

# OPERATORI BITWISE >

Operator	Effect	Conversions
&	bitwise AND	usual arithmetic conversions
	bitwise OR	usual arithmetic conversions
^	Bitwise XOR	usual arithmetic conversions
<<	left shift	integral promotions
>>	right shift	integral promotions
~	one's complement	integral promotions

```
int x=86, y=50;
printf("%x%y = %d\n", x&y);

0 1 0 1 0 1 1 0
& 0 0 1 1 0 0 1 0
-----
0 0 0 1 0 0 1 0
```

```
int x=86, y=50;
printf("%x|y = %d\n", x|y);

0 1 0 1 0 1 1 0
| 0 0 1 1 0 0 1 0
-----
0 1 1 1 0 1 1 0
```

```
int x=86, y=50;
printf("%x^y = %d\n", x^y);

0 1 0 1 0 1 1 0
^ 0 0 1 1 0 0 1 0
-----
0 1 1 0 0 1 0 0
```

```
int x=86;
printf("x<<2 = %d\n", x<<2);

0 1 0 1 0 1 1 0 << 2
-----
0 1 0 1 0 1 0 0 0
```

```
int x=86;
printf("x >> 1 = %d\n", x>>1);

0 1 0 1 0 1 1 0 >> 1
-----
0 1 0 1 0 1 1
```

# OPERATORI >



Operator	Direction	
() [] -> .	left to right	
! ~ ++ -- - + (cast) * & sizeof	right to left	un aiuto per ricordare la tabella:
* / %	left to right	• gli operatori con precedenza massima sono le parentesi delle funzioni, l'operatore per gli indici e gli operatori di accesso ai membri;
+ -	left to right	• seguono gli operatori unari;
<< >>	left to right	• seguiti a loro volta da quelli aritmetici (operatori moltiplicativi hanno priorità più alta di quelli additivi);
<= > >=	left to right	• tutti i tipi di assegnamento hanno priorità più bassa, tranne che rispetto all'operatore virgola, la cui priorità è minima.
== !=	left to right	
&	left to right	
^	left to right	
	left to right	
&&	left to right	
	left to right	
:	right to left	
= += and all combinations	right to left	
,	left to right	

## associatività

3  
la somma di un float e un int richiede di convertire l'intero in float per eseguire l'operazione

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

main() {
    int i, j;
    float f;
    i = 5; j = 2;
    f = 3.0;

    f = f + j / i;
    printf("value of f is %f\n", f);
    exit(EXIT_SUCCESS);
}
```

poiché l'operatore della divisione aveva tipi int da entrambe le parti, l'operazione aritmetica eseguita è una divisione intera, con risultato zero

2  
poiché quello è il tipo corretto per l'assegnamento, non ci sono ulteriori conversioni

- come si fa a riconoscere un operatore binario
  1. ++ e -- sono sempre operatori **unari**
  2. l'operatore immediatamente a destra di un operando è un operatore **binario**
  3. tutti gli operatori alla sinistra di un operando sono **unari**
- gli operatori **unari** hanno **alta precedenza**; quindi l'espressione

**a + b++ + c**

- ha due operatori **unari** applicati a **b**. poiché tutti gli operatori **unari** associano da destra a sinistra (**R L**), benché il segno '-' venga prima dell'operatore di incremento quando si legge, la parentesizzazione corretta è:

**a + -(b++) + c**

# CLASSI DI MEMORIZZAZIONE >

- **Auto:** Allocate e deallocate quando la funzione viene eseguita
- **Extern:** Variabili visibili ovunque.

# Vediamo prima dichiarate fuori poi tramite keyword:

## extern

```
#include <stdio.h>

int a = 1, b = 2, c = 3;
int f(void);

int main(void) {
    printf("%d\n", f());
    printf("%d%d%d\n", a, b, c);
    return 0;
}

int f(void) {
    extern int a;
    int b, c;

    a = b = c = 4;
    return (a + b + c);
}
```

variabili globali rispetto a tutte le funzioni dichiarate dopo; continuano a esistere all'uscita dal blocco o dalla funzione

essendo dichiarate al di fuori di una funzione hanno classe di memorizzazione **extern**, anche se la parola chiave **extern** non è specificata

il meccanismo delle variabili esterne costituisce un modo per passare le informazioni. tuttavia per aumentare la modularità del codice e per ridurre gli effetti collaterali, è preferibile utilizzare il **passaggio dei parametri**

Marco Botta - Laboratorio di Sistemi Operativi, corso di Informatica

- **Register**: Per variabili da salvare nei registri

## register

- classe di memorizzazione che ha come obiettivo l'aumento della **velocità di esecuzione**
- segnala al compilatore che la variabile corrispondente dovrebbe essere memorizzata in **registri di memoria ad alta velocità**  
qualora il compilatore non possa allocare un registro fisico, viene utilizzata la classe **automatica** (il compilatore dispone solo di una parte dei registri, che possono invece essere utilizzati dal sistema)
- quando la velocità è importante, il programmatore può scegliere **poche** variabili alle quali viene fatto più frequentemente accesso (per esempio, variabili di ciclo e parametri delle funzioni)

```
register int i;
for(i = 0; i < LIMIT; ++i)
    ...
}
```

## register

- all'uscita dal blocco, il registro viene liberato
- **register int i;** equivale a **register i;**
- la variabile **register** è di norma dichiarata nel punto più vicino possibile al punto in cui viene utilizzata, per consentire la massima disponibilità di **registri fisici**, utilizzati solo quando necessario

- **Static**: Conserva il val. vecchio finito un ciclo

## classe static

- variabili **static** servono per permettere a una variabile locale di mantenere il valore precedente al rientro in un blocco

```
void f(void) {
    static int count = 0;
    ++count;
    ...
}
```

alla prima chiamata della funzione count viene inizializzata a zero;

alle chiamate successive non viene più inizializzata, ma mantiene il valore che aveva alla precedente chiamata di funzione

- **Static Extern**: user. globali private nel file

## variabili static extern

- questo tipo di classe di memorizzazione fornisce meccanismo di '**privatezza**' (insieme di restrizioni sulla visibilità di variabili o funzioni che sarebbero altrimenti accessibili) fondamentale per la **modularità** dei programmi
- le variabili **statiche esterne** sono variabili **esterne** con visibilità ristretta: sono **accessibili dal resto del file** in cui sono dichiarate
- non sono disponibili alle funzioni precedentemente definite all'interno del file o definite all'interno di file differenti, anche se tali funzioni utilizzano la parola chiave **extern** relativamente alla classe di memorizzazione

## esempio

```
void f(void) {
    ...
    // v non disponibile
}

static int v;

void g(void) {
    ...
    // v disponibile
}
```

- l'idea è cioè di disporre di una variabile **globale** per una **famiglia di funzioni**, e al contempo **privata** per il file

# ARGOMENTI RIGA DI COMANDO

argomenti dalla riga di comando

```
int main(int argc, char *argv[]){
    int i;
    printf("argc = %d\n", argc);
    for(i=0; i<argc; ++i)
        printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}

$ ./lavagna primo secondo terzo quarto quinto
argc = 6
argv[0] = ./lavagna
argv[1] = primo
argv[2] = secondo
argv[3] = terzo
argv[4] = quarto
argv[5] = quinto
```

*argc* contiene il numero dei parametri sulla riga di comando  
*argv* è un array di puntatori a *char*: un array di stringhe, ciascuna delle quali è una parola sulla riga di comando

# PUNTOATORI A FUNZIONI

pointers to functions

```
int func(int a, float b);
```

- their **declaration** is easy: write the declaration as it would be for the function.
- and simply **put brackets around the name and a \* in front of it**: that declares the pointer. because of precedence, if you don't parenthesize the name, you declare a function returning a pointer:

```
/* function returning pointer to int */
int *func(int a, float b);
/* pointer to function returning int */
int (*func)(int a, float b);
```

Marco Botta - Laboratorio di Sistemi Operativi, corso B - turno T3

62

usage

- once you've got the pointer, you can assign the address of the right sort of function just by using its name;
- like an array, a function name is turned into an address when it's used in an expression. you can call the function using one of two forms:

```
(*func) (1,2);
func(1,2);
```

# usage example

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

void func(int);

int main(void){
    void (*fp)(int);
    fp = func;
    (*fp)(1);
    fp(2);
    return 0;
}
void func(int arg){
    printf("%d\n", arg);
}
```

# PRE- PROCESSORE

preprocessore

- le righe che iniziano con il simbolo **#** sono dette **direttive al preprocessore**
- una direttiva al preprocessore ha effetto a partire dal punto in cui si trova
  - fino alla fine del file stesso, o
  - fino al raggiungimento di un'altra direttiva che ne neghi l'effetto

preprocessore

- le direttive al preprocessore possono contenere **espressioni complesse**, come costrutti **if-then-else**
- usando le direttive possiamo scrivere meta-programmi
- queste istruzioni sono **interpretate** e rimosse dal preprocessore **prima che il codice arrivi al compilatore**

alterazioni del linguaggio

```
#include <stdio.h>
#define PROGRAM int
#define BEGIN main() {
#define END }
#define ESEMPIO_DI_USO_DELLE_MACRO \
    "n funziona?? \n"
#define WriteLn printf
PROGRAM
BEGIN
    WriteLn(ESEMPIO_DI_USO_DELLE_MACRO);
END;
#include <stdio.h>
int
main() {
    printf("n funziona?? \n");
};
```

le direttive costituiscono uno strumento molto potente per alterare il linguaggio

Marco Botta - Laboratorio di Sistemi Operativi, corso B - turno T3

85

## #define

```
#define identifier token_stringopt
#define SECS_PER_DAY (60*60*24)
```

- il preprocessore sostituisce ogni occorrenza di **identifier** con **token\_string<sub>opt</sub>**, eccetto le occorrenze all'interno di stringhe tra virgolette
- il preprocessore sostituisce con (60\*60\*24) ogni occorrenza della costante **SECS\_PER\_DAY** per il resto del file
- l'utilizzo di **#define** può migliorare la **chiarezza** e la **portabilità** dei programmi: le costanti simboliche migliorano la documentazione con **nomi mnemonici** e **significativi**
- minor numero di interventi per manutenzione e modifiche del codice

## esempi

```
#define SQ(x) ((x) * (x))
int i = 3;
```

```
printf("SQ(%d) = %d", i, SQ(i));
```

[rimiazamento] ((3) \* (3))

\$ SQ(3) = 9

```
#define SQ(x) ((x) * (x))
int i = 3;
```

```
printf("SQ(%d) = %d", i, SQ(i));
```

((3) \* (3))

\$ SQ(3) = 9

```
printf("SQ(7+d) = %d", i, SQ(7+i));
```

((7+i) \* (7+i))

\$ SQ(7+3) = 100

## esempi (di errori)

```
#define SQ(x) x * x
```

```
printf("SQ(%d) = %d", i, SQ(i));
```

3 \* 3

\$ SQ(3) = 9

```
printf("SQ(7+d) = %d", i, SQ(7+i));
```

... 7+i \* 7+i è ben diverso da (7+i) \* (7+i) ??????

\$ SQ(7+3) = 31

## esempio

```
char *time,
    *date,
    *file;
int line,stdc;

```

```
time = __TIME__;
date = __DATE__;
file = __FILE__;
line = __LINE__;
stdc = __STDC__;
```

```
printf("__TIME__: %s\n", time);
printf("__DATE__ = %s\n", date);
printf("__FILE__ = %s\n", file);
printf("__LINE__ = %d\n", line);
printf("__STDC__ = %d\n", stdc);
```

```
$ ./prova
__TIME__ = 02:55:57
__DATE__ = Nov 1 2022
__FILE__ = prova.c
__LINE__ = 53
__STDC__ = 1
```

101

## macro con parametri

```
#define identifier(identifier, ..., identifier)
token_stringopt
```

- non ci possono essere spazi fra il primo identificatore e l'apertura della tonda
- nell'elenco dei parametri possono esserci zero, uno o più identificatori

```
#define SQ(x) ((x) * (x))
```

l'identificatore **x** è un parametro che viene sostituito nel testo successivo

NB: la sostituzione è un rimpiazzamento di stringhe, e avviene senza valutare la correttezza sintattica

Marco Botta - Laboratorio di Sistemi Operativi, corso B - turno T3

93

## esempi

```
#define SQ(x) ((x) * (x))
int i = 3;
```

```
printf("SQ(%d) = %d", i, SQ(i));
```

((3) \* (3))

\$ SQ(3) = 9

```
printf("SQ(7+d) = %d", i, SQ(7+i));
```

((7+i) \* (7+i))

\$ SQ(7+3) = 100

## macro predefinite

### macro

### valore

**LINE** A decimal constant representing the current line number.

**FILE** A string representing the current name of the source code file.

**DATE** A string representing the current date when compiling began for the current source file. It is in the format "mmm dd yyyy", the same as what is generated by the asctime function.

**TIME** A string literal representing the current time when compiling began for the current source file. It is in the format "hh:mm:ss", the same as what is generated by the asctime function.

**STDC** The decimal constant 1. Used to indicate if this is a standard C compiler.

## output del preprocessore

- invece di semplificare le cose, un uso eccessivo delle macro **complica il debug** dei programmi
- istruzione **gcc -E nome\_file.c**
- dalla **documentazione di gcc**:

### -E option

Stop after the preprocessing stage; do not run the compiler proper. The **output** is in the form of **preprocessed source code**, which is sent to the standard output or to a file named with the **-o** option



101

## COMPILAZIONE CONDIZIONALE ➤

### compilazione condizionale

```
#if      constant_integral_expression
#endif   identifier
#ifndef  identifier
#elif   constant_integral_expression
#else   -
#endif   identifier
```

- l'espressione costante intera utilizzata nelle direttive al preprocessore non può contenere l'operatore **sizeof** o un cast

### esempio

- delle strutture di controllo a disposizione del preprocessore...

```
#define DEBUG 0

#ifndef DEBUG
printf("DEBUG definito\n");
#endif DEBUG
printf("DEBUG diverso da zero\n");
#else
printf("DEBUG zero\n");
#endif
#else
printf("DEBUG non definito\n");
#endif
```

Marco Botta - Laboratorio di Sistemi Operativi, corso B - turno T3

105

# PROGRAMMAZIONE MODULARE

header file  
 con la direttiva `#include`  
 richiediamo al preprocessore di  
 rimpiazzare il token  
`#include "mio_header_file.h"`  
 con il contenuto del file  
`mio_header_file.h`

```
#include "mio_header_file.h"
int main() {
    int i = 3;
    int j = 5;
    printf("i+j = %d\n", somma(i,j) );
    printf("i*j = %d\n", moltiplica(i,j) );
}
```

```
// questo file è mio_header_file.h
// i miei prototipi sono memorizzati qui
//
int moltiplica(int primo, int secondo);
int somma(int primo, int secondo);
// inoltre possiamo aggiungere anche
// l'implementazione delle funzioni
// online...
int moltiplica(int primo, int secondo) {
    return (primo*secondo);
}
int somma(int primo, int secondo) {
    return (primo+secondo);}
```

`mio_header_file.h`



più moduli...

```
#include "somma.h"
int somma(int primo, int secondo) {
    return (primo+secondo);
}
```

`somma.c`

```
#include "prodotto.h"
int moltiplica(int primo, int secondo) {
    return (primo*secondo);
}
```

`prodotto.c`

```
int somma(int primo, int secondo);
```

`somma.h`

```
int moltiplica(int primo, int secondo);
```

`prodotto.h`

```
#include "somma.h"
#include "prodotto.h"

int main() {
    int i = 3;
    int j = 5;
    printf("i+j = %d\n", somma(i,j));
    printf("i*j = %d\n", moltiplica(i,j));
}
```

`113`

gli header files possono contenere:

- prototipi di funzioni
- definizioni di tipi (strutture, unioni, tipi enumerativi, typedef)
- macro `#define`
- istruzioni `#pragma` per il compilatore
- variabili globali
  - crea una variabile globale in ogni modulo che include il file header a meno che la variabile sia dichiarata `extern`
- implementazione di funzioni inline
  - le chiamate di funzioni inline sono direttamente rimpiazzate dal corpo delle funzioni stesse

Marco Botta - Laboratorio di Sistemi Operativi, corso B - turno T3

112

per evitare inclusioni multiple

```
#ifndef __PLUTO_H__
#define __PLUTO_H__

#include "pippo.h"

type pluto(arg1,arg2);
#endif
```

`pluto.h`

```
#ifndef __PIPPO_H__
#define __PIPPO_H__

#include "pluto.h"

type pippo(arg1,arg2);
#endif
```

`pippo.h`

possiamo utilizzare `#ifndef` e `#define` per fare sì che il contenuto sia incluso una sola volta, quando è raggiunta la direttiva `#include`.

solo se `__PLUTO_H__` e `__PIPPO_H__` non sono ancora definiti sono inserite le righe fra `#ifndef` e `#endif`  
 questo meccanismo impedisce ulteriori inclusioni

## make

- il **Makefile** elenca un insieme di target, le regole per la compilazione e l'istruzione da eseguire per compilare a partire dai sorgenti

**riga di dipendenza**, che indica da quali file oggetto dipende il file **target**; inizia dalla prima colonna della riga

```
nome_target: file1.o file2.o fileN.o
    gcc -o nome_target file1.o file2.o fileN.o
file1.o: file1.c file1.h
    gcc -c file1.c
...
fileN.o: fileN.c fileN.h
    gcc -c fileN.c
```

Marco Botta - Laboratorio di Sistemi Operativi, corso B - turno T3

**riga d'azione** o di comando, che indica come il programma deve essere compilato nel caso sia stato modificato almeno uno dei file .o
 

- deve iniziare con una tabulazione
- dopo una riga di dipendenza è possibile specificare più di una riga d'azione

## l'utility make

- la make-utility è uno strumento che può essere utilizzato per **automatizzare** il processo di compilazione
- in generale è **più flessibile** degli ambienti integrati (IDE, integrated development environments) come MS Visual .NET, Xcode, NetBeans, o Borland C++
- si basa sull'utilizzo di un file (**makefile**) che **descrive le dipendenze** presenti nel progetto
- è quindi possibile utilizzare il comando **make** che si avvale della **marcatura temporale** dei files e **ricompila i target file** che sono più vecchi dei sorgenti

## make

```
nome_target: file1.o file2.o fileN.o
    gcc -o nome_target file1.o file2.o fileN.o
...
clean:                                ulteriore target, che è possibile invocare
    rm -f *.*                            per rimuovere tutti i file oggetto
```

- nell'invocare make da linea di comando è possibile specificare quale target compilare

\$ make **file1.o** //crea solo file1.o

- se viene invocato senza opzione dalla linea di comando, utilizza il **target di default**, il primo specificato nel makefile

\$ make //crea file1.o file2.o fileN.o e nome\_target

- per ricompilare cancellando tutti i file oggetto

\$ make **clean**

# VARIABILI D'AmbIENTE

## Environment List

- For example, the environment variable SHELL is set to be the **pathname** of the shell program itself.
- Many programs interpret this variable as the name of the shell that should be executed if the program needs to execute a shell.

```
$ echo $SHELL
/bin/bash
```

## Environment List

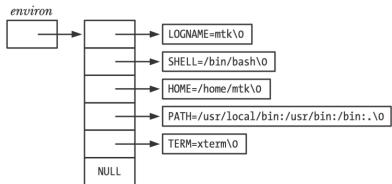
```
$ mia_var=valore_var // creazione variabile
$ export mia_var      // aggiunta della
                        // variabile nell'
                        // environment del
                        // processo
```

blocco  
environment list

- In most shells, a value can be added to the environment using the **export** command
- The **printenv** command displays the **current environment list**.

## Accessing environ from C programs

- Within a C program, the environment list can be accessed using the global variable **char \*\*environ**. Like **argv**, **environ** points to a NULL-terminated list of pointers to null-terminated strings.



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

extern char ** environ;
// di qui in poi è possibile utilizzare
// environ
int main(int argc, char** argv) {

    char** cursor;
    for(cursor=environ; *cursor!= NULL; ++cursor) {
        puts(*cursor);
    }

    return (EXIT_SUCCESS);
}
```