Corso di Laurea in Ingegneria e Scienze Informatiche
a.a. 2017/2018

Appunti dalle Lezioni
del corso di
Sistemi Operativi

Integrazione su Linguaggio C: Modello di Compilazione Moduli e Variabili

Vittorio Ghini
vittorio.ghini@unibo.it

Queste parte di lezioni serve ad integrare la conoscenza del linguaggio C appresa nel corso di Programmazione.
Si approfondiscono gli argomenti relativi alla costruzione di programmi composti da più moduli implementati in linguaggio ANSI C, evidenziando:

- le possibilità offerte dagli strumenti di compilazione,
- la necessità di proteggere le variabili ed i modi per proteggerle.

Caratteristiche del linguaggio C

- utilizzo frequente di chiamate a **funzioni**.
- **debole controllo sui tipi di dato**. A differenza del Pascal, il C permette di operare con assegnamenti e confronti su dati di tipo diverso, in qualche caso solo mediante un type cast (conversione di tipo) esplicito.
- **linguaggio strutturato**. Il C prevede costrutti per il controllo di flusso, quali raggruppamenti di istruzioni, blocchi decisionali (if-else), selezione di alternative (switch), cicli con condizione di terminazione posta all'inizio (while, for) o posta alla fine (do) e uscita anticipata dal ciclo (break).
- **programmazione a basso livello** facilmente disponibile
- implementazione dei **puntatori** (ampio uso di puntatori per memoria, vettori, strutture e funzioni)
- **portabilità** sulla maggior parte delle architetture.
- disponibilità di **librerie standard**.
- la grande libertà messa a disposizione dai puntatori e dagli array rende facile commettere errori, soprattutto con array e puntatori.

La definizione del linguaggio C originario fu stabilita nel 1978 da Brian Kernighan e Dennis Ritchie, che progettarono il linguaggio per scrivere il sistema operativo Unix. Tale dialetto di C è noto come K&R.

Al fine di rendere il linguaggio piu' accettabile a livello internazionale, nel 1989 venne messo a punto uno standard internazionale chiamato **ANSI C** (American National Standards Institute), noto anche col nome **C89**. Questo è lo standard che useremo in questo corso.

L'anno successivo, con pochissime modifiche, lo stesso standard fu promulgato con il nome di C90 dall'*International Organisation for Standardisation* (ISO).

Successivamente sono stati pubblicati degli standard meno restrittivi per il linguaggio C, tra i quali ISO C99, GNU99 ed altri ancora.

Sviluppo di un Programma in linguaggio C

Lo sviluppo di un programma in C richiede le seguenti fasi:
Scrittura, Compilazione, Esecuzione e Debugging.

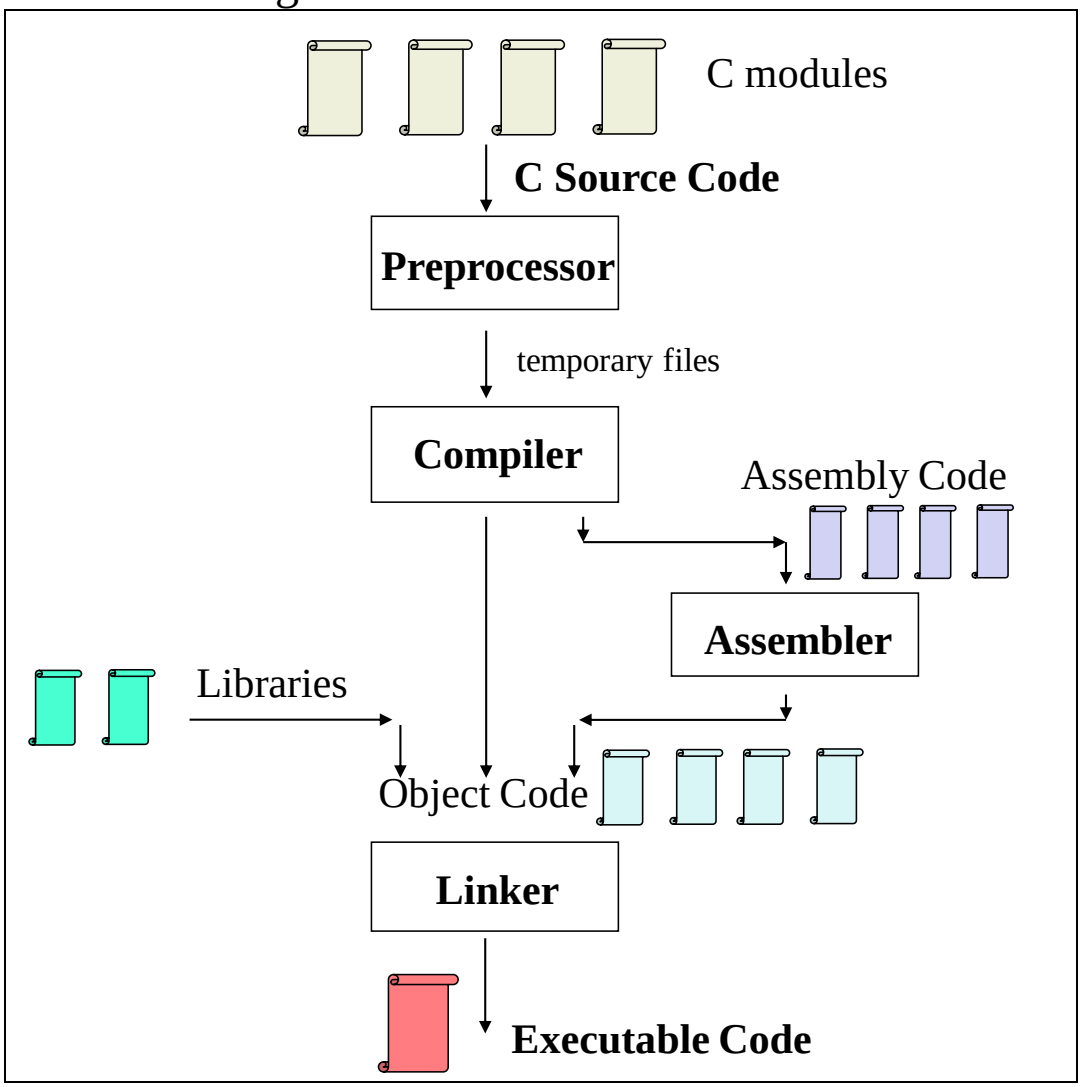
- **Scrittura del Programma in Linguaggio C:**
  - Per creare un file contenente un programma C si usa un qualsiasi text editor quale "edit" "notepad" in pc con sistemi operativi dos-windows, oppure vi, emacs, xedit, nano in host con s.o. unix o linux.
  - Il nome del file deve avere l'estensione "**.c**", ad esempio, prog.c.
  - Il contenuto, ovviamente, deve rispettare la sintassi C.
- **Compilazione:**
  - La compilazione di un programma scritto in C si effettua mediante programmi detti **compilatori** quali Microsoft C Compiler (cl.exe) in sistemi dos-windows, o Gnu C Compiler (gcc) in sistemi Unix.
  - Questi compilatori in realtà svolgono 3 funzioni: **preprocessing**, **compilazione vera e propria** (eventualmente passando da uno step intermedio che consiste nella generazione di file in codice assembly) per generare dei moduli oggetto, **linking** (collegamento) dei vari moduli oggetto e delle eventuali librerie.
  - queste tre funzioni, in ambiente unix sono solitamente eseguite da tre programmi diversi (preprocessor, compiler, linker) che vengono attivati da un programma di coordinamento anch'esso comunemente detto compilatore. In ambienti microsoft spesso preprocessing e compilazione sono eseguiti da uno stesso programma, ed il linking è effettuato da un altro programma chiamato dal compilatore. Esiste inoltre un utility (il make) che permette di eseguire le varie fasi solo per quei files che sono stati modificati più recentemente (in particolare i files modificati dopo l'esecuzione dell'ultima compilazione), limitando in tal modo il lavoro del compilatore.

COMPILATORI A LINEA DI COMANDO		
\ ambiente fase \	DOS-WINDOWS (Microsoft)	UNIX-LINUX (GNU)
make utility	nmake.exe	make
preprocessing	cl.exe	cpp
compilazione	cl.exe	**gcc**
linking	link.exe	ld

- **Esecuzione e debugging**:
  - L'eseguibile ottenuto mediante compilazione e link viene eseguito per verificarne il funzionamento e scoprire eventuali errori.
  - N.B. Un programma sviluppato in C che sembra funzionare dopo la prima compilazione ha sicuramente degli errori nascosti !!!!
  -
- Esistono i cosiddetti "ambienti di sviluppo integrati" (IDE) quali ad es il Microsoft Visual C, che nascondono all'utente questi aspetti della compilazione, mediante un'interfaccia visuale, e che consentono anche di effettuare il debugging in maniera più facile, seguendo passo a passo l'esecuzione del programma.

Il Modello di Compilazione per il C

Gli step essenziali della compilazione per un programma sviluppato in C sono rappresentati nel seguente schema:



- NOTA BENE: Il passaggio attraverso il linguaggio assembly solitamente non è necessario, e viene utilizzato solo a scopo di debugging.

- **Il Preprocessore**

Il preprocessore prende in input un file di codice sorgente C e:

1) processa le cosiddette **direttive al preprocessore**, che sono essenzialmente:
   - **Inclusione di file**
     #include <nomefile> oppure con " " al posto di < >
     prende il contenuto del file nomefile e lo inserisce al posto della direttiva
   - **Definizione di Simbolo**
     #define SIMBOLO DefinizioneDiSimbolo
     definisce un simbolo, e da quel momento, ogni volta che il preprocessore incontra SIMBOLO lo sostituisce con DefinizioneDiSimbolo.
     Esempio: #define LENGTH 100
     #undef SIMBOLO
     elimina la definizione di SIMBOLO, da quella riga in poi non esiste piu'
   - **Definizione di Macro (con o senza argomenti)**
     #define MACRO(X) DefinizioneDiMacro
     definisce una Macro, e da quel momento, ogni volta che il preprocessore incontra MACRO(qualcosa) lo sostituisce con DefinizioneDiMacro, in cui al posto della X sostituisce qualcosa. Ad esempio:
     #define MiniMacro(X, Y) i=(X); func((X)+(Y))
     MiniMacro(a, 10); ---> i=(a); func((a)+ (10)) ;
   - **Compilazione Condizionale su esistenza simbolo**
     #ifdef SIMBOLO
     istruzioni C /* eseguite se prima esiste #define SIMBOLO */
     #else
     istruzioni C /* eseguite se prima NON esiste #define SIMBOLO */
     #endif
     #ifndef SIMBOLO
     #endif
   - **Compilazione Condizionale su condizione costante numerica intera**
     #if Espressione1_COSTANTE_intera
     istruzioni C /* eseguite se l'espressione1 e' vera */
     #elif Espressione2_COSTANTE_intera
     istruzioni C /* eseguite se l'espressione2 e' vera */
     #else
     istruzioni C /* eseguite se sono false sia espressione1 che espressione2 */
     #endif

2) elimina i commenti contenuti nel sorgente.

3) genera così un nuovo file senza commenti e senza più necessità di effettuare sostituzioni, pronto per essere processato dal compilatore.

Un esempio di come lavora il processore:

prima

```
/* file header.h */
extern int    i;
extern double  f;

/* fine header.h */
```

prima

```
/* file prova.c */
#include "header.h"

/* size vettore */
#define SIZE 10

int vettore[SIZE];
```

dopo il passaggio col preprocessore

```
/* file prova.E */
extern int i;
extern double  f;

int vettore[10];
```

Altre direttive al preprocessore

#

il solo carattere pound # non produce effetto.

#error messaggio di errore definito dal programmatore

interrompe la precompilazione come se fosse accaduto un errore e visualizza il messaggio messaggio di errore definito dal programmatore.

Macro Predefinite

Le prime due sono utili per generare messaggi di errore

__LINE__ Valore decimale del numero della linea corrente del sorgente.
__FILE__ Stringa del nome del file in corso di compilazione.

__DATE__ Stringa della data di compilazione (formato Mmm dd yyyy).
__TIME__ Stringa dell'ora di compilazione (formato hh:mm:ss).
__STDC__ Contiene il valore 1 se il compilatore e' conforme allo standard ANSI.
Macro poco significativa, da' 1 anche se uso flag -std=c99

Operatori Utilizzabili dentro le Macro

L'operatore ## permette di concatenare il contenuto di due argomenti della macro.

Ad esempio:

```
#define   Macro( X , Y )     X=X##Y
int val,  val1,  val2;
Macro ( val, 2 );            Viene espansa in:          val=val2;
```

Se un parametro formale e' preceduto dal carattere pound #, il suo valore attuale e' espanso testualmente come stringa. Esempio:

```
#define DEBUG_OUT(expr)      fprintf(stderr, #expr " = %g\n", (float)(expr))
allora ...
DEBUG_OUT(x*y+z);        Viene espanso in:
                           fprintf(stderr, "x*y+z" " = %g\n", (float)(x*y+z))
```

- **Il Compilatore.**

Il compilatore prende in input il codice ottenuto come output dal preprocessore, e crea il codice assembly, cioè un codice che (semplificando) è mappabile 1 a 1 con il codice macchina ma è scritto in una forma umanamente comprensibile.

- **L'Assemblatore (assembler).**

Prende in input il codice assembly, e crea il codice oggetto, cioè del codice macchina (comprensibile per il calcolatore) in cui ancora esistono dei riferimenti non risolti.

- **Il Linker.**

detto anche collegatore, si occupa di prendere in input i vari files di codice oggetto, sia quelli generati dall'assemblatore, o direttamente dal compilatore a partire al codice del programmatore, sia quello delle librerie disponibili, e di generare il file eseguibile, collegando tutti i riferimenti a variabili e funzioni da un file all'altro.
In particolare si occupa di collegare, senza farlo vedere all'utente, una porzione di codice oggetto, reso disponibile come libreria di sistema, che:

- **prepara l'ambiente per il processo quando questo va in esecuzione**
- **passa il controllo alla funzione main** che l'utente deve avere definito.

Questa porzione di codice si occupa ad esempio, ma non solo, di consegnare al programma i parametri passati tramite la riga di comando, e questo senza che l'utente se ne debba occupare.

In genere, è necessario **indicare quali librerie il linker deve utilizzare**, passando al linker stesso alcuni flag nella linea di comando.
Alcune librerie sono utilizzate per default, altre devono essere esplicitamente indicate.

Gli ambienti integrati di sviluppo (IDE) in genere nascondono o almeno facilitano la soluzione delle problematiche relative alla scelta delle librerie da utilizzare.

Struttura di un Programma C

Un programma C ha in linea di principio la seguente forma:

- **Direttive per il preprocessore**
- **Definizione di tipi**
- **Prototipi di funzioni**, con dichiarazione dei tipi delle funzioni e delle variabili passate alle funzioni)
- **Dichiarazione delle Variabili Globali**
- **Dichiarazione Funzioni**, dove ogni dichiarazione di una funzione ha la forma:
  Tipo NomeFunzione(Parametri)
  {
  Dichiarazione Variabili Locali
  Istruzioni C
  }

```
#include <stdio.h>

typedef struct point {
            int   x;    int   y;
            };
int       f1(void);
void    f2(int i, double g);


int   sum;


int main(void)
{
      int j;
      double g=0.0;
      for(j=0;j<2;j++)
            f2(j,g);
      return(2);
}

void f2(int i, double g)
{
      sum = sum + g*i;
}
```

Variabili

Tutte le variabili devono essere dichiarate prima di essere usate.
La **dichiarazione delle variabili** è così fatta:

Tipo ElencoVariabili;

dove Tipo è uno dei tipi di dati ammessi dal C, e ElencoVariabili è composto da uno o più identificatori validi separati da una virgola.
In questo modo ogni identificatore che compare in ElencoVariabili diventa una variabile di tipo Tipo.

esempi:

```
int         i;          /* i è una variabile di tipo int. */
long int    l1, l2;     /* l1 ed l2 sono long int */
float       f,g,x,y;    /* f, g, x, y sono variabile in virgola mobile */
```

Le variabili assumono caratteristiche diverse, in particolare caratteristiche di visibilità (scope) da parte delle funzioni , in dipendenza della posizione in cui avviene la dichiarazione.

A seconda della posizione in cui avviene la dichiarazione, si distinguono tre tipi di variabili:

- Variabili **Locali**.
- Parametri **Formali**.
- Variabili **Globali**.

Variabili Locali

Definiamo **Blocco di Istruzioni** una sequenza di istruzioni C racchiusa tra una parentesi graffa aperta ed una parentesi graffa chiusa.
Il corpo di una funzione (il codice C che implementa una funzione) è un caso particolare di Blocco. Esempi di Blocchi:

Corpo di Funzione

```
int funcA (double f)
{
   int j;    /* corretto */
   printf("corpo di funcA")
   int K;  /* ERRORE */
}
```

Interno di ciclo for oppure blocco if

```
if ( 1 )
{
   int j;
    printf("ciclo for")
}

func(J);    ERRORE
qui J NON e' visibile
```

Ovunque, usando il Trucco aperta-chiusa {}

```
printf("codice C");
 {
   int j;
    printf("ciclo for")
 }
```

Una variabile Locale può essere dichiarata **dentro un qualunque blocco**, ma in questo caso sempre e solo **all'inizio** del blocco, cioè mai dopo che nel blocco sia stata scritta un'istruzione diversa da una dichiarazione), ed in tal caso:

- la variabile verrà detta **Locale al blocco**,
- potrà essere acceduta solo dall'interno del blocco stesso,
- cioè non è visibile fuori dal blocco,
- e avrà un ciclo di vita che inizierà nel momento in cui il controllo entra nel blocco, e terminerà nel momento in cui il controllo esce dal blocco.

Le variabili locali sono caricate sullo stack quando il controllo entra nel blocco considerato, e vengono eliminate quando il controllo esce dal blocco in cui sono state dichiarate.
Quando una variabile è dichiarata nel corpo di una funzione, è locale alla funzione, e assomiglia alle variabili locali del Pascal.

Variabili come Parametri Formali

Sono le variabili che definiscono, nell'implementazione di una funzione, i parametri passati come argomenti alla funzione.
Sono esattamente equivalenti ai parametri formali delle funzioni o procedure del Pascal.
Per default i dati di tipo semplice sono passati per valore, come in Pascal.
Invece i dati di tipo matrice sono passati per puntatore (la modalità **var** del pascal).

C	Pascal

int func(float f , int i)
{
  printf ("param: f=%f i=%d\n,f,i);
}
	func(real : f, i : integer) :integer ;
	begin
	  writeln('param: f=',f, ' i=', i);
	end;

Come per le variabili locali, anche i Parametri Formali

- potrà essere acceduta solo dall'interno della funzione in cui è stata dichiarata,
- cioè non è visibile fuori dalla funzione,
- avrà un ciclo di vita che inizierà nel momento in cui il controllo entra nella funzione, e terminerà nel momento in cui il controllo esce dal blocco.
-

I parametri Formali vengono caricati sullo stack quando il controllo entra nel blocco considerato, e vengono eliminati quando il controllo esce dal blocco in cui sono state dichiarate. Se il Parametro è passato per puntatore, è il puntatore ad essere caricato sullo stack.

Variabili Globali e Specificatore extern

Le variabili Globali sono quelle variabili che sono dichiarate **fuori da tutte le funzioni**, in una posizione qualsiasi del file.
Una tale variabile allora verrà detta **globale**, perchè

- potrà essere acceduta da tutte le funzioni che stanno **nello stesso file** ma sempre **sotto alla dichiarazione della variabile stessa**,
- potrà essere acceduta da tutte le funzioni che stanno **in altri file in cui esiste una dichiarazione extern per la stessa variabile,** ma sempre **sotto alla dichiarazione extern** della variabile stessa,
- e avrà durata pari alla durata in esecuzione del programma.

Per default, una variabile globale NomeVariabile è visibile da tutti i moduli in cui esiste una dichiarazione di variabile extern di NomeVariabile, ovvero una dichiarazione siffatta:

extern tipo NomeVariabile;

che è la solita dichiarazione di variabile preceduta però dalla parola extern.

Una tale dichiarazione dice al compilatore che:

1. nel modulo in cui la dichiarazione extern è presente, la variabile NomeVariabile non esiste,
2. ma esiste in qualche altro modulo,
3. e che il modulo con la dichiarazione extern è autorizzato ad usare la variabile,
4. e quindi il compilatore non si deve preoccupare se non la trova in questo file,
5. perchè la variabile esiste da qualche altra parte.
6. Sarà il Linker a cercare in tutti i moduli fino a trovare il modulo in cui esiste la dichiarazione **senza extern** per la variabile NomeVariabile.

La **variabile NomeVariabile viene fisicamente collocata solo nel modulo in cui compare la dichiarazione senza extern**, (che deve essere uno solo altrimenti il Linker non sa cosa scegliere) e precisamente nel punto in cui compare la dichiarazione. Nei moduli con la dichiarazione extern invece rimane solo un riferimento per il linker.

Protezione dagli Accessi esterni al modulo: **Variabili Globali** e specificatore **Static**

Se vogliamo che una certa variabile globale NomeVariabile, collocata in un certo file, non sia accedibile da nessun altro modulo, dobbiamo modificare la sua dichiarazione in quel modulo, facendola precedere dalla keyword **static** ottenendo una dichiarazione di questo tipo.

static tipo NomeVariabile;

In tal modo, quella variabile potrà ancora essere acceduta dalle funzioni nel suo modulo, ma da nessun altro modulo.

Esempio, Problemi con variabili globali, all'interno dello stesso file in cui le variabili globali sono definite

```
#include <stdio.h>

int K=2;                                /* variabile globale visibile      */
                                        /* da tutte le funzioni            */

int main(void)
{
        int         i=34;

        printf("i = %d \n",  i  );      /* stampa i cioè 34, corretto      */
        int         J=0;            /* ERRORE, dichiarazione dopo istr. */
        printf("K = %d \n", K  );      /* stampa K cioè 2, corretto       */
        printf("g = %f \n", g  );      /* NON "VEDE" g, ERRORE            */
        funzione1();
        exit(0);
}

double      g=13;

void funzione1(void)
{
        printf("g = %f \n", g  );      /* stampa g, cioè 13, corretto     */
        printf("i = %d \n",  i  );      /* NON VEDE i, ERRORE              */
}
```

Esempio, Problemi tipici in programmi con più moduli.

Il nostro programma è costituito da due moduli, var.c e main.c.

- main.c contiene il main del programma, ed alcune funzioni, tra cui la funzione f , che accetta come parametro formale un intero e lo stampa.
- var.c contiene alcune variabili intere, alcune (A)globali, altre (C) globali ma statiche e quindi visibili solo dentro il modulo var.c.
- Non esiste una variabile B da nessuna parte.

```
/* file var.c */
int A=1;
static int C;
```

```
#include <stdio.h>          /* file main.c */
extern int A;
extern int C;
void f(int c){   printf("c=%d\n",c);  } /*stampa intero */
void main(void)
{
   f(A);   /* corretto                                */
   f(B);   /* error C2065: 'B' : undeclared identifier*/
   f(C);   /* error LNK2001:unresolved external symbol _C*/
}
```

Il modulo main.c contiene due errori, perchè:

1) con l'istruzione f(C) main tenta di accedere alla variabile C che non può vedere perchè è protetta dallo specificatore static che la rende visibile solo dentro var.c.

- Il compilatore non si accorge dell'errore perchè main.c ha una dichiarazione extern per C, e il compilatore si fida e fa finta che C esista e sia accedibile in un qualche altro modulo.
- Il linker invece, che deve far tornare i conti, non riesce a rintracciare una variabile C accedibile, e segnala l'errore indicando un "**error LNK2001: unresolved external symbol**" perche non trova C.

2) con l'istruzione f(B) main tenta di accedere alla variabile B che non è definita nel modulo main.c, nemmeno da una dichiarazione extern.

- il compilatore si accorge dell'errore e lo segnala con il messaggio "**error C2065: 'B' : undeclared identifier**".

Protezione dagli Accessi esterni alla funzione per variabili che debbono mantenere un valore tra una chiamata alla funzione e la successiva ad una funzione.

Lo specificatore static, appliccato ad una variable locale ordina al compilatore di collocare la variabile non più nello stack all'atto della chiamata alla funzione, ma in una locazione di memoria permanente (per tutta la durata del programma), come se fosse una varabile globale.
Ma a differenza della variabile globale, la variabile locale static sarà visibile solo all'interno del blocco in cui è stata dichiarata.
L'effetto è che la variabile locale static:

- viene inizializzata una sola volta, la prima volta che la funzione viene chiamata.
- mantiene il valore assunto anche dopo che il controllo è uscito dalla funzione, e fino a che non viene di nuovo chiamata la stessa funzione.

vediamo un esempio di utilizzo, per contare il numero delle volte che una data funzione viene eseguita.

```
#include <stdio.h>          /* file contaf.c */
void f(void)
{
      static int    contatore=0;    /* viene inizializzato solo una volta */
      contatore = contatore + 1;
      printf("contatore =%d\n", contatore);    /*stampa contatore */
}

void main()           /* per vedere cosa succede in f*/
{
      int i;
      for( i=0; i<100; i++ )
            f();
}
```

Un esempio da non seguire:

Uno degli errori più comuni per chi comincia a programmare in C, consiste nell'ostinarsi a voler scrivere codice "stretto" e poco commentato.

E' sempre un errore, perchè:

1) il codice va modificato nel tempo, ed il codice scritto in forma compatta è più difficile da capire, anche per chi l'ha scritto.

2) L'aggiunta di parentesi tonde e spazi per rendere più visibile e comprensibile il codice non diminuisce le prestazioni.

Come curiosità, vediamo un esempio **DA NON SEGUIRE**, di codice C (estensione Kernighan&Ritchie, e non ANSII) **scritto in forma compatta**, funzionante, che stampa a video una filastrocca inglese.

```
#include <stdio.h>

main(t,_,a)char *a;
{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86, 0, a+1 )+a)):1,t<_?main(t+1, _, a ):3,main ( -94, -27+t, a
)&&t == 2 ?_<13 ?main ( 2, _+1, "%s %d %d\n" ):9:16:t<0?t<-72?main(_,
t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+\
,/+#n+,/#;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!/\
+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')(){}#}w')){}{nl]'/+#n';d}rw' i;# ){n\
l]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#\
n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
#'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/")
:t<-50?_==*a ?putchar(a[31]):main(-65,_,a+1):main((*a == '/')+t,_,a\
+1 ):0<t?main ( 2, 2 , "%s"):*a=='/'||main(0,main(-61,*a, "!ek;dc \
i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

Non è noto in quale manicomio sia stato internato, colui che ha scritto questo codice C.

Operatore sizeof()

Il C mette a disposizione un operatore unario, che restituisce la dimensione della variabile o dello specificatore di tipo passato in input.
Serve a conoscere le dimensioni di alcuni tipi di dati, che potrebbero cambiare al variare dell'architettura su cui il programma deve girare, come ad esempio cambiano le dimensioni degli interi int.

L'operatore sizeof prende in input o una variabile o un identificatore di tipo, e restituisce la dimensione in byte del dato. Il dato può anche essere un dato definito dall'utente, non solo un tipo di dato primitivo.

es:

```
int I;
printf("dimensione di I: %d \n", sizeof(I) );  /* stampa 2 in Windows */
                                               /* stampa 4 in Linux */

printf("dimensione del float: %d \n", sizeof(float) );    /* stampa 4 */
```

L'operatore sizeof è molto importante per la portabilità del codice, da un'architettura ad un'altra.

Particolarità dell'operatore unario sizeof è che viene valutato non durante l'esecuzione del programma, ma al momento della compilazione

---

Operatore offsetof()

Il C mette a disposizione un operatore unario, che restituisce **lo scostamento, in byte, dell'inizio di un campo di una struttura rispetto all'inizio della struttura stessa**.
Serve a conoscere dove inizia veramente un campo di una struttura poiché, per questioni di allineamento e di dimensioni dei tipi di dato, tale inizio potrebbero cambiare al variare dell'architettura su cui il programma deve essere eseguito.

L'operatore offsetof prende in input due argomenti:

- il primo argomento è il nome della struttura,
- il secondo argomento è il nome del campo all'interno della struttura.

es:

```
typedef struct struttura {   int a;    char c;  long int L; } STRUTTURA;
printf("offset di L dentro STRUTTURA: %d \n",
                        offsetof (  STRUTTURA , L  )     );
```

L'operatore offsetof è molto importante per la portabilità del codice, da un'architettura ad un'altra.

Particolarità dell'operatore binario offsetof è che viene valutato non durante l'esecuzione del programma, ma al momento della compilazione