Computer Science

Functions

Functions in C language

- Introduction
- Definition
- Declaration
- Variable types in relation to functions
- Function call
- Exit from a function
- `main()` function arguments
- Recursive functions
- Pointers to functions
- Complex declarations

Introduction (I)

- **Functions** are statement blocks that form the programs in C. All program activity occurs in them.
- Each function is a private, independent and indivisible code and data block.
  - A function can have access just to its own local variables and to global external ones
  - Any function can be accessed from outside just by calling it
  - They are equivalent to subroutines or procedures in other programming languages

Introduction (II)

- All C programs consist at least of one function: `main()`
  - Programs start execution always with `main`
- To maximize program portability, a function should:
  - Be generic
  - Receive information just through its parameters, i.e.
  - Not use external variables

Introduction (III)

- Example: Program to read a set of numbers and obtain its maximum, minimum and mean:

```
#include <stdio.h>
#define N 10
main()
{
  int max, min, med, listnum[N];
  Readdata(listnum, N);
  max = Maximum(listnum, N);
  min = Minimum(listnum, N);
  med = Mean(listnum, N);
  printf("Máximum: %d, Minimum: %d, Mean: %d",
           max, min, med);
  return 0;
}
```

Introduction (IV)

- **Advantages** of using functions
  - Code is structured and organized in independent blocks
  - Data are isolated
  - Error localization is easier
  - Functions can be tested separately
  - Same function can be used in different programs.
- **Disadvantages**
  - Source code may be larger.
  - In execution, call and return requires additional time.

In general **advantages are much more valuable** than disavantages

Function definition (I)

- The general form of a **function definition** in C is:

```
returntype functionname(parameterlist)
  {      /* Body of the function */
         Data declaration
         Statements;
         Return expressions;
  }
```

  - `returntype` is the data type of the value the function returns (`int` by default)
  - `functionname` identifies the name of the function

Function definition (II)

- The **`parameterlist`** refers to the type, order and number of the **formal parameters** of the function
  - They get the values that are passed to the function
  - They work as variables inside the function
  - The list has the following format:

```
type1 ident1, type2 ident2, … typeN identN
```

    - `typeX` represents any valid type
    - `identX` is the identifier of the variable

Function definition (III)

- Example: Function that receives a list of numbers and returns the maximum

```
int Maximum(int list[], int numdat)
{
  int i, max;
  max = list[0];
  for (i=0 ; i<numdat ; i++)
    if (max<list[i]) max=list[i];
  return max;
}
```

Function declaration (I)

- Function **declaration** or **prototype** describes the function:
  - It must be placed before the first function call, preferably at the beginning of the program before `main` function
  - It informs the compiler about the function and its characteristics, so
  - It prevents mistakes in the function call related to
    - Data types
    - Number of parameters

Function declaration (II)

- Format:

```
return_type function_name(parameter list);
```

Where `return_type`, `function_name` and `parameter list` have the same meaning that in the function definition

  - If the function does not receive arguments, it must be explicitly declared as `void`
  - If it does not return anything `return_type` must be `void`

Function declaration (III)

- There may be an indeterminated number of parameters:
  - Indicated by «...» in the parameter list
  - There must be at least one defined parameter before the «...»
- Example: Valid declarations:

```
int maximum();
int maximum(int [], int);
int maximum(int [], ...);
int maximum(int lista[], int numdat);
                  /* The last one is preferably  */
```

Variable types in relation to functions (I)

- **Local** or **automatic variables**:
  - They are declared within the function (optionally with the modifier `auto`)
    - Unknown/unused outside the function.
    - They just exist while function execution, so
    - They don’t keep their value among calls, unless they are explicitly declared as `static`
    - Stored in a temporal memory part, the ***stack***

Variable types in relation to functions (II)

- **Formal parameters**
  - They are the **local variables** that receive the **function arguments** that are send to the function in each call, so their types must be coincident.
  - They are declared in the function definition

Variable types in relation to functions (III)

- **External/global variables**
  - Declared outside all functions, preferibly before `main`
  - They can be accessed/modified from any point of the program and from any function
  - So they are stored in memory during all execution time
  - Must be declared `extern` in each function that uses them
  - Initialized automatically to zero
  - **Disadvantages**:
    - Functions that use them are less portable and generic
    - As they can be modified in any part of the program, they must be used with care to prevent “interferences”
    - They imply a permanent memory occupation and a larger program size.

Functions call (I)

- A **function call** is made writing the name of the function and its arguments.
- Arguments can be passed to the function by two ways:
  - **By value**
    - Arguments are copied in the corresponding formal parameters.
    - Chages made within the function **do not affect the variables used in the call**
  - **By reference**
    - Arguments passed to the functions are memory addresses of the variables (pointers).
    - The function can change the contents of the address and therefore **can change the variable used in the call**.

Functions call (II)

- To **pass an array** to a function, the argument is the address of the first element of the array (**pointer**).
  - The function can change any element of the array
  - The function must know the dimensions of the array.
    - With a 1D array, it must know its limits:
      - The number of elements
      - If it is a string, the null character \0
    - With a multidimensional array:
      - The number of dimensions
      - The total number of elements.

Functions call (III)

- Example: maximum() function with prototype

  `int maximum(int list[], int numdat)`

  - Receives
    - The address of an array of integers `list`
    - The number of elements in the array `numdat`
  - Returns an integer: the maximum of the array `max`
  - After the call `max=maximum(array, ndata);`
    - `ndata` does not change
    - The elements in `array` (`array[0],array[1],...`) may change.
    - `max` will change

Functions call (IV)

- Structures and unions can be passed to a function as any other variable:
  - When passed by **value**, a copy is made.
    - With big and complex structures, memory size and execution time increase.
  - When passed by **reference**:
    - Function call is fast (just an address is passed).
    - Function can change values of variables in the calling function.

Exit from a function (I) - `return`

- `return` statement allows to exit from a function and go back to the point where it was called

```
return expression;
```

  - `expression` represents the value to be returned
    - It must be of the type the function expects

- It can be placed anywhere and more than once.
- Closing bracket «`}`» means as well function ending and return to the calling point
- By default the retun type is `int`.

Exit from a function (II) - `exit`

- `exit()` forces the end of the program in the point where is placed
  - It returns the control to the OS
  - Defined in the file `stdlib.h`

`main()` function arguments (I)

- `main()` function can exchange information with the OS:
  - Receive arguments from command line
  - Return a value
- Prototype

```
int main(int argc, char *argv[]);
```

  - `int` indicates that it returns an integer (default)

`main()` function arguments (II)

- `argc` and `argv[]` are optional parameters to receive arguments:
  - `argc` is an integer indicating the number of arguments, considering the name of the program as the first one
  - `argv` is a pointer to an array of character strings that contains the arguments.
    - Each element of the array points to one argument in the command line: (`argv[0]` to the program name, `argv[1]` to the next argument…)
    - Separator in command line is just an space.

`main()` function arguments (III)

- `main()` receives as many strings as there are character sets separated by spaces in the command line

- Example: If `cp` was a C program, typing

```
cp -f origin_file destiny_file
```

in the `main()` function of the program there will be:

  - `argc=4`
  - `argv[0]="cp"`
  - `argv[1]="-f:"`
  - `argv[2]="origin_file"`
  - `argv[3]="destiny_file"`

Recursive functions (I)

- **Recursion** is the possibility that a function calls itself
  - When this happens:
    - Previous execution remains suspended and its parameters are stored in memory
    - A successive return must take place
  - Usually there is a conditional statement to finish recursion
  - Recursivity levels must be limited to a small number explicity or by the algorithm (risk of infinity loops)
- When programming recursive functions notice that:
  - `auto` and `register` variables are initialized every call
  - `static` variables are just initialized the first call

Recursive functions (II)

- Advantages
  - Sometimes they allow to create clearer and simpler versions of some algorithms
- Disadvantages
  - Usually they they increase both used memory and execution time
  - Difficult to understand

Recursive functions (III)

- Example: Program to show natural numbers up to the one introduced with the keyboard (I)

```
#include <stdio.h>
void present (int num);    /* Function prototype */

main()
{
  int n;
  printf("Introduce a number: ");
  fflush(stdin);
  scanf("%d", &n);
  present(n);        /* Call to the function */

  return 0;
}
```

Recursive functions (IV)

- Example: Program to show natural numbers up to the one introduced with the keyboard (I)

```
void present(int num)         /* Recursive function */
{
  if (num==1) printf ("%d\t", num);
                    /* Si num == 1 print and finish */
  else
  {
     present(num-1);     /* Si num!=1 decrement num
                              and calls to itself */
     printf("%d\t", num);
  }
}                       /* When returning from calls
                              numbers are printed */
```

Complex declarations (I)

- Combination of
  - *Pointer to* operator «`*`»
  - Array brackets «`[]`»
  - *Parenthesis* «`()`» to group operations or for functions

  Give rise to complex declarations difficult to understand

- To interpret correctly the declarations:
  1. Start with the identifier and go right
     - Parenthesis indicates that is a function
     - Brackets indicates that is an array
  2. Go left and check if there is a «`*`» indicating a pointer
  3. Apply fomer rules to each level of parenthesis from inside to outside

Complex declarations (II)

- Examples

```
int (*list)[20];      /* list is a pointer to an
                      array of 20 integers */

char *data[20];       /* data is an array of 20
                       pointers to character */

void (*busc)();      /* busc is a pointer to a
                      function that does not
                      return anything*/
```