Esercitazione 01

https://politecnicomilano.webex.com/meet/gianenrico.conti

Gian Enrico Conti

E00 - C Programming

Topics

Teoria:

A. Concetto di modulo e la realizzazione in C
B. Suddivisione del codice tra .c e .h
C. Include guards
D. Translation unit
E. Flusso di compilazione

Esercizi:

1. Esempio programma con un modulo con dati locali statici e metodi di accesso ed un file main che utilizza il modulo. Compilazione manuale
2. Suddivisione del codice tra .c e .h
3. Include guards
4. Altre direttive di preprocessore: #if, #else, #elif, #endif, #undef
5. Esempio di compilazione condizionale per debugging (#ifdef DEBUG … #endif)
6. e definizione della macro sulla linea di comando (-DDEBUG)
7. Tutti i prototipi di main()
8. Passaggio degli argomenti e parsing “manuale” di argomenti (semplice)
9. Esempio di getopt()
10. Accesso all’ambiente mediante envp e mediante getenv()

Intro to Shell - small recap

- la shell NON copy/paste
- Create sempre cartella x vs progetti esempio: **mkdir** EX01
- Entrare in una cartella **cd** es: cd EX01
- Risalire di un livello **cd..**
- File presenti ls -la (list **all** even hidden in **list** mode)
- Editor: nano o vi
- Primo file: nano hello.c (Ctl X per uscire da nano)

Compilatore *gcc*

- **Compilatore in grado di trasformare il codice sorgente C in codice macchina**

```
gcc [options] <filename>
```

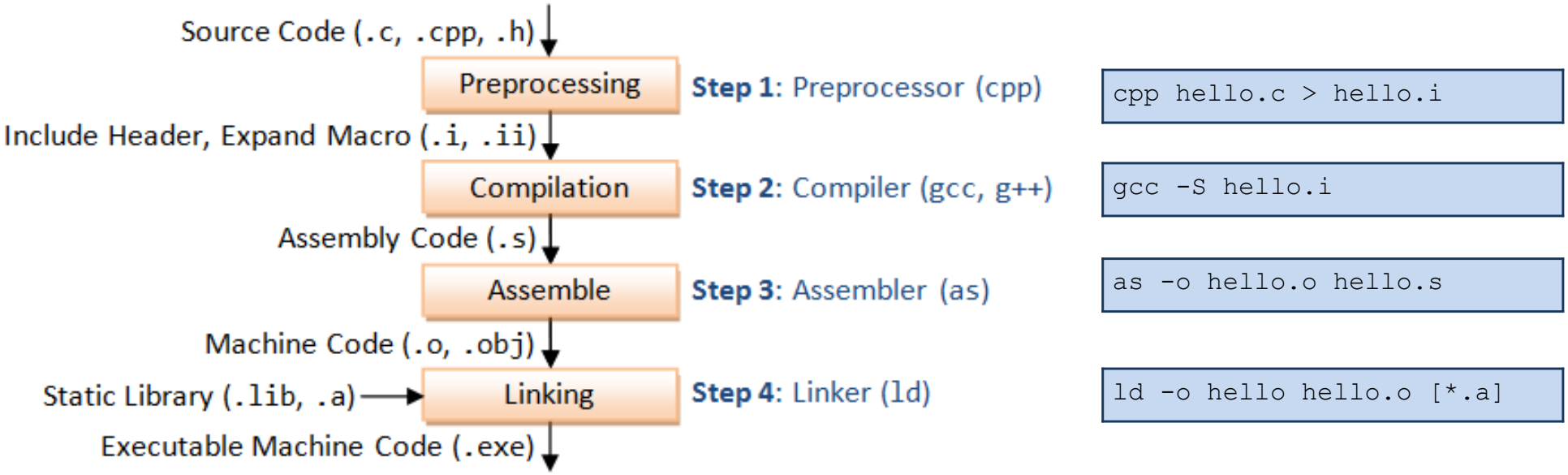
- **Tra le opzioni più importanti troviamo:**
  - `-o outputfile` specifica il nome del file di output
  - `-Wall` attiva tutti i warning
  - `-g` genera simboli aggiuntivi per *gdb*
  - `-v` attiva la modalità *verbose*
  - `-lm` linking libreria *math.h*
  - `-S` genera i file *assembly*
  - `-D name` Definisce *name* come macro, con definizione 1.

Fasi della compilazione *gcc*

Come si passa da un codice sorgente C ad un programma eseguibile?

```
gcc hello.c -o hello
```

- Il compilatore GCC compie questo processo in 4 passi successivi



- L'alternativa è chiedere al GCC di salvare i file intermedi prodotti durante la compilazione

```
gcc -save-temps hello.c -o hello
```

gcc and Shell:

- Primo file: nano hello.c (Ctl X per uscire da nano)
- gcc... gia detto.. **gcc hello.c -o hello** produrra' un eseguibile hello (no extension need0
- Run dell ns eseguibile: **./hello**

Nota: NON riscrivre ogni volta tutto il comando!

Freccia in su e li vedete tutti!

Preprocessore e compilazione condizionale

- **Il preprocessore legge un sorgente C e produce in output un altro sorgente C, dopo avere espanso in linea le macro, incluso i file e valutato le compilazioni condizionali o eseguito altre direttive.**
- **Il preprocessore agisce principalmente sulle keyword**
  - #include
  - #define
  - …
- **Esistono direttive del preprocessore che consentono la compilazione condizionata, vale a dire la compilazione di parte del codice sorgente solo sotto certe condizioni. Questo è possibile attraverso le keyword**
  - #if, #ifdef, #ifndef
  - #else, #elif
  - #endif

Parametri *argv* e *envp*

Dichiarazioni possibili

- `int main(int argc, char *argv[])`
- `int main(int argc, char *argv[], char *envp[])`
- `int main(void)`

Significato dei parametri

- `argc` numero degli argomenti
- `argv` vettore di puntatori a char che contiene la lista dei parametri passati al main
  - `argv[0]` restituisce sempre il nome del programma;
- `envp` restituisce le variabili d'ambiente

Ritorno dal main

- Il main ritorna di default con `return 0,` se non viene specificato altro dal programmatore.
- Tradizionalmente lo standard C prevede solo due possibili stati di uscita dal main:
  - `return 0;` EXIT_SUCCESS - indica che il programma ha avuto successo
  - `return x;` con $x \neq 0$ EXIT_FAILURE - in pratica , il significato dei valori di ritorno diversi da zero può essere gestito dal programmatore

Classi di memorizzazione

- **Definiscono le regole di visibilità delle variabili e delle funzioni quando il programma è diviso su più file.**
- **Variabili e funzioni hanno un attributo che specifica una tra 4 classi di memorizzazione possibili.**

- **Le classi di memorizzazione in C possono essere:**
  - *static* allocazione di memoria e visibilità
  - *extern* per variabili e funzioni

Classe *static* - memoria

- **Una variabile locale statica è una variabile di una funzione che vede associato uno spazio per tutto il tempo che il programma è in esecuzione. Una variabile statica conserva il proprio valore (anche se inaccessibile) tra una chiamata e l'altra della funzione in cui è definita. La parola riservata che specifica tale attributo è** `static`

- **Esempio:** questa funzione stampa il numero di volte che è stata chiamata

```
void f(void)
{
     static int count = 0;
     ...
     printf("%d",++count);
}
```

Classe *static* - visibilità

- **Un secondo uso della parola riservata `static` riguarda la possibilità di limitare la visibilità di variabili globali o funzioni. Una variabile globale o una funzione con attributo di memorizzazione `static` sono visibili esclusivamente nel file d'appartenenza a partire dal punto in cui sono dichiarate.**

Esempio: file1.c

```
void f(void)
{
          ...  /* qui s non è disponibile */
}
static int s; /* variabile globale statica */
void g(void)
{
          ...  /* qui s è disponibile */
}
```

file 2.c

```
extern int s; /* errore: s non è disponibile */
void g(void)
{
          s = 2;  /* errore */
}
```

Classe *extern*

- **L'uso dell'attributo esterno riferito a variabili locali rappresenta il modo che una funzione adotta per accedere a variabili globali definite in altri file. Una variabile locale esterna non è quindi memorizzata nel record di attivazione della funzione. La parola riservata che specifica tale attributo è** `extern`

- **L'attributo** `extern` **utilizzato nella definizione di un prototipo di funzione rappresenta un'indicazione data al compilatore che la definizione completa della funzione si trova in un altro file.**

Puntatori

Operatore di referenziazione “Reference” (&) e dereferenziazione (*)

- Utilizzando l’operatore di referenziazione, ovvero &, seguito immediatamente dal nome di una variabile è possibile estrarne l’indirizzo, ovvero referenziarla.
- Utilizzando l’operatore di “dereferenziazione”, ovvvero *, seguito immediatamente dall’indirizzo di una variabile è possibile estrarne il valore, ovvero dereferenziarla.

Puntatore

- Un puntatore è una speciale variabile, in grado di contenere l’indirizzo di un’altra variabile
- La dichiarazione di un puntatore avviene anteponendo al nome della variabile l’operatore di dereferenziazione*:

```
int pluto=10;   /* variabile */
int *pippo=&pluto;      /* puntatore pippo alla variabile pluto */
```

- Il valore NULL viene utilizzato per inizializzare puntatori di qualunque tipo specificando che essi non puntano a nessuna zona di memoria esistente

Fin qui tutto OK…

Puntatori

Puntatori a funzioni

- Si dichiarano come i prototipi delle funzioni, con l'accortezza di includere il nome della funzione tra parentesi e far precedere allo stesso l'operatore di dereferenziazione

Esempio

```
#include <stdio.h>
void   pippo(int i){
      printf("Valore: %d\n",i);
}

void (*foo) (int);

int main(){
   int i=10,j=5;
   pippo(i);
   foo=pippo;
   foo(j);
   (*foo)(j);
}
```

Quale delle due chiamate a foo è quella corretta? Cosa stampa foo?

→ Entrambe, una volta assegnato al puntatore di funzione l'indirizzo della funzione che si vuole chiamare non fa differenza il fatto di dereferenziare o meno il puntatore a funzione al momento della chiamata. `foo(j)` stampa 5.

Struct, union e typedef

- **Struct** Le *struct* del C sostanzialmente permettono l'aggregazione di più variabili, in modo simile a quella degli array, ma a differenza di questi non ordinata e non omogenea (una struttura può contenere variabili di tipo diverso).

```
struct <name> {
   field1;
   field2;
   …
}
```

- **Union** Il tipo di dato *union* serve per memorizzare (in istanti diversi) oggetti di differenti dimensioni e tipo, con, in comune, il ruolo all'interno del programma

```
union {
   field1;
   field2;
   …
}
```

- **Typedef** Per definire nuovi tipi di dato viene utilizzata la funzione typedef

Nota sulla "potenza" di #define:

```
#define MAX 100;

int main(void)
{
   int a = MAX
}
```

attenzione al ;

Il codice e' legale! Il pre-processore dara'

```
int main(void)
{
   int a = 100;
}
```

Al compilatore!!

Esercizio 1: Esempio programma con un modulo con dati locali statici e metodi di accesso ed un file main che utilizza il modulo. Compilazione manuale (Ex01)

```c
#include <stdio.h>
#include "lib.h"

int main(void)
{
    f();
    f();

    set_global(100);
    printf("%d\n", get_global());
    printf("%d\n", twice);
}
```

```c
// lib.c

int twice = 0;
static int global = 0;

static void inc(void)
{
    global++;
    twice = global * 2;
}

void f(void)
{
    inc();
}

int get_global(void)
{
    return global;
}

void set_global(int newValue)
{
    global = newValue;
}
```

```
gcc  main.c lib.c -o ex1
```

Esercizio 2: Suddivisione del codice tra .c e .h (EX_2

Si noti:

```
static int counter = 0;
```

E gli "accessor" per accedere/manipolare la v.
Senza renderla globale

```
#include <stdio.h>

#include "fatt.h"
#include "counter.h"

int main()
{
    int i;

    printf("%d\n", stato());
    for(i=0;i<5;i++)
        printf( "%d\n", incfatt() );

    for(i=0;i<3;i++)
        printf( "%d\n", decfatt() );

    return 0;
}
```

```
gcc  main.c fatt.c counter.c -o ex2
```

Esercizio 3: Include guards

Esempio:

```c
//  struct.h

typedef struct {
    int x;
    int y;
} punto;
```

```c
//  LibA.h
//  UsingGuards


#include "struct.h"

void DoA(punto P);
```

```c
//  LibB.h
//  UsingGuards


#include "struct.h"

void DoB(punto P);
```

```c
#include <stdio.h>
#include "LibA.h"
#include "LibB.h"


int main(int argc, const char *
argv[]) {
    punto P;
    DoA(P);
    DoB(P);
    return 0;
}
```

Esercizio 3: Include guards

- LibA deve includer struct.h
- LibB deve includer struct.h
- Main deve includere LibA e LibB...ma

```c
#include <stdio.h>
#include "LibA.h"
#include "LibB.h"


int main(int argc, const char *
argv[]) {
    punto P;
    DoA(P);
    DoB(P);
    return 0;
}
```

esegiuamo..

```
gcc main.c LibB.c LibA.c -o main
```

Esercizio 3: Include guards II

```
./struct.h:14:3: error: typedef redefinition with different types
      ('struct punto' vs 'struct punto')
} punto;
  ^
./LibA.h:9:10: note: './struct.h' included multiple times, additional include
      site here
#include "struct.h"
         ^
LibA.c:10:10: note: './struct.h' included multiple times, additional include
      site here
#include "struct.h"
         ^
./struct.h:14:3: note: unguarded header; consider using #ifdef guards or #pragma
      once
} punto;
  ^
```

Ovviamente il pre-processore avra' riportato nell'intermediate DUE VOLTE ..

Esercizio 3: Include guards III

Fix:

```
#ifndef struct_h
#define struct_h

typedef struct {
    int x;
    int y;
} punto;

#endif /* struct_h */
```

In tal modo il preprocessore "entra" solo al primo "giro": al secondo la define esiste.....

Esercizio 4: Altre direttive di preprocessore: #if, #else, #elif, #endif, #undef

Due aspetti:

- predefined
- user defined

Esempi MacOS / Linux (VM)

.....

```
#if defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined(__NT__)
   //define something for Windows (32-bit and 64-bit, this part is common)
   #ifdef _WIN64
      //define something for Windows (64-bit only)
   #else
      //define something for Windows (32-bit only)
   #endif
#elif __APPLE__
```

Esercizio 5: Esempio di compilazione condizionale per debugging (#ifdef … #endif)

```c
#define N 3

//#define ALL

int main(int argc, const char * argv[]) {
    int a = 2*N;
#ifdef ALL
    float f = 1.0;
#endif
    return 0;
}
```

```
gcc -Wall --save-temps main.c -o main
```

E vediamo gli intermedi...

"Scommentiamo" la define..

Esercizio 6: e definizione della macro sulla linea di comando:

Aggiugiamo una #define nella CMD line:

```
VERBOSE
```

sara':

```
gcc -Wall -DVERBOSE main.c -o main
```

Controlliamo l' intermedio e facciamo comunque il run.

Esercizio 7: Tutti i prototipi di main()

TH:

Dichiarazioni possibili

- `int main(int argc, char *argv[])`
- `int main(int argc, char *argv[], char *envp[])`
- `int main(void)`

Esercizio 7:

```
//  EX_7
//
//  Created by ing.conti on 01/03/21.

#include <stdio.h>
int main(int argc, const char * argv[]) {

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

```
 ./hello
found 1 arguments
./hello
```

Provate con:

./hello CIAO MONDO

3 args.. il 1' e' il nome del vs EXE.

Esercizio 8: Passaggio degli argomenti e parsing “manuale” di argomenti (semplice)

Vogliamo leggere due numeri da cmd line e effettuare la somma

Si noti che i parametri (argv) sono STRINGHE -> atti

Esercizio 8: Passaggio degli argomenti e parsing “manuale” di argomenti (semplice)

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

int main( int argc, char* argv[] )
{
    int n, m;
    if( argc != 3 )
    {
        printf("ERROR: %s <num1> <num2>\n",argv[0]);
        return -1;
    }
    // [0] e' il nome delle exe!
    n = atoi( argv[1] );
    m = atoi( argv[2] );

    printf("%d\n", n + m);

    return 0;
}
```

```
./sum 22 33
55
```

Esercizio 9: Esempio di getopt()

From:

https://man7.org/linux/man-pages/man3/getopt.3.html

```
int
       main(int argc, char *argv[])
       {
           int flags, opt;
           int nsecs, tfnd;

           nsecs = 0;
           tfnd = 0;
           flags = 0;
           while ((opt = getopt(argc, argv, "nt:")) != -1) {
               switch (opt) {
.....
```

Esercizio 10: Accesso all’ambiente mediante envp e mediante getenv()

Poremmo usare le v.

$PATH
$HOME

oppure...

- `printenv`

`Da cmd line x vedere le v. di ENV.`

Esercizio 10B: Accesso all’ambiente mediante getenv() a v. "Nostre" da shell:

Da shell:

```
VARNAME="my value"
```

```
char * custom = getenv ("VARNAME");
printf ("Your custom  %s\n", custom);
```

NON esce...

Esercizio 10B II

Da shell:

```
EXPORT VARNAME="my value"
```

```
char * custom = getenv ("VARNAME");
printf ("Your custom  %s\n", custom);
```