_var[]={1,2,3,4,5};
printf("%i\n",  my_var[2]);
printf("%i\n",*(my_var+2));
```

- Why does it work?
  - ➔ Elements of array allocated in adjacent memory zones

Strings in C

- Bad news: no specialized type
- String = char pointer → memory zone ended with `\0`
- Example: `char* my_string="abcd" ;`
  - ➔ Reserves 5 bytes in memory (4 + trailing `\0`)
  - ➔ Assigns address of first bye to `my_string`



  - ➔ Actual values in memory: ASCII codes {97,98,99,100,0}

Strings in C: Example

```
char* my_string = "Hello";

printf("%c\n", *my_string  );
printf("%s\n",  my_string  );

printf("%c\n",   my_string[1]  );
printf("%s\n", &(my_string[1]) );

printf("%c\n", *(my_string+2) );
printf("%s\n",   my_string+2  );
```

- What will it print?

Array, Char & String: Example

```
int main( int argc, char** argv) {
  int i ;
  char x[] = "it's a wonderful world";
  char delta = 'a' - 'A';

  printf("%i\n", delta );

  for(i=0; i < sizeof(x)-1; i++) {
    if (x[i]>='a' && x[i]<='z') x[i] -= delta ;
  } // EnFor

  printf("%s\n", x );
} // EndMain
```

What does this program do?

Notes: Char ≠ string

- '..' very important in previous example
  - ➔ What happens if you use double quotes ".." instead?
  - ➔ Why?

Dangers with Strings in C

- Strings never-ending cause of bugs in C
- 2 main problems
  - ➔ Unbounded copies (buffer overflow, code injection...)
  - ➔ Forgetting trailing \0

```
char my_string[10];
int i=0 ;

while (argv[1][i]!=0) {
  my_string[i]=argv[1][i];
  i++;
}
my_string[i]=0;
```

Where's the problem?

What we have not seen

- Enum, structs, and unions
- The role of header files (*.h)
- Memory allocation (see later unit)
- Preprocessor macros
- Static variables
- Function pointers
- Functions to handle strings
- Functions with variable args (like printf)

Summary

At the end of this session, you should be able to:

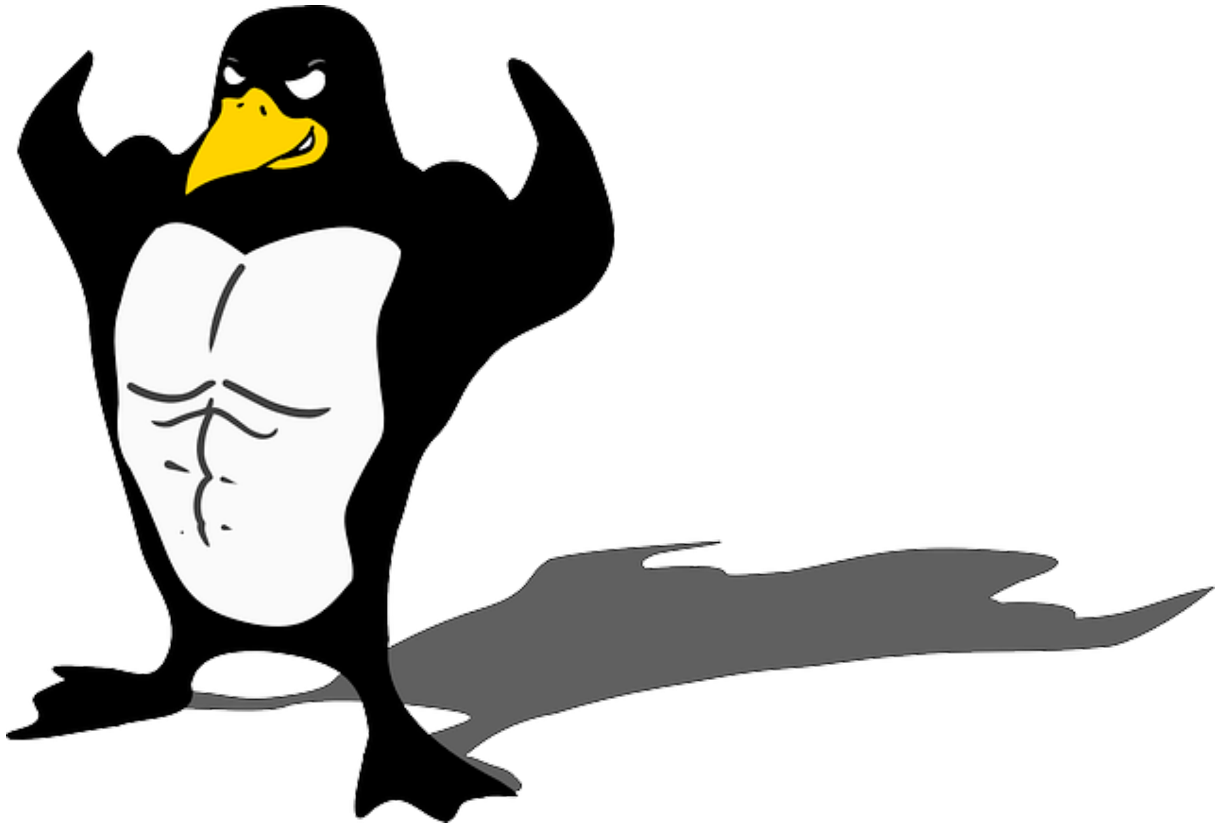
- Present and explain C's basic types;
- Analyse and create simple C programs;
- Understand programs using the dual nature of char;
- Understand how strings are encoded and used in C;
- Analyse simple uses of pointer arithmetic and casting;
- Analyse and explain simple string-related bugs;
- Explain the close link between arrays and pointers.

Further Readings

- "Integer (computer science)" @ wikipedia
  - ➔ http://en.wikipedia.org/wiki/Integer_%28computer_science%2
- x86 quick ref
  - ➔ http://faydoc.tripod.com/cpu/index_m.htm
- limits.h
  - ➔ http://en.wikibooks.org/wiki/C_Programming/C_Reference/limits.h
- Difference between char* and char[]
  - ➔ http://stackoverflow.com/questions/1335786/c-differences-between-char-pointer-and-array

Bonus Material

- Not exam material



Representation of Basic Types

- C supports a number of basic integer types
  - → short, int, long, ... (+char)
- How are they represented in memory ?
  - → bad news : size depends on architecture (≠ Java!)
  - → on 64bit computers usually
    - int : 4 bytes
    - short : half an int (2 bytes)
    - long : twice an int (8 bytes)
- How to know at runtime ? → sizeof() operator

Example

```c
#include <stdio.h>

int main( int argc, char** argv) {

  printf("%-8s: %4lu\n", "char"  , sizeof(char)  );
  printf("%-8s: %4lu\n", "short" , sizeof(short) );
  printf("%-8s: %4lu\n", "int"   , sizeof(int)   );
  printf("%-8s: %4lu\n", "long"  , sizeof(long)  );

} // EndMain
```

```
$ gcc size_of_example.c -o size_of_example
$ ./size_of_example
```

Fine Control

- If controlling representation is important
  - → #include <stdint.h>
- Defines a number of fixed-size number types
  - → int8_t, int16_t, int32_t, int64_t (signed)
  - → uint8_t, uint16_t, uint32_t, uint64_t (unsigned)
- Important for some embedded / low level software

Unsigned Integers

- Default integer types are signed
- But can be made unsigned with '**unsigned**' keyword
  - ➔ Impact on computations and results
  - ➔ Sometimes not always intuitive
- Example:
  - ➔ what is the result of **i** and **j**? (assuming short on 2 bytes)

```
unsigned short i ;
short          j ;

i = 32767 ;
j = 32767 ;

i++ ;
j++ ;
```

Full Code

```
int main( int argc, char** argv) {

  unsigned short i ;
  short          j ;

  i = 32767 ;
  j = 32767 ;

  i++ ;
  j++ ;

  // the compiler applies the correct division operation
  printf("%hu\n",i/32768);
  printf("%hi\n",j/32768);

  // beware : printf does not check that types are correct
  // (%hi = short int, %hu = unsigned short int)
  printf("%hu, %hi\n",i,j);
  printf("%hu, %hi\n",j,i);

} // EndMain
```

```
$ gcc -g example_signed_unsigned.c
$ ./a.out
```

Disassembling

```
objdump -S -Mintel a.out
```

```
printf("%hu\n",i/32768);
400515: 0f b7 45 fc             movzx  eax,WORD PTR [rbp-0x4]
400519: 66 c1 e8 0f             shr    ax,0xf
40051d: 0f b7 d0                movzx  edx,ax
400520: b8 8c 06 40 00          mov    eax,0x40068c
400525: 89 d6                   mov    esi,edx
400527: 48 89 c7                mov    rdi,rax
40052a: b8 00 00 00 00          mov    eax,0x0
40052f: e8 ac fe ff ff          call   4003e0 <printf@plt>
printf("%hi\n",j/32768);
400534: 0f bf 45 fe             movsx  eax,WORD PTR [rbp-0x2]
400538: 8d 90 ff 7f 00 00       lea    edx,[rax+0x7fff]
40053e: 85 c0                   test   eax,eax
400540: 0f 48 c2                cmovs  eax,edx
400543: c1 f8 0f                sar    eax,0xf
400546: 89 c2                   mov    edx,eax
400548: b8 91 06 40 00          mov    eax,0x400691
40054d: 89 d6                   mov    esi,edx
40054f: 48 89 c7                mov    rdi,rax
400552: b8 00 00 00 00          mov    eax,0x0
400557: e8 84 fe ff ff          call   4003e0 <printf@plt>
```

- The compilers knows i unsigned, j signed when dividing

Note on printf(..) formatting

```
printf("%hu, %hi\n",i,j);
40055c: 0f bf 55 fe          movsx  edx,WORD PTR [rbp-0x2]
400560: 0f b7 4d fc          movzx  ecx,WORD PTR [rbp-0x4]
400564: b8 96 06 40 00       mov    eax,0x400696
400569: 89 ce                mov    esi,ecx
40056b: 48 89 c7             mov    rdi,rax
40056e: b8 00 00 00 00       mov    eax,0x0
400573: e8 68 fe ff ff       call   4003e0 <printf@plt>
printf("%hu, %hi\n",j,i);
400578: 0f b7 55 fc          movzx  edx,WORD PTR [rbp-0x4]
40057c: 0f bf 4d fe          movsx  ecx,WORD PTR [rbp-0x2]
400580: b8 96 06 40 00       mov    eax,0x400696
400585: 89 ce                mov    esi,ecx
400587: 48 89 c7             mov    rdi,rax
40058a: b8 00 00 00 00       mov    eax,0x0
40058f: e8 4c fe ff ff       call   4003e0 <printf@plt>
```

- printf(..) does **not** check the types of passed parameters
  - ➔ Parameters interpreted according to formatting string
  - ➔ If inconsistent with actual type: potential bug

More on Pointers and Arrays

```
// Declares 5 contiguous integers
int array[5];
// Arrays can be used as pointers
int *ptr = array;
// Pointers can be indexed with array syntax
ptr[0] = 1;
// Arrays can be dereferenced with pointer syntax
*(array + 1) = 2;
// Pointer addition is commutative
*(1 + array) = 3;
// Subscript operator is commutative
2[array] = 4;
// 5*sizeof(int), size of array
sizeof(array)
// sizeof(int*), 64bit address on 64bit machine
sizeof(ptr)
```

See: https://en.wikipedia.org/wiki/Pointer_%28computing%29

Playing with Structs

- Imagine the following definitions

```
typedef struct {
  unsigned short age;
  char name[10];
} student ;

student all_students[2];
```

Playing with Structs

- What do you think will happen?

```
int i;

strcpy(all_students[1].name,"Malo");
all_students[1].age  = 20;

strcpy(all_students[0].name,"Childeric-Mael-Erwan");
all_students[0].age  = 22;

printf("Voici les membres de la classe:\n");

for(i=0;i<2;i++) {
  printf("%s a %hu ans\n",
          all_students[i].name,
          all_students[i].age);
} // EndFor
```

Notes

- 'M' = 77, 'a' = 97
  - ➔ "Ma" = 97 * 256 + 77 = 24909
  - ➔ When interpreted as unsigned short in **little** endian!
- A case of memory corruption
  - ➔ Even worse when corrupted location = pointer
  - ➔ Usually the cause of "segmentation fault ./a.out"
- Generally fixe-size char[] + dynamic content = bad idea
  - ➔ Either immutable char* s = "Hello";
  - ➔ Or mutable → dynamic allocation (more on this later)