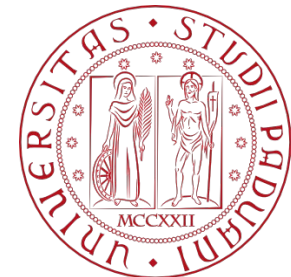


COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

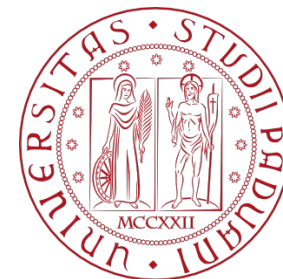
luigi.rizzo@unipd.it

October 2023-January 2024



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Struct, arithmetic expressions
evaluation, functions, bitwise
operators, input and output hints




UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- Structures are **derived data types**, built using elements (members) of other types
 - A structure is defined with the keyword: **struct**
 - Example:

```
struct student {  
    char surname[20];  
    char name[20];  
    unsigned int id;  
    unsigned int birthYear;  
    char gender;  
};
```

Variables declared within the parentheses are the members of the struct



- The type of the members of a struct can be any:
 - int, float, char, ...
 - arrays, other structs, unions, ...
 - Example:

```
struct    point    {  
int x;  
int y;  
};  
struct    rectangle {  
    struct point highLeft;  
    struct point lowRight;  
};
```

- Structure definitions do not reserve any memory space but create a new data type
- Structure type variables are declared like all other "standard" type variables (i.e. int, float, char ...)

```
struct    student std1, std2;
```

```
struct    student {  
...  
} std1, std2;
```

- Two operators are used to access individual members:
 - The structure member operator: `.`
 - The structure pointer operator: `->`
 - Let's see two examples:

```
printf("%s\t%s\n", std1.name, std1.surname);
```

(Assuming that the pointer `stdPtr` was initialized with the `std1` address)

```
printf("%s\t%s\n", stdPtr->name, stdPtr->surname);
```

- It is a good idea to combine the definition of structures with typedef for greater readability.
- Basically, typedef is used to assign symbolic names (aliases) to data types
- Usually struct and typedef are used together, but a classic example of using typedef is also the following:

```
typedef unsigned char Byte;
```

- There is some similarity between typedef and #define (both allow you to define aliases/symbolic names), however:
 - typedef is limited to defining aliases for types
 - #define is “more general” (e.g. we can define aliases for values) and is processed by the pre-processor, while typedef statements are processed by the compiler

- Types can use a different number of bytes depending on the architecture/compiler
 - For example int can use 2 or 4 bytes
- A very useful (and used) operator in C is sizeof
 - It is a unary operator that returns the number of bytes necessary to allocate a type or variable

```
sizeof(int);           //    4
```

```
char    name[20];  
  
sizeof(name);         //   20
```


- The arithmetic of C is not always the same as how we carry out the calculations (we must take this into account).

$$\frac{3}{2} * 2$$

- How does C calculate the expression?

Hypothesis 1

$$\frac{3}{2} * 2 = 3$$



NO because the expressions are not executed in the expected order:

$$(3/2)*2$$

- How does C calculate the expression?

Hypothesis 2

$$\left(\frac{3}{2}\right) * 2 = 1.5 * 2 = 3$$



NO because $3/2$ is an operation between integers, not between reals (the numbers do not have the decimal part, 3.0). The expression below is correct:

$$3.0/2.0*2.0=3$$

So which is the result of $3/2*2$?

Hypothesis 3

$$\left(\frac{3}{2}\right) * 2 = 1 * 2 = 2$$

- In C the operands of an expression must have the same type: what happens if we try to add an integer and a real?

$$3.0 + 2 = ?$$

- C transforms the integer into a real (it is possible to force the opposite transformation, we will see it in the next slide):

$$3.0 + 2.0 = 5.0$$

Explicit type conversions can be forced ("coerced") in any expression, with a unary operator called a cast.

In the construction

(type name) expression

the expression is converted to the named type by the conversion rules above.

The precise meaning of a cast is as if the expression were assigned to a variable of the specified type, which is then used in place of the whole construction.

For example, the library routine `sqrt` expects a double argument, and will produce nonsense if inadvertently handled something else.

So if `n` is an integer, we can use

```
sqrt((double) n)
```

to convert the value of `n` to `double` before passing it to `sqrt`

In C, a function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. They encapsulate a set of instructions that can be executed repeatedly or as needed. With properly designed functions, it is possible to ignore how a job is done; knowing what is done is sufficient. C makes the use of functions easy, convenient and efficient.

Let's write a function of our own.

C has no exponentiation operator, we will illustrate the mechanics of function definition by writing a function `power(m,n)` to raise an integer `m` to a positive integer power `n`.

Here is the function `power` and a main program to exercise it.

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

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

```
/* test power function */
```

```
main()
```

```
{
```

```
    int i;
```

```
    for (i = 0; i < 10; ++i)
```

```
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

```
    return 0;
```

```
}
```

```
/* power: raise base to n-th power; n >= 0 */  
int power(int base, int n)  
{  
    int p = 1;  
    for (int i = 1; i <= n; ++i)  
        p = p * base;  
    return p;  
}
```


A function definition has this form:

```
return-type function-name(parameter declarations, if any)  
{  
  declarations  
  statements  
}
```

A function in C is defined using the *return_type* (the type of value the function returns), a *function_name* (an identifier for the function), and a list of parameters (input values, if any).

The first line of power itself,

```
int power(int base, int n)
```

declares the parameter types and names, and the type of the result that the function returns.

Function definitions can appear in any order, and in one source file or several, although no function can be split between files.

The function body contains a sequence of statements that define the function's behavior. It can include variable declarations, calculations, and control structures like loops and conditionals.

To use a function, you call it by its name and pass any required arguments. The function executes its code and may return a value.

For example, the function `power` is called twice by `main`, in the line

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Each call passes two arguments to `power`, which each time returns an integer to be formatted and printed.

Parameters are variables defined in the function's declaration. Arguments are values passed to the function when it's called. The values of arguments are assigned to the parameters inside the function.

The declaration

```
int power(int base, int n);
```

just before *main* says that *power* is a function that expects two *int* arguments and returns an *int*. This declaration, is called a function prototype.

Parameter names are optional in a function prototype, so for the prototype we could have written

```
int power(int, int);
```

Well-chosen names are good documentation, however, so better use them.

The return statement is used to send a value back to the calling code. The return type in the function declaration specifies the type of value that the function can return.

The integer value that power computes is returned to main by the return statement. Any expression may follow return:

return expression;

A function needs not return a value. There can be a return statement at the end of main too. Since main is a function like any other, it may return a value to its caller, which is in effect the environment in which the program is executed. Typically, a return value of zero implies normal termination; non-zero values signal unusual or erroneous termination conditions.

- C provides a wide range of built-in library functions (e.g., printf, scanf, strlen) that simplify common tasks and can be used in your programs.
- Functions can call themselves, which is known as recursion. Recursive functions are often used to solve problems that can be broken down into smaller, similar subproblems.
- If a function doesn't need to return a value, you can specify *void* as the return type.

Arguments - Call by Value/Reference



In C, all function arguments are passed "by value." This means that the called function is given the values of its arguments in temporary variables rather than the originals. This doesn't affect the original value outside the function.

To modify the original values of variables within a function, you can pass pointers to those variables. This allows you to achieve "pass by reference" behavior. The caller must provide the address of the variable to be set (technically a pointer to the variable), and the called function must declare the parameter to be a pointer and access the variable through it.

In the case of arrays, when the name of an array is used as an argument, the value passed to the function is the location or address of the beginning of the array - there is no copying of array elements.

Functions can be used to encapsulate error-handling code, which helps keep the main program flow cleaner and more focused on the primary task.

Operators in C

	Operator	Type
Unary operator →	+ +, - -	Unary operator
Binary operator {	+, -, *, /, %	Arithmetic operator
	<, <=, >, >=, ==, !=	Relational operator
	&&, , !	Logical operator
	&, , <<, >>, ~, ^	Bitwise operator
	=, +=, -=, *=, /=, %=	Assignment operator
Ternary operator →	?:	Ternary or conditional operator



Bitwise operators in C are used to perform operations at the bit level, allowing you to manipulate individual bits of data. These operators are essential for tasks like working with flags, setting or clearing specific bits, and optimizing memory usage. There are six bitwise operators in C: AND (&), OR (|), XOR (^), NOT (~), left shift (<<), and right shift (>>).

Bitwise AND (&):

The bitwise AND operator (&) performs a bitwise AND operation on each pair of corresponding bits of two integers. It returns a 1 if both bits are 1, otherwise, it returns 0.

1010 & 1101 → 1000

Bitwise OR (**|**):

The bitwise OR operator (`|`) performs a bitwise OR operation on each pair of corresponding bits of two integers. It returns a 1 if at least one of the bits is 1.

1010 | 1101 → 1111

Bitwise XOR (**^**):

The bitwise XOR operator (`^`) performs a bitwise XOR (exclusive OR) operation on each pair of corresponding bits of two integers. It returns a 1 if the bits are different, and 0 if they are the same.

1010 ^ 1101 → 0111

Bitwise NOT (\sim):

The bitwise NOT operator (\sim) inverts the bits of an integer. It changes 0s to 1s and 1s to 0s.

$\sim 1010 \rightarrow 0101$

Left Shift (\ll):

The left shift operator (\ll) shifts the bits of an integer to the left by a specified number of positions filling vacated bits with zero. Bits that are shifted out of the left side are discarded. It effectively multiplies the number by 2 raised to the power of the shift count.

$1010 \ll 2 \rightarrow 101000$

Right Shift (\gg):

The right shift operator (\gg) shifts the bits of an integer to the right by a specified number of positions. It effectively divides the number by 2 raised to the power of the shift count.

- For unsigned integers, the leftmost bits are filled with zeros.
- For signed integers, the leftmost bits are filled with the sign bit (0 for positive values, 1 for negative values) to maintain the sign of the number.

$1010 \gg 2 \rightarrow 10$

Practical Use Cases:

Bitwise operators are commonly used for low-level programming, such as working with hardware registers, data compression, encryption, and image processing.

They are also used to optimize memory usage by packing multiple Boolean flags or options into a single integer, which saves space and speeds up access times.

Important Note:

Be cautious when working with bitwise operators, as incorrect usage can lead to unexpected results or undefined behavior. It's essential to understand how these operators work at the bit level and use them with care.

Input and output are not part of the C language itself.

We shall use the standard library, a set of functions that provide input and output, string handling, storage management, mathematical routines, and a variety of other services for C programs.

The library implements a simple model of text input and output. A text stream consists of a sequence of lines; each line ends with a newline character.

The simplest input mechanism is to read one character at a time from the standard input, normally the keyboard, with **getchar**:

```
int getchar(void)
```

getchar returns the next input character each time it is called, or EOF when it encounters end of file. The symbolic constant EOF is defined in `<stdio.h>`.

In many operating systems, a file may be substituted for the keyboard by using the `<` convention for input redirection: if a program uses getchar, then the command line *program <infile* causes program to read characters from infile instead of standard input.

The input may come also from another program via a pipe mechanism: on many systems, the command line *otherprogram | program* runs the two programs otherprogram and program and pipes the standard output of otherprogram into the standard input for program.

The function

int putchar(int)

is used for output: `putchar(c)` puts the character `c` on the standard output, which is by default the screen. `putchar` returns the character written, or EOF if an error occurs. Again, output can usually be directed to a file with *>filename*: if program uses **putchar**,

program >outfile

will write the standard output to `outfile` instead. If pipes are supported,

program | anotherprogram

puts the standard output of `program` into the standard input of `anotherprogram`.

Each source file that refers to an input/output library function must contain the line

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

before the first reference. When the name is bracketed by < and > a search is made for the header in a standard set of places (for example, on UNIX systems, typically in the directory /usr/include).

An example, considering the program lower, that converts its input to lower case.

```
#include <stdio.h>  
#include <ctype.h>  
main() /* lower: convert input to lower case */  
{  
int c  
while ((c = getchar()) != EOF)  
putchar(tolower(c));  
return 0;  
}
```

The function `tolower` is defined in `<ctype.h>`; it converts an upper case letter to lower case and returns other characters untouched.

``functions'' like `getchar` and `putchar` in `<stdio.h>` and `tolower` in `<ctype.h>` are often macros, thus avoiding the overhead of a function call per character.

The output function **`printf`** translates internal values to characters. The declaration is

```
int printf(char *format, arg1, arg2, ...);
```

`printf` converts, formats, and prints its arguments on the standard output under control of the format. It returns the number of characters printed.

The function **`scanf`** is the input analog of `printf`, providing many of the same conversion facilities in the opposite direction. The declaration is

```
int scanf(char *format, ...);
```

`scanf` reads characters from the standard input, interprets them according to the specification in format, and stores the results through the remaining arguments.