C Programming

Grégory Mermoud

School of Architecture, Civil and Environmental Engineering

EPFL, SS 2008-2009

http://disal.epfl.ch/teaching/embedded_systems/

Outline

- Week 4: main concepts introduced
- Today: **consolidation** and **refinement** of your understanding of C
- Further details about **control** structures
- Variables and other **data** structures
- Functions and parameter passing
- Pointers
- Memory organisation and dynamic allocation of memory (advanced topic)

Data and control

- **Any** computer program has two components:
  - Data structures
  - Control structures
- Control structures update data structures.

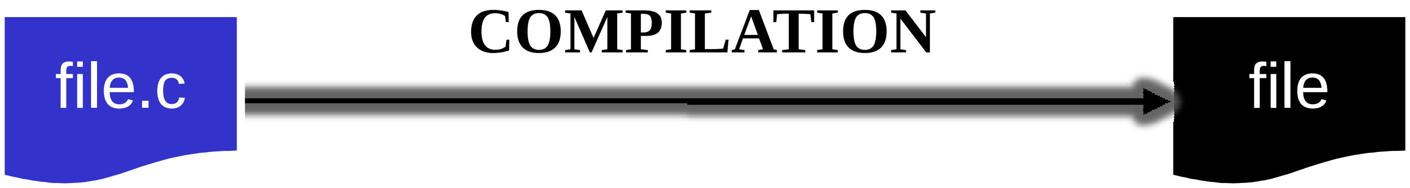
```
int main() {
  int i, j;
  double A[3][3];

  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      A[i][j] = 0.0;
    }
  }
  A[0][2] = 2.0;
  return 0;
}
```

Create a 3x3 matrix A

Initialize all elements of A to 0.0

Set A(0,2) to 2.0

From C code to executable code



file.c

COMPILATION

file

source file

executable

```
int main() {
  int a = 5;
  double b = 4.3;
  return a * b;
}
```

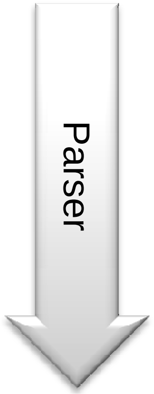
```
10100101010010
10100101001010
10010100101001
00010101001111
00100101010100
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

What does the compiler do?

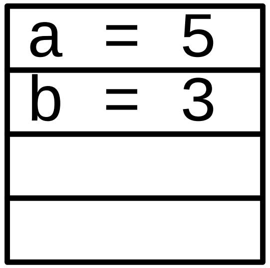
SIMPLIFIED VERSION!

```
int main() {
  int a = 5;
  int b = 3;
  b = a * b;
  return a;
}
```



1) It parses the code

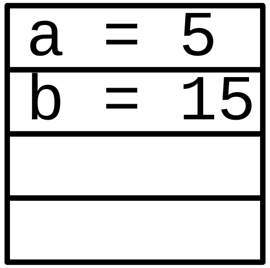
```
int main() {
  int a = 5;
  int b = 3;
  b = a * b;
  return a;
}
```


a = 5
b = 3

Computer memory

2) Upon declaration of a variable, it allocates some memory for it.

```
int main() {
  int a = 5;
  int b = 3;
  b = a * b;
  return a;
}
```


a = 5
b = 15

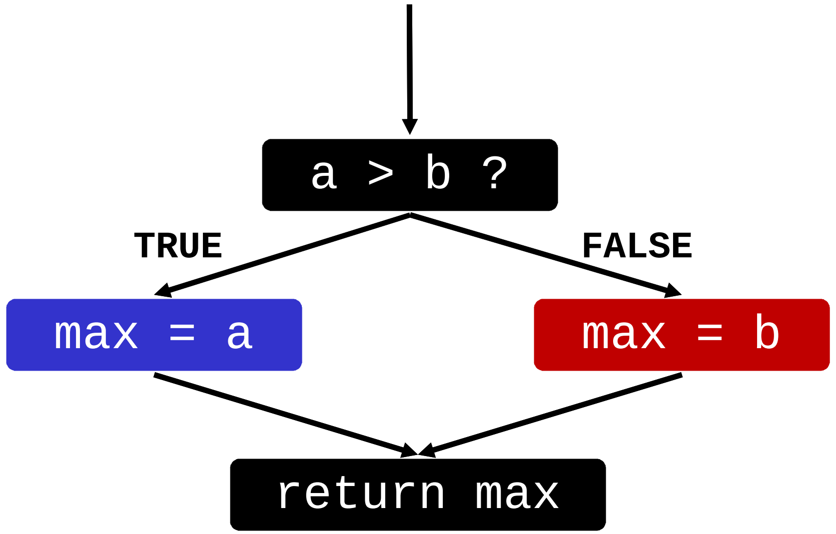
Computer memory

3) It generates executable code for each statement that modifies a data structure.

Controlling the execution flow

- There is one more thing that the compiler does: it controls the **execution flow** of the program.
- It does that by updating a very special variable that is internal to the microcontroller: the Program Counter (PC), which indicates what instruction must be executed next.
- As a C programmer, **you do not care about the PC**. The execution flow can be modified using **control statements**.

```
float max = 0.0;
float a = 5.0;
float b = 2.1;

if (a > b) {
  max = a;
} else {
  max = b;
}

return max;
```



Conditions

- Conditions can be expressed using logical expressions:

`>`	(greater than)
`<`	(less than)
`>=`	(for greater than or equal to)
`<=`	(for less than or equal to)
`!=`	(not equal)
`==`	(to test for equality)

**do not confuse a `==` 1 (equality)
with a `=` 1 (affectation)**

- In C90, there is no boolean variable (`true` or `false`). Instead, `true` is represented by any value not equal to 0 and `false` is represented by the value 0.

WRONG

```
int a = 0;

if (a = 1) {
   // this code is reached
} else {
   // this won't happen
}
```

CORRECT

```
int a = 0;

if (a == 1) {
   // this won't happen
} else {
   // this code is reached
}
```

Conditional branches

- The `switch` structure is very useful when the execution flow depends on the value of a single integral variable (int, char, short, long).

```
switch (a) {
   case 1:
   {
      // if a == 1, do this
      break; // jump to the rest of the code
   }
   case 2:
   {
      // if a == 2, do this
      break; // jump to the rest of the code
   }
   default:
   {
     // otherwise, do this
   }
}
// rest of the code
```

```
if (a == 1) {
   // if a == 1, do this
} else if (a == 2) {
   // if a == 2, do this
} else {
   // otherwise, do this
}
// rest of the code
```

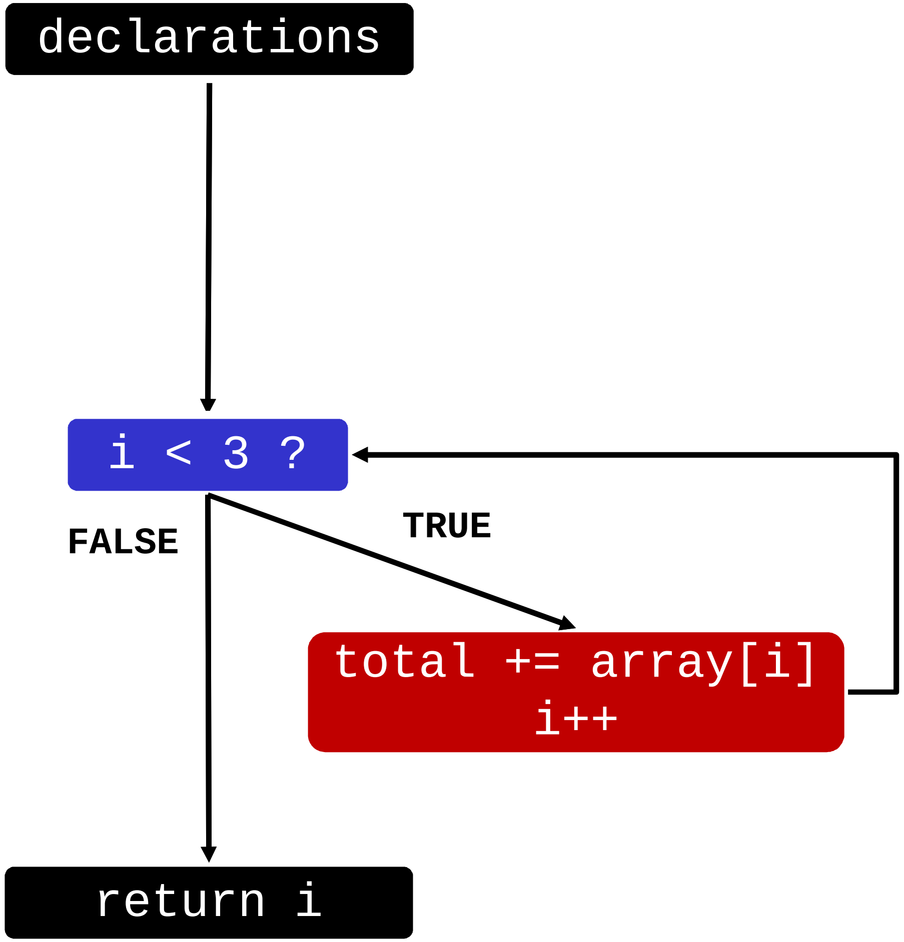
Both codes have exactly the same behavior!

Do not forget the `break` instructions, otherwise the statements in the rest of the `switch` will also be executed!

Conditional loops

Conditional loops are a combination of “if..then” and a jump.

```
int total = 0;
int i = 0;
int array[3] = {12,3,-5};

while (i < 3) {
   total += array[i];
   i++;
}

return  i;
```



declarations

i < 3 ?

FALSE

TRUE

total += array[i]
i++

return i

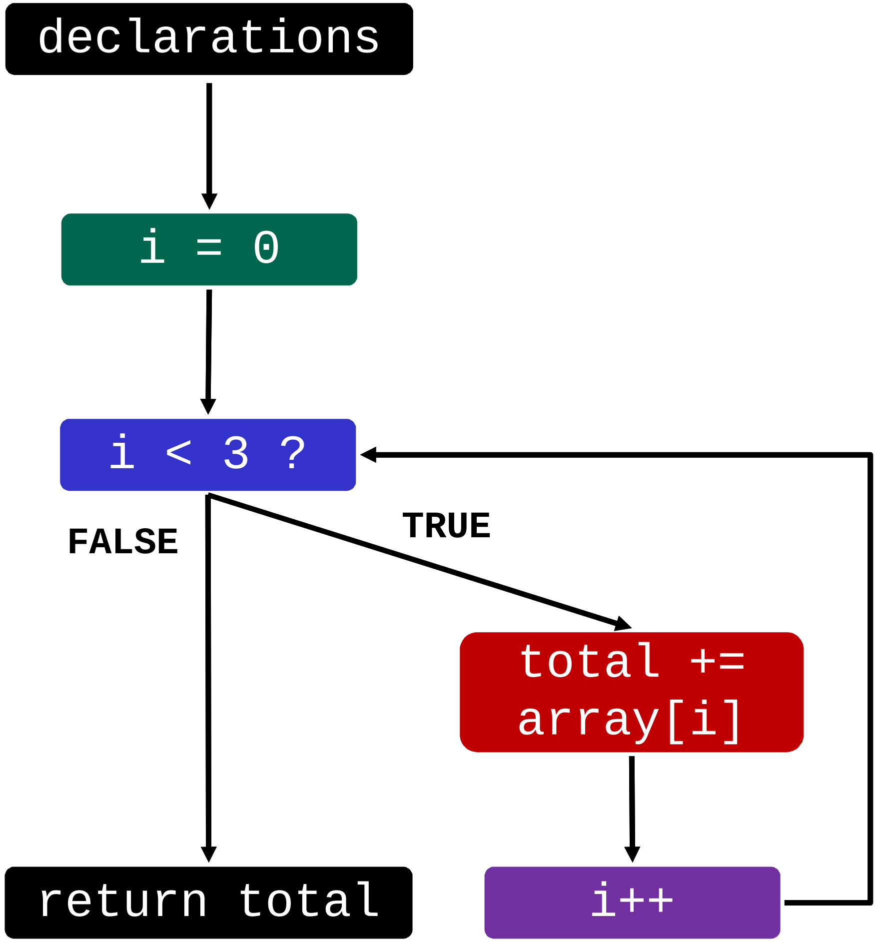
What is the value of total at the end of the program?

`total = 10`

Conditional loops

The loop `for` is useful when an iteration count (`i` in the example below) needs to be maintained, but the number of iterations must be known.

```
int total = 0;
int i = 0;
int array[3] = {12,3,-5};

for (i = 0; i < 3; i++) {
   total += array[i];
}

return total;
```



declarations

i = 0

i < 3 ?

FALSE

TRUE

total += array[i]

return total

i++

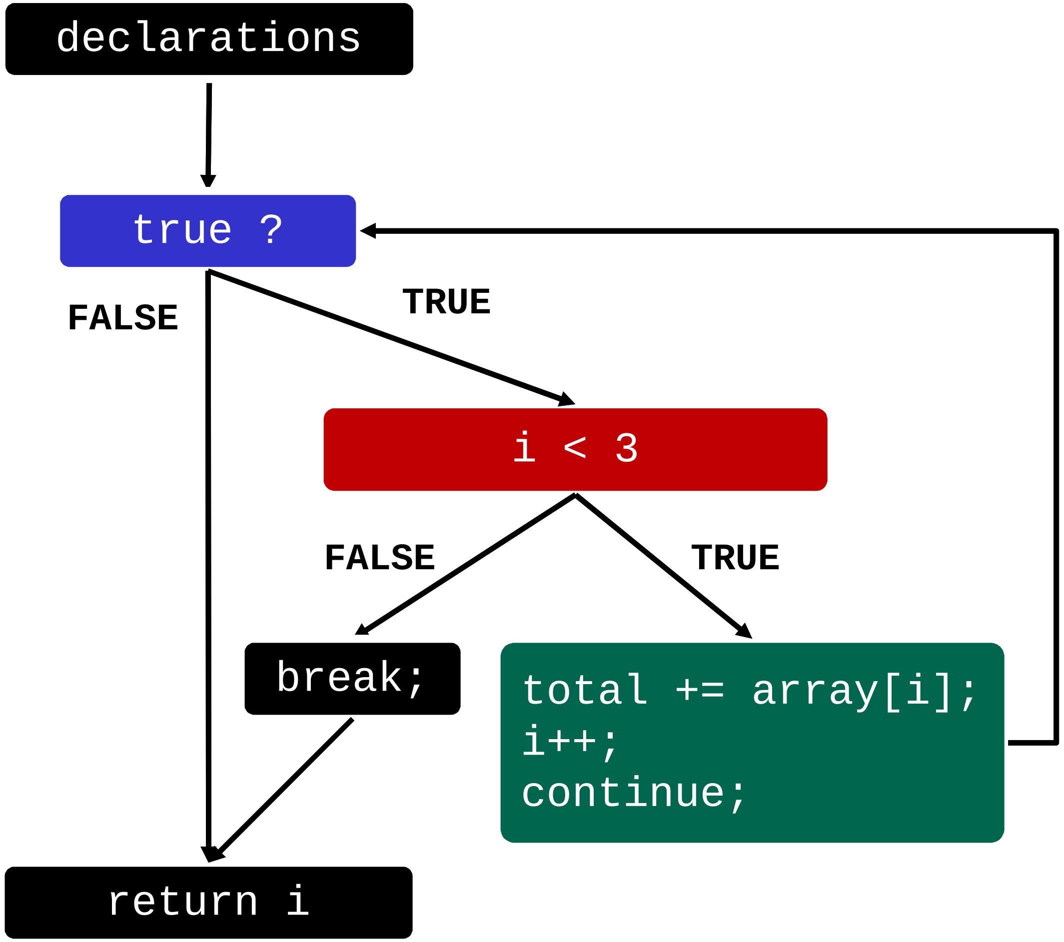
What is the value of total at the end of the program?

`total = 10`

`break` and `continue`

The statements `break` and `continue` cause the program to exit a loop or to jump directly to its next iteration, respectively.

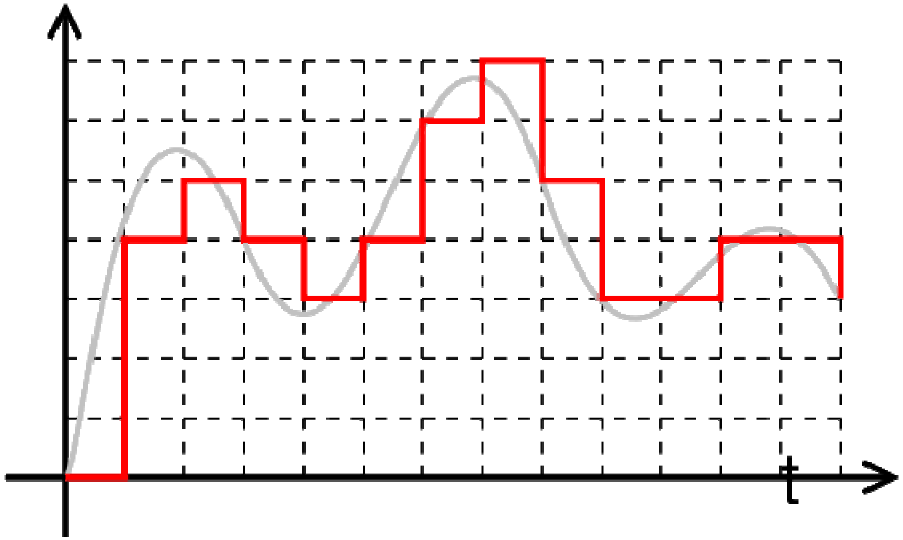
```
int total = 0;
int i = 0;
int array[3] = {12,3,-5};

while (1) {
   if (i < 3) {
      total += array[i];
      i++;
      continue;
   } else {
      break;
   }
   // unreachable code
}

return  i;
```



More about data structures

- Often, you need more complex data structures than simple variables. This is especially true in signal processing and related applications!
- For instance, how would you deal with a 1D signal in C? Using arrays!

```
int signal[50];
signal[0] = 0;
signal[1] = 4;
signal[2] = 5;
signal[3] = 4;
signal[4] = 3;
signal[5] = 4;
signal[6] = 6;
...
```



Quantization of a continuous signal (in grey) resulting in a digital signal (in red)

Arrays

- For an image, you can use a 2D array!

```
float epuck[640][480];
```

- And you can use nested loops to parse and process this image:

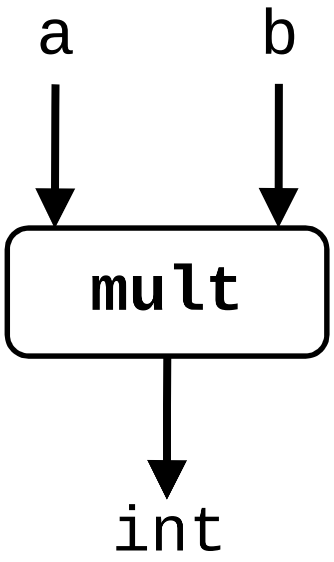
```
float epuck2[640][480];

for (i = 0; i < 640; i++) {
  for (j = 0; j < 480; j++) {
    epuck2[640-i-1][j] = epuck[i][j];
  }
}
```

What is the transformation performed by this program?

epuck

epuck2

Functions



- Functions must be declared using the following syntax:

```
type name(type1 arg1, type2 arg2, …);
```

- Here are some typical examples:

```
int mult(int a, int b);
double cos(double theta);
double norm(double v[]);
```

- Sometimes, you do not want your functions to return a value. You can use the keyword `void`!

```
void display_matrix(double m[][]);
void exchange(int a, int b);
```

Variable scope: local and global

- Any variable has a **scope**, i.e. a region where this variable can be used (read and/or write).
- In C, since variables must be declared at the beginning of the function, the scope of a variable is the function block:

```
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf(“Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

  printf(“Main: a = %d, b = %d\n", a, b);

  return 0;
}
```

scope of b

What about this b? It is a different variable, with a different scope!

- The scope of a variable does not extend beyond function calls!
- Use global variables if you want to use a **unique** variable in multiple functions.

Global variables

- A variable is **global** when it is declared outside of any block.
- Generally, try to avoid using them! If you want to use a constant value (known at compile time), rather use a **symbolic constant**.
- Using symbolic constants is way more efficient and allows the compiler to perform a better optimization of your code, but **you cannot change the value of this constant in the code!**

```
#include <stdio.h>

int unit_cost = 10; // global variable

int total_cost(int units) {
  return unit_cost * units;
}

int main() {
  int units = 12;
  int total = 0; unit_cost

  total = total_cost(units);

  printf(“%d units at %d CHF each cost %d 
CHF\n", units, unit_cost, total);

  return 0;
}
```

```
#include <stdio.h>

#define UNIT_COST 10 // symbolic constant

int total_cost(int units) {
  return UNIT_COST * units;
}

int main() {
  int units = 12;
  int total = 0; unit_cost

  total = total_cost(units);

  printf(“%d units at %d CHF each cost %d 
CHF\n", units, UNIT_COST, total);

  return 0;
}
```

Argument passing in C

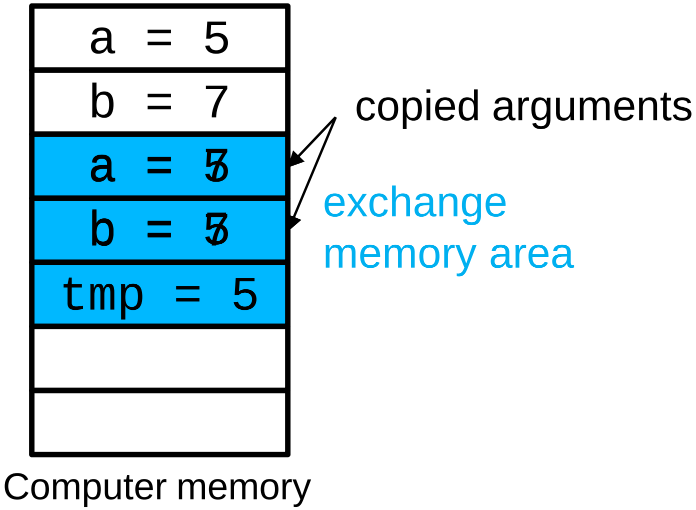
- Arguments are always passed *by value* in C function calls! This means that **local copies** of the values of the arguments are passed to the routines!

```
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf(“Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

  printf(“Main: a = %d, b = %d\n", a, b);

  return 0;
}
```

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 5, b = 7
```

What happens?

```
#include <stdio.h>

void exchange(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
  printf(“Exchange: a = %d, b = %d\n", a, b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(a,b);

  printf(“Main: a = %d, b = %d\n", a, b);

  return 0
}
```



Output:

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 5, b = 7
```

How to solve the problem?

- By using **pointers**, i.e. variables that contain the address of another variable!

```
#include <stdio.h>

void exchange(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
  printf(“Exchange: a = %d, b = %d\n", *a, *b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a,&b);

  printf(“Main: a = %d, b = %d\n", a, b);

  return 0;
}
```

Output:

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 7, b = 5
```

`int *a` and `int *b` are pointers!

Pointer?

A pointer is a **variable** that contains **the address of another variable**.

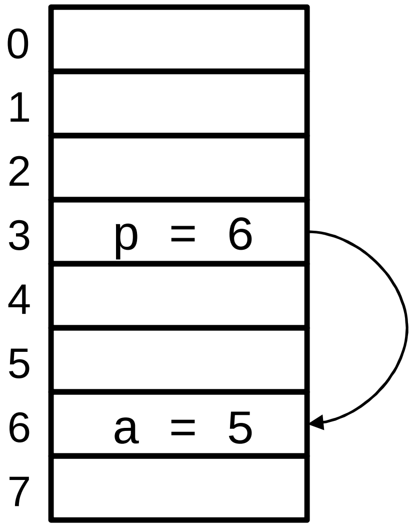
- A pointer can be declared as follows:

```
type* name
```

- To obtain the address of another variable, the operator & can be used:

```
int a = 5;
int* p = &a;
```

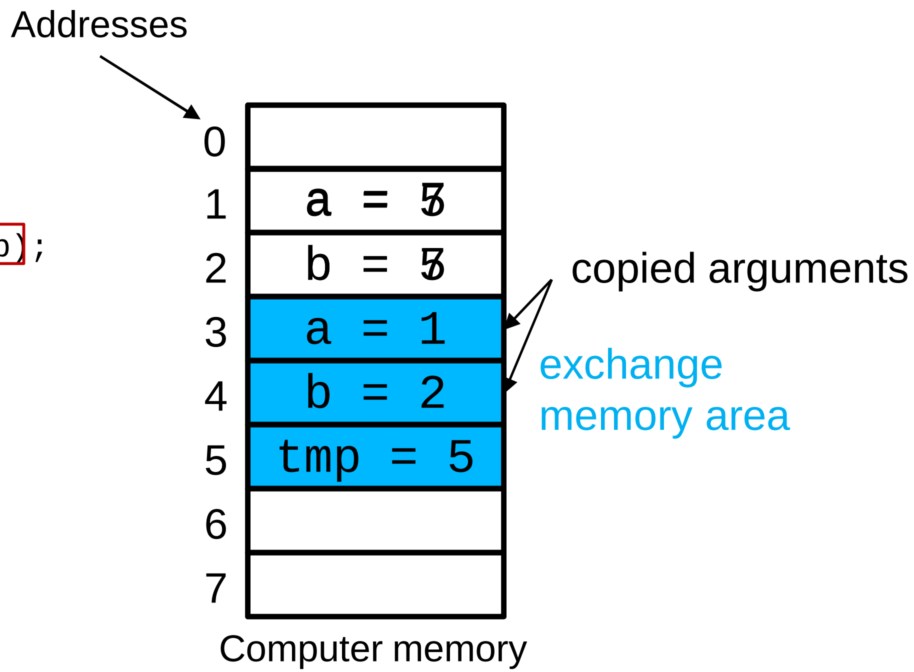
	Type	Address	Value
a	`int`	6	5
p	`int*`	3	6



Computer memory

What happens now?

```
#include <stdio.h>

void exchange(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
  printf(“Exchange: a = %d, b = %d\n", *a, *b);
}

int main() {
  int a = 5;
  int b = 7;

  exchange(&a,&b);

  printf(“Main: a = %d, b = %d\n", a, b);

  return 0;
}
```



Output:

```
computer:~> ./exchange
computer:~> Exchange: a = 7, b = 5
computer:~> Main: a = 7, b = 5
```

The operators * and &

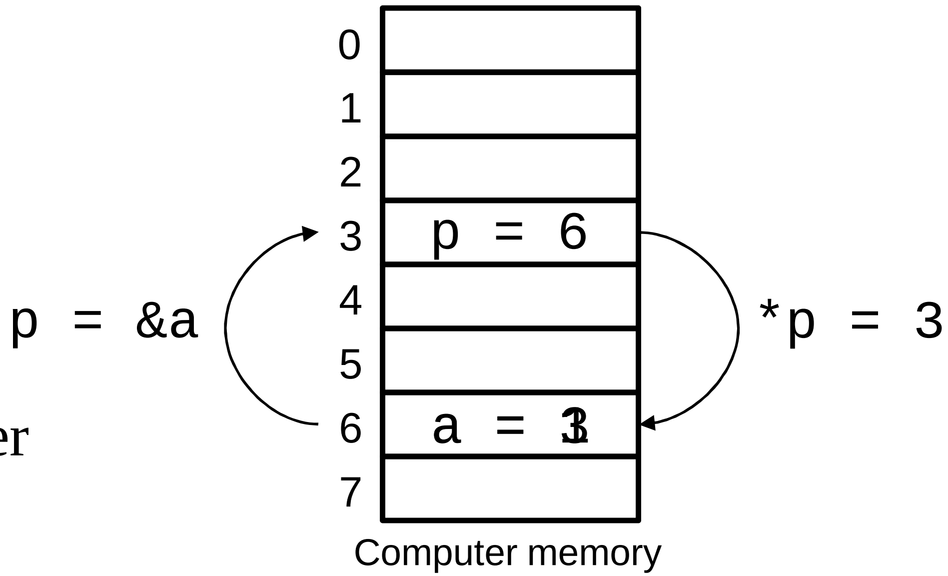
- The symbol * has two different meaning depending on the context.
- In a declaration, it indicates that we are declaring a pointer (i.e., a variable that contains the address of another variable):

```
int* p = &a;
```

- In other cases, it tells the compiler to **interpret the content of the variable as an address**, i.e. to read/write the data at the address in the variable:

```
*p = 3;
```

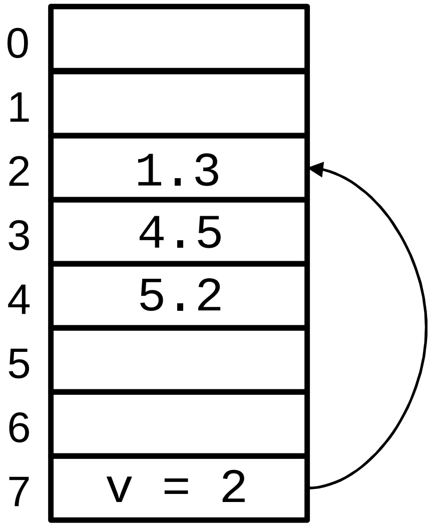
```
int a = 1;
```



Arrays

- Arrays and pointers are closely related. Actually, they are the exact same thing!

```
float v[3];
v[0] = 1.3;
v[1] = 4.5;
v[2] = 5.2;
```



Computer memory

	Type	Address	Value
v	`float*`	7	2
v[1]	`float`	v+1(3)	4.5

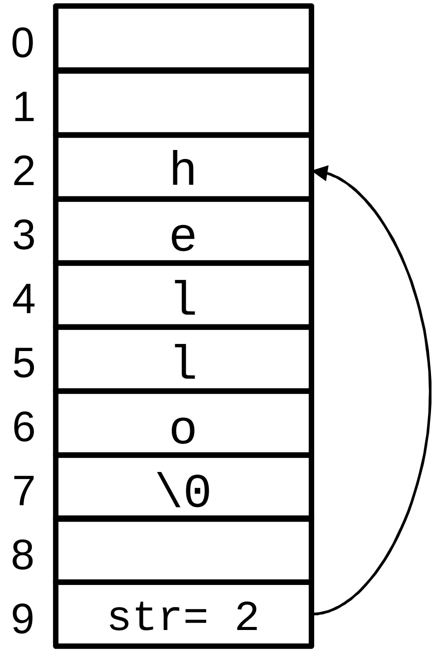
- The variable `v` is actually a pointer of type `float*`
- The expression `v[0]` is the same as *`v` or *(`v+0`)
- The expression `v[1]` is the same as *(`v+1`)

Strings

- There is no string type in C. Instead, we use arrays of `char`, i.e. the type `char*`.

```
char str[] = “hello”;
```

	Type	Address	Value
`str`	`char*`	`9`	`2`
`str[4]`	`char`	`word+4 (6)`	`‘o’`
`str[2]`	`char`	`word+2 (4)`	`‘l’`



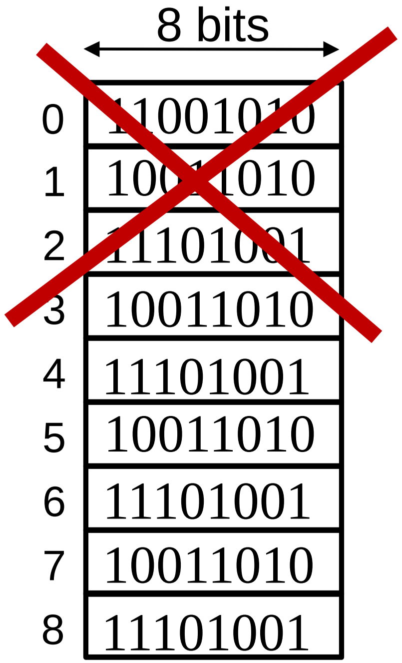
Computer memory

- You can use the `printf` to print out chains of character. It will read up to the character `‘\0’`.

```
printf(“%s”,str);     ──────→  computer:~> hello
printf(“%s”,str+3);   ──────→  computer:~> lo
```

Memory: a more realistic approach

- In a real computer, memory is organized into blocks of 8 bits, called **bytes**.
- On most modern computers , each byte has its own address.
- Memory is **limited,** not only in terms of the number of RAM modules that are installed, but also in terms of the number of addresses available.
- Furthermore, a program is not allowed to use (read and/or write) all bytes: some are reserved by the operating system. If you try to access them (using a pointer), your program will crash (segmentation fault or bus error).



~~0~~	~~11001010~~
~~1~~	~~10011010~~
~~2~~	~~11101001~~
~~3~~	~~10011010~~
4	11101001
5	10011010
6	11101001
7	10011010
8	11101001

8-bit computer memory

```
int *p = 1;
*p = 0;
```

→ segmentation fault (trying to write at address 1)

Binary addressing and hexadecimal notation

- Since everything is binary in a computer, addresses are also **binary**.
- For the sake of clarity, we generally write addresses in hexadecimal notation!

```
          0xB4CD
1011 0100 1100 1101
 B    4    C    D
```

- By convention, we add 0x in front of a hexadecimal expression.

8 bits

00000000	0x00	11001010
00000001	0x01	10011010
00000010	0x02	11101001
00000011	0x03	10011010
00000100	0x04	11101001
00000101	0x05	10011010
00000110	0x06	11101001
00000111	0x07	10011010
00001000	0x08	11101001

8-bit computer memory

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

The concept of “word”

- A **word** is a fixed-sized group of bits, which serves as a natural unit of data used by a particular computer architecture.
- Generally, **word sizes** are multiple of 8 bits, but this can vary as a function of the architecture.
- Note: generally, the memory is organized into multiple columns instead of a single one, so that the width of the memory is equal to a word size.

word size = 32 bits

0x00	10011010	11101001	10011010	11101001
0x04	10011010	11101001	10011010	11101001
0x08	10011010	11101001	10011010	11101001
0x0C	10011010	11101001	10011010	11101001

32-bit computer memory (e.g., Intel Pentium)

The size of the data types

- Each data type requires a certain number of bytes to be stored in memory, and this size can change as a function of the operating system (Windows, Linux, etc.) and the architecture of the system.
- The function `sizeof(`*`type`*`)` returns the size of the data type (in bytes).

```
printf("%d",sizeof(char));    /*  prints 1 */
printf("%d",sizeof(short));   /*  prints 2 */
printf("%d",sizeof(int));     /*  prints 4 */
printf("%d",sizeof(long));    /*  prints 4 */
printf("%d",sizeof(float));   /*  prints 4 */
printf("%d",sizeof(double));  /*  prints 8 */
```

The size of pointers

- **Reminder**: a pointer is a **variable** that contains **the address of another variable**.
- Therefore, the size of any pointer is **constant**, regardless of the data type that it points to (since it contains only the address of the variable, which does not depend on its type, obviously).

```
printf("%d",sizeof(char*));    /*  prints 4 */
printf("%d",sizeof(short*));   /*  prints 4 */
printf("%d",sizeof(int*));     /*  prints 4 */
printf("%d",sizeof(long*));    /*  prints 4 */
printf("%d",sizeof(float*));   /*  prints 4 */
printf("%d",sizeof(double*));  /*  prints 4 */
```

On a 32-bit computer

Dynamic allocation of memory (advanced topic)

- Java users often take for granted dynamical data structures such as `java.util.ArrayList`.
- These data structures are **dynamical** because they grow automatically in memory as you add data to them.
- In C, you **cannot** do that without managing memory yourself.
- In this code sample, for instance, the array `signal` can contain 50 integers and you cannot make it grow further.
- In many cases, you do not know at **compile time** the size of your data structure. In such cases, you need to **allocate memory dynamically!**

This value has to be a constant!

```
int signal[50];
signal[0] = 0;
signal[1] = 4;
signal[2] = 5;
signal[3] = 4;
signal[4] = 3;
...
```

Dynamic allocation of memory (advanced topic)

- To allocate a certain amount of memory, you can use the function `malloc(`*`size`*`)`, where *`size`* is the number of bytes of memory requested (which does not have to be constant).
- `malloc` returns a pointer to the first byte of memory which has been allocated.
- As a result, the static array declaration `int signal[50]` becomes, in its dynamic version:

```
int* signal = (int*) malloc(50 * sizeof(int));
signal[0] = 0;
signal[1] = 4;
signal[2] = 5;
signal[3] = 4;
signal[4] = 3;
...
```

This value does not have to be a constant!

Casting is required for compilation (without -> type error)

Freeing the memory

- If you allocated some memory dynamically, the compiler will **not** take care of freeing the allocated block of memory when you no longer need it.
- Use the function `free(void *ptr)` to make the block available to be allocated again.
- If you perform a `malloc` without its `free` counterpart, you will create a **memory leak**.
- Therefore, write a `free` for each `malloc` you write!
- After you freed memory, you can **no longer** access it!

```
#include <stdlib.h>

#define MAX_SIZE 1000000

int main() {
  int i;
  int *v; // a vector

  // create a vector of size i
  for (i = 1; i < MAX_SIZE; ++i) {
     v = (int*) malloc(i*sizeof(int));
     // do something with vector v
  }

  return 0;
}
```

2 TB (2000 GB) of RAM needed!

```
#include <stdlib.h>

#define MAX_SIZE 1000000

int main() {
  int i;
  int *v; // a vector

  // create a vector of size i
  for (i = 1; i < MAX_SIZE; ++i) {
     v = (int*) malloc(i*sizeof(int));
     // do something with vector v
     free((void*) v); // free memory
  }

  return 0;
}
```

4 MB of RAM needed!

Commented examples in C

Finding the maximum in an array

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

#define N_VALUES 5

int main() {
    int values[N_VALUES] = {1,5,2,7,3};
    int max = INT_MIN;
    int i = 0;

    for (i = 0; i < N_VALUES; ++i) {
        if (values[i] > max) {
           max = values[i];
        }
    }

    printf("The maximum is %d\n",max);

    return 0;
}
```

Includes needed functionalities (limits.h provides the constant INT_MIN that is the smallest defined integer)

Symbolic constants for the number of values in the array

Array declaration and initalization

Iteration using a for loop on the entire array

If we find a larger value than `max`, we update `max`

Print out the result

The program returns 0 because everything went well

Standard deviation

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

Includes needed functionalities (math.h provides the function `sqrt()`)

```
#define N_VALUES 10

int main() {
    float sample[N_VALUES] = {4.8, 4.6, 5.1, 5.9, 4.3, 5.0, 6.3, 5.4, 3.5, 5.0};
    float mean = 0.0;
    float std = 0.0;
    int i;

    for (i = 0; i < N_VALUES; ++i) {
        mean += sample[i];
    }
```

Compute the sum of all elements in the sample...

```
    mean /= (float) N_VALUES;
```

...and divide it by the number of elements to obtain the mean!

```
    for (i = 0; i < N_VALUES; ++i) {
        std += (sample[i]-mean)*(sample[i]-mean);
    }
    std /= (float) N_VALUES;
    std = sqrt(std);
```

Directly apply the formula of the standard deviation...

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2},$$

```
    printf("The standard deviation of the sample is %f\n",std);

    return 0;
}
```

Passing an array to a function

```
#include <stdio.h>

#define SIZE 3

void g(int array[], int const size) {
     int i;

     for (i = 0; i < size; ++i) {
          array[i] = 2 * (i+1);
     }
}

int main(void) {
     int i;
     int array[SIZE] = {0, 0, 0} ;

     g(array, SIZE);

     for (i = 0; i < SIZE; ++i) {
          printf(“%d:%d ", i, array[i]);
     }

     return 0;
}
```

- The two variables **`array`** and **`array`** are not the same (**`array`** is a copy of **`array`**)!
- However, since they are pointers (arrays = pointers) which point to the same memory portion, our function `g()` is still able to modify the content of the array.
- The function g() can also be declared like this:

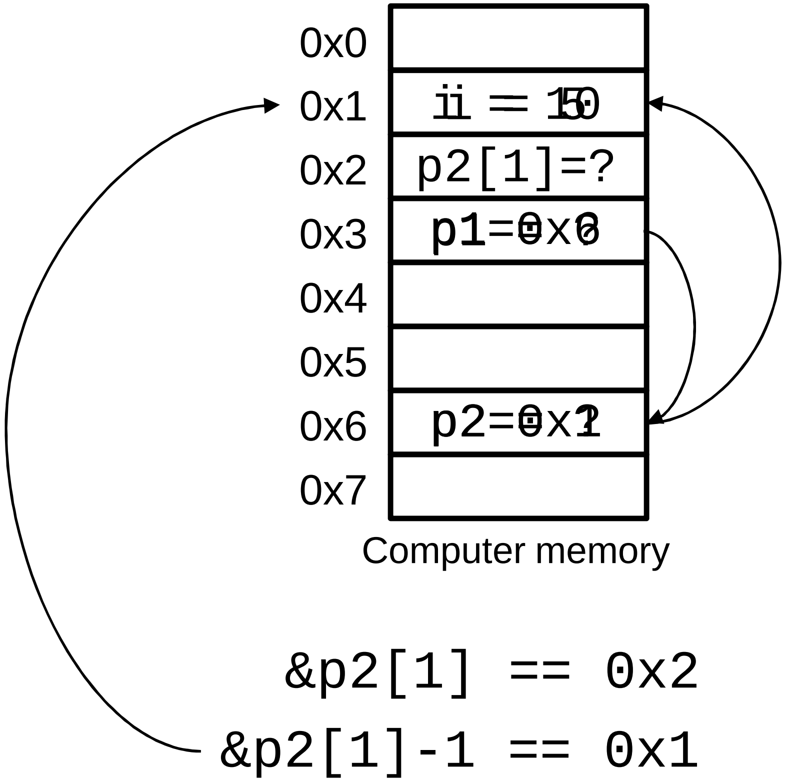
```
void g(int* array, int const size);
```

- Here is the output of the program:

```
computer:~> gcc –o array2fun array2fun.c
computer:~> ./array2fun
computer:~> 0:2 1:4 2:6
```

A (tortuous) pointer example

```
#include <stdio.h>

int main() {
   int i = 10;
   int** p1;
   int* p2;

   p1 = &p2;
   *p1 = &i;
   *(&p2[1]-1) /= 2;

   printf("i = %d", i);

   return 0;
}
```



```
&p2[1] == 0x2
&p2[1]-1 == 0x1
```

Output:

```
computer:~> ./pointers
computer:~> i = 5
```

Reading and references

- I strongly recommend that you read the tutorial "Pointers in the pocket", by Vlad Trifa (in French!).
- There are a lot of excellent C tutorials on the web:
  - http://www2.its.strath.ac.uk/courses/c/ by Steve Holmes
  - http://www.cs.cf.ac.uk/Dave/C/CE.html by A.D. Marshall
- And you can also find reference manuals:
  - The **C** Library Reference Guide http://www.acm.uiuc.edu/webmonkeys/book/c_guide/
  - **C** Language Syntax Reference http://www.cprogramming.com/reference/
- Whenever you do not remember how to use a function or a data/control structure, just do a **man** or **google** it!

Final note

- Thanks to Jean-Cédric Chappelier for making his course material available to me for this course!
- Thanks to Vlad Trifa for his hilarious (but very didactic) tutorial! This course has been largely inspired by this tutorial...
- Most of the code samples presented in these slides are available on Moodle. Compile, run and modify them in order to get a better understanding of the course material!
- To do that, you can use the following command:

```
computer:~> gcc –o myprog myprog.c
computer:~> ./myprog
```