Corso di Laurea in Ingegneria e Scienze Informatiche
a.a. 2021/2022

Appunti dalle Lezioni
del corso di Sistemi Operativi

Richiami di 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,
- le diverse versioni del linguaggio.

Storia del linguaggio C

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.

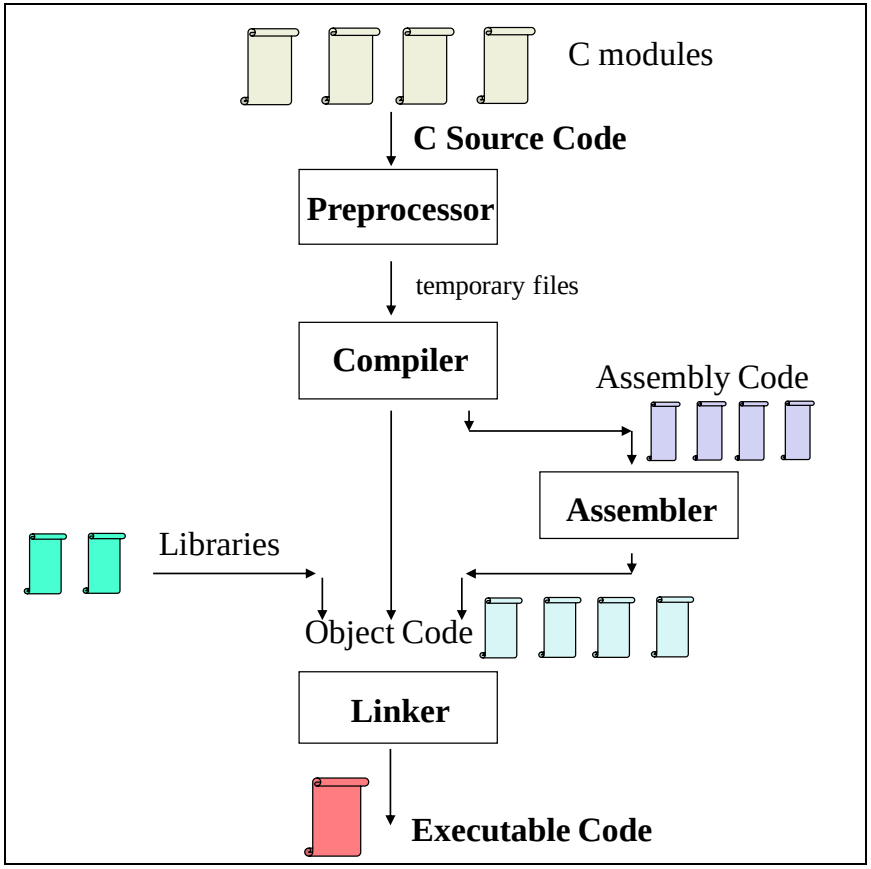
Al fine di rendere il linguaggio piu' accettabile a livello internazionale, nel 1989 venne messo a punto uno standard internazionale chiamato **ANSI C** (American National Standards Institute), noto anche col nome **C89**. Questo è lo standard che useremo in questo corso.

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 in linguaggio C: Il Modello di Compilazione

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



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

- 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.

make utility:	make
preprocessing:	cpp
compilazione:	gcc
linking:	ld

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**,
- 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 un solo modulo 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 tipici in programmi con più moduli.

Il nostro programma seguente 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;
```

```
/* file main.c */
#include <stdio.h>
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);   /* errore   'B' : undeclared identifier*/
   f(C);   /* errore    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 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.

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.

Compilazione e linking di programmi composti da più moduli.

qualche esempio