

Introduzione alla Programmazione in C

Lezione 2

Laboratorio di Programmazione I

Corso di Laurea in Informatica
A.A. 2017/2018



Calendario delle lezioni

Ogni lezione consta di una spiegazione assistita da slide, e seguita da esercizi in classe

Lezione 1 (19-20/09/2017) - Introduzione all'ambiente Linux

Lezione 2 (26-27/09/2017) - Introduzione al C

Lezione 3 (3-4/10/2017) - Tipi primitivi e costrutti condizionali

Lezione 4 (10-11/10/2017) - Costrutti iterativi ed array

Lezione 5 (17-18/10/2017) - Funzioni, stack e visibilità variabili

Lezione 6 (24-25/10/2017) - Puntatori e memoria

Lezione 7 (7-8/11/2017) - Debugging

Lezione 8 (14-15/11/2017) - Tipi di dati utente

Lezione 9 (21-22/11/2017) - Liste concatenate e librerie.

Lezione 10 (28-29/11/2017) - Ricorsione

Lezione 11 (5-6/12/2017) - CAML

Lezione 12 (12-13/12/2017) - Simulazione prova pratica

Libri di Riferimento del Corso

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

Al Kelley, Ira Pohl, *C. Didattica e programmazione*, Pearson Italia

Su DidaWiki trovate riferimenti a **materiali online** che coprono buona parte dei contributi del corso

```
http://didawiki.di.unipi.it/doku.php/  
informatica/pr1/start
```

Le origini del C



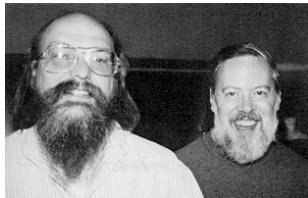
Sviluppato dai Bell Labs nella prima parte degli anni 1970 per la **programmazione del sistema operativo UNIX**

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

Si chiama C perchè 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 perchè 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)
- **Sicurezza!!**

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)
- **Sicurezza!!**

Caratteristiche fondamentali

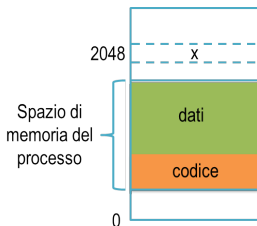
- Paradigma imperativo
 - **Assegnamento:** `variabile = espressione;`
 - **Tipi di dato primitivi e definiti dall'utente**
 - **Costrutti condizionali:** `IF...THEN...ELSE`
 - **Costrutti iterativi:** `FOR, WHILE, ...`
- **Strutturazione a blocchi**
 - Un programma C è una **collezione di funzioni** (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

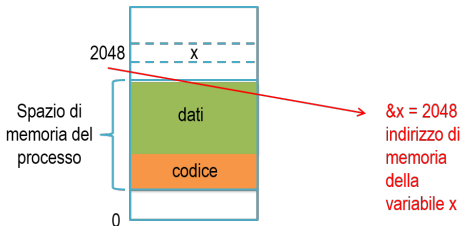
Il C e la memoria

- 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!!!)



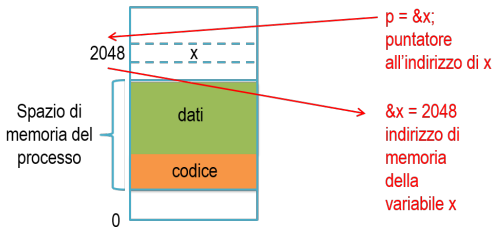
Il C e la memoria

- 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!!!)



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) { ... }
```

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) { ... }
```

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) */
}
```

Struttura delle Funzioni

Separare dichiarazioni e comandi!

```
int main(void) {
```

Dichiarazione variabili locali

```
int a, somma;  
int b = 1;
```

Comandi

```
a = 0;  
somma = a + b;  
return 0;
```

```
}
```

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>;
```

E' necessario **dichiarare tutte le variabili** utilizzate!

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>;
```

E' necessario **dichiarare tutte le variabili** utilizzate!

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>;
```

E' necessario **dichiarare tutte le variabili** utilizzate!

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>;
```

E' necessario **dichiarare tutte le variabili** utilizzate!

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!!

E' necessario **inizializzare sempre tutte le variabili!**

Contestualmente alla loro dichiarazione...

```
int a = 0;
```

..oppure successivamente

```
int a;
```

```
...
```

```
a = 0;
```

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!!

E' necessario **inizializzare sempre tutte le variabili!**

Contestualmente alla loro dichiarazione...

```
int a = 0;
```

..oppure successivamente

```
int a;
```

```
...
```

```
a = 0;
```


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!!

E' necessario **inizializzare sempre tutte le variabili!**

Contestualmente alla loro dichiarazione...

```
int a = 0;
```

..oppure successivamente

```
int a;
```

```
...
```

```
a = 0;
```

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!!

E' necessario **inizializzare sempre tutte le variabili!**

Contestualmente alla loro dichiarazione...

```
int a = 0;
```

..oppure successivamente

```
int a;
```

```
...
```

```
a = 0;
```

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!!

E' 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;`
- `somma = somma + 1;`

A sinistra `somma` indica il nome della variabile, a destra `somma` indica un valore.

- 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-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**
- In questa fase ci concentriamo su **pre-processore** e **compilatore**

Dal sorgente all'eseguibile

- Prima di poter essere **eseguito dal processore**, un programma deve essere
 - 1 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**
- In questa fase ci concentriamo su **pre-processore** e **compilatore**

Dal sorgente all'eseguibile

- Prima di poter essere **eseguito dal processore**, un programma deve essere
 - 1 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**
- In questa fase ci concentriamo 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`

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

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`

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

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`

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**

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 perchè 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
```

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 perchè 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
```

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 perchè 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`

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 perchè 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
```


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 perchè 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`

Il compilatore

- Trasforma il file preprocessato (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 preprocessato (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)

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 (1)

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 dirige 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 (1)

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 (2)

```
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
```

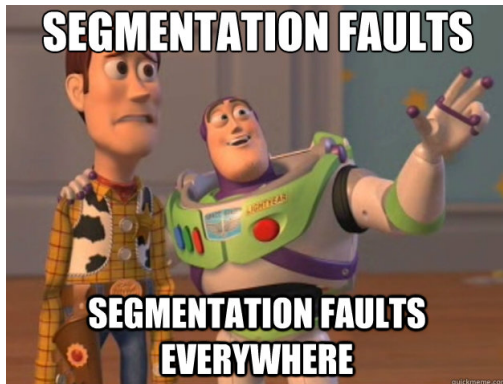
(**Linker**) Ho guardato in tutti i file oggetto ma non l'ho trovata: errore!

Esecuzione

E se riuscirete ad eliminare gli errori di compilazione...

Esecuzione

E se riuscirete ad eliminare gli errori di compilazione...



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

Commentare il Codice (1)

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 (1)

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 (2)

Perchè?

- 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 (2)

Perchè?

- 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! */`

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! */`

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 (1)

- 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 variabili 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 (1)

- 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 variabili 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 (1)

- 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 variabili 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 (2)

- 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`, compilate ed eseguite:

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

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 a disposizione funzioni di input/output

Per scrivere sullo schermo:

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

Pubblica a schermo il testo (**stringa di formattazione**) passato come argomento e va a capo

`\n` è un carattere di **escape**, sono caratteri speciali che iniziano per `\`:

- `\n`: va a capo;
- `\t`: inserisce un `tab`;
- `\%`: inserisce il simbolo `%`;
- ...

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

Libreria standard C che mette a disposizione funzioni di input/output

Per scrivere sullo schermo:

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

Pubblica a schermo il testo (**stringa di formattazione**) passato come argomento e va a capo

`\n` è un carattere di **escape**, sono caratteri speciali che iniziano per `\`:

- `\n`: va a capo;
- `\t`: inserisce un `tab`;
- `\%`: inserisce il simbolo `%`;
- ...

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

Libreria standard C che mette a disposizione funzioni di input/output

Per scrivere sullo schermo:

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

Pubblica a schermo il testo (**stringa di formattazione**) passato come argomento e va a capo

`\n` è un carattere di **escape**, sono caratteri speciali che iniziano per `\`:

- `\n`: va a capo;
- `\t`: inserisce un `tab`;
- `\%`: inserisce il simbolo `%`;
- ...

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

```
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** (in questo caso `age` di tipo `int`) dopo la virgola (in ordine di apparizione!!)

```
printf("Ciao Mondo, ho %d anni e %d amici!\n",  
age, age* 100);
```

Cosa stampa?

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

```
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** (in questo caso `age` di tipo `int`) dopo la virgola (in ordine di apparizione!!)

```
printf("Ciao Mondo, ho %d anni e %d amici!\n",  
age, age* 100);
```

Cosa stampa?

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 perche il programma faccia le seguenti cose:
 - Dichiarare una variabile intera `age`
 - Memorizzare nella variabile `age` la vostra età
 - Stampare a video il contenuto della variabile `age`
- Compilate ed eseguite come nell'esercizio precedente
- Eseguite una seconda volta il vostro programma reindirigendone l'output nel file `output.txt`: controllate il contenuto di quest'ultimo file di testo usando i comandi da shell

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**

Esercizio 4

Dato il seguente programma Hello World:

```
#include <stdio.h>

int main(void) {
    int i;
    printf("Ciao mondo. Vediamo fino a quanto so contare:\n");
    i = i + 1;
    printf("%d\n", i);
    i = i + 1 ;
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
    i++;
    printf("%d\n", i);
    i += 2;
    printf("%d\n", i);
    printf("Temo di non arrivare neanche a %d \n", --i);
    return 0;
}
```

- Che cosa succede se provate a **compilare** il programma?
Perchè?
- Che problema potrebbe esserci nell'**esecuzione** del codice?
Perchè?