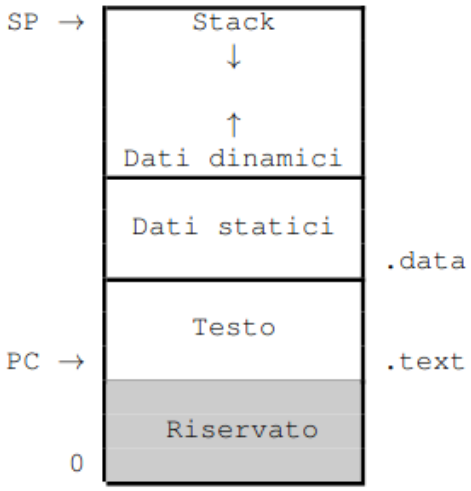
49. Strutturazione di un programma

Modello di gestione della memoria di un programma



L'area di memoria allocata ad un'esecuzione di un programma si divide in:

- **Area programmi**: destinata a contenere le istruzioni in linguaggio macchina del programma e si divide in:
  - Area riservata: tutte le istruzioni per eseguire il programma che dipendono dal sistema operativo;
  - **Area testo**: il codice assembly (in binario), del nostro programma col *main*
- **Area dati statici**: destinata a contenere variabili globali o allocate staticamente e le costanti del programma.
- **Area dati dinamici** (*heap*): destinata a contenere le variabili dinamiche (di dimensione non prevedibili a tempo di compilazione del programma). Il tutto avviene nei limiti di memoria a disposizione della macchina.
- **Area stack**: destinata a contenere le variabili locali e i parametri formali delle funzioni. In molti sistemi operativi ha dimensione fissa e non alterabile: MacOS, Windows e applicazioni destinate all'uso spaziale o per aerei (in queste applicazioni non si usano funzioni ricorsive o allocazioni dinamiche di memoria).

49.1 Scope di una definizione

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

- Scope o ambito
- Visibilità
- Durata

Lo scopo di una definizione è la porzione di codice in cui la definizione è attiva:

- **scope globale**: definizione attiva a livello di file (nel file in cui la si considera)
- **scope locale**: definizione attiva localmente (ad una funzione o ad un blocco di funzioni)

Esempio:

```
const float pi=3.1415;        // scope globale
int x;                        // scope globale
int f(int a, double x);       // scope locale (tutta la funzione f è definita con uno scope globale
                              visibile in tutto il file: diversamente da i parametri che sono locali a f)
{ int c;                      // scope locale
...}
int main()
{ char pi;                    // scope locale
…}
```

49.2 Visibilità di una definizione

Stabilisce quali oggetti sono visibili da un punto del codice (quando l'oggetto è visibile). 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

Esempio

```
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)
```

49.3 Durata di una definizione

Stabilisce il periodo in cui l'oggetto definito rimane allocato in memoria, cioè utilizzabile dal programma.

Globale o Statico: oggetto globale o dichiarato con *static*. Dura fino alla fine dell'esecuzione del programma e viene 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 e sono memorizzati nell'area stack.

Dinamico: oggetti allocati e deallocati da *new/delete*. Hanno scope determinato dalla raggiungibilità dell'indirizzo.
La loro durata è gestita dalle chiamate a *new* e *delete*: durano fino alla deallocazione con *delete* o alla fine del programma (in caso di mancata deallocazione). La loro dimensione non prevedibile a tempo di compilazione e sono memorizzati nell'area *heap*.

49.4 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 e 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 **!**
! *static* può anche essere applicato davanti al tipo restituito da una funzione. In questo caso la funzione è utilizzabile SOLAMENTE nel file in cui viene definita e non in altri file **!**
Potenziali sorgenti di errori: vanno usate con molta cautela!

Lo specificatore *static* applicato ad un oggetto di scope globale (ad esempio 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).
Infatti se ho due file e dichiaro una variabile *static* in un file, essa potrà essere usata solo in quel file.

49.5 Lo specificatore *extern*

Permette di dichiarare un oggetto e poi usare in quel file un oggetto (globale) definito in un altro file. Consente al compilatore di verificare la coerenza delle espressioni contenenti tali oggetti e 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.

Con *extern* quindi si usa una variabile in un file *a* che viene definita in un file *b* (nel file *a* la variabile viene solo dichiarata, in questo modo: *extern tipo var;*).

Se compiliamo due file insieme (*g++ file1.cc file2.cc* e otteniamo un unico *a.out*) r in uno usiamo una variabile definita in un altro file:

- se di tipo *static*, non possiamo usarla (neanche usando l’*extern*).
- se nel file in cui la usiamo la dichiariamo con un *extern* (globalmente) la possiamo usare (non la possiamo usare se la definiamo con static nell’altro file). Se usiamo nel main una variabile omonima la variabile che usa il programma è chiaramente l’omonima locale. Se non mettiamo *extern* non la possiamo usare.

Analogo discorso si fa con le funzioni (devo fare *extern tipo funz(…);* globalmente se voglio usare una funzione definita in un altro file).

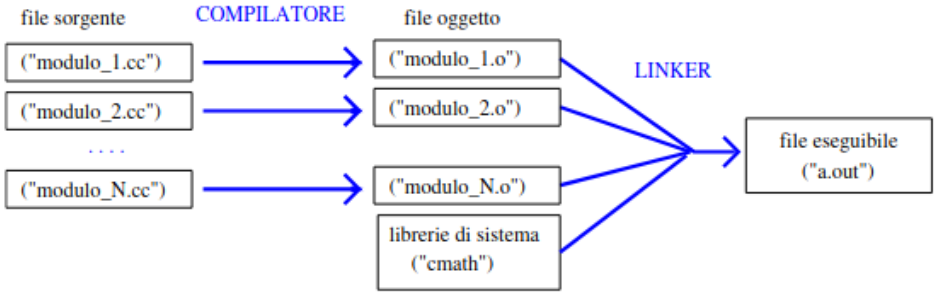
49.6 Dichiarazione vs definizione

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 un’implicita dichiarazione.

50. Programmazione su file multipli
I programmi possono essere organizzati su file multipli e presentare quindi un'organizzazione modulare: ogni file raggruppa un insieme di funzionalità (modulo). Può avvenire una compilazione separata di ogni modulo e linking finale dei vari file oggetto.

La programmazione su più file ha 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

50.1 Compilazione su più files



File sorgente tradotti nei rispettivi file oggetto uno alla volta.
g++ -c file.cc: per creare un file oggetto (*.o*) che contiene le informazioni del nostro codice in codice binario. Non è leggibile da noi ma solo dal compilatore.

I file oggetto collegati (linked) a librerie di sistema dal linker, generano un file eseguibile (default *a.out*, opzione *-o <nome>*)
g++ file1.o
g++ file1.o file2.o file3.o -o file.out

È anche possibile compilare e linkare direttamente i file senza passare dai file oggetto.
g++ file1.cc file2.cc file3.cc

! Ordine importante ! Prima il *main* poi i moduli.

50.2 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*. Ogni file *modulo_i* in cui è diviso in il nostro programma deve essere associato a un file *.h*.

Tutti i file *.cc* devono venire compilati e i risultanti file oggetto linkati.

50.3 Organizzazione del modulo
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.cc contiene le **definizioni** delle funzioni di *modulo_i*. Contiene all'inizio l'istruzione *#include "modulo_i.h"*, l'inclusione delle librerie usate e può contenere funzioni ausiliarie inaccessibili all'esterno (usare ***static***).

È come un file normale ma senza il *main*: ha solo le definizioni alle funzioni.
modulo_i.h contiene gli header delle funzioni di *modulo_i.cc.* Può contenere dichiarazioni di tipo, costanti, variabli globali, librerie e tutto ciò che viene usato in *modulo_i.cc*.

Per evitare di venire caricato ripetutamente deve utilizzare guardie di compilazione:
#ifndef MODULO_I_H *//**NOMEFILE_H*** (nome file in maiuscolo)
#define MODULO_I_H // *ifndef* è un comando che agisce sulla CPU (funziona come un isruzione *if* e se la risposta è affermativa si esegue *define* se no *endif)*
...
...
#endif

Al posto dei "…" vanno le dichiarazioni di funzioni (**non quelle *static!***), le librerie e altri costrutti (*struct*, *typedef*, …) usate nel file *.cc*.

Esiste anche direttiva:
#pragma once
Però non è standard e non è un approccio robusto.

Esempio

- File *disney.h*

```
#ifndef DISNEY_H
#define DISNEY_H
#include <iostream>
using namespace std;
void topolino();
void paperino();
#endif
```

- File *disney.cc*

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

- File *disney_main.cc*

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

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

50.4 Bad programming practices
Non includere direttamente in file .cc ma includere i file *.h*.

In fase di compilazione, bisogna compilare solamente i file *.cc*.

...
}

Nel file *.h*:

- dichiarazione di funzioni usate nei moduli (tranne quelle *static*)
- *using namespace std;* e *typedef … ;*
- librerie usate nei moduli (è possibile ometterle all'inizio dei moduli)
- *struct* che si usano nei moduli e nel *main* (non sarà necessario definire le *struct* nel *main* e nei moduli, ma basterà, per creare una nuova variabile di quella struttura, fare *struct_id* var). È possibile fare nel *.h* una forward declaration, ovvero scrivere solamente *struct struct_id;* e poi nei file in cui la si usa scrivere: *struct struct_id {};* e specificare i campi