Computer Science

Data types in C

Data in C language

- Introduction
- Basic data types and specifiers (or qualifiers, modifiers)
- Integer numbers
- Real numbers
- Size and range
- Other data types
  - Derived
  - User-defined
- Constants
  - Integer constants
  - Real constants
  - Character constants
  - Symbolic constants
- Variable declaration
  - Local variables
  - Global variables
- Variable initialization
- Other data type specifiers
  - Acces specifiers
  - Storage-class specifiers

Introduction to data types in C

- **Data** are the objects that are processed in computer programs
- In C, *variables* and *constants* must be **declared** before use
- Data declaration requires to specify:
  - Data type
  - Specifier (optional)
  - Identifier

```
              specifier datatype identifier;
Example:       unsigned   int       age
```

Basic data types and specifiers (I)

- When programming, the election of the **data types** to use will establish their main features:
  - Memory they occupy
  - Range of values they can store
  - How they are processed
- The required memory and the range for each data type depend on:
  - Compiler
  - Operating system
  - Computer

Basic data types and specifiers (II)

- Reserved words in C for **basic data types** are:

`char`	Character
`int`	Integer
`float`	Real
`double`	Real in double precission
`void`	No data (for functions that return no value)
`enum`	Enumerated type, list of integers/names

- The **specifiers** that can be applied to these basic data types are:
  - `signed`
  - `unsigned`
  - `long`
  - `short`

- Data are obtained combining basic types and specifiers.

Integers (I)

- Type to store integer quantities
  - `char` (`signed char`).
    - Normally occupies a byte (to store one ASCII character)
  - `int` (`signed int`).
    - Normally ocuppies 4 bytes
  - `short` (`signed short int`).
    - Normallly ocuppies 2 bytes
  - `long` (`signed long int`).
    - In 32 bits machines: 4 bytes; in 64 bits: 8 bytes
  - `enum`. *Enumerated type*. Variable that can take as argument a list of simbols

Integers (II)

- Size relation is always: `short ≤ int ≤ long`

- Internal representation of integers
  - Numbers **without sign**: pure binary
  - Numbers **with sign**: 2'complement

- Examples:
  - `int cantidad;`
  - `char letra`
  - `Short age`
  - `Long memoria`
  - `Enum week = {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};`

Reals

- Numerical quantities in scientific notation and with higher range
- Most extended format: IEEE754:
- Types
  - `float`. Real with simple precission
  - `double`. Real with double precission
  - `long double`. Real with double precission long format

Other data types

- **Void**
  - Void indicates a non-defined data type
  - It is used mainly for functions that don't return any value
- **Derived**
  - Complex data types obtained from fundamental ones
  - Arrays, function, pointers, structures and unions
- **User definided**
  - Created by the user with their own name and definition

```
typedef datatype newname;
typedef unsigned long int mytype;
```

Constants (I)

- **Constants** are fixed values that cannot be altered by the program in execution

- They can be:
  - Integer constants
  - Real constants
  - Character constants
  - Symbolic constants

Constants (II)

- **Integer constants** (I)
  - The compiler chooses the smaller data type compatible with the constant.
  - They can be expresssed in
    - *Decimal*: default option
      - The most significant bit cannot be `0`
      - Just numerical values from `0` to `9` are valid
    - *Octal*
      - The most significant bit is always `0`
      - Just numerical values from `0` to `7` are valid
    - *Hexadecimal*:
      - They always start with `0x`
      - Values from `0` to `9` and letters `A`, `B`, `C`, `D`, `E`, `F` (upper and lower case) are valid

Constants (III)

- **Integer constants** (II)
  - They have the following fields:
    - Prefix `0x` for hexadecimals or `0` for octals.
    - Sign (optional for positives)
    - Numerical value
    - Optional suffix to fix the size that the compiler must assign to it:
      - `U` for `unsigned`
      - `L` for `long`
      - `UL` for `unsigned long`
  - Examples: `-23L,   010,   0xF`

Constants (IV)

- **Real constants**
  - By default the compiler always create them `double`
  - They have the following fields:
    - Sign (optional for positives)
    - Integer part before the decimal point «`.`»
    - Fractional part after the point
    - Scientific notation with «`e`» or «`E`»
    - Optional suffix to fix the size that the compiler must assign to it:
      - `F` for `float`
      - `L` for `long double`
  - Examples:

```
35.78      1.25E-12      45F      33L
```

Constants (V)

- **Character constants** (I)
  - *One character constants* are `char` type and are expressed with with single quotation marks: `‘A’`
  - *Back slash* \ constant
    - It allows to represent ASCII character by its number. Use simple quotation marks: `‘\`*`ASCIIcode`*`’`
      - The *code* can be represented
        - In decimal up to 3 digits: `‘\ddd’`
        - In octal with two dígits: `‘\0oo’`
        - In hexadecimal with two dígits : `‘\0xhh’`

Constants (VI)

- **Character constants** (II)
  - Example:

```
‘6’     /* Character 6, ASCII code 0x36 */
‘\12’    /* ASCII code 13 (carriage return) */
‘\0x20’ /* ASCII code 32 (space) */
```

  - *String constants*
    - They are not a data type
    - The define a set of 1-byte characters stored consecutively
    - Represented with double quotation marks:

```
“This is a string constant”
```

    - The compiler stores the string and finish it with the **null character** «`’\0’`» to represent the end of the chain.

Constants (VII)

- **Symbolic constants**
  - They are defined with the directive `#define` :

    ```
    #define CONSTANTNAME Equivalence
    ```

    - `CONSTANTNAME` is the identifier of the symbolic constat (recommended in capitals)
    - *`Equivalence`* are the symbols that `CONSTANTNAME` is going to represent
    - When `CONSTANTNAME` appears in the program the compiler will substitute it with by *`Equivalence`*
  - Example:

    ```
    #define MAXIMUM 100  /* MAXIMUM takes de value 100 */
    #define SENTENCE “press a key”
    ```

Variables declaration (I)

- All variables must be *declared* before used so that the compiler assigns the required memory to them
- A variable *declaration* is a statement

```
      Datatype variablename;
```

- Examples:

```
        char letra;
        int actual, greater, lower;
        float resultado;
```

Variables declaration (II)

- Variables can be ***local***, ***global*** or ***formal parameters***.

- **Local variables** (also named *automatic variables* `-auto`)
  - Are declared within a function
  - Declaration must be at the beginning of the function
  - They are **just valid within the function**
  - They desappear when the function is executed
  - If the function is called many times, local variables are created and destroyed every time
  - They are stored in a special part of the menory, the **stack memory** (LIFO-Last Input First Output)

Variables declaration (III)

- **Global Variables**
  - Declared out of any function
  - Active during all program execution
  - Stored in a special part of the memory assigned by the compiler
  - Can be used by any function without restriction
  - Can be defined in another file (e,g, a *header*). In such a case they must be defined with `extern` specifier in the file where they are used.
  - Compiler initializes them to 0 when defined
  - Must be used with care:
    - They make functions less portable
    - They occupy memory during all program execution
    - They can give rise to many mistakes

Variables declaration(IV)

- **Formal Parameters** (or *parameters*)
  - Are the variables that receive the values that are passed to the function
  - Always local to the function
  - Declared in the same line than the function
  - Example:

```
long int Myfunction(int base, int exponente)
 {
    /* function statements */
  }
```

Variables inicialization

- It is used to assign the variable's first value
  - By default:
    - Global variables are initialized to 0
    - Locals variables just take the value that was in the memory position where that the compiler assign to them (rubbish in general)
  - It can be done in the same declaration with an assignment operator:

```
datatype variablename = initial value;
```

  - Example:

```
unsigneg int age = 25;
```

Other data specifiers (I)

- **Access specifiers**
  - The modify the *way a variable is accessed*
    - `const`. Set a variable as constant, i.e. it can be changed during all program execution.
    - `volatile`. Makes the variable posses special properties related to optimization (just for advanced programmers)

    - Example

```
  unsigned int const year = 2006;
```

Other data specifiers (II)

- **Storage-class specifiers**
  - Used to tell the compiler how the variable must be stored:
    - `extern`. Declares a variable that has been defined in a different file (they already have memory assigned)
    - `static`. (Inside a function) Declares a local variable that keeps its value among calls.
    - `static`. (Outside a function) Declares a global variable to be used just in the file where it is defined (private use)
    - `register`. Tells the compiler taht the variable must be stored in a register (fast access for heavily used variables)
    - `auto`. Declares a variable local to a function (is the default option)