Computer Architecture

Paul-André Melliès

Lecture 4 : Programming in C

A. basic types and control structures

```
// my very first program

#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

Preliminary schedule of the lectures

The basis of the language

- types, variables, expressions
- control structures (loops, conditionals)
- arrays and chains of characters
- structures

Intermediate

- functions
- pointers
- memory allocation

Advanced

- linked lists, binary trees
- recursive functions
- the UNIX system interface

Two great computer scientists && inventors

Dennis M. Ritchie (1941 - 2011)

Ken Thompson (1943)

Inventors of UNIX
and of the C programming language

First part

Basic types and syntactic elements

The basic components of the language

- variables — access to the memory
- operators — `+ - * < > ...`
- conditionals — `if (...) {...} else {...}`
- loops —
```
for (...) {...}
while (...) {...}
do {...} while (...)
```
- input / output —
```
printf (...)
scanf (...)
```

My very first program

Printing a few words on the screen

```
// my very first program

#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

./hello.out  **Hello World!**

My very first program

```
// my very first program

#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

The original program written in C

`// my very first program`	commentary (double slash + text)
`int main {...}`	defines of a function (the main function)
`printf (...)`	prints a chain of characters on the screen
`"Hello World!\n"`	the chain of character itself (with \n for line break)

Elucidating the purpose of each line

My very first program

The meaning of the two commands will be elucidated later in the course :

`#include <...>`	indication to the compiler (more precisely to the preprocessor)
`return 0`	the integer 0 is returned to the shell (0 = no error)

The program is written in a text editor and then compiled
(= transformed into an executable file) from a terminal :

```
$ emacs hello.c
$ gcc -Wall hello.c -o hello.out
```

The compiler gcc was developped by the Free Software Foundation initiated by Richard Stallmann in the 1980s. The three letters gcc stand for « The GNU Compiler ». The letters GNU are a recursive acronym for « GNU is not UNIX ».

General structure of a C program

A program is constructed as a sequence of functions

```
name of function (...) {
        ⋮
   sequence of instructions
        ⋮
}
```

body of the function == beginning and end of the block `{...}`

Each instruction ends with a semicolon `;`

The principal function main is launched at the start of the program.

Declaring variables, computing and printing

```
#include <stdio.h>
int main ()
{
   int x;      // new variable of integer type
   int y;      // new variable of integer type
   int z;      // new variable of integer type

   x = 10;     // store 10 in the variable x
   y = 20;     // store 20 in the variable y
   z = x+y;    // store in z the sum of x and y

// print the sum of x, y and z
printf ("the value computed is equal to %d", x+y+z);

return 0;
}
```

Declaring variables, computing and printing

☞ `int x;`

- reserves (or allocates) some piece of the memory sufficiently large to store an integer (**`int`**)
- calls this location **`x`**

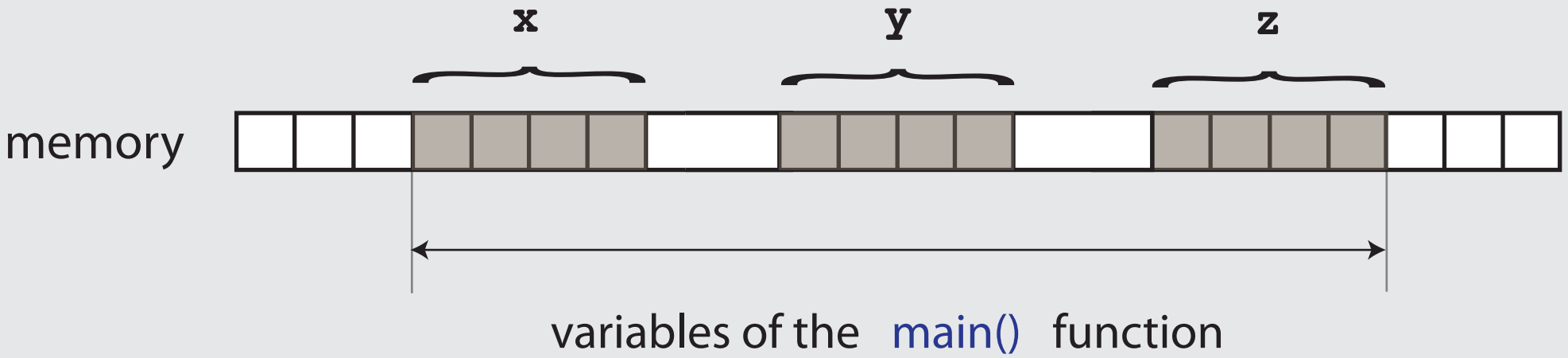
The variable **`x`** will be then used to refer to this specific memory location.

☞ The type **`int`** specifies the nature of the values stored at location **`x`** :

- characters
- floating point
- array of integers, etc...

```
type name_of_variable;
```
declaration of variable

Variables allocated in memory



Each square represents a byte = a sequence of eight bits
= a number between 0 and 255

p is large enough (8 bytes) to store the number [4 bytes]

int n;

long p;

p = 1000 000 000 000;

too large to be stored in n

4 bytes = 32 bits

To each data type corresponds a different amount of memory

n = p;

char

short

int

long

float

double

2^{32} values ?

$2^{32} = 4 \times 2^{30}$

$= 4 \times (2^{10})^3$

$\approx 4 \times (1000)^3$

$= 4\,000\,000\,000$

F F F F F F F F

Each block here represents a byte = a sequence of eight bits = a number between 0 and 255

Test yourself the size of your data types !!!

```
#include <stdio.h>

int main(void)
{
  printf("size of char = %zu\n",sizeof(char));
  printf("size of short = %zu\n",sizeof(short));
  printf("size of int = %zu\n",sizeof(int));
  printf("size of unsigned int = %zu\n",sizeof(unsigned int));
  printf("size of long = %zu\n",sizeof(long));
  printf("size of unsigned long = %zu\n",sizeof(unsigned long));
  printf("size of float = %zu\n",sizeof(float));
  printf("size of double = %zu\n",sizeof(double));
}
```

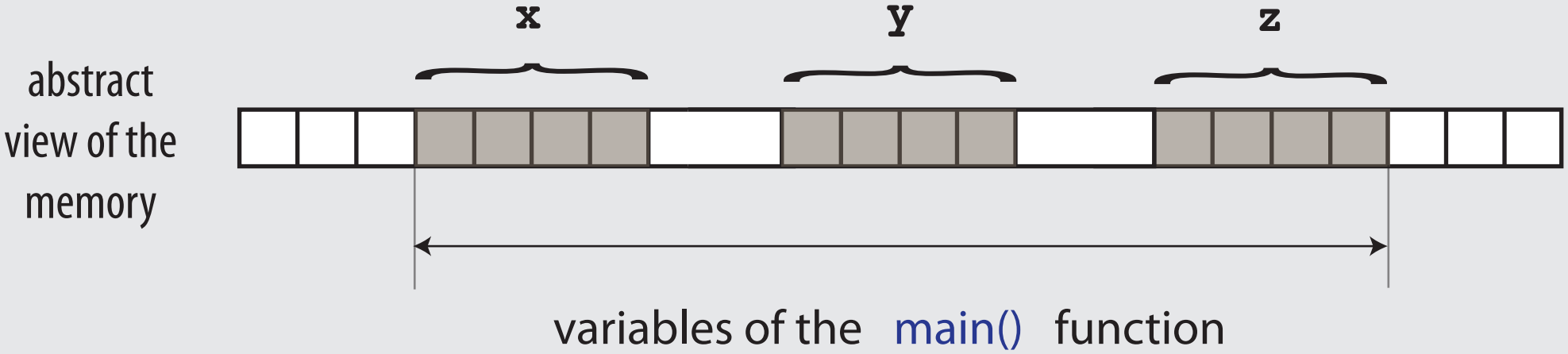
Test yourself the size of your data types !!!

```
size of char = 1
size of short = 2
size of int = 4
size of unsigned int = 4
size of long = 8
size of unsigned long = 8
size of float = 4
size of double = 8
```

Printf format identifiers

%d %i	Decimal signed integer
%o	Octal integer
%x %X	Hex integer
%u	Unsigned integer
%ld	Long decimal signed integer
%lu	Long unsigned integer
%c	Character
%s	String
%f	Double
%p	Pointer
%zu	Size of a type

Variables allocated in memory



Each square represents a byte = a sequence of eight bits
= a number between 0 and 255

Declaring variables, computing and printing

☞ `x = 10;` stores the value 10 in the memory location `x`

A point to remember :

- a variable which has not yet been assigned is called **not initialized**.
- the variables which have not been initialized may contain any value...

☞ `z = x + y;  // after x = 10; y = 20;`

- reads the current value of the variable `x` (that is 10)
- reads the current value of the variable `y` (that is 20)
- computes the sum (10 + 20) of these two values
- stores the result (30) in the variable `z` (and doing so, initializes `z`)

Declaring variables, computing and printing

☞ `printf("the value computed is equal to %d", x+y+z );`

- computes the sum of the current values of **x** and **y** and **z**
- prints the chain in quotation marks where **%d** is replaced by the result.

➡ `the value computed is equal to 60`

It is possible to print several integers :

```
printf("the sum of %d and %d is equal to %d", x , y , x+y );
```

➡ `the sum of 10 and 20 is equal to 30`

Reading on the keyboard

```
#include <stdio.h>
int main()
{
     int n;
     int m;

     printf("enter a first number :\n");
     scanf("%d",&n);    /* reading n on the keyboard */

     printf("enter a second number :\n");
     scanf("%d",&m);    /* reading m on the keyboard */

     printf("their sum is equal to %d", n+m );
     return 0;
}
```

Basic syntactic elements

Variable declaration

☞ It is not possible in the programming language C :

- to use an undeclared variable
- to declare a variable without indicating its type

The choice of names for the variables is essentially free :

```
x   result   weird_thing
```

In practice, the variable declarations of a function are done at the very beginning, followed by the instructions.

It is possible to declare several variables of the same type :

```
int x, y, z;
```

Basic syntactic elements

Variable declaration and immediate initialization

☞ `int x = 10;`

- declares the variable **`x`** of type **`int`**
- initializes the variable **`x`** with the value 10.

☞ It is also possible to mix declarations and initializations:

```
int x = 10, y = 20, z = x + y;
```

the declarations and initializations are performed sequentially.

Basic syntactic elements

The assignment operator =

```
int x;     // x undefined
int y;     // x and y undefined

x = 10;    // x is equal to 10, y undefined
y = x;     // value of x read and stored in y
           // x and y have same value 10

x = 20;    // x has value 20
           // y has still value 10
```

In that sense, the assignment operator enables one to copy, but is not retroactive

Basic syntactic elements

The assignment operator =

```
int x;     // x undefined
x = 1;     // x is equal to 1
x = x+1;   // x is incremented to 2
```

Alternative notation for incrementation

```
int x;     // x undefined
x = 1;     // x is equal to 1
x++;       // x is incremented to 2
```

Basic syntactic elements

The expressions

The allowed expressions are freely constructed as follows:

- the names of variables **`x y my_counter`**
- the constants of the appropriate type : **`10`** or **`-42`** for **`int`**
- the arithmetic operations (**`+`** , **`-`** , **`*`** , **`/`** , ...)
- parenthesis whenever it is necessary.

Some parenthesis may be omitted (following the priority rules)

Basic syntactic elements

The most usual types of the language

The type **`double`** for floating point numbers:

```
double x = 3.14;
```

The computations on these numbers are necessarily imprecise.

In order to print and to read these numbers, one uses **`%lf`**

```
printf("enter a number: ");
scanf("%lf", &x);  // -> 3.14 + Return
printf("The input value is %lf",x);
```

```
the input value is 3.14
```

Basic syntactic elements

The most usual types of the language : `double`

It is possible to store an int in a variable of type double but not the inverse without rounding up, or even an undefined result :

```
int n=1;
double pi = 1;          // pi is equal to 1.0
pi = pi + 2.14;         // pi becomes 3.14
n = pi;                 // compiles but n becomes 3
```

The value of **`n`** would be undefined if the value of pi were larger.

Basic syntactic elements

The most usual types of the language : **`char`**

The type **`char`** is the type used to represent characters.

The characters are represented in memory by numbers but the values of type **`char`** can also be written as **`'a'`**, **`'b'`**, ...

```
char c = 'a';          // in memory: 97
int n = c;             // n becomes equal to 97
```

`'a'` is equal to **`97`** **`'b'`** is equal to **`98`** etc...

It is possible to do arithmetic calculations on these numbers:

```
char c = 'a';  // in memory: 97
c = c + 1;     // c becomes 98, that is 'b'
```

Basic syntactic elements

The most usual types of the language : `char`

For reading and printing values of **`char`** as characters, one uses **`%c`**

```
char c;
scanf(" %c", &c);  //   input
printf("the integer associated to %c is %d", c, c);
```

```
the integer associated to a is 96
```

In the command **`scanf(" %c", c)`** the space before **`%c`** is only necessary after another **`scanf`** but it makes sense to add it systematically.

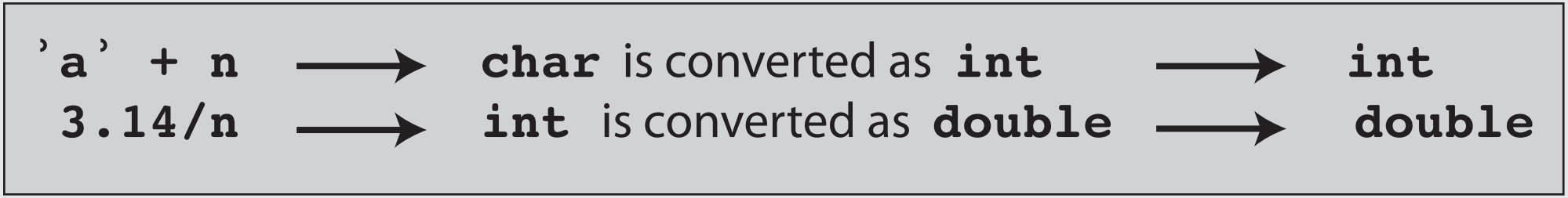
The values of type **`char`** lie between 0 and 255 (or sometimes 65525).
The number associated to a character is its ASCII code.

Basic syntactic elements

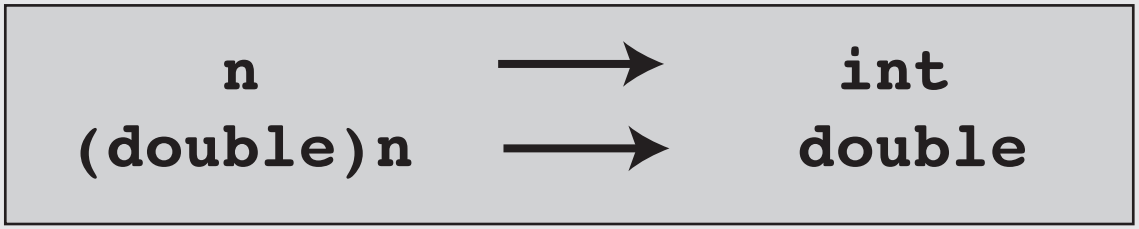
Typing of expressions and rules of conversion

The typing and conversion rules are precise but a bit complex.

For a start, one may remember that small types are coerced into larger types without loss of information whenever it is necessary :



In some cases, the cast may be also forced :



Second part

Control structures

Control structures

What we are going to see now :

- The structure **`if-else`**
- The conditional evaluation
- The loops **`for`** , **`while`** , **`do-while`**
- The structure **`switch`**
- The control instructions **`break`** and **`continue`**

Complement : some of the abridged notations of C

The if-else control structure

```
printf("enter two numbers :\n");
scanf("%d%d",&n,&m);

if (n > m) {
    max = n;
}
else {
    max = m;
}

printf("the largest one is %d\n", max);
```

If-else : general form

```
if (n > m){

     max = n;

}

else {

     max = m;

}
```

```
if ( boolean condition ){
          ⋮
     body of the if
          ⋮
}

else {
          ⋮
     body of the else
          ⋮
}
```

Note that a block reduced to a single instruction
may be written without any `{ ... }`

How multiple if-else are associated

The contents of the two bodies of the **`if-else`** are arbitrary and can be freely interchanged.

One can write an **`if`** without any **`else`** (but not the contrary !)

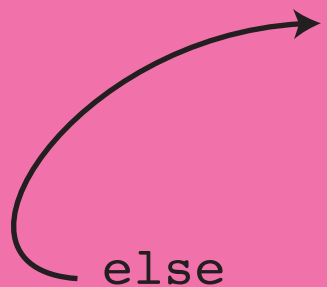
An **`else`** is always associated to the last **`if`** of same depth and not yet associated to any previous **`else`**.

Illustration :

```
if (expression)
    statement
else if (expression)
    statement
else if (expression)
    statement
else
    statement
```

The danger of nesting ifs

```
if (n > 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return i;
        }
else            /* WRONG */
    printf("error -- n is negative\n");
```



Example in Section 3.2 of Kernighan & Ritchie 1989

This kind of bugs is very difficult to detect...

It is thus a good practice to put braces **`{...}`**
when there are nested **`if`**s !!!

The syntax of conditions

Comparison operators

The usual form : comparison of two expressions

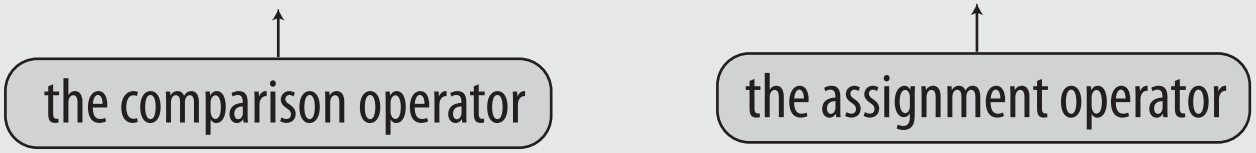
`expr op expr`

where **`op`** is one of the comparison operators below :

`<` **`<=`** **`>=`** **`>`**

`==` **`!=`**

☞ Beware : the equality is written **`==`** to distinguish it from **`=`**



The comparison operators are valid for all numeric types

The syntax of conditions

Comparison operators

It is also possible to compare expressions with different types :

```
char c; int n; double d;
// ... initialization of the variables ...

if (c <= n*2) {...}  // c promoted into int
if (n+1 <= d) {...}  // n+1 promoted into double
if (n != c+d) {...}  // n promoted into double
                     // c promoted into double
```

The syntax of conditions

Combining comparison combinators

Two conditions can be combined into a single one using :

`&&` the logical conjunction, true when its two components are true

`||` the logical disjunction, true when one of its components is true

```
if (n <= 1 || x > 3) {...}
```

The inverse of a condition can be constructed using :

`!` the logical negation, true when its unique component is false

`if (!(x==0)) {...}` is the same as `if (x!=0) {...}`

The evaluation of conditions

There is no boolean type in C

The type **`int`** is thus used to represent the truth values...

The conditions are values of type **`int`** like any other integer value.

A comparison evaluates as :

- **`1`** when it is satisfied
- **`0`** otherwise

Illustration :

`1 > 2` evaluates to **`0`**
`2 > 0` evaluates to **`1`**

One may thus write **`v = n > 2;`** to ask
that **`v`** receives the value **`0`** or **`1`** depending on the value of **`n`**

The evaluation of conditions

From the point of view of an **`if`** (of a **`for`** , of a **`while`** , etc...)

- all the expressions of non zero value are considered true
- all the expressions of zero value are considered false

Similarly for the boolean combinators && || !

```
 if (x + y){...}    is the same as  if (x + y != 0) {...}
if (!(x + y)){...}  is the same as  if (x + y == 0) {...}
```

Probably useless but does compile properly :

```
if (42) ...
```
the condition is always satisfied

```
if (0) ...
```
the condition is never satisfied

Beware : a common mistake !!!

A typical bug in C is the following one :

```
if (n = 0){ /* instead of n == 0 */
    ...
    ...
}
```

☞ Be warned of its dangers :

- this code will compile well and will produce an executable
- it will force the value of **`n`** to be zero
- it will never branch in the block **`if`** whatever the original value of **`n`** tested by the condition.

Assignment instructions have values !!!

☞ `n = 0`

- an instruction which gives the value `0` to `n`
- used here as an expression
- the value of this expression is the value received by `n`

☞ Every assignment instruction can be used in that way :

`x = y = 42` can be read as `x = ( y = 42 )`

is evaluated as follows :

- the variable `y` receives the value `42`
- this determines the value `42` of the expression `( y = 42 )`
- the variable `x` receives the value `42`

Assignment instructions have values !!!

```
if (n = 0) { ... }
```

- the variable `n` receives the value `0`
- the expression `(n = 0)` has thus value `0`
- as such, it is considered false
- the conditional thus branches on the `else` block.

```
if (n = 42) { ... }
```

- the variable `n` receives the value `42`
- the expression `(n = 42)` has thus value `42`
- as such it is considered true (because not zero)
- the conditional thus branches on the `if` block.

Assignment instructions have values !!!

Question :
But what is the point of treating assignments as expressions?

Answer :
This enables one to write more concise and clearer code in many situations of interest, like the following one :

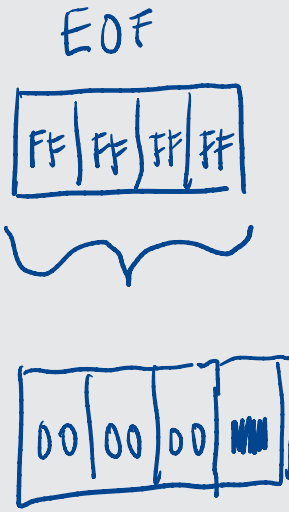
```
int c;

if ((c=getchar()) != EOF){
/* case when the character c has been read */
/* from the standard input */
}
else {
/* case when the program has reached the end */
/* of the standard input */
}
```

gets the first character in a file

the value of this assignment is the value produced by getchar()

End of File



Here, EOF is a signal End-Of-File emitted by the system when the end of the standard input has been reached

Conditional evaluation

```
int n, m;

printf("enter two numbers :\n");
scanf("%d%d",&n,&m);

printf("the largest one is %d", (n > m) ? n : m);
```

A variant of the construction **`if-else`** : select an expression among two possibilities depending on the boolean condition.

The `for` loop: illustration

```
// computing the first ten square numbers

int i,square;

for (i = 1; i <= 10 ; i=i+1){
    square = i * i;
    printf("The square of %d is equal to %d\n",i,square);
}
```

The `for` loop: general form

```
for (i = 1; i <= 10 ; i=i+1){ body of the loop };
```

where the variable `i` is called the counter of the loop

`i = 1` initialization instruction
the initial value of the counter is `1`

`i <= 10` boolean condition
the loop is performed as long as this condition holds

`i=i+1` incrementation instruction
the counter is incremented each time the body of the loop is executed

The `for` loop: general form

The method of using a counter in a loop is well-known, general and safe.

```
for (i = 1; i <= 10 ; i = i + 1){ body of the loop };
```

```
for (i = 1; i < 10 ; i = i + 1){ body of the loop };
```

```
for (i = 1; i < 12 ; i = i + 2){ body of the loop };
```

```
for (i = 10; i >= 0 ; i = i - 1){ body of the loop };
```

The `for` loop

The interval of the counter is not necessarily known before execution :

```
// computing the sum of the n first numbers

int i,n,sum;

printf("enter a positive number : ");
scanf("%d", &n);

sum = 0;
for (i = 1 ; i <= n ; i = i + 1){
    sum = sum + i;
}

printf("The sum of the %d first numbers is %d\n",n,sum);
```

Question : what happens when the integer **`n`** is negative ?

The `for` loop

Advice : avoid to alter the counter inside the body of the **`for`** loop

The following program is absurd... but it does compile !!!

```
for (i = 1 ; i < 2 ; i = i + 1){
     i = i - 1;
}
```

The only way to stop it is ctrl-C from the shell (= interruption).

Even more absurd :

```
for ( ; 1 ; );
```

- the initialization instruction is empty
- the conditional test is always successful
- the incrementation instruction is empty

Again, the only way to stop the loop is ctrl-C from the shell.

The `while` loop: illustration

Computation of the first power of 2 greater than 10000


```
int power, n;
n = 0;
power = 1;

while (power < 10000){
     power = power * 2;
     n = n + 1;
}

printf("Two to the power %d = %d is the first
power of two greater than 10000\n", n, power);
```

The `while` loop : general form

```
while (boolean condition){ body of the while };
```

- The condition is evaluated. If it is true (that is, non zero) then
  - the body of the while is executed
  - one goes back to ☞

  Otherwise, one carries on.

  It is possible that the body of the **`while`** is never executed when the condition is immediately false (equal to zero) .

  Nothing guarantees that the **`while`** loop will stop -- as such, it a bit more risky than a **`for`** loop.

Exercise

It is easy to encode a **`for`** loop using a **`while`** loop.
Explain how !

It is even easier to implement an infinite **`while`** loop.
Explain how !

The do-while loop

```
int n;

do {
    printf("Enter a positive number : ");
    scanf("%d",&n);
    if (n < 0) {printf("Sorry, I have said positive...\n")
} while (n < 0);
```

do { body of while } while (boolean condition);

- The body of the **while** is executed (it is thus executed at least once)

  The condition is then evaluated.
  - if it is true (that is, non zero) then one goes back to ☞
  - otherwise, one carries on and goes to the next instruction.

The switch structure

```
int n, m; char choice;

printf("Enter two numbers : ");
scanf("%d%d",&n,&m);

printf("What do you want to do with them ?\n");
printf("Add them (+) ?\n");
printf("Multiply them (*) ?\n");
scanf(" %c",&choice);

switch (choice) {
case '+': printf("Their sum is equal to : %d\n", n+m);
          break;
case '*': printf("Their product is equal to : %d\n", n*m);
          break;
default : printf("Unknown operation");
          break;
}
```

The `switch` structure

```
switch (expression) {
    case constant1 : sequence of instructions;
                     break;
    case constant2 : sequence of instructions;
                     break;
    ...
    default : sequence of instructions by default;
              break;
```

1. the expression is evaluated
2. the first sequence of instructions whose constant is equal to the value just computed is executed
3. the **`break`** instruction at the end of the sequence causes an immediate exit from the **`switch`**
4. the sequence of instructions at **`default`** is executed when the computed value is different from all constants

The `switch` structure

The **`break`** instruction is not necessary at the end of the instruction.
In that case, the next instruction of the **`switch`** is performed.
Similarly, it is not necessary to end the **`switch`** with a **`default`** case.

```
int n=2;

  switch (n) {
  case 0: printf("is switched ON at 0\n");
  case 1: printf("is switched ON at 1\n");
  case 2: printf("is switched ON at 2\n");
  case 3: printf("is switched ON at 3\n");
  case 4: printf("is switched ON at 4\n");
  }
printf("end of the switch");
```

```
is switched ON at 2
is switched ON at 3
is switched ON at 4
end of the switch
```

In that respect, the **`switch`** structure behaves just like a switch...

```c
#include <stdio.h>

int main(){  /* count digits, white spaces, others */

  int c, i, nwhite, nother, ndigit[10];

  nwhite = nother = 0;
  for (i=0; i < 10; i=i+1)
    {ndigit[i] = 0;}  /* initialization of the array of numbers */
  while ((c=getchar()) != EOF)
    {switch (c) {
      case '0': case '1': case '2': case '3': case '4':
      case '5': case '6': case '7': case '8': case '9':
        ndigit[c-'0'] = ndigit[c-'0']+1;
        break;
      case ' ': case '\n': case '\t':
        nwhite=nwhite+1;
        break;
      default:
        nother=nother+1;
        break;
      }
    }
  printf("digits =");
  for (i = 0; i<10; i=i+1)
    {printf(" %d", ndigit[i]);}
  printf(", white space = %d, other = %d\n", nwhite, nother);
  return 0;
}
```

declare the variables

while loop

c - '0'

Ascii — ASCII code of zero

digit in the file

ndigit[0] = 0

ndigit[9] = 0

The `break` instruction

More generally, the **`break`** instruction causes immediate exit from :

- the body of a **`switch`**
- the body of a **`for`** loop
- the body of a **`while`** loop
- the body of a **`do-while`** loop

```
for ( ... ){
      ...
      break;  // exit from the body
      ...
}
//  the execution carries on here
```

It is not possible to exit with one single **`break`** instruction from several nested bodies : one needs a **`break`** instruction for each level of nesting.

The `continue` instruction

The **`continue`** instruction enables one

- in the body of a **`for`** loop :
  to jump directly to the increment instruction

- in the body of a **`while`** loop or of a **`do-while`** loop :
  to jump directly to the evaluation of the condition

In other words, the **`continue`** instruction jumps immediately
to the end of the body of the loop.

Very often, the **`continue`** instruction can be simulated by an **`if`**.

Acknowledgements and references

The slides of this lecture are largely based on a very nice and cleverly crafted introductory course on the programming language C given by Vincent Padovani in my University Paris Diderot.

Padovani is a pedagogical master and I am greatly endebted to his art here !

I have also inserted in the lecture a series of slightly more demanding examples extracted from the marvelous and so instructive book by Kernighan and Ritchie.

Please read these programs carefully and try to understand what they do...
You will discover their beauty, and learn a lot from them !