Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2022/23

A. Apicella

Compilare un programma C da linea di comando

Se lo sai fare, NON hai bisogno di un IDE (DevC, CodeBlocks, ecc.)
Ricorda che:

> *An IDE, or Integrated Development

Environment, will turn you stupid. They are the worst tools if you want to be a good programmer because they hide what's going on from you, and your job is to know what's going on. They are useful if you're trying to get something done and the platform is designed around a particular IDE, but for learning to code C (and many other languages) they are pointless." (Zed Shaw, "Learn C the hard way")*

Installazione delle componenti necessari per la compilazione

necessari permessi di *sudo*

1. su distribuzioni basate su gestore di pacchetti *Advanced Package Tool* (e.g., Ubuntu, Mint) digitare:

   > `sudo apt install build-essential`

   > oppure

   > `sudo apt-get install build-essential`

2. su distribuzioni basate su gestore di pacchetti *Yum* (e.g., Red Hat) digitare

   > `su -c "yum groupinstall development-tools"`

GCC, the GNU Compiler Collection

> *The GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Ada, Go, and D, as well as libraries for these languages (libstdc++,...). GCC was originally written as the compiler for the GNU operating system. The GNU system was developed to be 100% free software, free in the sense that it respects the user's freedom.* (From: https://gcc.gnu.org/)

Editor

Utilizzate l'editor che preferite (tra questi):

1. nano
2. pico
3. emacs
4. vi/vim

Scrivete il codice del vostro programma, salvatelo (ad esempio `primo.c`) ed uscite dall'editor.

è importante l'estensione!!!

```
// primo programma
    #include <stdio.h>
    # define CONST 42
    int main()
    {
        printf("ciao\nLa risposta è %d", CONST);
    }
```

- Date il comando `gcc primo.c`
- Verrà generato il programma eseguibile dal nome `a.out`
- Per eseguirlo, digitare `./a.out`

GCC: cosa succede

> *When you invoke GCC, it normally does preprocessing, compilation, assembly and linking.* (From: https://man7.org/linux/man-pages/man1/gcc.1.html)

- La compilazione di un programma in *C* si compone di 4 fasi:
  1. Preprocessing
  2. Compiling
  3. Assembly
  4. Linking
- Ognuna di queste fasi prevede la creazione di un file temporaneo che può essere salvato utilizzando il parametro `--save-temps`.

- Il comando `gcc primo.c --save-temps` genererà:

1. `primo.i`, output della fase di Preprocessing
2. `primo.s`, output della fase di Compiling
3. `primo.o`, output della fase di assembly
4. `a.out`, output della fase di Linking (programma eseguibile)

Preprocessing stage

In questa fase:

1. eventuali commenti vengono rimossi
2. le macro (e.g., `#define`) vengono sostituite all'interno del file sorgente
3. i file inclusi attraverso direttiva `#include` vengono inseriti
4. vengono inserite delle informazioni aggiuntive sulle posizioni precise del codice all'interno dei rispettivi file inclusi (*linemarkers*, https://gcc.gnu.org/onlinedocs/cpp/Preprocessor-Output.html)

`cat primo.i`:

```
⋮
extern void funlockfile (FILE *__stream) __attribute__
((__nothrow__ , __leaf__));
# 858 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);

# 873 "/usr/include/stdio.h" 3 4
# 3 "primo.c" 2
# 4 "primo.c"
int main()
{
    printf("ciao\nLa risposta è %d", 42);
}
```

`gcc -E -P primo.c` produce l'output della fase di preprocessing senza linemarkers

```
⋮

extern int ftrylockfile (FILE *__stream) __attribute__
((__nothrow__ , __leaf__)) ;
extern void funlockfile (FILE *__stream) __attribute__
((__nothrow__ , __leaf__));
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);

int main()
{
    printf("ciao\nLa risposta è %d", 42);
}
```

Compiling stage

In questa fase viene generato il codice `assembly` del programma per la macchina target:

`cat primo.s`

```
    .file   "primo.c"
    .text
    .section    .rodata
.LC0:
    .string "ciao\nLa risposta \303\250 %d"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $42, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc

⋮
```

Assembly stage

In questa fase:

- il codice assembly viene convertito in linguaggio macchina;
- Solo le istruzioni presenti nel file vengono convertite, eventuali chiamate a funzioni (e.g., `printf(...)`) non vengono risolte;
- Il file risultante è anche detto **file oggetto** (NB: un file oggetto **NON** ha nulla a che vedere con la programmazione ad oggetti);
- Essendo in codice macchina, l'output non è leggibile utilizzando i classici tool di testo:

`cat primo.o`

```
ÿÿUHÿÿ*Hÿ=ÿÿÿ]ÿciao
La risposta è %dGCC: (Ubuntu 9.3.0-17ubuntu1~20.04)
9.3.0GNUÿzRx
```

```
\
%E@C
@@
%$primo.cmain_GLOBAL_OFFSET_TABLE_printf@@@@@@@@@@

@@@@@@@@
.symtab.strtab.shstrtab.rela.text.data.bss.rodata.comment.note.GI
stack.note.gnu.property.rela.eh_frame @%h0

&ee1e90|+B@R@j@e@@
               8


    8+@t
```

- Il file oggetto, avente estensione `.o`, può essere generato anche attraverso il flag `-c`: `gcc -c primo.c` genera il file oggetto `primo.o`, eseguendo quindi tutte le fasi della compilazione tranne la fase di linking (e quindi **non** generando il file eseguibile).

```
xxd -b primo.o
```

```
00000000: 01111111 01000101 01001100 01000110 00000010
00000001  | .ELF..
00000006: 00000001 00000000 00000000 00000000 00000000
00000000  | ......
0000000c: 00000000 00000000 00000000 00000000 00000011
00000000  | ......
00000012: 00111110 00000000 00000001 00000000 00000000
00000000  | >.....
00000018: 01100000 00010000 00000000 00000000 00000000
00000000  | `.....
0000001e: 00000000 00000000 01000000 00000000 00000000
00000000  | ..@...
00000024: 00000000 00000000 00000000 00000000 01111000
00111001  | ....x9
0000002a: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
00000030: 00000000 00000000 00000000 00000000 01000000
00000000  | ....@.
00000036: 00111000 00000000 00001101 00000000 01000000
00000000  | 8...@.
0000003c: 00011111 00000000 00011110 00000000 00000110
00000000  | ......
00000042: 00000000 00000000 00000100 00000000 00000000
00000000  | ......
00000048: 01000000 00000000 00000000 00000000 00000000
00000000  | @.....
0000004e: 00000000 00000000 01000000 00000000 00000000
00000000  | ..@...
00000054: 00000000 00000000 00000000 00000000 01000000
00000000  | ....@.
0000005a: 00000000 00000000 00000000 00000000 00000000
```



```
00000000  | ......
00000060: 11011000 00000010 00000000 00000000 00000000
00000000  | ......
00000066: 00000000 00000000 11011000 00000010 00000000
00000000  | ......
0000006c: 00000000 00000000 00000000 00000000 00001000
00000000  | ......
00000072: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
00000078: 00000011 00000000 00000000 00000000 00000100
00000000  | ......
0000007e: 00000000 00000000 00011000 00000011 00000000
00000000  | ......
00000084: 00000000 00000000 00000000 00000000 00011000
00000011  | ......
0000008a: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
00000090: 00011000 00000011 00000000 00000000 00000000
00000000  | ......
00000096: 00000000 00000000 00011100 00000000 00000000
00000000  | ......
0000009c: 00000000 00000000 00000000 00000000 00011100
00000000  | ......
000000a2: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
000000a8: 00000001 00000000 00000000 00000000 00000000
00000000  | ......
000000ae: 00000000 00000000 00000001 00000000 00000000
00000000  | ......
000000b4: 00000100 00000000 00000000 00000000 00000000
00000000  | ......
000000ba: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
000000c0: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
000000c6: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
000000cc: 00000000 00000000 00000000 00000000 00000000
00000110  | ......
000000d2: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
000000d8: 00000000 00000110 00000000 00000000 00000000
00000000  | ......
000000de: 00000000 00000000 00000000 00010000 00000000
00000000  | ......
000000e4: 00000000 00000000 00000000 00000000 00000001
00000000  | ......
000000ea: 00000000 00000000 00000101 00000000 00000000
00000000  | ......
000000f0: 00000000 00010000 00000000 00000000 00000000
00000000  | ......
000000f6: 00000000 00000000 00000000 00010000 00000000
00000000  | ......
000000fc: 00000000 00000000 00000000 00000000 00000000
00010000  | ......
00000102: 00000000 00000000 00000000 00000000 00000000
```

```
00000000  | ......
00000108: 11110101 00000001 00000000 00000000 00000000
00000000  | ......
0000010e: 00000000 00000000 11110101 00000001 00000000
00000000  | ......
00000114: 00000000 00000000 00000000 00000000 00000000
00010000  | ......
0000011a: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
00000120: 00000001 00000000 00000000 00000000 00000100
00000000  | ......
00000126: 00000000 00000000 00000000 00100000 00000000
00000000  | ... ..
0000012c: 00000000 00000000 00000000 00000000 00000000
00100000  | .....
00000132: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
00000138: 00000000 00100000 00000000 00000000 00000000
00000000  | . ....
0000013e: 00000000 00000000 01101000 00000001 00000000
00000000  | ..h...
00000144: 00000000 00000000 00000000 00000000 01101000
00000001  | ....h.
0000014a: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
00000150: 00000000 00010000 00000000 00000000 00000000
00000000  | ......
00000156: 00000000 00000000 00000001 00000000 00000000
00000000  | ......
0000015c: 00000110 00000000 00000000 00000000 10111000
00101101  | .....-
00000162: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
00000168: 10111000 00111101 00000000 00000000 00000000
00000000  | .=....
0000016e: 00000000 00000000 10111000 00111101 00000000
00000000  | ...=..
00000174: 00000000 00000000 00000000 00000000 01011000
00000010  | ....X.
0000017a: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
00000180: 01100000 00000010 00000000 00000000 00000000
00000000  | `.....
00000186: 00000000 00000000 00000000 00010000 00000000
00000000  | ......
0000018c: 00000000 00000000 00000000 00000000 00000010
00000000  | ......
00000192: 00000000 00000000 00000110 00000000 00000000
00000000  | ......
00000198: 11001000 00101101 00000000 00000000 00000000
00000000  | .-....
0000019e: 00000000 00000000 11001000 00111101 00000000
00000000  | ...=..
000001a4: 00000000 00000000 00000000 00000000 11001000
00111101  | .....=
000001aa: 00000000 00000000 00000000 00000000 00000000
```



```
00000000  | ......
000001b0: 11110000 00000001 00000000 00000000 00000000
00000000  | ......
000001b6: 00000000 00000000 11110000 00000001 00000000
00000000  | ......
000001bc: 00000000 00000000 00000000 00000000 00001000
00000000  | ......
000001c2: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
000001c8: 00000100 00000000 00000000 00000000 00000100
00000000  | ......
000001ce: 00000000 00000000 00111000 00000011 00000000
00000000  | ..8...
000001d4: 00000000 00000000 00000000 00000000 00111000
00000011  | ....8.
000001da: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
000001e0: 00111000 00000011 00000000 00000000 00000000
00000000  | 8.....
000001e6: 00000000 00000000 00100000 00000000 00000000
00000000  | .. ...
000001ec: 00000000 00000000 00000000 00000000 00100000
00000000  | .... .
000001f2: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
000001f8: 00001000 00000000 00000000 00000000 00000000
00000000  | ......
000001fe: 00000000 00000000 00000100 00000000 00000000
00000000  | ......
00000204: 00000100 00000000 00000000 00000000 01011000
00000011  | ....X.
0000020a: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
00000210: 01011000 00000011 00000000 00000000 00000000
00000000  | X.....
00000216: 00000000 00000000 01011000 00000011 00000000
00000000  | ..X...
0000021c: 00000000 00000000 00000000 00000000 01000100
00000000  | ....D.
00000222: 00000000 00000000 00000000 00000000 00000000
00000000  | ......
00000228: 01000100 00000000 00000000 00000000 00000000
00000000  | D.....
0000022e: 00000000 00000000 00000100 00000000 00000000
00000000  | ......
00000234: 00000000 00000000 00000000 00000000 01010011
11100101  | ....S.
0000023a: 01110100 01100100 00000100 00000000 00000000
00000000  | td....
00000240: 00111000 00000011 00000000 00000000 00000000
00000000  | 8.....
00000246: 00000000 00000000 00111000 00000011 00000000
00000000  | ..8...
0000024c: 00000000 00000000 00000000 00000000 00111000
00000011  | ....8.
.
```

.
.

Linking stage

In questa fase:

1. vengono risolte le chiamate a funzione
2. viene finalizzato il programma così da renderlo pronto per l'eventuale esecuzione (generazione file eseguibile)

Da notare che l'output finale di questa fase produce il file `a.out`. Come fare per cambiargli nome? Due soluzioni:

1. banalmente, si rinomina con `mv a.out primo`
2. si usa il flag `-o` $\rightarrow$ `gcc primo.c -o primo`, oppure `gcc primo.c --save-temps -o primo`

Da notare che i sistemi Linux/Unix non necessitano che i file abbiano estensione, quindi non è necessario che un file abbia suffisso `.exe` per essere eseguibile, a differenza dei sistemi Windows

Esecuzione

`./a.out`

```
ciao
La risposta è 42
```

Errori

In programmazione gli errori possono dividersi in un due principali macrocategorie:

1. errori di *sintassi*: per quanto fastidiosi, sono gli errori più "facili" da trovare. In presenza di questi errori (e.g., la mancanza di un `;` o di una `}`) il compilatore si rifiuta di compilare il programma, riportando messaggi di errore che possano aiutare nell'individuazione del/degli errore/i
2. errori *logici*: l'avvenuta compilazione non implica che il programma sia esente da errori di altra natura, in quanto il programma non fa quello che dovrebbe fare durante l'esecuzione.

Consiglio: non sottovalutare i *warning* dati dal compilatore

In [21]:
```c
// programma potenza.c (errato)
#include <stdio.h>

float power(float base, int esponente)
{
    float risultato = 1;
    for(int i; i < esponente; i = i+1)
    {
        risultato = risultato * base;
    }
    return risultato;
}

int main()
{
    float b   = 4;
    int   e   = 3;
    float pot = power(b,e);
    printf("il valore di %.1f^%d è %.1f\n", b, e, pot);
    return 0;
}
```

```
il valore di 4.0^3 è 1.0
```

Per trovare il problema posso:

- fare ispezione visiva del codice sorgente (quindi prima della compilazione), effettuando modifiche nel codice che mi aiutino ad individuare l'errore (e.g., inserimento di `printf(...)`)
- ispezionare il codice eseguibile durante l'esecuzione attraverso programmi di supporto (e.g., GDB)

Il debugger GDB

GDB (GNU project DeBugger) è un programma che permette di vedere cosa sta succedendo durante l'esecuzione di un programma. GDB permette di:

- Far iniziare il programma da ispezionare speficando ogni cosa che potrebbe condizionare il suo comportamento
- Far fermare il programma al verificarsi di specifiche condizioni
- Esaminare cosa è successo quando il programma si è fermato
- Cambiare alcune cose nel programma durante l'esecuzione

Prima di eseguire il GDB, conviene ricompilare il programma con il flag `-g`

```
gcc -g potenza.c -o potenza
```

il flag `-g` inserire all'interno del file eseguibile la *Tabella dei simboli*. Tale Tabella contiene informazioni utili a GDB per aiutarci nel debugging (e.g., a quali locazioni di memoria corrispondono i nomi di variabili e funzioni del codice sorgente originario).

Senza la Symbol table è comunque possibile effettuare il debug attraverso GDB, ma in maniera molto più difficile.

Avvio di GDB

Per avviare GDB si dà il comando `gdb nome_file_eseguibile`

`gdb potenza`

```
        For help, type "help".
        Type "apropos word" to search for commands related
  to "word"...
        Reading symbols from potenza...
        (gdb)
```

principali comandi GDB:

- `list` : mostra il codice sorgente (non disponibile senza Symbol Table)
- `run` : esegue il programma fino al primo break-point
- `print` : stampa il contenuto di una variabile in un dato istante
- `where` o `backtrace` : mostra il punto di esecuzione
- `break` : imposta un break-point
- `next` : avanza alla successiva linea di codice (senza entrare in una eventuale funzione)
- `step` : come next, ma entra anche nelle funzioni
- `display` : stampa ad ogni passo
- `watch` : imposta un watch-point
- `continue` : continua l'esecuzione
- `quit` : termina GDB

GDB: esempio di esecuzione

list

`list` visualizza il codice sorgente 10 righe alla volta

```
(gdb) list
1// programma potenza.c (errato)
2#include <stdio.h>
3
4float power(float base, int esponente)
5{
6    float risultato = 1;
7    for(int i; i < esponente; i = i+1)
8    {
9        risultato = risultato * base;
10   }
(gdb)
```

`list 1,100` visualizza il codice sorgente a partire dalla riga 1 alla 100 (se esiste)

```
(gdb) list 1,100
1   // programma potenza.c (errato)
2   #include <stdio.h>
3
4   float power(float base, int esponente)
```



```
5   {
6       float risultato = 1;
7       for(int i; i < esponente; i = i+1)
8       {
9           risultato = risultato * base;
10      }
11      return risultato;
12  }
13
14  int main()
15  {
16      float b   = 4;
17      int   e   = 3;
18      float pot = power(b,e);
19      printf("il valore di %.1f^%d è %.1f\n", b, e,
pot);
20      return 0;
21  }
22
(gdb)
```

break

permette di fissare un break-point ad una data riga (e.g., `break 18`) oppure ad ogni invocazione di una data funzione (e.g., `break power`). Durante l'esecuzione del programma, questo si fermerà alla riga impostata

```
(gdb) break 18
Breakpoint 1 at 0x11aa: file potenza.c, line 18.
```

run

il programma viene eseguito fino al primo break-point

```
(gdb) run
Starting program:
/home/andrea/Scrivania/MONNEZZA/potenza

Breakpoint 1, main () at potenza.c:18
18      float pot = power(b,e);
```

where

Per sincerarsi ulteriormente di dove ci troviamo possiamo usare il comando `where`

```
(gdb) where
#0  main () at potenza.c:18
```

print

Possiamo visualizzare il valore che assumono le variabili in quell'istante

```
(gdb) print b
$1 = 4

(gdb) print e
$2 = 3
```

step

Il programma può essere eseguito, da questo punto in poi, un'istruzione alla volta attraverso il comando `step` (se interessa esaminare ogni funzione istruzione per istruzioni) o `next` (se non interessa entrare nelle funzioni, ma solo ciò che succede all'uscita)

```
(gdb) step
power (base=3.0611365e-41, esponente=1431654973) at
potenza.c:5
5   {
```

il comando ci dice che sta per essere invocata la funzione la funzione `power(...)` (notare i parametri formali...). Andando ancora avanti...

```
(gdb) step
6       float risultato = 1;
(gdb) step
7       for(int i; i < esponente; i = i+1)
(gdb) step
11      return risultato;
```

Ci accorgiamo che il corpo del ciclo for non è eseguito. Insospettiti, esaminando il codice ci accorgiamo che la variabile `i` non è stata inizializzata. Bug individato.

continue

Digitiamo `continue` per far continuare (e terminare) il programma.

```
(gdb) continue
Continuing.
il valore di 4.0^3 è 1.0
[Inferior 1 (process 71212) exited normally]
(gdb)
```

Nel caso ci fossero stati altri breakpoints, il programma si sarebbe fermato ad ognuno di questi