Corso “Programmazione 1”
Capitolo 10: Strutturazione di un Programma

Docente:	Roberto Sebastiani - roberto.sebastiani@unitn.it
Esercitatori:	Mario Passamani - mario.passamani@unitn.it
	Alessandro Tomasi - alessandro.tomasi@unitn.it

C.D.L.:	Informatica (INF)
	Ing. Informatica, delle Comunicazioni ed Elettronica (ICE)
	Studenti con numero di matricola pari
A.A.:	2019-2020
Luogo:	DISI, Università di Trento
URL:	disi.unitn.it/rseba/DIDATTICA/prog1_2020/

Ultimo aggiornamento: 30 ottobre 2019, ore 11:29

Outline

Outline

Modello di gestione della memoria per un programma

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

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

Modello di gestione della memoria per un programma

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

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

Modello di gestione della memoria per un programma

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

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

Modello di gestione della memoria per un programma

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

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

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
  ...
}
```

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
  ...
}
```

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 interna, 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)
```

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 interna, 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)
```

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 interna, 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` 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

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` 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

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` 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

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 $\Longrightarrow$ vanno usate con molta cautela!

- Esempio di uso di variabile static locale:
  { `PROG_FILE_MULTIPLI/static.cc` }
- Esempio di uso di variabile static locale anziche' globale:
  { `PROG_FILE_MULTIPLI/fibonacci.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 $\Longrightarrow$ vanno usate con molta cautela!

- Esempio di uso di variabile static locale:
  { `PROG_FILE_MULTIPLI/static.cc` }
- Esempio di uso di variabile static locale anziche' 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` }

Nota su Dichiarazione e Definizione

Nota

- Un oggetto può essere dichiarato quante volte si vuole mentre può essere degli definito una volta sola
- Un oggetto dichiarato più volte deve essere dichiarato sempre nello stesso modo
- Ogni definizione è anche un'implicita dichiarazione

Outline

Programmazione su file multipli

I programmi possono essere organizzati su file multipli

- Organizzazione modulare
  - Ogni file raggruppa un insieme di funzionalità (modulo)
  - Compilati separatamente e linkati
- 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
  - ...

Programmazione su file multipli

I programmi possono essere organizzati su file multipli

- Organizzazione modulare
  - Ogni file raggruppa un insieme di funzionalità (modulo)
  - Compilati separatamente e linkati
- 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
  - ...

Programmazione su file multipli

I programmi possono essere organizzati su file multipli

- Organizzazione modulare
  - Ogni file raggruppa un insieme di funzionalità (modulo)
  - Compilati separatamente e linkati
- 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 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 linkati
- 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, variabli globali, ecc.
  - per evitare di venire caricato ripetutamente deve utilizzare guardie di compilazione:

```
#ifndef MODULO_I_H
#define MODULO_I_H
...
#endif
```

- `modulo_i.cc` contiene le definizioni delle funzioni di `modulo_i`
  - può contenere funzioni ausiliarie inaccessibili all'esterno (`static`)

Organizzazione di un programma su file multipli

- Un programma viene usualmente ripartito su $2N + 1$ file
  - Un file `file_main.cc` contenente la 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 linkati
- 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, variabli globali, ecc.
  - per evitare di venire caricato ripetutamente deve utilizzare guardie di compilazione:

```
#ifndef MODULO_I_H
#define MODULO_I_H
...
#endif
```

- `modulo_i.cc` contiene le definizioni delle funzioni di `modulo_i`
  - può contenere funzioni ausiliarie inaccessibili all'esterno (`static`)

Organizzazione di un programma su file multipli

- Un programma viene usualmente ripartito su $2N + 1$ file
  - Un file `file_main.cc` contenente la 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 linkati
- 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, variabli globali, ecc.
  - per evitare di venire caricato ripetutamente deve utilizzare guardie di compilazione:

```
#ifndef MODULO_I_H
#define MODULO_I_H
...
#endif
```

- `modulo_i.cc` contiene le definizioni delle funzioni di `modulo_i`
  - può contenere funzioni ausiliarie inaccessibili all'esterno (`static`)

Organizzazione di un programma su file multipli

- Un programma viene usualmente ripartito su $2N + 1$ file
  - Un file `file_main.cc` contenente la 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 linkati
- 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, variabli globali, ecc.
  - per evitare di venire caricato ripetutamente deve utilizzare guardie di compilazione:

```
#ifndef MODULO_I_H
#define MODULO_I_H
...
#endif
```

- `modulo_i.cc` contiene le definizioni delle funzioni di `modulo_i`
  - può contenere funzioni ausiliarie inaccessibili all'esterno (`static`)

Schema: Programma su un solo file

```
// 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 II

```
// ------------------- file disney.h ---------------
#ifndef DISNEY_H
#define DISNEY_H

void topolino();
void paperino();
#endif

// ------------------- file disney.cc --------------
#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

```
// ------------------- file 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` }

Come **non** organizzare un file su file multipli

Includere i file ".h" (e compilare tutti i file),
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

$\Longrightarrow$ naturale sorgente di errori, evitare tassativamente

```
// ------------- file disney_main.cc ----------
#include "disney.cc" // NO!!
...
```

```
// ------------ file disney_main.cc -----------
#include "disney.h" // SI!
...
```

Esercizi proposti

Vedere file `ESERCIZI_PROPOSTI.txt`