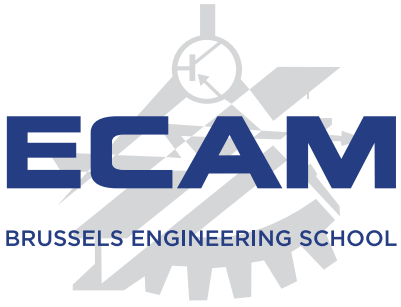


E301B C Programming

Session 1

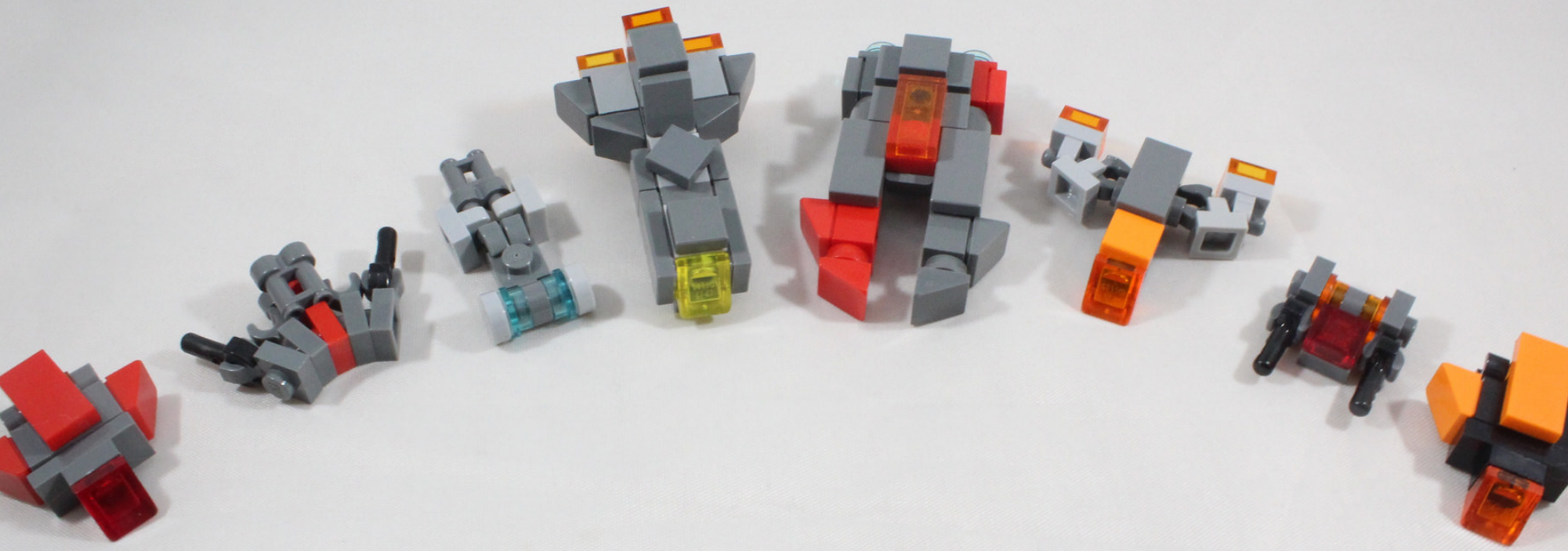
Introduction to C Programming

Sébastien Combéfis

Fall 2019

Objectives

- Compilation and execution of a program
  - Source code, machine code and intermediate files
  - Compilation chain, GCC and XC8 compilers and preprocessor
- Introduction to the basics of C programming
  - Variable, data type, literal forms and operators
  - Conditional and iterative control structures
  - Defining and calling procedures and functions



Compilation and Execution

Program

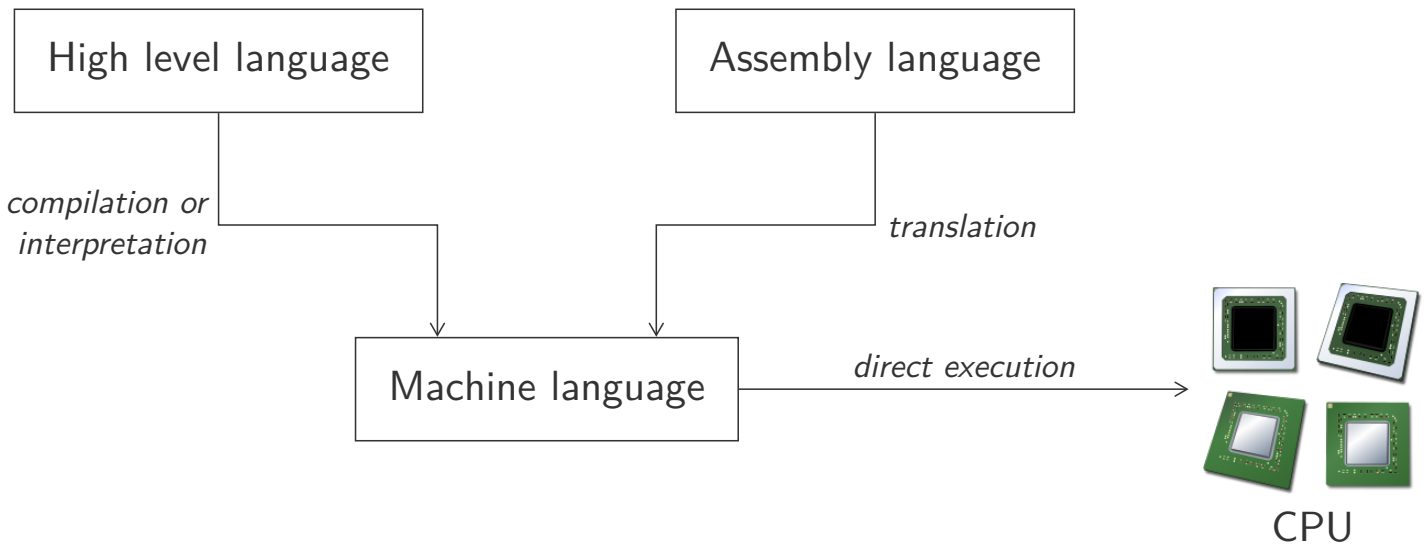
- A program is a sequence of statements
  - It receives data as input
  - It performs some computations
  - It produces a result as output
- Several ways to provide inputs and to produce outputs

  Command line parameters, request to the user, file, etc.

Executable Program

- Transformation from programming language to a machine one

  From a language with a more or less high abstraction level



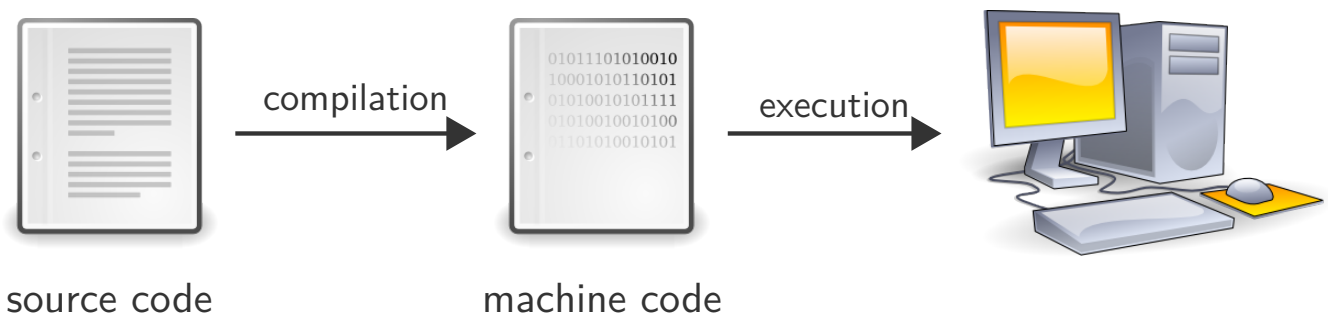
Compilation Chain

- Source code is written in a programming language

  Text file, readable and understandable by the human being

- A compiler transforms source code in machine code

  Binary file, readable and executed by the computer



Hello World

- Hello World program written with C programming language
  - Including a library with `#include`
  - Entry point of the program is the `main` function
  - Displaying text on standard output with the `printf` function

```c
#include <stdio.h>

int main()
{
    printf("Hello World\n");    // Display "Hello World"

    return 0;
}
```

GCC Compiler

- GCC, the GNU Compiler Collection, is a compiler system

  Initial release on May 23, 1987, stable (9.2) on August 12, 2019

- Using `gcc` command to compile source code to machine code

  From a `.c` text file to a `.exe/.out` binary file

```
$ ls
helloworld.c
$ gcc helloworld.c
$ ls
a.out  helloworld.c
$ ./a.out
Hello World
```

Executable File

- Executable file contains configuration and machine code

  Precise format depend on the operating system

- Linux systems mainly uses the ELF file format

  Executable and Linkable Format

```
$ file a.out
a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSv),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 2.6.32, BuildID[sha1]=
cfcfe0a9c7e41bc708b7dc3292e697acfa92f5ff, not stripped
```

Hello Led

- Hello Led program written with C programming language
  - `PORTD` bit 7 to output (0) and bits 6:0 are inputs (1)
  - Set `LAT` register bit 7 to turn on LED
  - Wait indefinitely to keep the program running

```
1  #include <xc.h>
2
3  void main(void)
4  {
5      TRISD = 0b01111111;
6      LATDbits.LATD7 = 1;
7      while (1)
8      ;
9  }
```

MPLAB® XC8 Compiler

- Specific MPLAB® compiler for PIC microcontrollers

  Last version 2.10 built on July 30, 2019

- Using `xc8` command to compile source code to machine code

  From a `.c` text file to a `.hex` memory image

```
$ xc8-cc -mcpu=18F46K20 helloled.c

Memory Summary
    Program space        used 18h (   24) of 10000h bytes (   0.0%)
    Data space           used  0h (    0) of   F60h bytes (   0.0%)
    Configuration bits   used  0h (    0) of     7h words (   0.0%)
    EEPROM space         used  0h (    0) of   400h bytes (   0.0%)
    ID Location space    used  0h (    0) of     8h bytes (   0.0%)
    Data stack space     used  0h (    0) of   F00h bytes (   0.0%)

$ file helloled.hex
helloled.hex: ASCII text
```

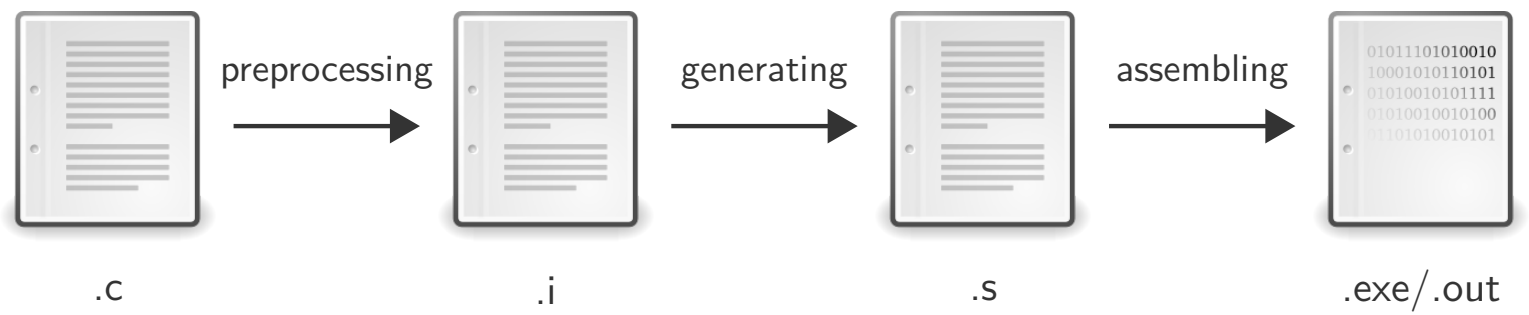
Intermediate File (1)

- A compiler creates intermediate files during compilation

  Different representations of the same program

- Several tools are used during the compile process

  Preprocessor, parser, code generator, assembler, linker, etc.



Preprocessor

- The preprocessor makes a first treatment on source code

  Transformation from one source code to another source code

- Only executes some specific preprocessor directives
  - `#include <path>`: includes the content of a file
  - `#define TOKEN value`: replace a text by another one
  - `#pragma config`: provides information to the compiler
  - and many others: `#ifdef`, `#ifndef`, etc.

Defining Constant (1)

- A constant is a value that do not change during execution

  Should be given a name to improve program readability

- Declaring a constant with the `#define` directive

  `#define CONSTANT_NAME value`

```c
#define TAX_RATE 0.21

int main()
{
    float price = 12.5; // Price of the item
    float total_price = price * (1 + TAX_RATE);

    return 0;
}
```

Defining Constant (2)

- The preprocessor transforms the source code file
  - It removes all the comments from the file
  - It replaces `TAX_RATE` by `0.21` everywhere in the file
- Transformed source code file can be asked to GCC compiler

  Use the `-E` option to get the intermediate `.i` source code file

```
1  #define TAX_RATE 0.21
2
3  int main()
4  {
5     float price = 12.5;
6     float total_price = price * (1 + 0.21);
7
8     return 0;
9  }
```

Intermediate File (2)

- Possible to save all the generated intermediate files
  - `.i` source code file obtained after preprocessing
  - `.s` assembly language source file after code generation
  - `.o` binary object file obtained after assembling
- Using the `-save-temps` option of GCC compiler

```
$ gcc -save-temps preprocessor.c
$ ls
a.out            preprocessor.i  preprocessor.s
preprocessor.c   preprocessor.o
```

Data and Variable

Variable

- A variable is used to store and manipulate data
  - Created and then initialised with its initial value
  - Value of a variable can be modified at any time
- A variable must be declared before being used

  Defining its name, its type and eventually an initial value

```
1  int a = 1;       // Declaration and initialisation
2
3  int b;           // Declaration
4  b = 3;           // Initialisation
5
6  a = 5;           // Modification
```

Data Type

- Integer number: `short int`, `int`, `long int`, `long long`

  The number of inhabitants in a country, students in a class, etc.

- Floating number: `float`, `double`, `long double`

  A price, the consumption of a car in liter, etc.

- Character: `char`

  A letter, the gender of a person (M/F/I), etc.

`int` Type

- An `int` represents an integer number

  Can hold positive and negative values by default

- Possibility to choose to have a signed integer or not
  - `signed int` to accept positive and negative values (default)
  - `unsigned int` to restrict to positive values

```
1  signed int a;              // a can hold any integer
2  a = -10;
3
4  unsigned int b = 20;       // b can only hold positive integers
5  a = a + b;
6
7  int c = -5;                // Same as signed int c
```

Literal Form of `int`

- Writing a constant integer number with its literal form

  Just write its digits one after the other, with a possible sign

- Using a suffix to explicitly determine the type
  - By default, integer literal is a `signed int`
  - Adding `u` or `U` for `unsigned` and `l` or `L` for `long int`

```
1  int i = 42;
2  unsigned int ui = 42U;
3
4  long l = 42L;
5  unsigned long ul = 42UL;
```

`float` Type

- A `float` represents a floating point number

  Number that contains a floating decimal point

- Not possible to represent all the real numbers

  Computations with float are sometimes approximate

```
1  float d = 0.123;          // d can hold any floating point number
2  d = d + 2.001;
3
4  float e;                  // float variable can also hold integers
5  e = 5;
6  e = 4.0;
```

Literal Form of `float`

- Writing a constant floating point number with its literal form

  Using the decimal point explicitly or using the scientific notation

- Using a suffix to explicitly determine the type
  - By default, floating point literal is a `double`
  - Adding `f` or `F` for `float` and `l` or `L` for `long double`

```
1  float f = 12.5F;
2
3  double d = 125e-1;
4  long double ld = 125e-1L;
```

char Type

- A `char` represents a single character

  Smallest storage unit occupying a single byte

- A character is nothing else than an integer number

  Number $\leftrightarrow$ character mapping with a table

```
1  char f = 'A';    // f holds the single character A (uppercase)
2
3  char g;          // g holds the character corresponding to number 65
4  g = 65;
```

Literal Form of `char`

- Writing a constant character with its literal form

  Using single quotes around the character

- Using the ASCII code to identify the character
  - Octal representation (base 8) with three digits
  - Hexadecimal representation (base 16) with two digits

```
1  char c = 'M';
2  char o = '\115';
3  char h = '\x4D';
```

printf Function (1)

- Print text to the standard output with `printf` function

  Often on the console from where the program was launched

- Can also be used for a formatted output

  Replacing markers in a template string by values

```c
printf("Hello\n");                                  // "Hello"

int age = 26;
printf("I'm %d y.o.\n", age);                       // "I'm 26 y.o."

int year = 1993;
int month = 9;
int day = 6;
printf("Born on %d-%d-%d", year, month, day); // "Born on 6-9-1993"
```

printf Function (2)

- Possible to limit the number of decimal places for floats

  The specifier `%.5f` shows five places

- Possible to prefix integer numbers with 0 or spaces

  The specifier `%05d` shows at least five digits, padding with 0

```
1  float f = 1.2345678;
2  printf("%.3f", f);          // Prints "1.234"
3
4  int a = 4;
5  printf("%3d", a);           // Prints "  4"
6  printf("%03d", a);          // Prints "004"
```

Occupied Memory Space (1)

- The `sizeof` function gives the memory space for a type

  Total space allocated for one variable of a given type

- Result gives the allocated memory space in bytes

  Result is a `size_t` value, a specific unsigned integer

```
1  size_t char_space = sizeof(char);      // Should be 1
2
3  size_t int_space = sizeof(int);        // Is typically 4 or 8
```

Occupied Memory Space (2)

- Integer number
  - `short int`: 2 bytes
  - `int`: 4 bytes
  - `long int`: 8 bytes
  - `long long`: 8 bytes
- Floating point number
  - `float`: 4 bytes
  - `double`: 8 bytes
  - `long double`: 16 bytes
- Character
  - `char`: 1 byte

Limits (1)

- Memory space for data types is highly machine dependent

  Depends on the physical architecture of the machine

- The `limits.h` file contains information about data types

  Number of occupied bits, minimal and maximal values, etc.

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

int main()
{
    printf("%d\n%d\n", INT_MIN, INT_MAX);

    return 0;
}
```

```
-2147483648
2147483647
```

Limits (2)

- Integer number

Type	Unsigned	Signed
`short int`	0 to 65 535	−32 768 to 32 767
`int`	0 to 4 294 967 295	−2 147 483 648 to 2 147 483 647
`long int`	0 to 18 446 744 073 709 551 616	−9 223 372 036 854 775 808 to 9 223 372 036 854 775 807

- Floating point number

Type	Possible values
`float`	$3{,}4 \cdot 10^{-38}$ to $3{,}4 \cdot 10^{38}$
`double`	$1{,}7 \cdot 10^{-308}$ to $1{,}7 \cdot 10^{308}$

Operation

Arithmetic Operator

- Four basic arithmetic operators between numbers

  Addition (+), subtraction (-), multiplication (∗) and division (/)

- Two specific operators for integer numbers
  - Integer division (/) computes the quotient
  - Modulo (%) computes the remainder of integer division

```
1  int a;
2  a = 2 + 3;                // a is 5
3
4  float b = 8.0 / a;        // b is 1.6
5
6  int c = a / 2;            // c is 2, quotient of 5 divided by 2
7  int d = a % b;            // d is 1, the remainder of 5 divided by 2
```

Modulo Operator

- The result of `a % N` is an integer in the interval [0; N[

  Smallest positive integer r so that $a = qN + r$

- Several possible applications to the modulo operator
  - Testing the parity of a number with `a % 2`
  - Testing the divisibility by N of a number with `a % N`

```
1  int a = 4 % 2;     // a is 0 between 4 is even
2  int b = 3 % 2;     // b is 1 between 3 is odd
3
4  int c = 20 % 4;    // c is 0 because 20 is divisible by 4
5  int d = 21 % 4;    // c is not 0 because 20 is not divisible by 4
```

Pseudo-Random Number (1)

- Initialising PRN generator with a seed with `srand` function

  The seed must be as random as possible

- Getting a pseudo-random number with the `rand` function

  Can only be used after the initialisation of the PRNG

```c
#include <stdlib.h>

int main()
{
    srand(3);           // initialising generator with seed 3

    int a = rand();     // a is a pseudo-random number
    int b = rand();     // b is a pseudo-random number

    return 0;
}
```

Pseudo-Random Number (2)

- Bring back integer number X in an interval [min; max]

  Using the transformation `(X % (max - min + 1)) + min`

- Width of the interval is given by $max - min + 1$

  Number of integer numbers in the interval

```
1  srand(10);
2
3  int a = (rand() % 4) + 2;     // Intervalle [2; 5]
4
5  int b = (rand() % 7) + 3;     // Intervalle [3; 9]
6
7  int c = (rand() % 11) - 5;     // Intervalle [-5; 5]
```

Shortened Operator

- Compound operators to shorten variable update

  `a = a ⋆ b` *is equivalent to* `a ⋆= b`, *with* ⋆ *any arithmetic operator*

- Incrementation/decrementation operator to add/subtract 1
  - Prefix notation `++i` first update variable before using it
  - Suffix notation `i++` produces value before variable update

```
1  int i = 2;
2  int j = 2;
3
4  printf("%d\n", ++i);          // Prints 3
5  printf("%d\n", j++);          // Prints 2
6  printf("%d%d\n", i, j);       // Prints 33
```

Comparison Operator

- Comparison operators result in an integer value (0 or 1)

  Depending on the success or failure of the comparison

- Six different comparison operators to compare values
  - `a > b`, `a < b`: strictly smaller or larger than
  - `a >= b`, `a <= b`: smaller or larger than
  - `a == b`, `a != b`: equal or different

```
1  int a = (5 >= 2);          // a is 1
2
3  int b = (5 < 1);           // b is 0
4
5  int c = (a == b);          // c is 0
```

Logical Operator

- Three logical operators `&&` (AND), `||` (OR) and `!` (NOT)

  Result of the operators can be described with a truth table

a	b	!a	a && b	a || b
0	0	1	0	0
0	1		0	1
1	0	0	0	1
1	1		1	1

```
1  int a = 2 == 2 && 3 == 3;          // a is 1
2
3  int b = 2 == 3 || 3 == 3;          // b is 1
4
5  int c = !(2 == 3) && 3 == 3;       // c is 1
```

Simplification of Condition

- Symmetry of "opposing" operators to eliminate NOT
  - `!(x != b)` is equivalent to `x == b`
  - `!(x > a)` is equivalent to `x <= a`
  - `!(x >= a)` is equivalent to `x < a`
- Morgan's rule to switch from AND to OR and inversely
  - `!(a && b)` is equivalent to `!a || !b`
  - `!(a || b)` is equivalent to `!a && !b`

Control Structure

`if` Statement

- Executing code depending on a condition
  - If the value is 0, the code is not executed
  - For all the other values, the code is executed
- Simple condition built with comparison operators

  Possible to build complex conditions with logical operators

```
1  int distance = 10;
2
3  if (distance < 5)
4     printf("The distance is strictly lower than 5");
5
6  printf("...continuation of the program...");
```

Multiple Alternative

- Code to execution if condition not satisfied with `else`

  Used with `if` statement to define alternative code

- Possible to define several execution alternative

  Adding as many `else if` clauses as possible

```
1  if (distance < 5)
2     printf("The distance is strictly lower than 5");
3  else if (distance <= 10)
4     printf("The distance is comprised between 5 and 10");
5  else
6     printf("The distance is strictly greater than 10");
7
8  printf("...continuation of the program...");
```

Code Block

- Put multiple statements as one code block

  Statements delimited by braces

- Possible to execute multiple statements in a `if`, `else`, etc.

  Required if more than one statement to execute

```
1  int distance = 10;
2
3  if (distance < 5) {
4      printf("The distance is strictly lower than 5");
5
6      // ...other statements...
7  }
8
9  printf("...continuation of the program...");
```

Ternary Operator

- Choosing between two expressions depending on a condition

  Can be done with an `if-else` instruction

- Ternary operator is a shorter notation to choose

  `condition ? if_true : if_false`

```
1  int age = 21;
2  float price = age < 18 ? 7.5 : 15;
```

`switch` Statement

- Choice between several constant values for a variable

  Can replace several nested `if` statements

- Work following cascading principle between several `cases`

  Need to exit the `switch` statement with `break` statement

```
1  switch (i) {
2     case 1:
3     case 3:
4     case 5:
5        printf("Odd");
6        break;
7
8     case 2:
9        printf("Even");
10
11    default:
12       printf("I do not know")
13 }
```

while Loop

- Repeat statements as long as a condition is satisfied

  Condition is a boolean expression, as with `if` statement

- Beware of the possibility to have infinite loops

  Happen when condition does not change in value, remains false

```
1  int i = 3;
2
3  while (i > 0) {
4      printf("%d\n", i);
5      i--;
6  }
7
8  printf("BOOM!");
```

do-while Loop

- The body of a `while` loop may not be executed once

  If the condition of the loop is initially false

- The `do-while` loop is executed at least once

  The condition is checked after the execution of the body

```
1  int i = 3;
2
3  do {
4      printf("%d\n", i);
5      i--;
6  } while (i > 0);
7
8  printf("BOOM!");
```

for Loop

- Repeat statements a certain number of times
  - Is often more readable than `while` loop for this situation
  - Dedicated variable used to count the number of iterations
- The `for` statement is composed of three optional elements

  The initialisation, the condition and the update statement

```
1  int i;
2
3  for (i = 3; i > 0; i--)
4     printf ("%d", i);
5
6  printf("BOOM!");
```

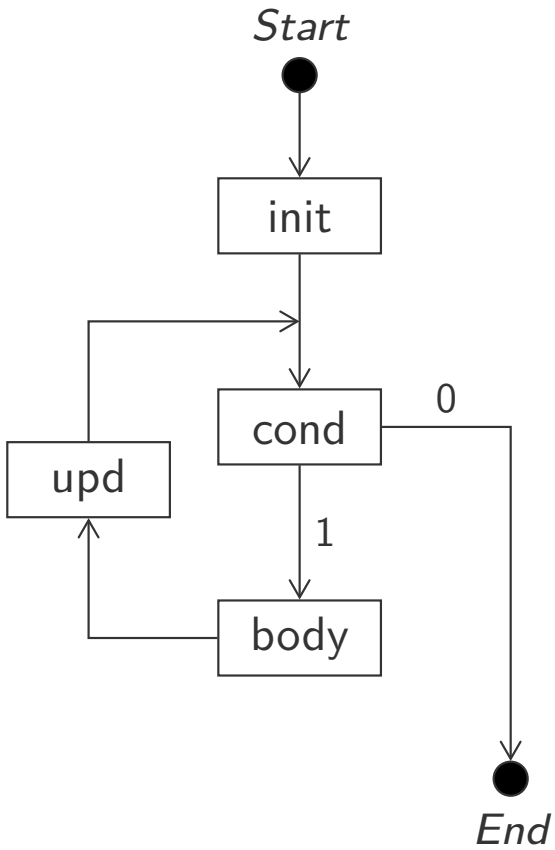
Loop Equivalence

- Similarity between `while`, `do-while` and `for` loops

  Can always rewrite a loop with any of the three loop statements

```
for (init; cond; upd) {
    body;
}
```

```
init;
while (cond) {
    body;
    upd;
}
```



Procedure and Function

Procedure and Function

- Block of code that can be easily called multiple times
  - Uniquely identified by a name
  - May received one or several parameters as input
  - May produce a return value as output
- Distinction to be made between procedure and function

  Depending on whether there is a return value or not

Procedure without Parameter

```
void procedure_name()
```

- Two elements in the signature of the procedure
  - `void` indicates that it is a procedure
  - Uniquely identified with its name, `procedure_name`
- A procedure defines an action to execute

  The body of a procedure is composed of statements to execute

say_hello Procedure

- A procedure is simply called with its name

  Followed by two parenthesis, without anything between

```c
#include <stdio.h>

// Procedure that prints Hello!
void say_hello()
{
    printf("Hello!\n");
}

// Main function
int main()
{
    say_hello();

    return 0;
}
```

Procedure with Parameter

```
void procedure_name(parameters_list)
```

- Parameters used to communicate information to the procedure

  Transmitted in local variables in the called procedure

- A parameter is described with two elements

  A type (`int`, `float`, `char`, etc.) and a name for the variable

- Limited scope for procedure parameters

  They are local to the body of the procedure

count_to Procedure

- Must specify effective parameters when calling procedure

  These values are copied into formal parameters of the procedure

```c
#include <stdio.h>

// Procedure which counts from 1 to max
void countTo(int max)
{
    int i;
    for (i = 1; i <= max; i++)
        printf("%d\n", i);
}

int main()
{
    countTo(2);

    return 0;
}
```

Prototype

- Procedure must be defined before being called

  Because compiler reads the source code file incrementally

- Possibility to declare the prototype of the procedure

  Makes it possible to call it before its definition

```
1  #include <stdio.h>
2
3  void countTo(int);
4
5  int main()
6  {
7      countTo(2);
8
9      return 0;
10 }
11
12 void countTo(int max) { /* ... */ }
```

Function without Parameter

```
return_type function_name()
```

- Two elements in the signature of the function
  - Returns a value with the defined `return_type`
  - Uniquely identified with its name, `function_name`
- A function defines a computation to execute

  The body of a function computes then return a value

random_die Function

- Function must return a value with the declared return type

The returned value can be retrieved at the function call

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

// Function which throws a die with six faces
int random_die()
{
    int die_value = (rand() % 6) + 1;
    return die_value;
}

int main()
{
    srand(3);
    int value = random_dice();
    printf("The value of the die is: %d\n", value);

    return 0;
}
```

`return` Statement

- The `return` statement defines the result of the function
  - Returns a value from the called function to the callee
  - This statement also directly quit the called function
- The type of the returned value must be the same as declared

  Otherwise the compiler will issue a compile error
- A function call is therefore just an expression

  Can be assigned to a variable, used to form a complex expression

Function with Parameter

```
return_type function_name(parameters_list)
```

- Parameters used to communicate information to the function

  Transmitted in local variables in the called function

- A parameter is described with two elements

  A type (`int`, `float`, `char`, etc.) and a name for the variable

- Limited scope for function parameters

  They are local to the body of the function

get_sum Function

- The `return` statement can directly take an expression

  The value of the expression is returned to the callee

```c
#include <stdio.h>

// Function which computes the sum between a and b
int get_sum(int a, int b)
{
    return a + b;
}

int main()
{
    int sum = get_sum(2, 3);
    printf("The sum of 2 and 3 is: %d.\n", sum);

    return 0;
}
```

Macro (1)

- Possible to define macros handled by preprocessor

  Parametrised text replacement similar to functions

- Very strict syntax to define a macro
  - No space between name and list of parameters parenthesis
  - More careful to parenthesise the parameters

```
1  #include <stdio.h>
2
3  #define MAX(A,B) ((A) > (B) ? (A) : (B))
4
5  int main()
6  {
7      int a = MAX(2,7);
8      printf("max(2,7,-3) = %d\n", MAX(a,-3));
9
10     return 0;
11 }
```

Macro (2)

- The `#define` directive is just a text replacement tool

  It works the same way for constants and macros

- Used to simplify the code and make it more readable

  Possible to define an exception mechanism with macros

```
1  // ... content of stdio.h ...
2
3  int main()
4  {
5      int a = ((2) > (7) ? (2) : (7));
6      printf("max(2,7,-3) = %d\n", ((a) > (-3) ? (a) : (-3)));
7
8      return 0;
9  }
```

References

- Richard M. Stallman and the GCC Developer Community, *Using the GNU Compiler Collection For GCC version 9.2.0*, 2019.
- Michael Boelen, *The 101 of ELF files on Linux: Understanding and Analysis*, May 15, 2019. https://www.linux-audit.com/elf-binaries-on-linux-understanding-and-analysis
- Microchip, *MPLAB® XC8 C Compiler User's Guide for PIC® MCU*, 2018. (ISBN: 978-1-5224-2815-2)

Credits

- https://www.flickr.com/photos/tuckermarley/36928554005
- https://openclipart.org/detail/100267/cpu-central-processing-unit
- https://www.flickr.com/photos/xiziluo/14107640658
- https://www.flickr.com/photos/soldiersmediacenter/12821095033
- https://www.flickr.com/photos/elpadawan/11140574194
- https://www.flickr.com/photos/mwichary/2220430874