Introduzione alla Programmazione in C

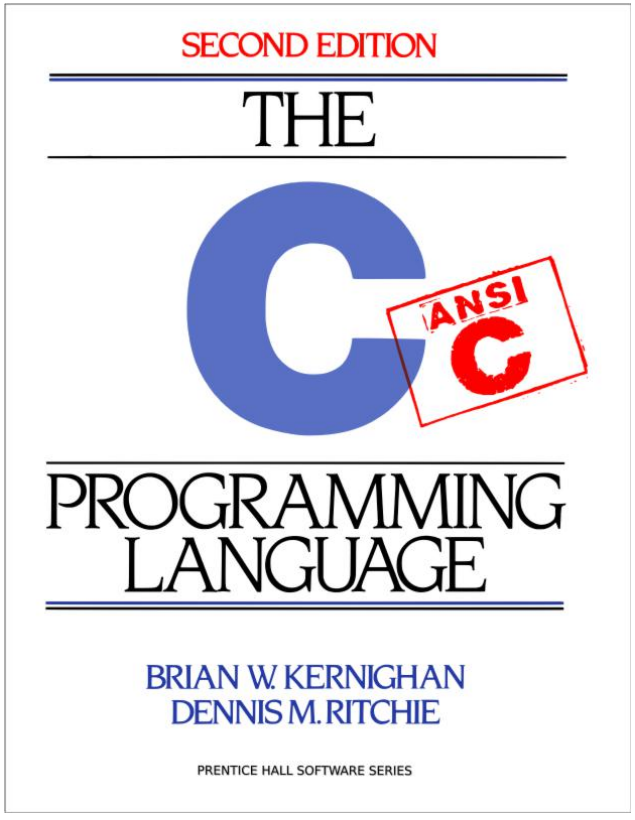
Lezione 2

(Fondamenti di Programmazione e)
Laboratorio

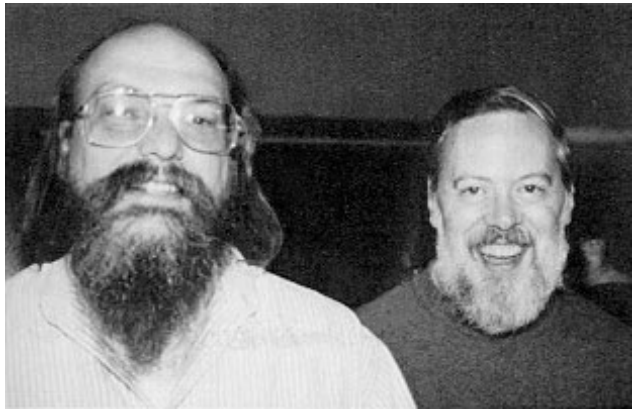
A.A. 2018/2019

Libri di Riferimento del Corso

Dennis M. Ritchie, Brian W. Kernighan, *Il linguaggio C. Principi di programmazione e manuale di riferimento* Pearson Italia



Le origini del C



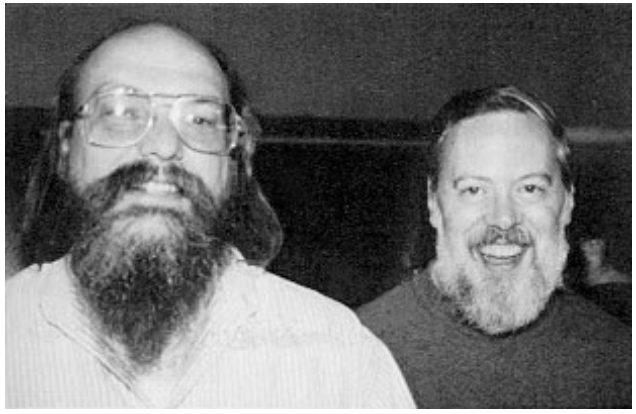
Sviluppato dai Bell Labs nella prima parte degli anni 1970 per la programmazione del sistema operativo UNIX

- Dennis Ritchie (C deus-ex-machina)
- Ken Thompson (UNIX)
- Brian Kernighan (Manuale di riferimento - Hello World)

Si chiama C perché rappresenta un evoluzione del linguaggio B

- Eredita la sintassi di B
- Introduce un sistema di tipi articolato e la possibilità di manipolare i bit in memoria
- Linguaggio standardizzato (ANSI-C) e multi-piattaforma

Le origini del C



Sviluppato dai Bell Labs nella prima parte degli anni 1970 per la programmazione del sistema operativo UNIX

- Dennis Ritchie (C deus-ex-machina)
- Ken Thompson (UNIX)
- Brian Kernighan (Manuale di riferimento - Hello World)

Si chiama C perché rappresenta un evoluzione del linguaggio B

- Eredita la sintassi di B
- Introduce un sistema di tipi articolato e la possibilità di manipolare i bit in memoria
- Linguaggio standardizzato (ANSI-C) e multi-piattaforma

Motivazioni

Perché usare il C?

- Visione a basso livello delle risorse
  - Memoria e dispositivi
  - Possibilità di incapsulare codice Assembler
- Capacità di manipolare i singoli bit
- Uso efficiente delle risorse
  - Ridotto uso memoria
  - Compilazione efficiente

Perché non usare il C?

- Scarso livello di astrazione
- Portabilità (compile once-run everywhere)

Motivazioni

Perché usare il C?

- Visione a basso livello delle risorse
  - Memoria e dispositivi
  - Possibilità di incapsulare codice Assembler
- Capacità di manipolare i singoli bit
- Uso efficiente delle risorse
  - Ridotto uso memoria
  - Compilazione efficiente

Perché non usare il C?

- Scarso livello di astrazione
- Portabilità (compile once-run everywhere)

Caratteristiche fondamentali

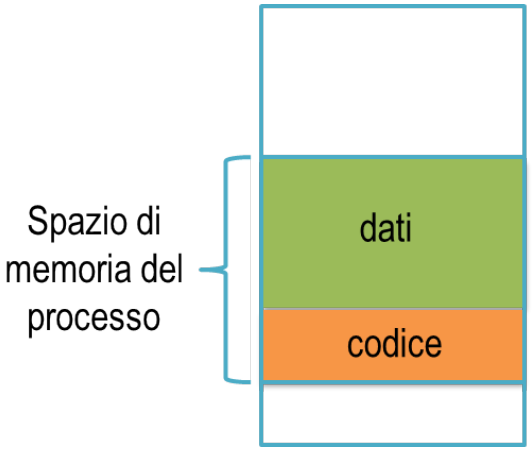
- Paradigma imperativo
  - Assegnamento: `variabile = espressione;`
  - Tipi di dato primitivi e definiti dall'utente
  - Costrutti condizionali: `IF...THEN...ELSE`
  - Costrutti iterativi: `FOR`, `WHILE`, ...
- Strutturazione a blocchi
  - Un programma C è una collezione di funzioni (NON è un linguaggio ad oggetti!)
  - Le funzioni possono essere distribuite su più files
  - La funzione `main` definisce l'inizio dell'esecuzione di un programma

Caratteristiche fondamentali

- Paradigma imperativo
  - Assegnamento: `variabile = espressione;`
  - Tipi di dato primitivi e definiti dall'utente
  - Costrutti condizionali: `IF...THEN...ELSE`
  - Costrutti iterativi: `FOR`, `WHILE`, ...
- Strutturazione a blocchi
  - Un programma C è una collezione di funzioni (NON è un linguaggio ad oggetti!)
  - Le funzioni possono essere distribuite su più files
  - La funzione `main` definisce l'inizio dell'esecuzione di un programma

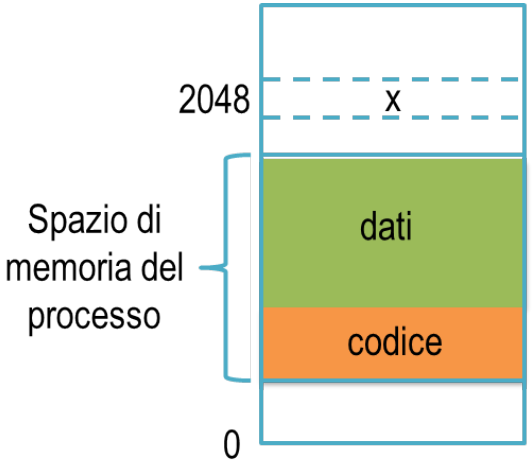
Il C e la memoria

- Le istruzioni ed i dati di un programma in esecuzione (processo) risiedono nella memoria del calcolatore
- Il C permette ad un programma di accedere alle porzioni di memoria a lui allocate
  - Puntatore: variabile che denota un indirizzo di memoria nello spazio di indirizzamento del processo
  - I puntatori permettono di manipolare il contenuto della memoria (sia i dati che le istruzioni!!!)



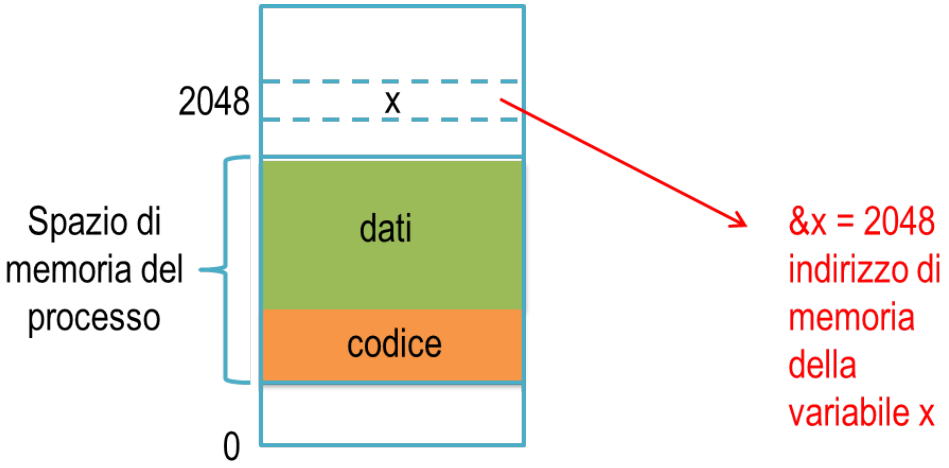
Il C e la memoria

- Le istruzioni ed i dati di un programma in esecuzione (processo) risiedono nella memoria del calcolatore
- Il C permette ad un programma di accedere alle porzioni di memoria a lui allocate
  - Puntatore: variabile che denota un indirizzo di memoria nello spazio di indirizzamento del processo
  - I puntatori permettono di manipolare il contenuto della memoria (sia i dati che le istruzioni!!!)



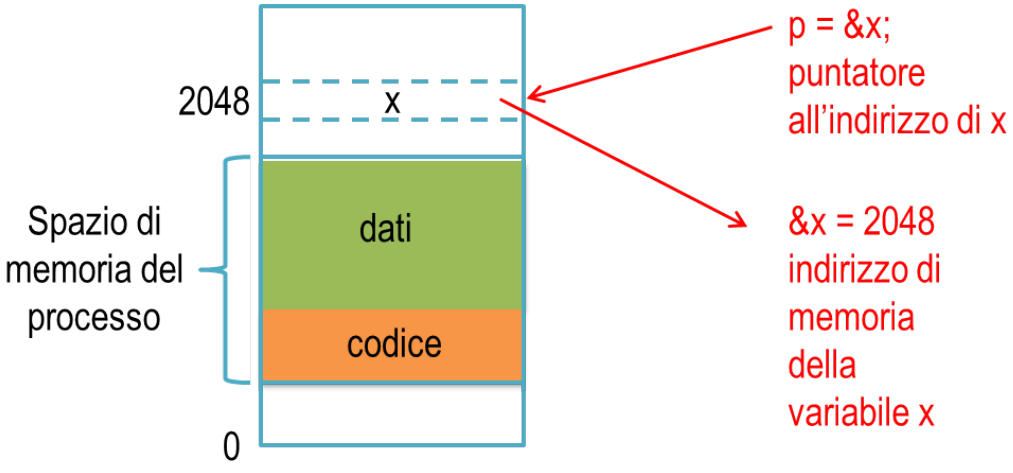
Il C e la memoria

- Le istruzioni ed i dati di un programma in esecuzione (processo) risiedono nella memoria del calcolatore
- Il C permette ad un programma di accedere alle porzioni di memoria a lui allocate
  - Puntatore: variabile che denota un indirizzo di memoria nello spazio di indirizzamento del processo
  - I puntatori permettono di manipolare il contenuto della memoria (sia i dati che le istruzioni!!!)



Il C e la memoria

- Le istruzioni ed i dati di un programma in esecuzione (processo) risiedono nella memoria del calcolatore
- Il C permette ad un programma di accedere alle porzioni di memoria a lui allocate
  - Puntatore: variabile che denota un indirizzo di memoria nello spazio di indirizzamento del processo
  - I puntatori permettono di manipolare il contenuto della memoria (sia i dati che le istruzioni!!!)



Struttura di un programma C

Il codice C è un file testuale con estensione .c (es. file.c) ed una struttura ben definita

Direttive per il Pre-Processore

```
#include <stdio.h>
#define MIACOSTANTE 0.1
```

Dichiarazioni Globali

```
int contatore = 0;
int funzione2(int a, int b);
```

Funzioni

```
int funzione1(int a) { ... }
int funzione2(int a,int b) { ... }
int main(void) { ... }
```

Un semplice programma C

Il più semplice programma C è costituito da un unico file con la sola funzione `main`

```
#include <stdio.h>

int main(void) {
    int a, somma;  /* Dichiarazione variabili di tipo Intero */
    int b = 1; /* Dichiarazione ed inizializzazione  variabile */

    a = 0; /* Assegnamento */
    somma = a + b; /* Somma di interi ed assegnamento */

    return  0; /* Valore restituito dal main al sistema operativo (0 -> OK) */
}
```

Dichiarazione variabili

Dichiarare una variabile le assegna un tipo ed un nome, es.

```
int a;
```

alloca una variabile di nome `a` e tipo `int` (Intero).

Sintassi allocazione:

```
<tipo_var> <nome_var>;
```

Variabili dello stesso tipo possono essere dichiarate insieme:

```
<tipo_var> <nome_var1>,..., <nome_varN>;
```

È necessario dichiarare tutte le variabili utilizzate!

Dichiarazione variabili e inizializzazione

Domanda

Che valore assume `a` dopo la sua dichiarazione con `int a;`?

Dichiarazione variabili e inizializzazione

Domanda

Che valore assume `a` dopo la sua dichiarazione con `int a;`?

Non si può dire: dichiarare una variabile alloca spazio in memoria, ma non ci dice niente sul suo contenuto!!

È necessario inizializzare sempre tutte le variabili!

Contestualmente alla loro dichiarazione....

```
int a = 0;
```

....oppure successivamente

```
int a;
....
a = 0;
```

Comandi C

- Assegnamento del valore di un'espressione ad una variabile
  - `somma = a + b;`
- Costrutti condizionali ed iterativi

```
if (somma > 0} {
  positivo = true;
} else {
  positivo = false;
}
```

- Chiamate a funzioni
  - `massimo = max(a,b);`

Dal sorgente all'eseguibile

- Prima di poter essere eseguito dal processore, un programma deve essere
  1. pre-elaborato o pre-processato (pre-processing)
  2. compilato (compiling)
  3. collegato (linking)
  4. caricato in memoria (loading)
- I file testuali .c sono parte del codice sorgente di un programma
  - Il compilatore trasforma il sorgente .c in codice oggetto .o (binario)
  - Il linking collega i file oggetto in un eseguibile
- Concentriamoci su pre-processore e compilatore

Dal sorgente all'eseguibile

- Prima di poter essere eseguito dal processore, un programma deve essere
  1. pre-elaborato o pre-processato (pre-processing)
  2. compilato (compiling)
  3. collegato (linking)
  4. caricato in memoria (loading)
- I file testuali .c sono parte del codice sorgente di un programma
  - Il compilatore trasforma il sorgente .c in codice oggetto .o (binario)
  - Il linking collega i file oggetto in un eseguibile
- Concentriamoci su pre-processore e compilatore

Dal sorgente all'eseguibile

- Prima di poter essere eseguito dal processore, un programma deve essere
  1. pre-elaborato o pre-processato (pre-processing)
  2. compilato (compiling)
  3. collegato (linking)
  4. caricato in memoria (loading)
- I file testuali .c sono parte del codice sorgente di un programma
  - Il compilatore trasforma il sorgente .c in codice oggetto .o (binario)
  - Il linking collega i file oggetto in un eseguibile
- Concentriamoci su pre-processore e compilatore

Pre-processing

- Fase preliminare alla compilazione
- Comporta la sostituzione di informazioni simboliche (testo) nel codice sorgente con un contenuto che viene specificato dal programmatore mediante direttive per il pre-processore
- Le direttive per il pre-processore vanno scritte in testa ai sorgenti C e sono precedute dal simbolo #
  - Inclusione file: `#include`
  - Macro: `#define`
  - Compilazione condizionale: `#ifdef...`

Non vi fate spaventare! In pratica si tratta di un sofisticato Trova e Sostituisci

Include

`#include PERCORSOFILE`

Dice al pre-processore di inserire il contenuto del file specificato con `PERCORSOFILE` all'interno del file C in cui si trova l'`include`

Due modi di specificare il percorso dei file:

- `#include <file>` - Il file viene cercato in un percorso standard (librerie), e.s. `/usr/include` su Linux
- `#include "file"` - Il file viene cercato a partire dalla directory corrente

Macro

```
#define NOME MACRO
```

Sostituisce ogni occorrenza di `NOME` con il testo in `MACRO`

Utile per definire costanti di uso comune, es.

```
#define MAX_INT +32767
```

***Ancora più utile perché permette di specificare testo parametrico

```
#define BIRRA(X) X bottiglie di birra sul muro
```

Quindi `BIRRA(MAX_INT)` diventerebbe?

Macro

```
#define NOME MACRO
```

Sostituisce ogni occorrenza di `NOME` con il testo in `MACRO`

Utile per definire costanti di uso comune, es.

```
#define MAX_INT +32767
```

***Ancora più utile perché permette di specificare testo parametrico

```
#define BIRRA(X) X bottiglie di birra sul muro
```

Quindi `BIRRA(MAX_INT)` diventerebbe?

Macro

```
#define NOME MACRO
```

Sostituisce ogni occorrenza di `NOME` con il testo in `MACRO`

Utile per definire costanti di uso comune, es.

```
#define MAX_INT +32767
```

***Ancora più utile perché permette di specificare testo parametrico

```
#define BIRRA(X) X bottiglie di birra sul muro
```

Quindi `BIRRA(MAX_INT)` diventerebbe?

```
+32767 bottiglie di birra sul muro
```

***Compilazione Condizionale

```
#ifdef MACRO
      TESTO1
#else
      TESTO2
#endif
```

Controlla se la `MACRO` è definita: se si, esegue le direttive in `TESTO1`; altrimenti, esegue le direttive alternative in `TESTO2`

Ad esempio è utile per fare l'include di un file solo la prima volta che viene incontrata la direttiva

Esistono altre istruzioni di compilazione condizionale: `#IF`, `#IFNDEF`,...

***Compilazione Condizionale - Esempio

Come viene rimpiazzato `DOMANDA(RISPOSTA)`[1]?

```
#define DOMANDA(X) Qual è la risposta alla domanda \
                       fondamentale sulla vita, \
                       l'universo e tutto quanto? X
#ifdef AUTOSTOP
     #define RISPOSTA 42
#else
     #define RISPOSTA
#endif
```

Se volessi ottenere una risposta cosa dovrei fare?

[1] va usata una `printf` nel corpo di `DOMANDA`

***Compilazione Condizionale - Esempio

Come viene rimpiazzato `DOMANDA(RISPOSTA)`?

```
#define DOMANDA(X) Qual è la risposta alla domanda \
                        fondamentale sulla vita, \
                        l'universo e tutto quanto? X
#ifdef AUTOSTOP
     #define RISPOSTA 42
#else
     #define RISPOSTA
#endif
```

Se volessi ottenere una risposta cosa dovrei fare?

Aggiungere `#define AUTOSTOP`
`DOMANDA(RISPOSTA)` → `Qual è la risposta alla domanda fondamentale sulla vita, l'universo e tutto quanto? 42`

Il compilatore

- Trasforma il file pre-processato (senza più `#include` o `#define`) in un file eseguibile che contiene
  - Alcune informazioni di carattere generale per il linking e il caricamento in memoria
  - Rappresentazione binaria di (parte dei) dati del programma (variabili globali)
  - Codice assembler eseguibile dal processore target
- Il compilatore C esegue una serie di controlli di correttezza
  - Sintassi dei comandi: es. terminazione con ";", parentesi bilanciate, ...
  - Coerenza dei tipi di dato: es. operatori algebrici usati solo con variabili numeriche, parametri delle funzioni, ...
  - Processing lineare: un pezzo di codice può usare solo variabili o funzioni dichiarate in precedenza

Il compilatore

- Trasforma il file pre-processato (senza più `#include` o `#define`) in un file eseguibile che contiene
  - Alcune informazioni di carattere generale per il linking e il caricamento in memoria
  - Rappresentazione binaria di (parte dei) dati del programma (variabili globali)
  - Codice assembler eseguibile dal processore target
- Il compilatore C esegue una serie di controlli di correttezza
  - Sintassi dei comandi: es. terminazione con ";", parentesi bilanciate, ...
  - Coerenza dei tipi di dato: es. operatori algebrici usati solo con variabili numeriche, parametri delle funzioni, ...
  - Processing lineare: un pezzo di codice può usare solo variabili o funzioni dichiarate in precedenza

GNU Compiler Collection (GCC)

- Sviluppato per sistemi Linux permette generare eseguibili per diverse piattaforme target
- Non è solo un compilatore ma è anche pre-processore e linker

Sintassi:

```
gcc [OPT] file.c -o fileExec
```

- `[OPT]` - Insieme di opzioni che controllano le funzionalità di GCC (es. `-E`, `-Wall`, `-g`)
- `file.c` - Nome del file sorgente
- `-o fileExec` - Imposta il nome del file eseguibile a `fileExec` (`a.out` di default)

Compilare un programma C (I)

Il comando

```
gcc file.c -o mioexec.out
```

fa sì che GCC esegua il pre-processing, la compilazione e il linking del codice in file.c generando l'eseguibile mioexec.out

Possiamo anche eseguire le tre fasi separatamente

- `gcc -E file.c` - pre-processa `file.c` e redirige il risultato sullo standard-out
- `gcc -c file.c` - compila `file.c` e genera il codice oggetto `file.o`
- `gcc file.o -o mioexec.out` - esegue solo il linking di `file.o` e genera l'eseguibile `mioexec.out`

Compilare un programma C (I)

Il comando

```
gcc file.c -o mioexec.out
```

fa sì che GCC esegua il pre-processing, la compilazione e il linking del codice in file.c generando l'eseguibile mioexec.out

Possiamo anche eseguire le tre fasi separatamente

- `gcc -E file.c` - pre-processa `file.c` e redirige il risultato sullo standard-out
- `gcc -c file.c` - compila `file.c` e genera il codice oggetto `file.o`
- `gcc file.o -o mioexec.out` - esegue solo il linking di `file.o` e genera l'eseguibile `mioexec.out`

Compilare un programma C (II)

```
gcc -Wall -pedantic -g file.c -o mioexec.out
```

- `-Wall -pedantic` opzioni che aumentano il numero di controlli e di messaggi di avvertimento (warnings) visualizzati
- `-g` opzione che include anche informazioni necessarie al debug
- Usate `--help` o `-v --help` per ottenere informazioni sulle opzioni disponibili per GCC

Errori di compilazione....

```
file.c:  In function 'main':
file.c:4:  warning:  implicit declaration of
function 'max'
Undefined symbols for architecture x86_64:  "_max",
referenced from:  _main in file.o
ld:  symbol(s) not found for architecture x86_64
collect2:  ld returned 1 exit status
```

Errori di compilazione....

```
file.c:  In function 'main':
file.c:4:  warning:  implicit declaration of
function 'max'
Undefined symbols for architecture x86_64:  "_max",
referenced from:  _main in file.o
ld:  symbol(s) not found for architecture x86_64
collect2:  ld returned 1 exit status
```

Errori di compilazione....

```
file.c: In function 'main':
file.c:4: warning: implicit declaration of
function 'max'
Undefined symbols for architecture x86_64: "_max",
referenced from: _main in file.o
ld: symbol(s) not found for architecture x86_64
collect2: ld returned 1 exit status
```

(Compilatore) Non conosco la funzione max: assumo che troverò l'implementazione dopo

Errori di compilazione....

```
file.c:  In function 'main':
file.c:4:  warning:  implicit declaration of
function 'max'
Undefined symbols for architecture x86_64:  "_max",
referenced from:  _main in file.o
ld:  symbol(s) not found for architecture x86_64
collect2:  ld returned 1 exit status
```

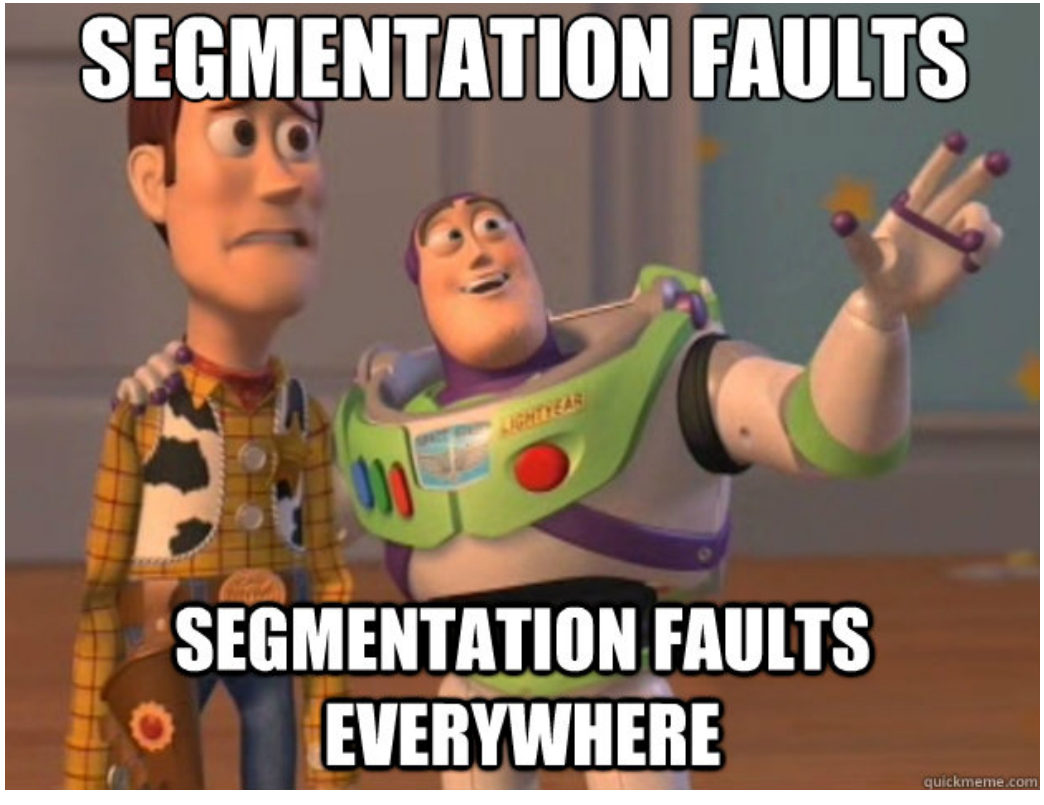
(Linker) Ho guardato in tutti i file oggetto ma non l'ho trovata: errore!

Esecuzione

E se riuscirete ad eliminare (tutti) gli errori di compilazione....

Esecuzione

E se riuscirete ad eliminare (tutti) gli errori di compilazione....



....ci saranno sempre gli errori a tempo di esecuzione

Commentare il Codice (I)

Programmare bene significa anche saper scrivere codice ben commentato e leggibile

```
//Commenti su singola linea

/* Si possono inserire anche commenti
   distribuiti su più linee  */
```

Commentate SEMPRE il vostro codice!!!

Commentare il Codice (II)

Perché?

- Spiegare come usare il vostro codice
- Spiegare come funziona il vostro codice
- Spiegare passaggi difficili nel codice

Per chi?

- Chiunque si troverà a modificare il vostro codice....
- ....inclusi voi stessi, dopo settimane, mesi o anni

Commentare il Codice (II)

Perché?

- Spiegare come usare il vostro codice
- Spiegare come funziona il vostro codice
- Spiegare passaggi difficili nel codice

Per chi?

- Chiunque si troverà a modificare il vostro codice....
- ....inclusi voi stessi, dopo settimane, mesi o anni

Nomi significativi

I commenti non sono l'unico modo di rendere leggibile il vostro codice

- Usate nomi di variabili che siano significativi..
  - `int l7; /* Che significa l7?  */`
  - `int somma; /* Decisamente più informativo! */`
- ....ma non esagerate con la verbosità dei nomi!
  - `int indice_ciclo_for; /* ????!  */`
  - `int i; /* ok!  */`

Convenzione nomi e altre regole stilistiche (I)

- I nomi definiti dalle macro `#define` usano tutte lettere maiuscole, eventualmente separando le singole parole con l'underscore: es. `MAX_INT`
- Di conseguenza non usate lo stile tutto maiuscolo per i nomi di varibili e funzioni
- Attenzione all'uso degli spazi: es. non vanno usati per gli argomenti delle funzioni

`massimo = max( a , b);` non va bene!

Convenzione nomi e altre regole stilistiche (II)

- Indentate sempre il codice in maniera che sia chiaro dove iniziano e finiscono i blocchi: es.

```
if (somma > 0) {
    positivo = TRUE;
} else {
    positivo = FALSE;
    if (somma == 0) {
        zero = TRUE;
    }
}
```

Hello World

```
#include <stdio.h>  /* Libreria standard C per funzioni di IO */

/* Il main definisce il punto di partenza del nostro programma.
* void -> non prende parametri in ingresso (in questo caso)
* int -> restituisce un intero */
int main(void) {

    /* Stampa a schermo la stringa passata come
       argomento */
     printf("Ciao mondo!\n");

   /* Valore restituito dal main al S.O. (0 -> OK) */
    return  0;
}
```

Scrivete il vostro "Hello World" in un file `hello.c`....comando per compilare?

Hello World

```
#include <stdio.h>  /* Libreria standard C per funzioni di IO */

/* Il main definisce il punto di partenza del nostro programma.
* void -> non prende parametri in ingresso (in questo caso)
* int -> restituisce un intero */
int main(void) {

    /* Stampa a schermo la stringa passata come
       argomento */
     printf("Ciao mondo!\n");

   /* Valore restituito dal main al S.O. (0 -> OK) */
    return  0;
}
```

Scrivete il vostro "Hello World" in un file `hello.c`, compilate ed eseguite:

```
$ gcc hello.c -o hello.out
$ ./hello.out
```

`stdio.h` - Funzioni per controllare l'I/O

Libreria standard C che mette per operazioni di input/output

Scrivere sullo schermo:

```
printf("Ciao Mondo!");
```

Pubblica a schermo il testo (stringa di formattazione) passato come argomento

```
printf("Ciao Mondo, ho %d anni!  \n", age);
```

La stringa di formattazione può contenere segnaposto (`%d` → `int`) che vengono rimpiazzati con il valore delle variabili (`int age`) dopo la virgola (in ordine di apparizione!!!!)

Esercizio 1

- Aprite l'editor gedit
- Create un file `hello.c`
- Scrivete in `hello.c` il vostro personale Hello World!
- Aprite una shell e posizionatevi sulla directory dove avete salvato `hello.c`
- Compilate `hello.c` usando GCC ed eseguitelo

Esercizio 2

- Create una copia di `hello.c`
- Modificate la copia perché il programma faccia le seguenti cose:
  - Dichiari una variabile intera `age`
  - Memorizzi nella variabile `age` la vostra età
  - Stampi a video il contenuto della variabile `age`
- Compilate ed eseguite come sopra

Esercizio 3

- Definire, con direttive del pre-processore, una costante MATRICOLA che contenga il vostro numero di matricola
- Stampare a schermo il testo

  "Ciao Mondo, ho X anni e il mio numero di matricola è Y"

  dove X è il contenuto della variabile `age` dell'esercizio precedente e Y è il valore di MATRICOLA (il che implica usare il comando `printf` nella `#define`)

***Esercizio 4

- Modificare il programma dell'Esercizio 3 perché il testo "Ciao Mondo, ho X anni e il mio numero di matricola è Y" sia definito dalla direttiva parametrica del pre-processore HELLO(X,Y) (va usata una printf nel corpo di HELLO(X,Y))
- Stampare il testo a schermo usando la direttiva HELLO(X,Y)
- Compilare ed eseguire il codice
- Usate GCC per mostrare a schermo l'output del pre-processore (senza compilazione e linking): che cosa è successo al vostro file C?