

# Corso “Programmazione 1”

## Capitolo 10: Strutturazione di un Programma

Docente: **Marco Roveri** - `marco.roveri@unitn.it`  
Esercitori: **Giovanni De Toni** - `giovanni.detoni@unitn.it`  
**Stefano Berlato** - `stefano.berlato-1@unitn.it`  
C.D.L.: Informatica (INF)  
A.A.: 2021-2022  
Luogo: DISI, Università di Trento  
URL: <https://bit.ly/2VgfYwJ>



Ultimo aggiornamento: 10 novembre 2021

# Terms of Use and Copyright

## USE

This material (including video recording) is intended solely for students of the University of Trento registered to the relevant course for the Academic Year 2021-2022.

## SELF-STORAGE

Self-storage is permitted only for the students involved in the relevant courses of the University of Trento and only as long as they are registered students. Upon the completion of the studies or their abandonment, the material has to be deleted from all storage systems of the student.

## COPYRIGHT

The copyright of all the material is held by the authors. Copying, editing, translation, storage, processing or forwarding of content in databases or other electronic media and systems without written consent of the copyright holders is forbidden. The selling of (parts) of this material is forbidden. Presentation of the material to students not involved in the course is forbidden. The unauthorised reproduction or distribution of individual content or the entire material is not permitted and is punishable by law.

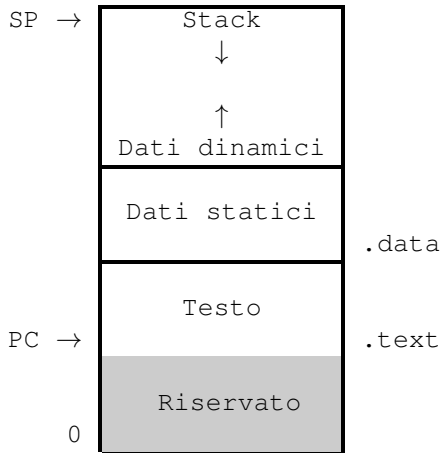
The material (text, figures) in these slides is authored mostly by Roberto Sebastiani, with contributions by Marco Roveri, Alessandro Armando, Enrico Giunchiglia e Sabrina Recla.

- 1 Modello di Gestione della Memoria in un Programma
- 2 Programmazione su File Multipli

# Modello di gestione della memoria per un programma (ripasso)

Area di memoria allocata ad un'esecuzione di un programma:

- **Area programmi**: destinata a contenere le **istruzioni** (in linguaggio macchina) del programma
- **Area dati statici**: destinata a contenere **variabili globali** o **allocate staticamente** e le **costanti** del programma.
- **Area heap**: destinata a contenere le **variabili dinamiche** (di dimensioni non prevedibili a tempo di compilazione) del programma.
- **Area stack**: destinata a contenere le **variabili locali** e i **parametri formali** delle funzioni del programma.



# Scope, Visibilità e Durata di una definizione

La definizione di un oggetto (variabile, costante, tipo, funzione) ha tre caratteristiche:

- Scope o ambito
- Visibilità
- Durata

# Scope di una definizione

È la porzione di codice in cui è attiva una definizione

- **Scope globale**: definizione attiva a livello di file
- **Scope locale**: definizione attiva localmente
  - ad una funzione
  - ad un blocco di istruzioni

```
const float pi=3.1415;      // scope globale
int x;                      // scope globale
int f(int a, double x);    // scope locale
{  int c;                  // scope locale
  ...}
int main()
{  char pi;                // scope locale
  ...}
```

# Visibilità di una definizione

Stabilisce quali oggetti definiti sono visibili da una punto del codice

- In caso di funzioni:
  - una definizione globale è visibile a livello locale, ma non viceversa
  - una definizione omonima locale maschera una definizione globale
- in caso di blocchi annidati
  - una definizione esterna è visibile a livello interno, ma non viceversa
  - una definizione omonima interna maschera una definizione esterna

```
1. const float pi=3.1415; // sono visibili:
2. int x;                // pi(1)
3. int f(int a, double x)
4. { int c;              // pi(1), a(3), x(3)
5.   ...}               // pi(1), a(3), x(3), c(4)
6. int main()
7. { char pi;           // x(2), f(3),
8.   ... }              // x(2), f(3), pi(7)
```

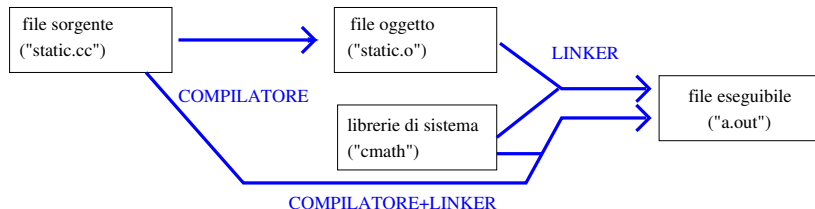
# Durata di una definizione

Stabilisce il periodo in cui l'oggetto definito rimane allocato in memoria

- **Globale o Statico**: oggetto globale o dichiarato con **static**
  - dura fino alla fine dell'esecuzione del programma
  - memorizzato nell'**area dati statici**
- **Locale o automatico**: oggetti locali a un blocco o funzione
  - hanno sempre scope locale
  - durano solo il periodo di tempo necessario ad eseguire il blocco o funzione in cui sono definiti
  - memorizzato nell'**area stack**
- **Dinamico**: oggetti allocati e deallocati da **new/delete**
  - hanno scope determinato dalla raggiungibilità dell'indirizzo.
  - durata gestita dalle chiamate a **new** e **delete**: durano fino alla deallocazione con **delete** o alla fine del programma
  - dimensione non prevedibile a tempo di compilazione
  - memorizzati nell'**area heap**



# Compilazione (un unico file) - recap



- **File sorgente** tradotto in un **file oggetto** dal **compilatore**
  - Es: `g++ -c static.cc`
  - Imp: file oggetto illeggibile ad un essere umano e non stampabile!
- File oggetto collegato (linked) a librerie di sistema dal **linker**, generano un **file eseguibile** (default `a.out`, opzione `-o <nome>`)
  - Es: `g++ static.o`
  - Es: `g++ static.o -o static`
  - File incomprensibili agli umani, ma eseguibili da una macchina
- Compilazione e linking possibile in un'unica istruzione
  - Es: `g++ static.cc`

# Lo specificatore **static**

Lo specificatore **static** applicato ad una **variabile locale** forza la durata della variabile oltre la durata della funzione dove è definita

- la variabile è allocata nell'**area dati statici**
- un'eventuale inizializzazione nella dichiarazione viene eseguita una sola volta all'atto dell'inizializzazione del programma
- il valore della variabile viene “ricordato” da una chiamata all'altra della funzione
- potenziali sorgenti di errori  $\implies$  **vanno usate con molta cautela!**

- Esempio di uso di variabile **static** locale:

```
{ PROG_FILE_MULTIPLI/static.cc }
```

- Esempio di uso di variabile **static** locale anziché globale:

```
{ PROG_FILE_MULTIPLI/fibonacci.cc }
```

# Lo specificatore `static` II

## Nota

Lo specificatore `static` applicato ad un **oggetto di scope globale** (es. funzioni, variabili, costanti globali) ha l'effetto di restringere la visibilità dell'oggetto al solo file in cui occorre la definizione

- concetto molto importante nella **programmazione su più file** (vedi slide successive)

# Lo specificatore **extern**

Lo specificatore **extern** consente di **dichiarare** e poi utilizzare in un file oggetti (globali) che sono **definiti** in un altro file

- consente al compilatore di
  - verificare la coerenza delle espressioni contenenti tali oggetti
  - stabilire le dimensioni delle corrispondenti aree di memoria
- l'oggetto dichiarato deve essere definito in un altro file
- il linker associa gli oggetti dichiarati alle corrispondenti definizioni

- **Esempio di uso di **extern**:**

```
{ PROG_FILE_MULTIPLI/extern.cc  
  PROG_FILE_MULTIPLI/extern_main.cc }
```

# Dichiarazione vs Definizione

## Nota

- Un oggetto può essere **dichiarato** quante volte si vuole ...  
... ma può essere **definito** una volta sola
- Un oggetto dichiarato più volte deve essere dichiarato sempre nello stesso modo
- Ogni **definizione** è anche una implicita **dichiarazione**

# Programmazione su file multipli

I programmi possono essere organizzati su file multipli

- Organizzazione modulare
  - Ogni file raggruppa un insieme di funzionalità (modulo)
  - Compilazione separata di ogni modulo e linking file oggetto
- Moltissimi vantaggi:
  - Rapidità di compilazione
  - Programmazione condivisa tra più persone/team
  - Riutilizzo del codice in più programmi
  - Produzione di librerie
  - Utilizzo di librerie prodotte da altri
  - Mantenibilità del codice
  - ...

# Organizzazione di un programma su file multipli

- Un programma viene usualmente ripartito su  $2N + 1$  file
  - Un file `file_main.cc` contenente la definizione della funzione `main()`
  - $N$  coppie di file `modulo_i.h` e `modulo_i.cc`, una per ogni modulo `modulo_i` che si vuole realizzare separatamente
  - Tutti i file “.cc” devono venire compilati e i risultanti file oggetto linkati

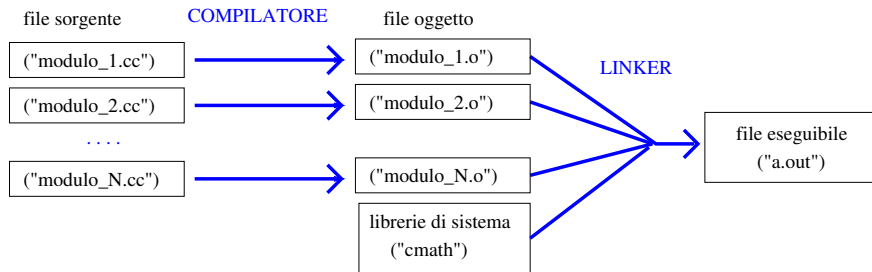
## Organizzazione di un programma su file multipli - II

- Ogni file “.cc” che utilizzi funzioni/tipi/costanti/variabili globali definiti in `modulo_i` deve inizialmente contenere l'istruzione: **#include** "modulo\_i.h"
- `modulo_i.h` contiene gli **header** delle funzioni di `modulo_i`
  - può contenere definizioni di tipo, costanti, variabili globali, ecc.
  - per evitare di venire caricato ripetutamente deve utilizzare **guardie di compilazione**:

```
#ifndef MODULO_I_H           // Esiste anche direttiva
#define MODULO_I_H           // #pragma once
...                           // Non e' standard e non e' robusto
...                           // come approccio (e.g. link,
                              // copie di file, ...)
#endif
```
- `modulo_i.cc` contiene le **definizioni** delle funzioni di `modulo_i`
  - contenere l'istruzione: **#include** "modulo\_i.h"
  - può contenere **funzioni ausiliarie inaccessibili all'esterno** (**static**)



# Compilazione (su più files) - recap



- File sorgente tradotti nei rispettivi **file oggetto** uno alla volta
- File oggetto collegato (linked) a librerie di sistema dal **linker**, generano un **file eseguibile** (default `a.out`)
  - Es: `g++ modulo_1.o modulo_2.o ... modulo_N.o`
- Compilazione e linking possibile in un'unica istruzione
  - Es: `g++ modulo_1.cc modulo_2.cc ... modulo_N.cc`

# Schema: Programma su un solo file

disney.cc

```
#include <iostream>
int pluto() {...};           // funzione ausiliaria
                               // non chiamata dal main()
void topolino() { ... };
void paperino() { ... };

int main() {
    ...
    switch(scelta) {
        case 1: topolino(); break;
        case 2: paperino(); break;
        ...
    }
}
```

## Schema: Programma su file multipli - I

disney.h

```
#ifndef DISNEY_H
#define DISNEY_H
void topolino();
void paperino();
#endif
```

disney.cc

```
#include <iostream>
#include "disney.h"
static int pluto() {...}; // funzione ausiliaria non chiamata
                          // dal main().
                          // Header non in disney.h!

void topolino() { ... };
void paperino() { ... };
```

## Schema: Programma su file multipli - II

disney\_main.cc

```
#include <iostream>
#include "disney.h"

int main() {
    ...
    switch(scelta) {
        case 1: topolino(); break;
        case 2: paperino(); break;
        ...
    }
}
```

# Organizzazione di un programma su file multipli: Esempi

- Programma su un solo file:

{ PROG\_FILE\_MULTIPLI/matrix\_v2\_typedef.cc }

- Stesso programma organizzato in file multipli:

{  
  PROG\_FILE\_MULTIPLI/matrix.h  
  PROG\_FILE\_MULTIPLI/matrix.cc  
  PROG\_FILE\_MULTIPLI/matrix\_main.cc  
}

# Bad programming practices

Cosa non fare e come non organizzare un programma su file multipli

- Includere i file “.h” (e compilare tutti i file, “.h” inclusi)
- **NON** includere i file “.cc” (e compilare solo il file chiamante) !!!
  - impedirebbe uso multiplo (non hanno guardie di compilazione )
  - romperebbe la modularità
  - impedirebbe la compilazione separata
- $\Rightarrow$  naturale sorgente di errori, **evitare tassativamente**

```
disney_main.cc
```

```
#include "disney.cc" // NNOO!!!!!!  
...
```

```
disney_main.cc
```

```
#include "disney.h" // SI!  
...
```

Vedere file `ESERCIZI_PROPOSTI.txt`