

PROGRAMMAZIONE PROCEDURALE

A.A. 2023/2024



COMMENTS



COMMENTS

- ④ There are two ways to insert a comment in C:
 - ✓ *block comments* begin with `/*` and end with `*/`, and
 - ✓ *line comments* begin with `//` and end with the next new line character.
- ④ You can use the `/*` and `*/` delimiters to begin and end comments within a line, and to enclose comments of several lines.
- ④ You can use `//` to insert comments that fill an entire line, or to write source code in a two-column format, with program code on the left and comments on the right:

```
double pi = 3.1415926536; // Pi is 3,14
```

COMMENTS 2

- ⊗ Inside the quotation marks that delimit a character constant or a string literal, the characters `/*` and `//` do not start a comment. For example, the following statement contains no comments:

```
printf( "Comments in C begin with /* or //.\\n" );
```

- ⊗ The first `*/` will terminate the opening of the multiline comment: not possible to nest block comments
- ⊗ You can insert `/*` and `*/` to comment out part of a program that contains line comments:

`/*` Temporarily removing two lines:

```
double pi = 3.1415926536;    // Pi is 3.14  
area = pi * r * r;         // Calculate the area  
Temporarily removed up to here */
```

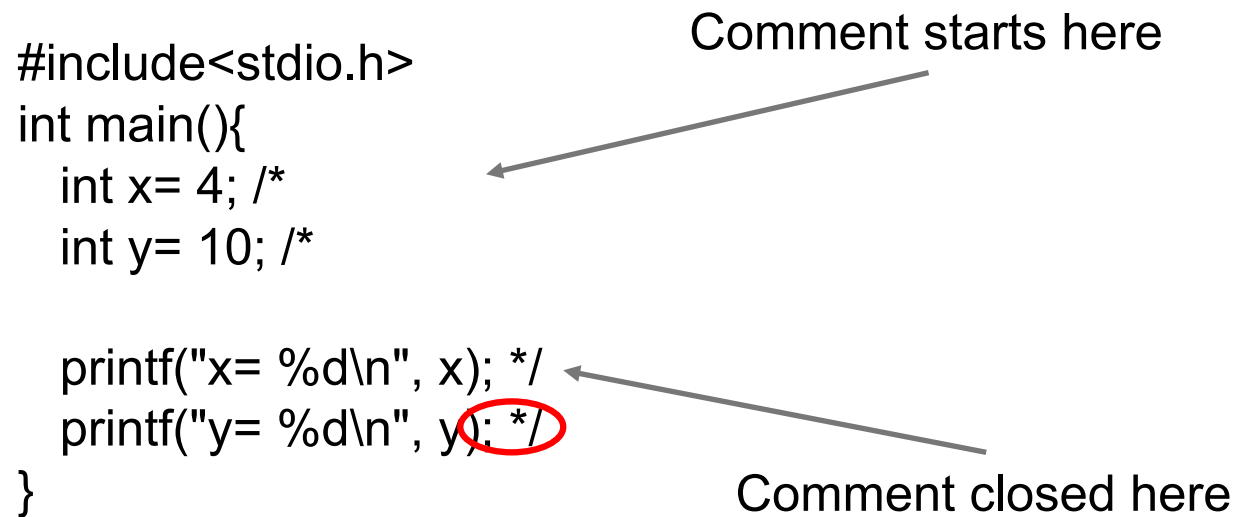
EXAMPLE

```
#include<stdio.h>
int main(){
    int x= 4; /*
    int y= 10; /*

    printf("x= %d\n", x); */
    printf("y= %d\n", y); */
}
```

Comment starts here

Comment closed here



ALL ABOUT COMMUNICATION

- 🌀 We write code to be executed by a computer, but to be *read* by humans. You write code for:
 - ✓ You right now, as you're writing it. The code has to be crystal clear so you don't make implementation mistakes.
 - ✓ You, a few weeks (or months) later as you prepare the software for release.
 - ✓ Other people on your team who have to integrate their work with this code.
 - ✓ The maintenance programmer (which could be you or another programmer) years later, when investigating a bug in an old release.

COMMUNICATION

- 🌀 It can also look pretty, but be unreasonably hard to maintain.

```
/******  
* This is a pretty comment. *  
* Note that there are asterisks on the *  
* righthand side of the box. Wow; it looks neat. *  
* Hope I never have to fix this tiypo. *  
*****/
```

It's cute, but it's not easy to maintain. If you want to change the comment text, you'll have to manually rework the right-hand line of comment markers.

BAD COMMENTING EXAMPLES

Stupid, redundant comments range from the classic example of byte wastage:

```
i=i+1; // increment i
```

```
// loop 5 times and add input values together
```

```
int sum= 0, count= 0, n;
```

```
while (count < 5) {
```

```
    scanf("%d", &n);
```

```
    sum= sum + n;
```

```
    count= count+1;
```

```
}
```


GOOD AND BAD COMMENTING

- ⌚ It's also common to see “old” code that has been surgically removed by commenting it out.
- ⌚ Do not remove code by commenting it out. It confuses the reader and gets in the way.
 - ✓ You can do it when you are still in the debugging phase, to check how different encodings work if you have a run-time error for instance.
- ⌚ Often, a clarification comment is a code smell. It tells you that your code is too complex. You should strive to remove clarification comments and simplify the code instead because, “good code is self-documenting.”

CHARACTER SETS



CHARACTER SETS

- ⊙ Accordingly, C defines two character sets:
 - ✓ the **source character set** is the set of characters that may be used in C source code, and
 - ✓ the **execution character set** is the set of characters that can be interpreted by the running program.

- ⊙ In most implementation they are the same.

BASIC SOURCE AND EXECUTION CHARACTER SETS

The letters of the Latin alphabet

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

The decimal digits

0 1 2 3 4 5 6 7 8 9

The following 29 punctuation marks

! " # % & ' () * + , - . / : ; < = > ? [\] ^ _ { | } ~

The five whitespace characters

Space, horizontal tab, vertical tab, new line, and form feed

PLUS...

WHITESPACE CHARACTERS

☉ Whitespace characters are

- ✓ '\f' form feed;
- ✓ '\n' *newline*;
- ✓ '\t' *horizontal tab*;
- ✓ '\v' *vertical tab*.

IDENTIFIERS



IDENTIFIERS

- ④ The term *identifier* refers to the names of variables, functions, macros, structures and other objects defined in a C program. Identifiers can contain the following characters:
 - ✓ The letters in the basic character set, a–z and A–Z. Identifiers are case-sensitive.
 - ✓ The underscore character, `_`.
 - ✓ The decimal digits 0–9, although the first character of an identifier must not be a digit.

RESERVED KEYWORDS

- ④ The following **44 keywords** are *reserved* in C, each having a specific meaning to the compiler, and must not be used as identifiers:
- ④ auto extern break float case for char goto const if
continue inline default int do long double register else
restrict enum return short signed sizeof static struct
switch typedef union unsigned void volatile while
_Alignas _Alignof _Atomic _Bool _Complex _Generic
_Imaginary _Noreturn _Static_assert _Thread_local.

EXAMPLES

④ The following examples are valid identifiers:

✓ x

✓ dollar

✓ While

✓ error_handler

✓ scale64

④ The following are not valid identifiers:

✓ 1st_rank switch y/n x-ray

MORE ON IDENTIFIERS

- ④ When choosing identifiers in your programs, remember that many identifiers are already used by the C standard library. These include the names of standard library functions, which you cannot use for functions you define or for global variables.
- ④ There is no limit on the length of identifiers. However, most compilers consider only a limited number of characters in identifiers to be significant : 31 or 63.

EXAMPLE

```
#include <stdio.h>
int main()
{
    int firstNumber, secondNumber, sumOfTwoNumbers;

    printf("Enter two integers: ");

    // Two integers entered by user is stored using scanf()
function
    scanf("%d %d", &firstNumber, &secondNumber);

    // sum of two numbers is stored in variable sumOfTwoNumbers
    sumOfTwoNumbers = firstNumber + secondNumber;

    // Displays sum
    printf("%d + %d = %d", firstNumber, secondNumber,
sumOfTwoNumbers);

    return 0;
}
```

SCOPE AND BINDING



IDENTIFIER SCOPE

- ④ The *scope* of an identifier refers to that part of the translation unit in which the identifier is meaningful. Or to put it another way, **the identifier's scope is that part of the program that can "see" that identifier.**
- ④ The type of scope is determined by the location at which you declare the identifier. Different kinds of scope are possible:
 - ✓ *File scope:* If you declare an identifier outside all blocks and parameter lists (functions), then it has file scope. You can then use the identifier **anywhere after** the declaration and up to the end of the translation unit.

IDENTIFIER SCOPE

- ④ *Block scope*: identifiers declared within a block have block scope. You can use such an identifier only **from its declaration to the end of the smallest block containing that declaration**. The smallest containing block is often, but not necessarily, the body of a function definition.

IDENTIFIER

- ⊙ It is not possible to have two variables with the same identifier with the same scope

```
int main() {  
    int a= 5;  
    float a= 6.5;  
}
```

```
int main() {  
    int a= 5;  
    float a= 6.5;  
    int c= a + 1; // which a?  
}
```

EXAMPLE 1

```
int p;  
  
int fun2 (void)  
{  
    int r= 3;  
    int s = r;  
    while( r != s)  
    {  
        r--;  
    }  
    return r;  
}
```

File scope



p, fun2

Block scope



r,s



BINDING

- ④ A *binding* (process) is an association between a **name** and **the thing it names**. *Binding time* is the time at which a binding is created or, more generally, the time at which any implementation decision is made.
- ④ The textual region of the program in which a binding is active is its *scope*.
- ④ In C the scope of a binding is determined statically, that is, at compile time
- ④ Called static or dynamic scoping
 - ✓ In C we have static binding

EXAMPLE 2

```
int main()
{
    {
        int x = 10, y = 20;
        {
            printf("x = %d, y = %d\n", x, y);
            {
                int y = 40;
                x++;
                y++;

                printf("x = %d, y = %d\n", x, y);
            }

            printf("x = %d, y = %d\n", x, y);
        }
    }
    return 0;
}
```

x = 10, y = 20
x = 11, y = 41
x = 11, y = 20

EXAMPLE 3

```
int main()
{
    int x = 1, y = 2, z = 3;
    printf(" x = %d, y = %d, z = %d \n", x, y, z);
    {
        int x = 10;
        float y = 20;
        printf(" x = %d, y = %f, z = %d \n", x, y, z);
        {
            int z = 100;
            printf(" x = %d, y = %f, z = %d \n", x, y, z);
        }
    }
    return 0;
}
```

x = 1, y = 2, z = 3

x = 10, y = 20.000000, z = 3

x = 10, y = 20.000000, z = 100

so

- ④ It is possible to use an identifier again in a new declaration nested within its existing scope. If you do so, then the new declaration must have block scope, and the block must be a subset of the outer scope.
- ④ In such cases, the new declaration of the same identifier *hides* the outer declaration, so that the variable or function declared in the outer block is not *visible* in the inner scope.

EXAMPLE

```
int main()
{
    {
        int x = 10;
    }
    {
        printf("%d", x);
    }
    return 0;
}
```

Error: x is not accessible here

error: 'x' undeclared (first use in this function)

EXERCISE

```
#include <stdio.h>

int f1( void );
int f2( int x, int a );

int a ;

int main()
{
    int a, b, c ;

    a = 7 ;
    b = f1() ;
    c = f2( a, b ) ;
    printf( "%d %d %d\n", a, b, c ) ;
}
```

What is the output?

```
int f1( void )
{
    a = 12 ;
    printf( "%d ", a ) ;
    return( a + 5 ) ;
}

int f2( int x, int a )
{
    printf( "%d ", a ) ;
    return( x * a ) ;
}
```

12 17 7 17 119

EXAMPLE OF DYNAMIC BINDING

- Perl's keyword "my" defines a statically scoped local variable, while the keyword "local" defines dynamically scoped local variable.

Output:
20

```
# A perl code to demonstrate
dynamic scoping
$x = 10;
sub f
{
    return $x;
}
sub g
{
    # Since local is used, x uses
    # dynamic scoping.
    local $x = 20;

    return f();
}
print g()."\n";
```

HOW TO CHOOSE NAMES OF IDENTIFIERS

- ④ A name conveys the identity of an object; it describes the thing, indicates its behaviour and intended use.
- ④ A misnamed variable can be *very* confusing.
- ④ Do not create unnecessarily long variable name
- ④ Choose different names for different scopes in order to avoid confusion

- ④ See the difference
 - ✓ $a = b * c;$
 - ✓ `weekly_pay = hours_worked * pay_rate;`

SU LIBRO

📍 Pagina 46

📍 Sezione 5.13

📍 Pagina 346

