Lezione 8

Visibilità degli identificatori e tempo di vita degli oggetti

Ripasso dichiarazioni oggetti

- Finora abbiamo scritto le dichiarazioni di variabili e costanti con nome solo all'interno di blocchi
  - Nel corpo della funzione main, ad esempio:
    `main() { int i ; cin>>i ; int a ; … }`
  - Nel corpo delle altre funzioni, ad esempio:
    ```
    void fun() {cout<<"ciao"<<endl ; int c ; ...}
    main() { }
    ```
  - Nel corpo delle istruzioni di scelta ed iterative, quando tale corpo era definito con istruzioni composte, ad esempio:
    ```
    if (a > 1) {
          cout<<"caso 1"<<endl;
          int c ;
          …
    }
    ```

Dichiarazioni globali

- Si possono però scrivere le dichiarazioni di variabili e costanti con nome **all'esterno del corpo delle funzioni**
  - In questo caso, se il programma sta su un solo file, si parla di dichiarazioni e di **identificatori globali**
  - Se si tratta di definizioni di variabili o costanti con nome ed il programma sta su un solo file, si parla di **variabili o costanti con nome globali**
- In generale, un programma scritto su di un solo file ha la seguente struttura in C ed in C++

Struttura programma C

```
#include <stdio.h> // in generale inclusione di tutti i file necessari

<dichiarazioni_globali>

<intestazione_funzione>
{
        <dichiarazioni>
        <istruzioni diverse da dichiarazioni>
}

<dichiarazioni_globali>

<intestazione_funzione>
{
        <dichiarazioni>
        <istruzioni diverse da dichiarazioni>
}

...

<dichiarazioni_globali>

main()
{
        <dichiarazioni>
        <istruzioni diverse da dichiarazioni>
}
```

Struttura programma C++

```
#include <iostream> // in generale inclusione di tutti i file necessari

<dichiarazioni_globali>

<intestazione_funzione>
{
        <istruzioni>
}

<dichiarazioni_globali>

<intestazione_funzione>
{
        <istruzioni>
}

...

<dichiarazioni_globali>

main()
{
        <istruzioni>
}
```

Dichiarazioni locali

- Gli identificatori dichiarati nei blocchi sono invece comunemente denominati locali (al blocco)
  - Si dice che la dichiarazione è locale ad un dato blocco
  - Se si tratta di definizioni di variabili o costanti con nome, si parla di variabili o costanti con nome locali (al blocco)

Introduzione

- In questa lezione vediamo in quali punti del programma si può utilizzare un certo oggetto ed in quali momenti della vita del programma tale oggetto è presente in memoria
  - In funzione di dove è stato dichiarato
- Per procedere dobbiamo prima completare la descrizione delle caratteristiche di un oggetto
  - Diciamo che una certa caratteristica è
    - **statica** se non cambia durante l'esecuzione del programma
    - **dinamica** se cambia durante l'esecuzione del programma
  - Esemplifichiamo questo concetto nel caso di variabili ...

Caratteristiche di una variabile

- **Nome:** definito mediante un identificatore
- **Indirizzo (lvalue):** locazione di memoria a partire dalla quale è memorizzata la variabile
- **Valore (rvalue):** contenuto dell'area di memoria associata alla variabile
- **Tipo:** insieme di valori che la variabile può assumere e di operazioni ad essi applicabili

Statiche/dinamiche ?

- **Nome:** statico o dinamico?
- **Indirizzo (lvalue):** statico o dinamico?
- **Valore (rvalue):** statico o dinamico?
- **Tipo:** statico o dinamico?

Statiche/dinamiche

- **Nome:** statico
- **Indirizzo (lvalue):** statico
- **Valore (rvalue):** dinamico
- **Tipo:** statico
  - Il C/C++ è un linguaggio con tipizzazione statica e forte (il compilatore controlla il rispetto del tipo)
  - Il tipo di una variabile **non cambia** durante l'esecuzione del programma

Visibilità e tempo di vita

- **Campo di visibilità** (**scope**) di un identificatore: parte (del testo) di un programma in cui l'identificatore può essere usato
- **Tempo di vita di un oggetto**: intervallo di tempo in cui l'oggetto è mantenuto in memoria
  - in FORTRAN: tempo di vita statico
    - Ogni oggetto vive quanto tutto il programma
  - in C, C++, Pascal: tempo di vita in generale **dinamico**
    - Gli oggetti possono nascere e morire durante l'esecuzione del programma

Regole di visibilità

- Regole di visibilità: regole con cui si stabilisce il campo di visibilità degli identificatori
- Due alternative
  - **Regole di visibilità statiche**: il campo di visibilità è stabilito in base al testo del programma, cioè in base a dove sono dichiarati gli identificatori
    - C/C++, Java, Pascal
  - **Regole di visibilità dinamiche**: il campo di visibilità è stabilito dinamicamente in base alla sequenza di chiamata delle funzioni
    - LISP ed alcuni linguaggi di scripting

Casistica

- Riassumendo, in C/C++ il campo di visibilità degli identificatori e, come vedremo, il tempo di vita degli oggetti, dipendono da dove sono collocate le corrispondenti dichiarazioni nel testo del programma
- Considereremo tre casi
  1) Identificatori dichiarati all'interno del blocco che definisce il corpo di una funzione diversa dal **`main`** o di un qualsiasi altro blocco diverso dal corpo del **`main`**, o nella lista dei parametri formali di una funzione diversa dal **`main`**
  2) Identificatori dichiarati nel blocco del **`main`**
  3) Identificatori globali

Terminologia

- In base a quanto detto in precedenza, nei primi due casi si tratta di **identificatori locali ad un blocco**
  - In particolare nel primo caso si tratta di un blocco che non coincide con il blocco che definisce il corpo del **`main`**
  - Quindi può trattarsi anche di un blocco innestato nel blocco del main
- Partiamo dal primo caso

Caso 1

- Il **campo di visibilità** degli identificatori dichiarati all'interno di un blocco è il blocco stesso
  - dal punto in cui sono dichiarati fino alla fine del blocco
  - Con solo la seguente eccezione
    - Se nel blocco compare un blocco innestato in cui è dichiarato un identificatore con lo stesso nome, l'oggetto del blocco esterno rimane in vita, ma l'identificatore non è visibile nel blocco innestato
- In quanto al **tempo di vita**, se si tratta di definizioni di variabili/costanti, queste ultime hanno tempo di vita che va dal momento della definizione fino alla fine dell'esecuzione delle istruzioni del blocco (senza eccezioni)

Esempio

```
funzione1()
{
      int a = 3 ;
      cout<<a ;
      int bz=5;
      bz++;
      cout<<bz;
}

main()
{
      …
      funzione1() ;
      …
}
```

Il campo di visibilità dell'identificatore **a** va dal punto in cui è definita fino alla fine del blocco, ossia del corpo della funzione. Il suo tempo di vita va dall'istante in cui viene eseguita la sua definizione fino all'istante in cui termina l'esecuzione delle istruzioni presenti nel blocco. In pratica la variabile **a** ha *tempo di vita* e visibilità limitati a tutto e solo il corpo di **funzione1()**

Esempio

```
funzione1()
{
      int a = 3 ;
      cout<<a ;
      int bz=5;
      bz++;
      cout<<bz;
}

main()
{
      …
      funzione1() ;
      …
}
```

Il campo di visibilità dell'identificatore **bz** va anch'esso dal punto in cui è definita fino alla fine del blocco, ossia del corpo della funzione.
Il suo tempo di vita va dall'istante in cui viene eseguita la sua definizione fino all'istante in cui termina l'esecuzione delle istruzioni presenti nel blocco.
In pratica la variabile **bz** ha *tempo di vita* e visibilità limitati ad una parte del blocco della **funzione1()**

Esempio blocco innestato 1/4

```
funzione1()
{
      int bz=5;
      int ps=9;
      ...
      {
         int ps=bz+5;
         cout<<ps ;
      }
      cout<<ps;
      ...
}
```

Le variabili **bz** e **ps** qui definite hanno tempo di vita relativo a tutto il corpo di **funzione1()**

Cosa stampa?

Cosa stampa?

Esempio blocco innestato 2/4

```
funzione1()
{
     int bz=5;
     int ps=9;
     ...
     {
        int ps=bz+5;
        cout<<ps ;
     }
     cout<<ps;
     ...
}
```

10

Esempio blocco innestato 3/4

```
funzione1()
{
     int bz=5;
     int ps=9;
     ...
     {
        int ps=bz+5;
        cout<<ps ;   10
     }
     cout<<ps;    9
     ...
}
```

Esempio blocco innestato 4/4

```
funzione1()
{
      int bz=5;
      int ps=9;
      ...
      {
         int ps=bz+5;
         cout<<ps ;
      }
      cout<<ps;
      ...
}
```

L'identificatore **bz** ha visibilità relativa a tutto il corpo di **funzione1()**.
L'identificatore **ps** dichiarato nel blocco esterno ha visibilità relativa a tutto il corpo di **funzione1()**, **fatta eccezione per il blocco interno**.

Variante 1/2

```
funzione1()
{
      int bz=5;
      int ps=9;
      ...
      {
         ps=bz+5;
         cout<<ps ;
      }
      cout<<ps;
      ...
}
```

Cosa stampa?

Cosa stampa?

Variante 2/2

```
funzione1()
{
      int bz=5;
      int ps=9;
      ...
      {
         ps=bz+5;
         cout<<ps ;   10
      }
      cout<<ps;    10
      ...
}
```

Esempio parametro formale

```
funzione1(int ps)
{
      int bz=3;
      ...
      {
         int ps=bz+5;
         cout<<ps ;
      }
      cout<<ps;
      ...
}
```

La variabile **bz** ed il parametro formale **ps** hanno tempo di vita relativo a tutto il corpo di **funzione1()**

L'identificatore **bz** ha visibilità relativa a tutto il corpo di **funzione1()**.
Il parametro formale **ps** ha visibilità relativa a tutto il corpo di **funzione1()**, **fatta eccezione per il blocco interno**.

Caso 2

- Vediamo il caso di identificatori dichiarati nel blocco che definisce il corpo del **`main`**
  (ma non in un blocco annidato nel corpo del **`main`**, che ricadrebbe nel precedente caso)
  - Se si tratta di definizioni di variabili/costanti, vivono quanto la funzione **`main`**, ovvero hanno **tempo di vita pari alla durata del programma**
  - Ma, in generale **non sono visibili in qualunque parte del programma**, perché il **`main`** è una funzione come tutte le altre
    - In modo simile al caso precedente sono visibili solo dal punto in cui sono dichiarati fino alla fine del corpo del **`main`**
      - Con l'esclusione di blocchi annidati in cui siano dichiarati identificatori con lo stesso nome

Esempio

```
fun()
{
   int bz=5;
   bz++;
}

main()
{
   int as=10;
   int xt=5;
   xt=xt*as;
   fun();
   cout<<xt;
}
```

Come sappiamo, la variabile **bz** ha tempo di vita e scope limitato al corpo di **fun()**

La variabile **as** e la variabile **xt** hanno **tempo di vita pari alla durata del programma**, ma scope limitato al blocco del **main()**

Sono visibili da dentro il corpo di fun() ?

Risposta

- No

Caso 3: Identificatori globali

- Gli identificatori dichiarati all'esterno del corpo delle funzioni sono visibili
  - dal punto in cui sono dichiarati (non prima!)
  - fino alla fine del file
    a meno della seguente eccezione:
    - Se nel blocco di altre funzioni è dichiarato un identificatore con lo stesso nome, l'oggetto globale rimane in vita, ma l'identificatore non è visibile nel blocco innestato
- Se si tratta di definizioni di variabili/costanti, le relative variabili/costanti hanno **tempo di vita pari alla durata dell'intero programma** (senza eccezioni)

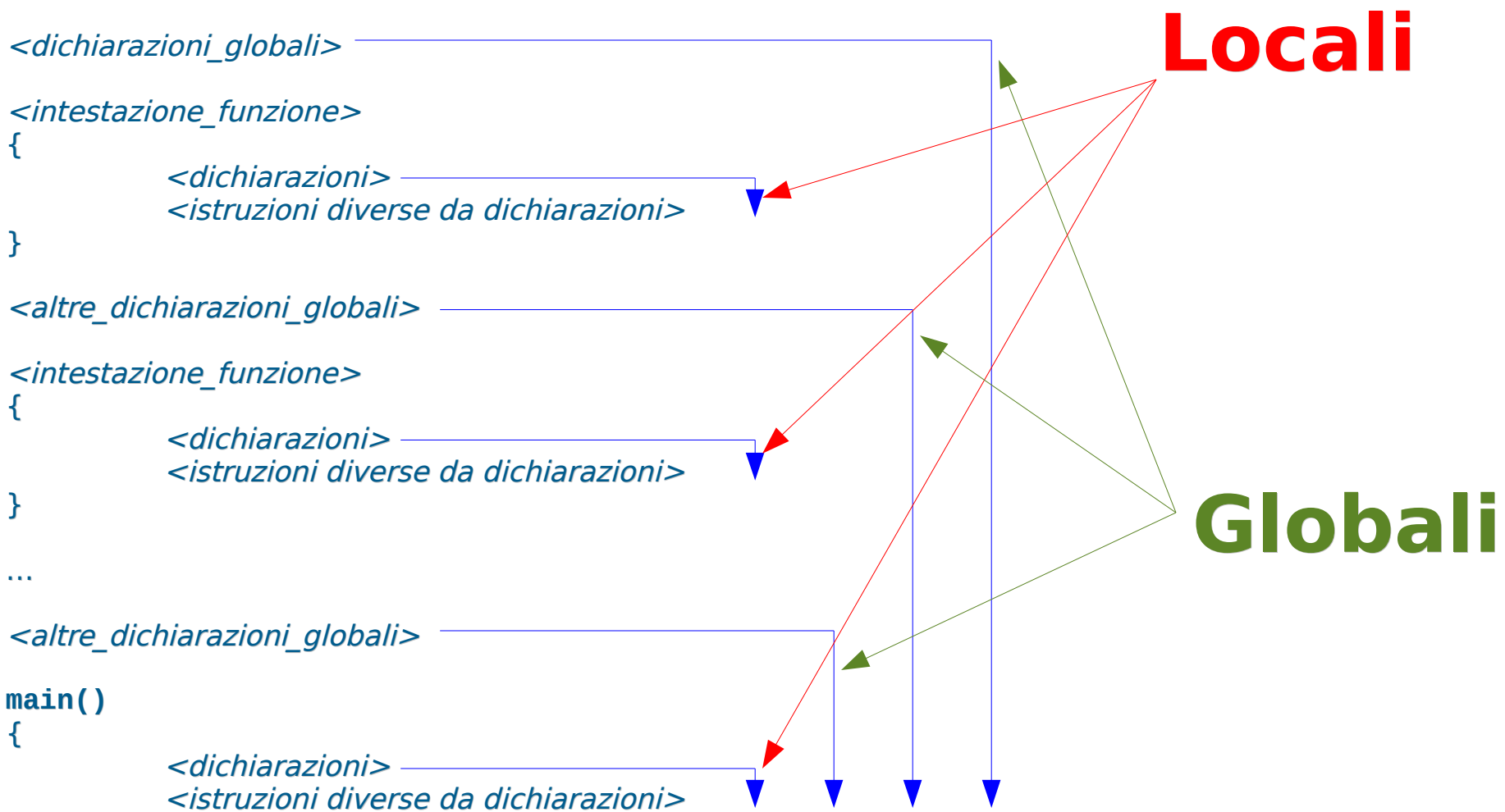
Esempio generale C/C++ 1/2

```
...

<dichiarazioni_globali>

<intestazione_funzione>
{
        <dichiarazioni>
        <istruzioni diverse da dichiarazioni>
}

<altre_dichiarazioni_globali>

<intestazione_funzione>
{
        <dichiarazioni>
        <istruzioni diverse da dichiarazioni>
}

...

<altre_dichiarazioni_globali>

main()
{
        <dichiarazioni>
        <istruzioni diverse da dichiarazioni>
}
```

Campo di visibilità identificatori

Esempio generale C/C++ 2/2

```
...

<dichiarazioni_globali>

<intestazione_funzione>
{
        <dichiarazioni>
        <istruzioni diverse da dichiarazioni>
}

<altre_dichiarazioni_globali>

<intestazione_funzione>
{
        <dichiarazioni>
        <istruzioni diverse da dichiarazioni>
}

...

<altre_dichiarazioni_globali>

main()
{
        <dichiarazioni>
        <istruzioni diverse da dichiarazioni>
}
```



Esercizio

```
#include <iostream>

int  x=0;

void funct1()
{ x = x + 3; }

void funct2()
{
     int x = 10;
     funct1();
     cout<<x;  /* x = ?? */
}

main()
{
   x++;
   cout<<x;  /* x = ?? */
   funct2();
   cout<<x;  /* x = ?? */
}
```

Esercizio

```
#include <iostream>

int  x=0;

void funct1()
{ x = x + 3; }

void funct2()
{
     int x = 10;
     funct1();
     cout<<x;  /* x = ?? */
}

main()
{
   x++;
   cout<<x;  /* x = 1 */
   funct2();
   cout<<x;  /* x = ?? */
}
```

Esercizio

```
#include <iostream>

int  x=0;

void funct1()
{ x = x + 3; }

void funct2()
{
     int x = 10;
     funct1();
     cout<<x;  /* x = ?? */
}

main()
{
   x++;
   cout<<x;  /* x = 1 */
   funct2();
   cout<<x;  /* x = 4 */
}
```

Esercizio

```
#include <iostream>

int  x=0;

void funct1()
{ x = x + 3; }

void funct2()
{
     int x = 10;
     funct1();
     cout<<x;  /* x = 10: NON E' LA x GLOBALE !!! */
}

main()
{
   x++;
   cout<<x;  /* x = 1 */
   funct2();
   cout<<x;  /* x = 4 */
}
```

Unicità identificatori

- L'associazione di un identificatore ad un oggetto deve essere unica per tutto il suo campo di visibilità
- Esempio di errore:

```
{
 int intervallo = 5;
 int  Y, Z, intervallo; // ERRORE: il terzo
                        // identificatore è già
                        // associato ad un altro
                        // oggetto
}
```

Valori iniziali variabili

- Il valore iniziale di una variabile non inizializzata dipende da dove tale variabile è definita
  - All'interno di un blocco: corpo di una funzione (incluso la funzione **`main`**) o qualsiasi altro blocco
    - **Valore iniziale casuale**
      - Notevole fonte di errori!
  - Al di fuori di tutti i blocchi
    - Valore iniziale 0

Esercizi

- Svolgere l'esercizio *mostra_visibilita.cc* della settima esercitazione

Domanda

- Se un componente di un sistema elettronico, meccanico o quant'altro non funziona
- E' più facile risolvere il problema
  - Se il malfunzionamento del componente può dipendere solo da malfunzionamenti di ciò che sta dentro il componente stesso?
  - Oppure se il guasto del componente potrebbe dipendere anche da un malfunzionamento di un qualsiasi altro componente del sistema?

Risposta

- Ovviamente è più facile nel primo caso
  - Nel caso peggiore basta semplicemente sostituire il pezzo guasto con uno funzionante
  - Si accorciano i tempi ed i costi di manutenzione

Dipendenze

- Questo stesso tipo di problemi si hanno nei sistemi software
- Si dice che tra due parti di un programma vi è una dipendenza se il corretto funzionamento di una delle due parti dipende dalla presenza e dal corretto funzionamento dell'altra parte
- All'aumentare delle dimensioni di un programma, uno degli obiettivi principali della progettazione è
  - Ridurre al minimo le **dipendenze** tra le varie parti del programma
- Questo obiettivo è in conflitto con l'uso di variabili e costanti con nome globali

Oggetti globali e complessità

- Gli oggetti globali **aumentano la complessità dei programmi**
  - Creano effetti collaterali: modificando il valore di una variabile globale si modifica il comportamento di tutte le funzioni che la usano: **dipendenza**
  - Quindi, se si usa una data variabile globale, per capire come funziona il programma e per evitare di commettere errori bisogna tenere in mente tutti i punti in cui è o può essere usata !!!
- Al contrario, gli oggetti locali riducono la complessità dei programmi
  - Bisogna sempre cercare di usare oggetti locali, e ricorrere a quelli globali solo quando non vi è altra soluzione!

Esempio dal mondo reale ...

- http://www.safetyresearch.net/blog/articles/toyota-unintended-acceleration-and-big-bowl-“spaghetti”-code
- Controllate il numero di variabili globali che è riportato essere presente nel codice incriminato ...