Computer Architecture (Practical Class)

Introduction to the C Programming Language

aka C for Java Programmers

Luís Nogueira Raquel Faria

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

{lmn,arf}@isep.ipp.pt

2020/2021

The C programming language

- C is a general-purpose, imperative programming language, developed in the 70es at Bell Labs
- Its history is closely related with the UNIX operating system (OS) as it was created out of the necessity to port the OS to different machines
- It was designed to be simple and small. The book "The C Programming Language, Second Edition" by Brian Kernighan and Dennis Ritchie describes the whole language, the standard library and several examples and exercises with 261 pages
- C is widely used for systems programming (OS kernel, drivers, compilers, ...) and embedded systems
- Examples: Linux OS and MySQL have been written in C

Why C?

- C ranks amongst the most popular programming languages (1st in the TIOBE Index of 2020, 2nd in 2019)
- C influenced many other programming languages (C++, Java, Objective-C, Swift, C#, PHP, Go, ...)
- Provides access to low-level mechanisms such as memory management, explicit initialization and error detection
- C usually produces very efficient programs

Philosophical reason

- Helps understanding what happens in the system from the UI to the electrons ;-)

Operators, conditionals, loops, ...

- Operators, conditionals, loops and other languages constructs are similar to Java
- Operators:
  - Arithmetic: +,-,*,\,%, ++, –, +=, -=, *=, ...
  - Relational: <,>,<=,>=,==,!=
  - Logical: &&, ||, !, ? :
  - Bit: &,|,^,!,«,»
- Language constructs:
  - if( ){ } else { }
  - while( ){ }
  - do { } while( )
  - for(i=0; i<100; i++){ }
  - switch( ) { case 0: ... }
  - break, continue, return
- No exception handling statements

C vs Java

what	C	Java
type of language	function oriented	object oriented
basic programming unit	function	class = ADT
portability of source code	possible with discipline	yes
portability of compiled code	no, recompile for each architecture	yes, bytecode is 'write once, run anywhere'
compilation	creates machine language code	creates Java virtual machine language bytecode
execution	loads and executes program	interprets byte code
variable auto-initialization	not guaranteed	all variables must be initialized; compile-time error to access uninitialized variables

Source: `http://introcs.cs.princeton.edu/java/faq/c2java.html`

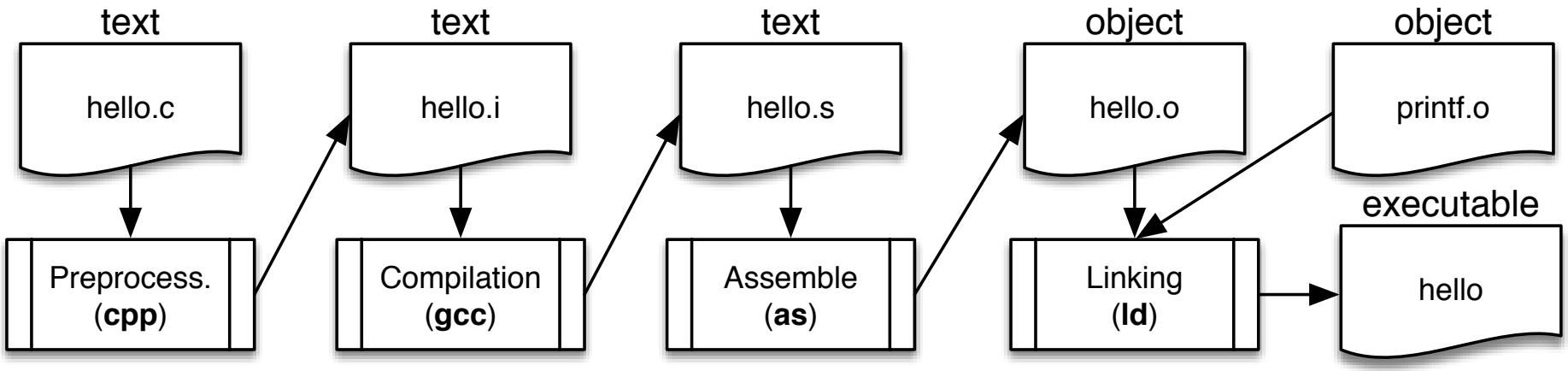
"Hello World" program in C

Listing 1: hello.c (get it: http://codepad.org/q5x1Quqa)

```c
/*
 our program uses printf(), defined in the C standard library "stdio"
 lines staring with '#' are called preprocessor directives, and do not
 have a ';' at the end.
 (this is a multiline comment)
*/
#include <stdio.h>

/*
 the function main() returns an integer and receives no arguments (void)
 main() is the first function to be called when the program is executed
*/
int main(void)
{
        /* printf() prints formatted output;
           \n is a newline */
        printf("Hello, World ! \n");

        // the main function returns the value 0 (single line comment)
        return 0;
}
```

Compiling C programs

- C programs must be transformed into machine-code so they can be executed, in a process called **compilation**
- The compilation process involves several other steps:
  - preprocessing; compilation; assembling; linking



We will use a compiler called the *GNU C Compiler* - ***gcc***

- Compile hello world program: `gcc -Wall hello.c -o hello`
- The gcc option "-save-temps" tells it to keep all files generated during compilation
- Example: `gcc -save-temps hello.c -o hello`

Important gcc switches

option	**Description**
-Wall	all warnings – **use always!**
-o*filename*	output filename for object or executable
-c	compile only, do not link; used to create an object file (.o) for a single (non-main) .c file (module)
-g	insert debugging information
-E	stop after the preprocessing stage; output goes to standard output
-v	show information about gcc and/or compilation process
-S	performs preprocessing and compilation only; that is, convert C source into assembly
-save-temps	keep temporary files created (.i, .s, .o, ...)
-l*library-name*	link with library called *library-name*
-I*dir*	add *dir* to the list of dirs to be searched for header files
-L*dir*	add *dir* to the list of dirs to be searched for the libraries specified with -l;

More at: `http://goo.gl/exA6gY`

Notes about gcc output

- Always read the output of gcc carefully!
  - Gives pretty good indications about the origin of the error
  - Several sources of errors:
    - preprocessor: missing include files;
    - parser: syntax errors;
    - assembler: syntax errors in assembly code (only if you are coding assembly);
    - linker: missing libraries.
  - Often, one error causes lots of subsequent errors
    - fix first error, and then retry – ignore the rest.
  - Often, errors are caused by previous mistakes
    - for example a missing ';' often will cause an error in the subsequent line(s).

Compile with *-Wall* and *do not* ignore warnings!

- Warnings often provide indication about errors that will manifest themselves at runtime

The C preprocessor

- The C preprocessor (cpp) allows defining macros, which are brief abbreviations for longer constructs
- Preprocessor directives start with a '#' at the beginning of the line and are used for:
  - Inserting content of another file into file to be compiled: #include
  - Conditional compilation: #if; #ifdef
  - Definition of macros and constants: #define.
- Before compilation, the preprocessor reads the source code and transforms it.

The C preprocessor

- Example 1:

```
#include <stdio.h> // searches for stdio.h in system defined directories
#include "mydefs.h" // searches for mtdefs.h in the current directory
```

- Example 2:

```
#define MAX 100
#define check(x) ((x) < MAX)
if check(i) { ...  }
```

  - Becomes:

```
if ((i) < 100) { ...  }
```

Use the C preprocessor with caution

- It is easy to introduce subtle errors
- Not visible in debugging
- Code hard to read

C data types

Integer types

Type	Storage size	Value range
char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
int	4 bytes (IA-32)	-2,147,483,648 to 2,147,483,647
unsigned int	4 bytes (IA-32)	0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Floating-point types

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

C data types - Example declarations

```
char c='A';

char c=100;

int i=-2343234;

unsigned int ui=100000000;

float pi=3.14;

double long_pi=0.31415e+1;
```

Remarks on C data types

- The storage size of some types **varies** among architectures. E.g. A *long* is 4 bytes in IA32 and 8 bytes in x86-64 machines
- *char* is misleading. It is a numeric type that happens to be sometimes used to store ASCII character codes
- The *void* type comprises an empty set of values; it is an incomplete type that cannot be completed.
  - You cannot define variables of type *void*, however *void* can be used to:
    - indicate that a function has no parameters. E.g. `int func(void);`
    - indicate that a function has no return. E.g. `void func(int n);`
    - define a pointer that does not specify the type it points to (more on this in the following lectures). E.g. `void* ptr;`
- Two kinds of type conversions
  - Implicit: automatic type conversion by the compiler.
    E.g.: `int a=1000; char b=a; // b=-24 (lower 8 bits of a=11101000...)`
  - Explicit: explicitly defined by the programmer.
    E.g.: `float f=1.2; int d=(int)f; // d=1`

Using `sizeof`

- C has a unary operator `sizeof`, that can be used to get the storage size of variables and data types, *measured in the number of char type storage size*.
- Examples:
  - `sizeof(int)`: returns the size of *int*
  - `sizeof int`: (same as above) returns the size of *int*
  - `sizeof a`: returns the size of the variable *a*
  - `sizeof(char)`: returns the size of type char; **guaranteed to always be 1**

Important

- While, for most modern systems, the char type has 8 bits, *there is no guarantee that this is always true*.
- The number of bits of type char is defined in the CHAR_BIT constant in `<limits.h>` .

- Check the file `<limits.h>` for the sizes and limits of the integer types
  - e.g. CHAR_MAX, CHAR_MIN, INT_MAX, INT_MIN
- Check the file `<float.h>` for the sizes and limits of the floating-point types
  - e.g. FLT_MIN, FLT_MAX

Using `sizeof`: Example (1/3)

- The next slide will present an example using `sizeof` and several constants from `<limits.h>` and `<float.h>`

`printf()` format specifiers quick reference

- %d or %i: Signed decimal integer
- %u: Unsigned decimal integer
- %f: Decimal floating point, lowercase
- %E: Scientific notation (mantissa/exponent), uppercase
- %c: Character
- %s: String of characters
- see: `http://www.cplusplus.com/reference/cstdio/printf/`

Using sizeof: Example (2/3)

Listing 2: sizeof.c (http://codepad.org/sPFcuyKy)

```c
#include <stdio.h> // needed for printf
#include <limits.h> // needed for CHAR_BIT, INT_MAX, INT_MIN
#include <float.h> // needed for FLT_MAX, FLT_MIN, FLT_DIG

int main() {
   char n='A';

   printf("\nStorage size for variable n: %u\n", sizeof(n));

   printf("\nStorage size for char: %u\n", sizeof(char));
   printf("Number of bits in a char: %u\n", CHAR_BIT);

   printf("\nStorage size for int: %u\n", sizeof(int));
   printf("Minimum int value: %u\n", INT_MIN );
   printf("Maximum int value: %u\n", INT_MAX );

   printf("\nStorage size for float : %d \n", sizeof(float));
   printf("Minimum float positive value: %E\n", FLT_MIN );
   printf("Maximum float positive value: %E\n", FLT_MAX );
   printf("Precision value for float: %d\n", FLT_DIG );

   printf("\nStorage size for double=%u\n", sizeof(double));

   return 0;
}
```

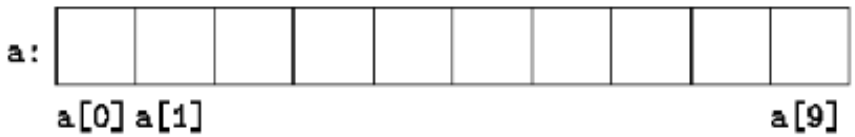
Using sizeof: Example (3/3)

Output of the example (Listing 2; sizeof.c)

```
Storage size for variable n: 1

Storage size for char: 1
Number of bits in a char: 8

Storage size for int: 4
Minimum int value: -2147483648
Maximum int value: 2147483647

Storage size for float : 4
Minimum float positive value: 1.175494E-38
Maximum float positive value: 3.402823E+38
Precision value for float: 6

Storage size for double=8
```

Arrays in C (1/4)

- C allows to define arrays of elements of the same type
- Examples of arrays, with statically defined sizes (size is fixed):
  - `int a[10]; // array of 10 integers`
  - `int a1[]={1, 2, 3, 4, 5}; // array of 5 integers, initialized to 1, 2, 3, 4 and 5;`
  - `int a2[1000]={0}; // array of 1000 integers, all initialized to 0;`
  - `short s[100]; // array of 100 shorts`
  - `float m[10][10]; // 10x10 matrix of floats`
- For an array containing N elements, indexes are $0..N-1$, accessed using `a[0]`, `a[1]`, `...`, `a[N-1]`
- Arrays are stored as a continuous linear arrangement of elements



Arrays in C (2/4)

gcc **does not check** when you access invalid indexes

- `int x[10]; x[10] = 5;` is an overflow of the array, and will result in undefined behaviour (it may work for a while...usually results in a segmentation fault and program termination)

An array **cannot be the target of an assignment**

- Assume an array `int v[10]`, declared previously. The following statement is **not valid**: `v = {1, 2, 3, 4, 5};`
- We can only assign values "in bulk" to an array when it is declared. After the declaration, a valid statement would be: `for (i=0; i<5; i++) v[i] = i+1;`
- If you want to copy arrays, use `memcpy(dest, src, size);`

Arrays in C (3/4)

C does not remember how large arrays are (i.e., no length attribute)

- `sizeof` works **only** for statically defined arrays, within the scope they are declared

- Example:

```
{
  int a[10];
  printf("%u", sizeof(a)); // prints 40 in IA32
}
```

- The {} define the scope of these statements

- Size of array can be computed with `sizeof(a) / sizeof(a[0])`

Arrays in C (4/4)

Passing the address of an array to a function

- When the array is passed as an argument to a function, the size information is not available

```
void func(int a[ ]) {
  printf("%u", sizeof(a)); // prints 4 in IA-32
}
```

- More on this in the folowing classes...

- The solution is for the programmer to maintain the length of the array:
  - by passing the size as an argument of the function;
  - by using a globally defined constant;
  - by defining a data structure for storing the array and its size together;
  - by defining a value that indicates the end of the the array (e.g. an int array ends with a -1 value).

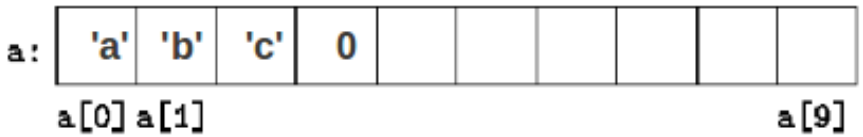
Strings in C (1/2)

C **does not have** a specific data type for strings

- Strings are just char arrays with a NUL (‘\0’) terminator (value zero)

```
char a[10]="abc";
```

- Defines an array of 10 chars. The first 3 chars will have the characters ‘a’, ‘b’, and ‘c’. The fourth will be the NUL terminator:



Strings in C (2/2)

Listing 3: funcao.c (http://codepad.org/gZ65gm8F)

```
int xpto(char s[])
{
    int c=0;

    while (s[c]!=0) c=c+1;

    return c;
}
```

- What is the functionality of the function?
  - A. The function returns the ASCII code of the last character.
  - B. The function returns the number of elements of the string.
  - C. The function returns the number of words of the string.
  - D. None of the above.

Copying strings

Remember!

- Arrays (strings - arrays of char - included) cannot be target of assignments after the declaration

- The following statement is valid, and declares an array s[6], initialized with the characters 'H', 'e', 'l', 'l', 'o', '\0': `char s[]="Hello";`
- However, after declaring `s[]`, you **cannot assign it a new value**: `s="World"; // this is not a valid statement!`
- Keep in mind that strings should be copied with `strncpy(dest_str, src_str, n_chars)`: `strncpy(s, ''World'', 5);`
- Also, note that it is the programmer's responsibility to check that the destination string has enough storage.

Example: Compute the average of two integers

Listing 4: avg.c (http://codepad.org/Jy9uXsjQ)

```c
#include <stdio.h> /* for declaration of printf */

/* globals (really necessary?) */
int n1=6, n2=4, avg=0;

/* this function computes the (integer) average of two integers */
int calc_avg(int a, int b) {
  int c=0;   /* local variable */
  c=(a+b)/2;
  return c;
}

/* the program starts by executing this function */
int main(void) {

  avg = calc_avg(n1, n2); /* call function and save return */
  printf("Avg = %d\n", avg);

  return 0;      /* returns 0 */
}
```

Good C code

- Good code should be mostly self-documenting
  - Variables and function names should generally help making clear what you are doing
  - Comments should not describe what the code does, but why. What the code does should be self-evident (assume the reader knows C)
  - Do comment: each source file, function headers, large blocks of code, tricky bits of code (e.g. bit manipulations)
- Use C-style naming conventions:
  - E.g. prefer `get_radius()` to `GetRadius()`;
  - *i* and *j* for loop variables.
- Bodies of functions, loops, if-else statements, etc. should be indented

Good C code

- Define constants and use them. Constants make your code more readable, and easier to change
- Avoid global variables. Pass variables as arguments to functions
- Initialize variables before using them!
- Use good error detection and handling. *Always* check return values from functions, and handle errors appropriately

Making the best use of C

- Read `https://goo.gl/5u9aCo` for advice on how to use the C language

Practice

- Implement a C program that reads 10 integers into an array and computes their average.
- The average should be calculated in a separate function, but its value should be printed by the main().