# Sistemas Operativos: Tópicos de C
# Operating Systems: C Topics

## 2º MIEIC

Pedro F. Souto (pfs@fe.up.pt)

February 17, 2012

# Sumário

# The `main()` function

- Every program must have one function named `main()`
- The smallest C program is:

```
int main() {
}
```

- Before a C program can be executed it must be compiled. E.g. using `gcc`:

```
gcc -Wall main.c -o main
```

- To run the resulting program (binary/executable) you need only to type its name in a terminal:

```
./main
```

- What does it do?

# The most famous C program

```
#include <stdio.h>
int main() {
    printf("Hello World!\n");
}
```

What does it do?

# The `printf()` function

- This is one of the most useful functions of the C library
  - You'll use it often for debugging
- It has a variable number of arguments. It's prototype is (check `man 3 printf`):

  ```
  int printf(const char *format, ...);
  ```

  `format` is a string that specifies the format of the output. It may be include:
  - Either text that is output verbatim, e.g. `Hello World!`
  - Or **conversion specifiers**, e.g. `%d`, which specify how each of the following arguments should be interpreted. (Check the Wikipedia for a nice description of the conversion specifiers.)

  `...` this specifies a variable number of arguments
  - One per conversion specifier
  - The type of each argument should match that of the corresponding conversion specifier

# Sumário

# What are variables?

- Variables are a programming language concept used to store values
  - The values stored in a variable can change, hence its name

```
int main() {
    int i;

    for( i = 0; i < 10; i++) {
        printf("%d\n", i);
    }
}
```

- We can think of variables as **user defined registers**
  - In C, however, it is more useful to think of them as **memory locations**

# Primitive types of C variables

**Integer Types** types of integer values

- There are several of them `char`, `short`, `int`, `long`, `long long`
- Can be either signed or `unsigned`

**Floating Point Types** types of floating point values

- Can be either single precision (`float`) or double precision (`double`)

- C supports also composite types, such as arrays and structs

# How relevant is a data type?

**Determines the possible values** E.g. a variable of type `char` can store a value between -128 and 127

- Whereas an `unsigned char` variable can store a value between 0 and 255

**Determines the operations that can operate on the values of the type** E.g. the operator `%` requires that both operators be integer types

# Type conversions

- Sometimes C automatically converts a type in another, specially to perform arithmetic operations
- Programmers can also use **casts** to force the program to convert a value of one type to the value of another type

```
int n, m;
double x;


x = (double)n/m;
```

- Type conversions in C can be tricky and may generate unexpected results
  - This is not usual in the kind of programs you'll develop in SO, but ...
  - Check this CERT page for a discussion of some issues

# Scope of a variable (declaration)

- The **scope** of a variable declaration is the region of the program text in which that declaration holds
- In C, the scope of a variable can be:

  Global i.e. the declaration holds in the entire program text
  - Using the `static` keyword, one can limit the scope of a variable declaration to a module, i.e. a C source file

  Local i.e. the declaration holds only in the compound statement (i.e. the statement delimited by matching '{' '}'), after the point where it occurs
  - In earlier versions of the C standard, declarations had to occur always at the beginning of a compound statement before any statement

# Sumário

# C functions

- A C function is very similar to a mathematical function
- In C, there are 3 constructs related to functions:

  **Definition** which specifies the set of instructions to be executed when a function is invoked

  ```
  int sum(int n, int m) {
      return n + m;
  }
  ```

  **Invocation** which determines the execution of the instructions in the function definition, with some particular values for its arguments

  ```
  p = sum(a,1);
  ```

  **Declaration** which specifies the prototype of a function, i.e. its name, the type of the returned value and the type of each of its arguments

  ```
  int sum(int n, int m);
  ```

- The declaration of a function should appear in a program before its invocation.

# Parameter Passing in C (1/2)

## Formal vs. Actual Parameters

**Formal parameters** parameters that appear in a function definition:

```
int sum(int n, int m) {
    return n+m;
}
```

**Actual parameters** parameters that appear in a function invocation:

```
p = sum(1,a);
```

## Pass-by-Value

- The **formal parameters** are **variables local** to the function
- The **value** of each **actual** parameter is **copied** to the corresponding **formal** parameter

# Parameter Passing in C (2/2)

- Copying of parameter values may be inefficient, specially for large variables, such as arrays and structs. In C:
  - Structs are rarely used as function arguments
  - Arrays implementation is such that passing them as parameters is not less efficient than passing the value of a variable, but ...
- The value of the actual parameters after the function call is equal to their value before the function call.
  - The following function does not do what you may expect:

```
void swap(int n, int m) {
    int t;

    t = n;
    n = m;
    m = t;
}
```

when it is called as follows:

```
swap(p,q);
```

# Sumário

# Pointers

Def. A **pointer** is a **variable** whose value is the **address** of a memory location that contains a value of a **given type**

```
char *p1;  // the value pointed by p1 is a char
int *p2;   // the value pointed by p2 is an int
```

## Comments

- If the type requires more than one byte, the address is that of the first memory location with the value
- The size of a pointer depends on the architecture of the underlying processor (actually, nowadays of the platform)
  - Many Intel processors support both 32-bit and 64-bit address operation
  - Sometimes code (not only of the OS, but also of the applications) depends on the "width of the architecture"
    - In the case of Intel it does not affect only the size of the addresses
- Pointers `p1` and `p2` must not be used before they are initialized

# C Pointer Operators

- `*` This is the **dereference** operator. E.g. in

  ```
  c = *p1;
  ```

  `*p1` denotes the value of the memory location pointed by `p1`
  - It is also used to declare a pointer (`char *p1;`)
- `&` This is the **reference** operator, i.e. the operator that extracts the address of a variable. E.g.:

  ```
  char c, *p;

  p = &c;
  ```

  - To initialize a pointer usually we use
    either the `&` operator
    or functions that return addresses of the appropriate type

# Using Addresses as Function Arguments

- Let's use pointers as formal parameters:

```
void swap(int *p, int *q) {
    int t;

    t = *p;
    *p = *q;
    *q = t;
}
```

  and addresses as actual parameters:

```
int m, n;
...
swap(&m,&n);
```

- This is one of the main uses of pointers/addresses in C
  - Parameters are still passed by-value
  - However, as they are addresses, using the dereference operator, we achieve the same effect as if they were passed by-reference (almost always)

# scanf() (man 3 scanf)

```
#include <stdio.h>

int main() {
    int n;
    printf("Enter an integer ...");
    scanf("%d", &n);
    printf("\nRead %d \n", n);
}
```

- `scanf()` is the reciprocal of `printf()`
  - Whereas `printf()` allows a program to output data, `scanf()` allows a program to input data
- Like in `printf()`:
  - the first argument of `scanf()` is a format string that should contain conversion specifiers
  - for each conversion specifier there should be one additional argument with the address of a memory location of appropriate type

# Strings and Pointers (1/2)

- A **string** is a sequence of characters
- In a programming language they are usually implemented by storing each of the characters in a string in consecutive memory locations
- Thus either of the followind suffice to process a string:
  - The address of the location with the first character and the length of the string;
  - The address of the location with the first character and the address of the location with the last character

# Strings and Pointers (2/2)

- C uses a slight variant of the second alternative
  - It uses a **sentinel**, i.e. a character with a special value, `0`, also known as **end-of-string** character
  - This way, when processing a string all we need to know is the address of the memory location of the first character in the string. E.g. in:

    ```
    printf("Hello, World!\n");
    ```

    The compiler:
    - Initializes a region of memory with the characters in the string `Hello, World!`
      `n`, followed by the end-of-string character
    - Puts the address of the first location of that region of memory as actual argument of this invocation of `printf()`

# Arrays and Pointers (1/3)

- In C, an **array** is a sequence of values of the same type

```
int main() {
    int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int i;
    for( i = 0; i < 10; i++ )
        printf("a[%d] = %d\n", i, a[i]);
}
```

  - The indices of an array range from 0 to N-1, where N is the number of elements (length) of the array
- As we would expect, the elements of an array are stored in order in a contiguous memory region

# Arrays and Pointers (2/3)

- C keeps only the address of the first memory location of that region, i.e. of the first element in the array
  - The value of the name of an array, in the example `a`, is the address of the first element in the array, `&a[0]`
- C does not store the length of the array anywhere
  - If included in the definition, it is used only for space allocation when the array is defined, but it is forgotten afterwards

```
int a[10]; // a is an array of 10 integers
```

  - It is up to the programmer to ensure that it uses only valid indices, i.e. there is no **array bound checks**
- C supports **pointer arithmetic**
  - I.e. arithmetic operators when applied to pointers/addresses may yield different results from those when applied to integers of the same size

to operate on arrays

# Arrays and Pointers (3/3)

```
#include <stdio.h>

int main() {
    int a[] = {0, 1, 2};
    int i, *p;
    printf("Array a[] @ 0x%p\n", a);
    for( i = 0, p = a; i < 3; i++, pp++ ) {
        printf("a[%d] (@ %p) = %d\n", i, &a[i], a[i
        printf("a[%d] (@ %p) = %d\n", i, p, *p);
    }
    return 0; // So that the compile does not compl
}
```

- Likewise, difference between pointers/addresses of the same type yields what we would expect