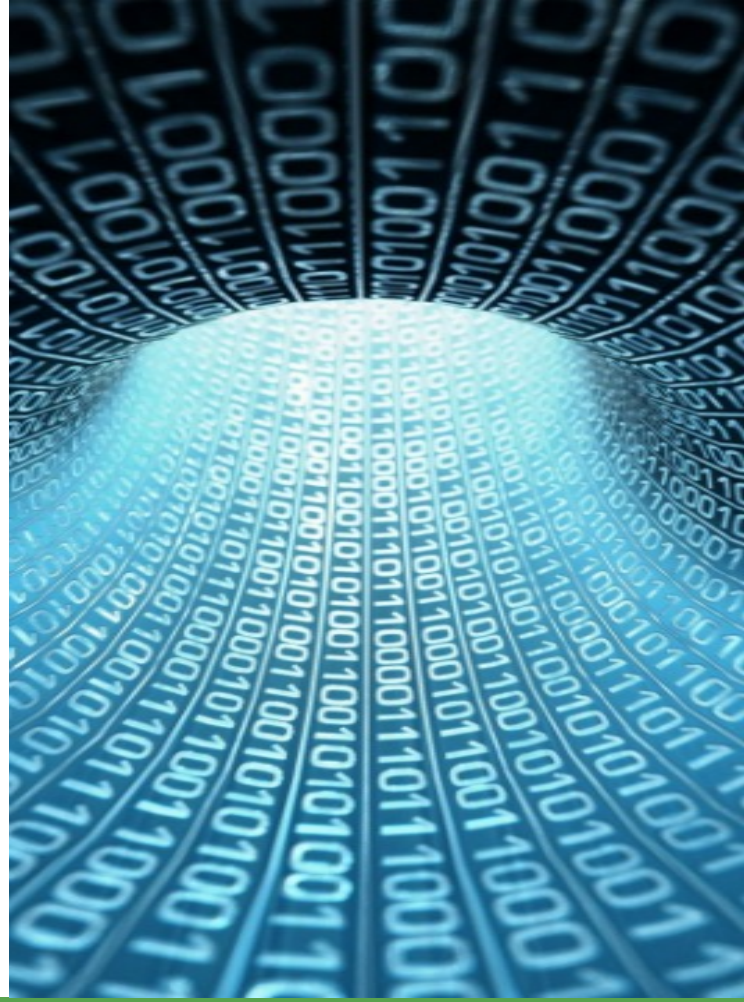
POLITECNICO
DI TORINO
Dipartimento
di Automatica e Informatica

C language in brief

SYNTACTIC ASPECTS, BASIC TYPES AND CONSTRUCTS



Contents

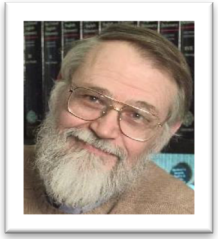
- A little history
- Syntactic aspects
  - **The structure of a C program**
- Basic data types and I/O
  - **Primitive data types (scalar),** boliche
  - **I/O operations** (on stdin/stdout and text files)
- Control flow constructs
  - **Conditional and iterative constructs**
  - **functions** and parameters passing
- Aggregated data types
  - **arrays** (vectors and matrices of integers, floats and characters)
  - **strings** and arrays of strings
  - **structs** (composite data types)

Birth of C language

- Developed between 1969 and 1973 at AT&T Bell Laboratories (Ken Thompson, B. Kernighan, Dennis Ritchie)
  - For internal use
  - Linked to the development of Unix OS
- In 1978 “The C Programming Language” is published, firt official documentation of C language
  - For friends: the “K&R”

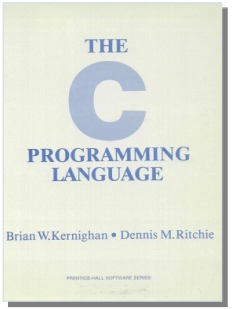
AT&T Bell Laboratories

Ken Thompson



Brian Kernighan

Dennis Ritchie



History

- C versions and standards
  - K&R (1978)
  - C89 (ANSI X3.159:1989)
  - C90 (ISO/IEC 9899:1990)
  - **C99 (ANSI/ISO/IEC 9899:1999, INCITS/ISO/IEC 9899:1999)**
  - C11
  - C17m C20
- Not all the compilers are standard!
  - GCC: *almost* C99, with some missing parts, extensions
  - Borland & Microsoft: *more or less* C89/C90
  - CLion (the IDE we will use in labs): C99

Current diffusion

- C language was traditionally the most popular programming language, now no more…
- It is still one of the most diffused embedded programming language
- Syntax of all the main programming languages (including Pytjon and Java) is derived from C
- **There is no Computer Engineering without C!!!**

  (sorry for you…)

General characteristics of C language

- C language is:
  - High level, imperative
    - ... But also not too much high level (very close to Hardware, **much more** than Python)
  - Structured
    - ... With few exceptions
  - Typed
    - Each object has a type
  - Elementary
    - Few *keyword*
  - *Case sensitive*
    - In identifiers, upper-case is different from lower-case!
  - Portable (with few exceptions)
  - Standard ANSI

Python vs C

PYTHON

1. High level language, very flexible, prefers code readability to efficiency
2. Interpreted
3. Blocks identified by indentation
4. Types of objects (variables, values returned by functions) are implicit
5. Having a main function is a “good practice”, not mandatory

C

1. Much lesser versatile, lower level (closer to HW), prefers efficiency to code readability (it is the language of OS and embedded systems)
2. Compiled
3. Blocks identified by { } and keywords
4. Explicit declaration of type “always” needed (variables, formal parameters, returned value…)
5. main function is “mandatory”

First example (python vs C)

```
##
#  Demonstrate the print function

#  Print 7
print(3 + 4)

#  Print Hello World! on two lines
print("Hello")
print("World!")

#  Print multiple values
#  with a single print function call
print("My numbers are", 3 + 4, "and", 3 + 10)

#  Print messages with empty line in between
print("Goodbye")
print()
print("Hope to see you again")
```

```
//
//  Demonstrate the print function
#include <stdio.h>

int main (void) {

  printf("%d\n", 3 + 4); // Print 7

  //  Print Hello World! on two lines
  printf("Hello\n")
  printf("World!\n")

  //  Print multiple values
  //  with a single print function call
  printf("My numbers are %d and %d\n",
          3+4, 3+10); // can be on multiple lines

  //  Print messages with empty line in between
  printf("Goodbye\n\nHope to see you again\n");

  return 0;
}
```

Second example (python vs C)

```
##
# This program simulates an elevator panel
# that skips the 13th floor.
#

# Obtain floor number from the user as an integer
floor = int(input("Floor: "))

# Adjust floor if necessary.
if floor > 13 :
   actualFloor = floor - 1
else :
   actualFloor = floor

# Print the result.
print("The elevator will travel to the ”
      "actual floor", actualFloor)
```

```
/*
   This program simulates an elevator panel
   that skips the 13th floor. */

#include <stdio.h>

int main (void) {
  // declare/define variables
  int floor, actualfloor;

  // Obtain floor number from the user as an integer
  printf ("Floor: ”);
  scanf (“%d”, &floor);

  /* Adjust floor if necessary. */
  if (floor > 13)
    actualFloor = floor – 1;
  else
    actualFloor = floor;

  // Print the result.
  printf("The elevator will travel to the "
         "actual floor %d\n", actualFloor);
  return 0;
}
```

Second example (C: lines do not count!)

```
/*
   This program simulates an elevator panel
   that skips the 13th floor. */
#include <stdio.h>

int main (void) {
  // declare/define variables
  int floor, actualfloor;

  // Obtain floor number from the user as an integer
  printf ("Floor: ”);
  scanf (“%d”, &floor);

  /* Adjust floor if necessary. */
  if (floor > 13)
    actualFloor = floor – 1;
  else
    actualFloor = floor;

  // Print the result.
  printf("The elevator will travel to the "
         "actual floor %d\n", actualFloor);
}
```

```
/*
   This program simulates an elevator panel
   that skips the 13th floor. */
#include <stdio.h>

int main (void) {
  // declare/define variables
  int floor,
      actualfloor;

  // Obtain floor number from the user as an integer
  printf ("Floor: ”); scanf (“%d”, &floor);

  /* Adjust floor if necessary. */
  if (floor > 13) actualFloor = floor – 1;
  else actualFloor = floor;

  // Print the result.
  printf("The elevator will travel to the ”);
  printf("actual floor %d\n", actualFloor);

  return 0;
}
```

Third example: print a square of *

```
#include <stdio.h>
int main (void)
{
  int n, i, j;

  n = 0;
  printf("Insert an integer >= 2: ");
  scanf("%d", &n);
  if (n < 2){
    printf("Error: value < 2\n");
    return -1;
  }

  for(i=0; i<n; i++)
    printf("*");
  printf("\n");

  for(i=2; i<n; i++){
    printf("*");
    for(j=2; j<n; j++)
      printf(" ");
    printf("*\n");
  }

  for(i=0; i<n; i++)
    printf("*");
  printf("\n");

  return 0;
}
```

Execution

```
Insert an integer >= 2: 1
Error: value < 2
```

```
Insert an integer >= 2: 4
****
*  *
*  *
****
```

Forth example: verify order

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

const int MAXC=50;

int verifyOrder(FILE *fp);

int main(void) {
  char namein[MAXC+1];
  FILE *fin;

  printf("name of input file: ");
  scanf("%s", namein);
  fin=fopen(namein,"r");

  if (verifyOrder(fin)==1)
    printf(”File %s is ordered\n", namein);
  else
    printf(”File %s is not ordered\n",
           namein);
  fclose(fin);
  return 0;
}
```

```
int verifyOrder(FILE *fp) {
  char row0[MAXC+1], row1[MAXC+1];

  fgets(row0,MAXC,fp);

  while (fgets(row1,MAXC,fp)!=NULL) {
    if (strcmp(row1,row0)<0)
        return 0;
    strcpy(row0,row1);
  }
  return 1;
}
```

Syntactic elements of C language

- **Reserved words** (*keyword*)
  - Es. break, if, for, int, float
- **Identifiers**
  - free, predefined
- **Literal constants**
  - Numbers, characters, strings
- **Special characters** (symbols: *parenthesis, punctuation, operators*)
  - Es. {} () [] , ; . ->

Example of C program

```c
/* simple C program */
#include <stdio.h>
int main(void)
{
    int max, A, B;

    scanf("%d%d",&A,&B);
    if (A >= B)
        max = A;
    else
        max = B;
    printf("%d\n",max);
    return 0;
}
```

/* comments */
keywords
Free identifiers
predefined
Literal constants
Special characters

Reserved words (keywords)

Words “reserved” for syntactic/semantic purposes

- Cannot be used for other purposes
- They are the “building blocks” of the syntax of the language

In standard C they are 32

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Identifiers

- They refer to one of these objects:
  - Constants
  - Variables
  - Types
  - Subprograms (functions)
- Rules:
  - They start with either alphabetic character or “_”
  - They contain only alphanumeric characters or “_”
  - Case sensitive

Literal constants and special characters (symbols)

- Literal constants
  - They represent numeric values, characters and strings:
    - Integer numbers: 10 -72 025 0x24
    - Real numbers: 3.14159 1.7E+12
    - Characters: 'H' ';' '\0' '\n'
    - Strings: "Hello World!\n"
- Special characters (symbols)
  - They are used to:
    - Group/separate parts of a program (es. { } , ; …)
    - Represent operations within expressions (es. > < >= <= + - * / & ++ -- …)

Comments

- Free text in a program
- Ignored by the compiler
- Is there for the programmer, not for the system!
- Format:
  - Multi-line: between /* and */
  - Nested comments are not possible
  - Single line: Starts with // until end of line (it was only C++, even C starting from C99)
- Example:
  - /* This is a comment! (It could be in multiple lines) */
  - /* This /* will generate */ an error */
  - // This is another comment

Structure of a C program

A C program consists of a sequence of:

- Directives to the pre-processor
- Functions (main function is one of them, mandatory)
- Data
  - Variables
  - Constants
  - Expressions
- Instructions
  - Declaratives
  - Operatives
- Comments

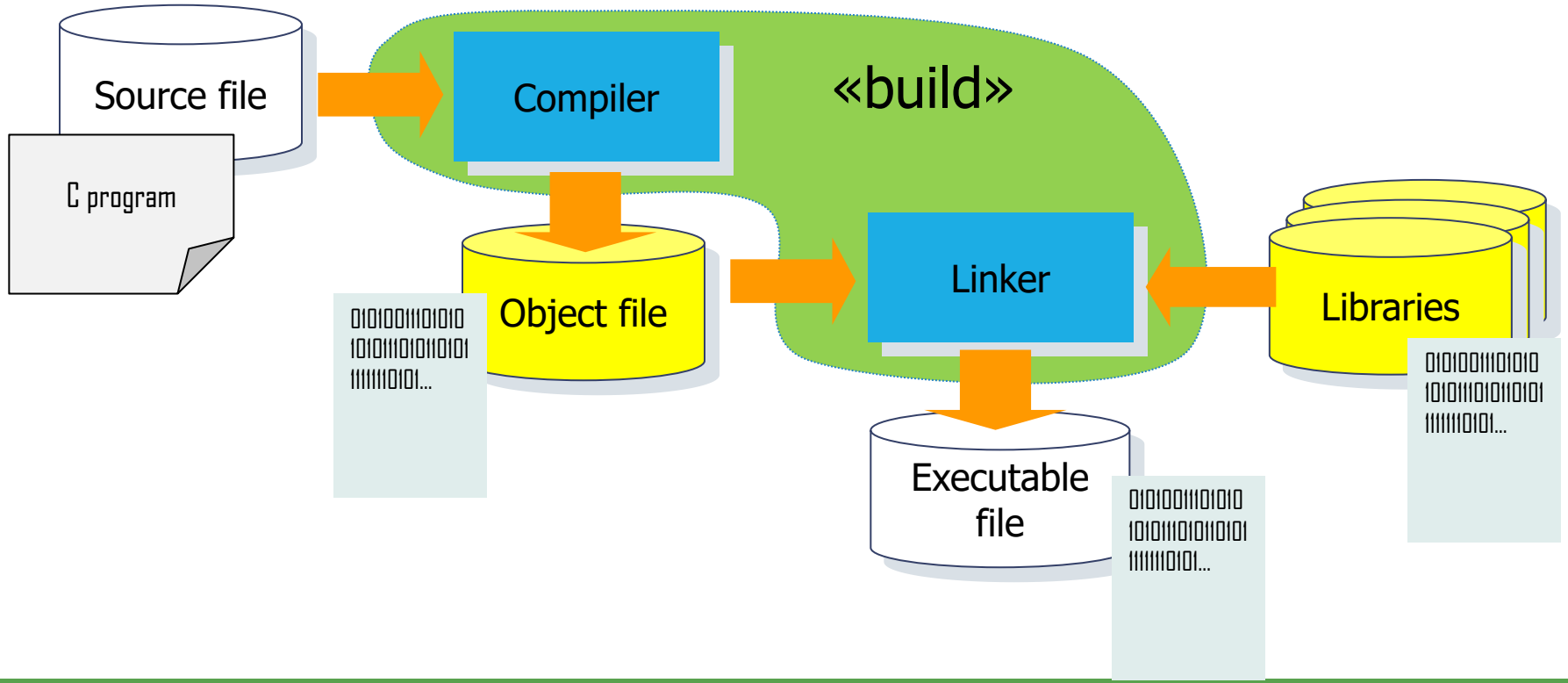
Structure of a simple C program

```
<directives to preprocessor>
<global declarations, if any>

int main (void) {
  <declarative instructions>
  <operative instructions>
  return 0;
}
```

```
#include <stdio.h>
/* there are no global declarations*/
int main(void) {
   int max, A, B;
   scanf("%d%d",&A,&B);
   if (A >= B)
      max = A;
   else
      max = B;
   printf("%d\n",max);
   return 0;
}
```

Program translation: from source file to executable



Object file and libraries

- Object files do not include commands (or instructions) to interact with hardware and operating system
- These commands are in the system library
  - They contain common-use commands, already pre-compiled
- The linker is a piece of software that links the object file to the libraries, in order to generate an executable file, that contains ALL the information that is necessary for the execution of the program

The data

- Types
- Declaration of variables and constants
- Assignments
- Expressions
- Cast

Data declaration

- **In C, all the objects need to be declared before they can be used!!**
- Data declaration (or definition) consists in in:
  - **Allocating** suitable space in memory for data storage
  - **Giving** a univocal name to that space in memory
- More specifically, we need to provide:
  - **Name** (identifier)
  - **Type**
  - **Modality of access** (variable or constant)

Primitive data types (scalar)

- Are the ones directly provided by C language
- Identified by keywords!
  - char ASCII characters
  - int integers (2's complement)
  - float real numbers (single precision floating point)
  - In C99: _Bool boolean (true: 1 or false: 0)
- The actual size in memory of these data types depends on the architecture of the system (it is not specified by the language)
  - |char| = 8 bit = 1 Byte always, in any architecture

Primitive data types (scalar)

- Types
  - Basic: int, char, float
  - Variants: (unsigned/signed, short, long, double)
- Constants (values) → they can never modified during the program
  - literals:
    - 10 -72 0250x24
    - 3.14159 1.7E+12
    - 'H' ';' '\0' '\n' "Hello world!\n"
  - Symbolic (identifier associated to a value):

```
#define N 100
#define PIGRECO 3.14159
#define hash '#‘
```

```
const int N = 100;
const float PIGRECO = 3.14;
const char hash ='#';
```

Modifiers of basic types

They are constituted by keywords that precede the basic types

- Sign: signed/unsigned
  - Can be applied to either char or int
  - signed: numeric value with sign
  - unsigned: numeric value without sign
- Dimension: short/long
  - Can be applied to int
  - The int keyword can be even omitted (it is implicit)
- In C99: Complex numbers / imaginary part:
  - _Complex
  - _Imaginary

Modifiers of basic types

- Integer numbers
  - [signed/unsigned] short [int]
  - [signed/unsigned] int
  - [signed/unsigned] long [int]
  - [signed/unsigned] long long [int] (in C99)

- Real numbers
  - float
  - double
  - long double (in C99)
  - float _Complex (in C99)
  - double _Complex (in C99)
  - long double _Complex (in C99)

Declaration of variables

- Examples:
  - int x;
  - char ch;
  - long int x1, x2, x3;
  - double pi;
  - short salary;
  - long y, z;
- Always use identifiers that means something, for readability!!!
  - Examples:
    - int x0a11; /* NO */
    - int value; /* YES */
    - float radius; /* YES */

Declaration of constants (symbolic)

- Identifier associated to a value (non-modifiable)
- Two options:

1. We declare a variable and we make it “non-modifiable”

   const <type> <constant_name> = <value> ;

Examples:

- const double PIGRECO = 3.14159;
- const char SEPARATOR = ‘$’;
- const float RATE = 0.22;

2. Directive to the preprocessor (#define) that associate an identifier to a literal constant

Examples:

- #define PIGRECO 3.14159
- #define SEPARATOR ‘$’
- #define RATE 0.22

Constants (examples)

- Examples of values that can be associated to a constant:
  - char values:
    - ‘f’
  - int, short, long:
    - 26
    - 0x1a,0X1a
    - 26L
    - 26u
    - 26UL
  - float, double:
    - -212.6
    - -2.126e2, -2.126E2, -212.6f

Special characters

- “Predefined” characters
  - ‘\b’ backspace
  - ‘\f’ form feed
  - ‘\n’ line feed
  - ‘\t’ tab
- Other non-printable and/or “special” characters
  - “escape sequences”
    - \<ASCII code, base 8 in 3 digits>
  - Examples:
    - ‘\007’
    - ‘\013’

Visibility of the variables

- Each variable can be used within a specific range of visibility (scope)
- Global variables
  - Defined outside main()
- Local variables
  - Defined within main()
  - More in general, defined within a block of the program

Blocks structure

- In C, it is possible to create blocks by enclosing a set of instructions within { } braces
- Meaning: Delimitation of a visibility scope of data (variables, constants)
- Corresponds to a “sequence” of instructions in a flow-chart
- Example:

```
{
   int a=2;
   int b;
   …
}
```

`a` e `b` are defined only within the block!

Visibility of variables: Example

```
/* this program is incomplete, there are declarative
instructions but no operative instructions */
int n;
double x;
int main()
{
    int a,b,c;
    double y;
    {
        int d;
        double z;
    }
}
```

Assignment

- **Meaning:** to assign = to give a value to a variable
- **Typical use**: to modify the value of a variable
- **Syntax:** `<identifier> = <value>;`
  - On the left: variable to modify,
  - = symbol: assignment operator
  - On the right: expression that generates the value to be assigned
- **Type:** value needs to be ***compatible*** with the type of the variable
  - For example, we cannot assign a float value to a char variable
  - It is possible to assign int value to a float variable or viceversa. But we need to know the conversion rules!!! (CAST)

Assignment

- Value can be:
  - A constant (literal or symbolic)
  - A variable
  - An expression containing constants, variables, operators or even calls to a function
- Examples
  - `a = 42;`
  - `c = NUM;`
  - `b = a;`
  - `d = a + b * division(a, NUM) + 5;`

Expression

- **Meaning:** formulas representing arithmetic/logic computations, involving operands and operators
- **Operands** can be:
  - constants (literal and/or symbolic)
  - Variables (the expression uses the current value of the variable at the time the expression is executed).
  - Calls to functions
- **Evaluation and result:**
  - If the expression contains only constants, the result does not depend on the time it is executed
  - If it contains variable, result depends on the current value of the variables at the time of execution

Expressions (examples)

```
5 – 10
3 * 3.14
a – 10
b * 3.14
(x + 5) * (x - y)
2*(a + b) - (a*a - b*b)
(2 * PIGRECO * r)

10 * 20  + 1 // = 200 + 1 =  201
10 * (20 + 1) // = 10 * 21 =  210
```

Cast (problem)

- Expressions and assignments typically ***require*** variables and constants of compatible type:
  - Arithmetic operations requires numbers of same type (ex. all `int` or all `float`)
  - Assignment requires that the expression on the right has a result value of the same type of the variable on the left.
- Example: In principle, given two x and y **`float variables`**, following instruction should **(SHOULD)** be wrong:
  - `x = 1;`
  - `y = x * 2;`

  Because the 1 and 2 constants are int and not float

Cast (problem)

- Expressions and
  constants of con
  - Arithmetic opera
    `float`)
  - Assignment requ
    the same type of the vari

They should be re-written as follows:

```
x = 1.0;
y = x * 2.0;
```

- Example: In princip
  following instruc
  - `x = 1;`
  - `y = x * 2;`

  Because the 1 and 2 constants are int and not float

two x and y **`float variables`**,
should **(SHOULD)** be wrong:

Cast (solution)

- A strict compatibility between variables / constant is
  - Formally desirable,
  - Too restrictive, in the practice.
- We accept a COMPROMISE between rigor and practicality
- CAST: operation of type conversion
  - Generates a value of a specific type, starting from one of another type
  - Can be implicit: applied automatically
  - Can be explicit: by preceding the variable (or expression) with the new type it needs to be converted to, written between round brackets ()

Cast (rules)

- Data types hierarchy, based on correspondence/inclusions rules:
  - Ex. {char} ⊂{int} ⊂ {float} ⊂{double}
  - Promotion: a less “general” type is promoted to a most general one (ex. int is converted to float)
- Implicit CAST: the promotion happens whenever possible
  - Ex. If an arithmetic operation involves int and float variables: int is converted to float before executing the operation.

    6/1.2 implicitly is executed as 6.0/1.2
- Explicit CAST: we can convert data types even on the opposite direction, but we lose some information
  - Ex. If we cast from float to int we lose the decimal part:

  (int) 7.8 becomes 7

Cast (examples)

```
/* IMPLICIT cast*/
  float x, y;
  ...
  x = 10; // converted to = 10.0;
  y = x * (2/3.0); // converted to y = x * (2.0/3.0);
  ...
  int a_ascii, n;
  ...
  /* 97 (the ASCII code of 'a', converted to an integer value) is assigned
    to a_ascii,  26 (difference between ASCII codes of 'z' and 'a', seen as
    integer) is assigned to n */
  a_ascii = 'a';
  n = 'z' - 'a' + 1;
```

Cast (examples 2)

```
/* x is assigned 1.75, but the assignment i = x
  produces a conversion to integer, with truncation: i is assigned the value 1.
  float x; int i;
  ...
  x = 7.0/4;
  i = x;

/* explicit cast*/
  float x, y, z, w;
  ...
  x = 10.5;
  y = (float)((int)x * (3/2)); // y = 10.0
  w = x * (float)(3/2); // w = 10.5
  z = x * (float)3/(float)2;  // z = 15.75
```

Summarizing

Definition/declaration:

```
int number;
char a, b, c;
float num_real;
```

Expressions:

```
5 - 10
3 * 3.14
a - 10
b * 3.14
(x + 5) * (x - y)
2*(a + b) - (a*a - b*b)
(2 * PIGRECO * r)
```

Summarizing

Assignment:

```
tmp = a;  // standard assignment
a = b;
b = a;
```

Initialization:

```
int a = 1, b = 2; // definition with initialization
```

Cast (type conversion):

```
x = 10.5;
y = (float)((int)x * (3/2));  // y = 10.0
w = x * (float)(3/2);  // w = 10.5
z = x * (float)3/(float)2; // z = 15.75
```

Control flow constructs

- Conditional construct
  - if (with or without else)
  - switch-case
    - Multiple selector, but ONLY with int or char data types!
- Iterative constructs
  - while, do-while
    - General loops (driven by boolean expressions)
  - for
    - Loops with apriori known number of iterations (with counting)

Conditional expressions

- Logic expressions:
  - Conditions that enable/disable the block that needs to be executed
- if

```
if (condition) {
  // instructions to be executed if condition is true
}
else {
  // if condition is false
}
```

- If only one instruction in either block, { } can be omitted
- There can be multiple conditions and nested if constructs

Switch: multiple selection

```
switch (selector) {
  case 0: printf("Option n. 0\n"); break;
  case 1: printf("Option n. 1\n"); break;
   ...
  default: printf("None of the options above\n");}
}
```

Iterative constructs (loops)

```
while           while (condition_to_continue) {
                    ...
                }

do … while      do {
                  printf("Insert a positive number");
                  scanf("%d", &x);
                } while (x<=0);

for             for(i = 0; i < 10; i++) {
                  printf("Insert an integer value:  ");
                  scanf("%d", &vet[i]);
                }
```

Input/Output

- I/O on text files (of characters). We won't deal with binary I/O (fread, fwrite)
  - stdin, stdout, stderr (always automatically open)
  - fopen/fclose for all other files
- I/O operations:
  - Formatted I/O: (f)printf and (f)scanf (%d, %c, %f, %s)
  - Of lines/strings: (f)gets, (f)puts ((f)printf ("%s"))
  - Of single characters: (f)getc/getchar, (f)putc ((f)printf ("%c"))
- Output (easier)
- Input
  - Easy if format is fixed
  - Complicated if format is free

Input/output (including files)

- Open/close a file:
  - FILE *fp;
  - fp=fopen("myfile.txt","r");
  - …
  - fclose(fp);
- I/O:
  - formatted (includes almost all the others):
    - fscanf, fprintf, scanf, printf, (sscanf, sprintf)
  - strings:
    - fgets, fputs, gets, puts
  - characters:
    - fgetc, fputc, getchar, putchar

Input/output of characters (examples)

```
// Examples: getc, fgetc, getchar
  char a, b, c;
  FILE *fp;
  fp=fopen("myfile.txt","r");
  ...
  // reads one character from file pointed by fp
  a = getc(fp);
  // reads one character from file pointed by fp
  b = fgetc(fp);
  // reads one character from keyboard (stdin)
  c = getchar();
  // reads one character from keyboard (stdin)
  c = getc(stdin);
   ...
  fclose(fp);
```

```
// Examples: putc, fputc, putchar
  char a = 'x', b = 'y', c = 'z';
  FILE *fp;
  ...
  fp=fopen("myfile.txt","w");
  // writes one character on file pointed by fp
  putc(a, fp);
  // writes one character on file pointed by fp
  fputc(b, fp);
  // writes one character on video (stdout)
  putchar(c);
 // writes one character on video (stdout)
  putc(c, stdout);
   ...
  fclose(fp);
```

Input/output of strings (examples)

```
// Example of gets
  char mystring[5];
  // reads from keyboard (stdin) to mystring
  gets(mystring);

// Example of fgets
  char str[50];
  FILE *fp;
  fp=fopen("myfile.txt","r");
  // reads from file pointed by fp to str
  fgets(str,10,fp)
  fclose(fp);
```

```
// Example of puts
  char mystring[5]="ciao";
  // writes the string in mystring to video (stdout)
  puts(mystring);

// Example of fputs
  FILE *fp;
  char mystring[5]="ciao";
  fp=fopen("myfile.txt","w");
  // output in file
  fputs (mystring,fp);
  fclose(fp);
```

Input/output of strings (examples)

```
// Example of gets
  char mystring[5];
  // reads from keyboar  (stdin) to mystring
  gets(mystring);

// Example of fgets
  char str[50];
  FILE *fp;
  fp=fopen("myfile.txt","r");
  // reads from file pointed by fp to s
  fgets(str,10,fp)
  fclose(fp);
```

```
// Example of puts
  char mystring[5]="ciao";
  // writes the string in mystring to video (stdout)

  fclose(fp);
```

A string is a vector (array) of characters, of FIXED dimension.
It always end with a special '\0' character
mystring can contain 5 characters in total (4 characters + '\0')

Input/output of strings (examples)

```
// Example of gets
  char mystring[5];
  // reads from keyboard (stdin) to mystring
  gets(mystring);

// Example of fgets
  char str[50];
  FILE *fp;
  fp=fopen("myfile.txt","r");
  // reads from file pointed by fp to
  fgets(str,10,fp)
  fclose(fp);
```

```
// Example of puts
  char mystring[5]="ciao";
  // writes the string in mystring to video (stdout)
  puts(mystring);
```

gets reads all characters typed on the keyboard by the user, until the newline or EOF (end of file). It automatically adds ‘\0‘ at the end and discards the ‘\n’.
There might be more characters than the ones mystring can contain (RISK!!).

Input/output of strings (examples)

```
// Example of gets
  char mystring[5];
  // reads from keyboard (stdin) to
  gets(mystring);

// Example of fgets
  char str[50];
  FILE *fp;
  fp=fopen("myfile.txt","r");
  // reads from file pointed by fp to str
  fgets(str,10,fp)
  fclose(fp);
```

```
// Example of puts
                                    lout)
                    p;
  char mystring[5]="ciao";
  fp=fopen("myfile.txt","w");
  // output in file
  fputs (mystring,fp);
  fclose(fp);
```

Reads at most 9 (10-1) characters from the file and stores them in str, adding ‘\0‘ at the end
Allows to "protect" str in case there are more characters than the ones that str can store.
Differently from gets, it stores the ‘\n’, too.

Formatted I/O

- Allows to specify a string that
  - (output) needs to be printed,

or

  - (input) corresponds to the way the input is provided.

- The string can include format directives (starting with % operator) that specify how to treat the input/output representing numeric or textual data. Main format directives are:
  - %c for single characters,
  - %s for strings,
  - %d for decimal integers,
  - %f for float

Formatted I/O

- Formatted I/O includes the possibility to read/write
  - Single characters (format directive "%c")
  - Strings (format directive "%s").
- Many redundant options
  - Single characters can be read/written in two alternate ways
    - Using formatted I/O (scanf,fscanf,printf,fprintf with directive "%c")
    - With functions getc, fgetc, putc, fputc, getchar, putchar
  - Strings can be read/written in two alternate ways
    - Using formatted I/O (scanf,fscanf,printf,fprintf with directive "%s”)
    - With functions fgets, fputs, gets, puts.

Formatted I/O

- Formatted I/O f… …ludes the possibility to read/write … "%c")
- …haracters can be read/written in two alternate ways
  - …rmatted I/O (directive "%c")
  - Wit… …ctions getc, fgetc, putc, fputc, getchar, putchar
- Strings can be read/written in two alternate ways
  - Using formatted I/O (directive "%s")
  - With functions fgets, fputs, gets, puts.

fputs and puts can be completely replaced by formatted output (fprintf, printf) with format "%s"

Formatted I/O

- ...udes the possibility to read/write
  ... "%c")
  - ...haracters can be read/writte
    - ...rmatted I/O (directive "%c")
    - Wit... ...ctions getc, fgetc, putc, fputc, getch...
  - Strings can be read/written in two a...
    - Using formatted I/O (directive "%s")
    - With functions fgets, fputs, gets, puts.

fputs and puts can be completely replaced by formatted output (fprintf, printf) with format "%s"

Formatted input with fscanf e scanf with format "%s" is not exactly the same as fgets and gets!
Input with %s uses spaces as separators, while fgets e gets read whole lines (including spaces, if present) until newline character.

Formatted input/output (examples)

```
// Example: use of scanf
  int n;
  scanf("%d",&n); // reads int from stdin (keyboard)
// Example: use of fscanf.
  FILE *fp;
  int n;
  fp=fopen("myfile.txt","r");
  ...
  fscanf(fp,"%d, &n); reads int from file

// Example: effect of spaces.
  char str1[50], str2[50]; FILE *fp;
  fp=fopen("people.txt","r");
  fscanf(fp, "%s", str1); // reads until first space
  rewind(fp); // resets fp to go back to beginning of file
  fgets(str2, 50, fp); // reads whole line
```

```
// Example: use of printf
  int n=5;
  printf("%d",n); // prints int to stdout (video)

// Example: use of printf
  FILE *fp;
  int n=5;
  fp=fopen("myfile.txt","w");
  ...
  fprintf(fp,"%d",n); // prints int to file
```

Formatted input/output (examples)

```
// Example: use of scanf
  int n;
  scanf("%d",&n); // reads int from stdin (keyboard)
// Example: use of fscanf.
  FILE *fp;
  int n;
  fp=fopen("myfile.txt","r");
  ...
  fscanf(fp,"%d, &n); reads int from file

// Example: effect of spaces.
  char str1[50], str2[50]; FILE *fp;
  fp=fopen("people.txt","r");
  fscanf(fp, "%s", str1); // reads until
  rewind(fp); // resets fp to go ba  to beginning of file
  fgets(str2, 50, fp); // reads whole line
```

```
// Example: use of printf
  int n=5;
  printf("%d",n); // prints int to stdout (video)
```

Suppose the first line of file is
This is an example

fscanf (str1, ...) will read This
fgets (str2, ...) will read the whole line
This is an example

Formatted input/output

- You will need a LOT of practice (laboratories, exercises, examples)

- Careful not to lose synchronization input and format
  - spaces, newlines, inconsistent inputs
  - never mix up inputs of single characters (%c, getc, fgetc) and/or whole lines (newline included) with input that uses spaces/newlines as separators (it is not wrong per se, but it is very complicated: handle with care!)

End of file test

- EOF constant (typically EOF = -1)
  - fscanf(…)==EOF
  - if file not finished, fscanf returns the number of % records that were correctly read
  - getc/fgetc(…)==EOF
  - if file not finished, they return the ASCII code (int) of the character that was read

- fgets(…)==NULL
  - if file not finished, fgets returns a pointer to the string that was read (destination of the input)

- feof() function:
  - for example: if (eof(fp))
  - for example: while (!feof(fp))
  - CAREFUL: feof() becomes true only AFTER it tries reading after the end-of-file!!!! (easy to do an extra-reading by mistake)

End of file test

- EOF constant (typically EOF = -1)
  - fscanf(…)==EOF
  - if file not finished, fscanf returns the number of % records that were correctly read
  - getc/fgetc(…)=
  - if file acter that was read
- fgets(
  - if file s read (destination of the input
- feof()
  - for example: if (eof(fp))
  - for example: while (!feof(fp))
  - CAREFUL: feof() becomes true only AFTER it tries reading after the end-of-file!!!! (easy to do an extra-reading by mistake)

```
while (fscanf("%d%f",…)!=EOF) {
   ...
}
// or
while (fscanf("%d%f",…)==2) {
   ...
}
```

End of file test

- EOF constant (typically EOF = -1)
  - fscanf(…)==EOF
  - if file not finished, fscanf returns the number of % records that were correctly read
  - getc/fgetc(…)==EOF
  - if file not finished, they return the ASCII code (int) of the character that was read

- fgets(…)==NULL
  - if file not finished, fge… a pointer to the string that was read (destination of the input)

```
while (fgets(fp,MAX,line)!=NULL) {
   ...
}
```

- feof() fur…
  - for exam…
  - for exam…
  - CAREFUL: feof() becomes true only AFTER it tries reading after the end-of-file!!!! (easy to do an extra-reading by mistake)

End of file test

- EOF constant (typically EOF = -1)
  - fscanf(…)==EOF
  - if file not finished, fscanf returns the number of % records that were correctly read

```
while (!feof(fp)) {
   fscanf("%s",line);
   printf("I read: %s\n", line);
}
```

vas read

tination of the input)

- feof() function:
  - for example: if (eof(fp))
  - for example: while (!feof(fp))
  - CAREFUL: feof() becomes true only AFTER it tries reading after the end-of-file!!!! (easy to do an extra-reading by mistake)

End of file test

- EOF constant (typically EOF = -1)
  - fscanf(…)==EOF
  - if file not finished, fscanf returns the number of % records that were correctly read

```
while (!feof(fp)) {
   fscanf(”%s”,line);
   printf(”I read: %s\n”, line);
}
```

TRIES TO READ AFTER END OF FILE!!

…was read

…tination of the input)

- feof() function:
  - for example: if (eof(fp))
  - for example: while (!feof(fp))
  - CAREFUL: feof() becomes true only AFTER it tries reading after the end-of-file!!!! (easy to do an extra-reading by mistake)

End of file test

- EOF constant (typically EOF = -1)
  - fscanf(…)==EOF

```
while (!feof(fp)) {
  fscanf(”%s”,line); // tries to read a line
  // if it was an end of file, it discards line
  if (!feof(fp))
    printf(”I read: %s\n”, line);
}
```

- feof() function:
  - for example: if (eof(fp))
  - for example: while (!feof(fp))
  - CAREFUL: feof() becomes true only AFTER it tries reading after the end-of-file!!!! (easy to do an extra-reading by mistake)

Functions

- Function as a subprogram
  - Written once, used multiple times
    - content: it performs operations on parameters and local variables and provides result by return
  - interface
    - prototype, caller
    - Parameters passing
      - by value
      - by reference (in C it does not exist: it is obtained by passing pointers by value)
- Rules are similar to Python:
  - Interface: formal parameters – actual parameters (arguments)
  - Body of the function
    - Formal parameters are a "copy" of the actual ones (local variables initialized with the same value of the arguments)
    - Exception: arrays are "shared" with the caller

Example of function (python vs C)

```
##
#  main function

def main() :
    result = cubeVolume(2)
    print("A cube with side length 2 has volume",
      result)

#  cubeVolume function

def cubeVolume(sideLength) :
    volume = sideLength ** 3
    return volume

main
```

```
# include <math.h>  // pow function is in math library
# include <stdio.h>

//  cubeVolume function prototype
int cubeVolume(int sideLength);


// main function
int main(void) {
  int result;
  result = cubeVolume(2);
  printf("A cube with side length 2 has volume %d\n",
result);
}

// cubeVolume function
int cubeVolume(int sideLength) {
  int volume = pow(sideLength,3);
  return volume;
}
```

Example of function (python vs C)

```
##
#  main function

def main() :
    result = cubeVolume(2)
    print("A cube with side length 2 has volume",
      result)

#  cubeVolume function

def cubeVolume(sideLength) :
    volume = sideLength ** 3
    return volume

main
```

```
# include <math.h>  // pow function is in math library
# include <stdio.h>

//  cubeVolume function prototype
int cubeVolume(int sideLength);

// main function
int main(void) {
  int result;
  result = cubeVolume(2);
  printf("A cube with side length 2 has volume %d\n",
result);
}

// cubeVolume function
int cubeVolume(int sideLength) {
  int volume = pow(sideLength,3);
  return volume;
}
```

Interface: formal parameters

Example of function (python vs C)

```
##
#  main function

def main() :
    result = cubeVolume(2)
    print("A cube with side length 2 has volume",
      result)

#  cubeVolume function

def cubeVolume(sideLength) :
    volume = sideLength ** 3
    return volume

main
```

Body of function

```
# include <math.h>  // pow function is in math library
# include <stdio.h>

//  cubeVolume function prototype
int cubeVolume(int sideLength);


// main function
int main(void) {
  int result;
  result = cubeVolume(2);
  printf("A cube with side length 2 has volume %d\n",
result);
}

// cubeVolume function
int cubeVolume(int sideLength) {
  int volume = pow(sideLength,3);
  return volume;
}
```

Example of function (python vs C)

```
##
#  main function

def main() :
    result = cubeVolume(2)
    print("A cube with side length 2 has volume",
      result)

#  cubeVolume function

def cubeVolume(sideLength) :
    volume = sideLength ** 3
    return volume

main
```

Function call with arguments (actual parameters)

```
# include <math.h>  // pow function is in math library
# include <stdio.h>

//  cubeVolume function prototype
int cubeVolume(int sideLength);


// main function
int main(void) {
  int result;
  result = cubeVolume(2);
  printf("A cube with side length 2 has volume %d\n",
result);
}

// cubeVolume function
int cubeVolume(int sideLength) {
  int volume = pow(sideLength,3);
  return volume;
}
```

Aggregated data types

- arrays: vectors and matrices
  - Aggregate of values of homogeneous type, with position indexes
    - int v[100]; float M[10][10];
    - X = V[i]*M[j][k];
    - Dimensions APRIORI KNOWN (constant) -> typically over-sized and under-utilized

- strings
  - Monodimensional arrays of “special” characters
  - Processed using library functions (strlen, strcmp, strcpy, …) → # include <string.h>
  - Always end with '\0‘ (string terminator)

- struct
  - Heterogeneous Aggregates (fields can be of different types)
  - Fields identified by names, as if they were local (struct) variables

Vectors (monodimensional arrays)

- AGGREGATED values all of the SAME TYPE, accessed by INDEXING

```
int age[20], height[20], i;
float ageAvg = 0.0;

for(i=0; i<20; i++) {
  scanf("%d%d", &age[i], &height[i]);
  ageAvg += age[i];
}
ageAvg = ageAvg/20;
```

Matrices (multidimensional arrays)

Example 1

```
int diagonal_matrix[3][3] = { { 1, 0, 0 },
                              { 0, 1, 0 },
                              { 0, 0, 1 }  } ;
```

Example 2

```
float M2[N][M], V[N], Y[M];   // M and N are constants
for (r=0; r<N; r++) {
  Y[r] = 0.0;
  for (c=0; r<M; c++)
    Y[r] = Y[r] + M2[r][c]*V[c];
}
```

Strings

- They are NOT a new data type:
  - They are simply a monodimensional array of char
  - Terminated by ‘\0’
- They can be processed:
  - Character by character (like any other array)
  - As a whole:
    - Using I/O functions for strings: ex. fgets, sscanf, fscanf/fprintf (with %s)
    - Using library functions (including <string.h>): ex. strcmp, strlen, strcpy, strcat, ...

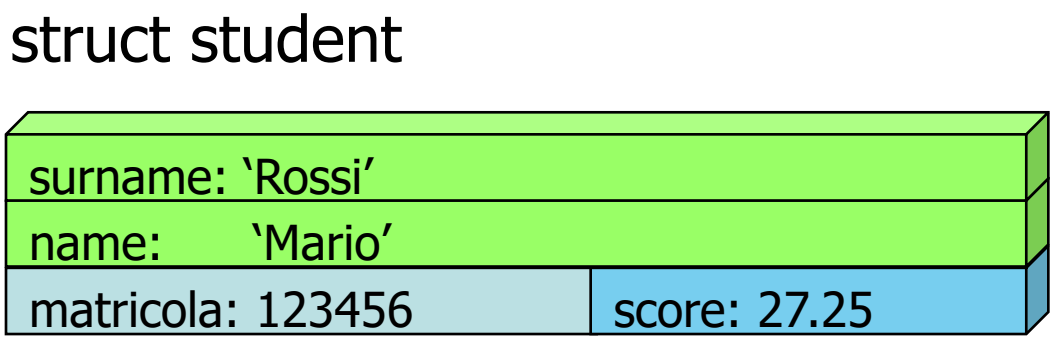
Struct

- Structures (struct data type)
  - Aggregated data type
  - Fields identified by names
  - Differently from arrays, it aggregates heterogeneous data types
- Ex.

```
struct student
{
  char surname[MAX], name[MAX];
  int matricola;
  float score;
};
```

Heterogeneous aggregated data type (struct)

- Heterogeneous information can be aggregated into a single entity as fields of a struct



Struct types

- Heterogeneous aggregated type in C is a struct. Same as record of other programming languages
- A struct is composed by fields:
  - Fields are either basic data types or other structs
  - Each field in a struct can be accessed by means of its identifier (unlike arrays, where elements are accessed by indexing)

```
struct student
{
    char surname[MAX], name[MAX];
    int matricula;
    float score;
};
```

A new data type

- The new type is `struct student`
- Keyword `struct` is mandatory

```
struct student
{
    char surname[MAX], name[MAX];
    int matricola;
    float score;
};
```

New data type

Name of the struct

- Same rules as for the names of the variables
- Names of `struct` need to be different from the names of other struct (they can be the same as the name of other variables, but better avoid…

```
struct student
{
   char surname[MAX], name[MAX];
   int matricola;
   float score;
};
```

Fields

New data type

Name of the struct

- Fields correspond to local variables of the struct
- Each field has a type and an identifier

From C to programming

- Constructs and rules of the language
  - Given for granted!!! (almost)

- Programming = "from problem to solution" (using C language)
  - Strategy -> problem solving
  - Experience and personal skills
  - Learn from proposed solutions
  - NEWS: Classification of problems

Categories of problems we will see…

	Without arrays	**With vectors/matrices**
Numerical	2nd degree eq. Series and numeric successions …	Groups statistics Operations on sets of numbers Prime numbers generation Sum/product of matrices
Enoding	Conversion of bases (ex. binary/decimal) Criptograpy …	Conversion between bases Re-encoding of texts using conversion tables
Text proc.	String manipulations Menu of options Graphs	Counting characters in a text Graphs Text formatting
Verify/ select	Verify order/data consistency Verify moves in a game Filter a list of data Search maximum/minimum Partial sorting	Verify unicity or repetitions of data Select data based on specific acceptance criteria Search data from a table (based on name/string) Selection sorting