Visibilità e tempo di vita delle variabili

(più sintesi di alcuni concetti della prima parte del corso)

Struttura dei programmi C/C++

Un programma C/C++ deve essere contenuto in uno o più file (per ora si assume tutto in un file**)**:

- Una parte contenente direttive per il ***pre-processore***: **#** tipicamente per l'inclusione di file di intestazione (**#include**)
- Definizione della **funzione speciale** `main()`

- Definizione di eventuali funzioni `funz1()`, `funz2()`, …
- Eventuali prototipi di funzioni
- Eventuali definizioni/dichiarazioni di costanti/variabili al di fuori delle funzioni

1. Direttive al *pre-processore*

SINTASSI: #comando

1. **#include** (utilizzata per includere altri file)

   Esempio: #include <cmath>

   Esempio: #include “miofile.h”

2. Funzione main ()

- `main()` è una funzione speciale con tre caratteristiche:
  - deve essere sempre presente
  - è la prima funzione che viene eseguita ovunque si trovi all'interno del file (stessa cosa vale nel caso di programma su più file)
  - quando termina l'esecuzione del **main()**, termina il programma
- In C la funzione `main()` contiene due sezioni, racchiuse entro il ***blocco*** denotato da parentesi `{ }`
  - **Parte dichiarativa**
  - **Parte esecutiva**

Esempio C

Esempio

```
#include <iostream>

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

Direttive al pre-processore

Parte dichiarativa

Parte esecutiva

Ciascuna **dichiarazione** è separata dall'altra dal **;**

Ciascuna **istruzione** è separata dall'altra dal **;**

Non vi sono altri simboli speciali per separare parte dichiarativa da parte esecutiva

Esempio C++

Esempio

```
#include <iostream>

main()
   { <dichiarazione o istruzione>
     <dichiarazione o istruzione>
      ...
   }
```

Direttive al pre-processore

Concetto di blocco

<blocco in C++> ::=

```
{
    <istruzione o dich.>
    <istruzione o dich.>
    ...
}
```

<blocco in C> ::=

```
{
        [ <dichiarazioni >; ]
        <istruzioni>;
}
```

NOTE

- dopo un blocco non occorre il punto e virgola (esso *termina* le istruzioni semplici, non *separa* istruzioni)
- il campo d'azione delle <dichiarazioni > che compaiono entro il blocco è il blocco stesso
- i blocchi possono essere innestati

3. Concetto di funzione

- L'astrazione di *funzione* è presente in tutti i linguaggi di programmazione di alto livello.
- È l'astrazione del concetto di *operatore* su un tipo di dato, ovvero è un *componente software* che cattura l'astrazione matematica di funzione:

$$f : A \times B \times ... \times Q \rightarrow S$$

con molti possibili ingressi ed **una sola uscita** (corrispondente al risultato). Esempio:

```
int funzione1 ([ parametri; ] )
 {
   ...
 }
```

possono anche mancare

Esempio struttura dei programmi C

```
#include <stdio.h>

dichiarazioni;
funzione1()
   { <dichiarazioni>
     <istruzioni>  }

dichiarazioni;

funzione2()
   { <dichiarazioni>
     <istruzioni>  }

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

Esempio struttura dei programmi C++

```
#include <iostream>

dichiarazioni;
funzione1()
   { <dichiarazioni mescolate ad istruzioni>
   }

dichiarazioni;

funzione2()
   { <dichiarazioni mescolate ad istruzioni>
   }

main()
   { <dichiarazioni mescolate ad istruzioni>
   }
```

Tempo di vita e *scope* (*regole di visibilità*) delle variabili

Caratteristiche di una variabile (*già note*)

Nome: identificatore

Indirizzo (lvalue): locazione di memoria a partire dalla quale è memorizzato l'oggetto riferito dalla variabile

Valore (rvalue): è rappresentato nell'area di memoria associata alla variabile

Tipo: specifica l'insieme di valori che la variabile può assumere e le operazioni ad essi applicabili

Caratteristiche di una variabile *completamento*

Nome: statico

Indirizzo (lvalue): statico

Valore (rvalue): dinamico

Tipo: statico

- **Il C/C++ è un linguaggio con tipizzazione statica e forte**
- **Il tipo di una variabile non cambia durante l'esecuzione**

Visibilità e tempo di vita

Campo di visibilità (*scope*): parte di programma in cui l'identificatore può essere usato

- in C, C++, Pascal $\rightarrow$ determinato staticamente
- in LISP $\rightarrow$ determinato dinamicamente

Tempo di vita: intervallo di tempo in cui l'oggetto è mantenuto in memoria

- in FORTRAN $\rightarrow$ allocazione statica
- in C, C++, Pascal $\rightarrow$ allocazione dinamica

NOTA: *Scope* e *tempo di vita* possono non coincidere

Regole di visibilità *statiche* e *dinamiche*

- 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
  - Regole di visibilità **dinamiche**: il campo di visibilità è stabilito dinamicamente in base *alla sequenza di chiamata* delle funzioni

Dichiarazioni, visibilità e tempo di vita

Vi sono tre casi principali di dichiarazione di identificatori:

1. Identificatori dichiarati all'interno di un blocco (di una funzione o di un qualsiasi altro blocco) o nella lista dei parametri formali di una funzione
2. Identificatori dichiarati nel blocco del *main*()
3. Identificatori dichiarati all'esterno di funzioni

Caso 1: Variabili interne ad un blocco

- **Il campo di visibilità degli identificatori dichiarati all'interno di un blocco è il blocco stesso**
- Sono **visibili:** dal momento della dichiarazione fino alla fine del blocco (con una eccezione!)
- Se si tratta di definizioni, le relative variabili hanno **tempo di vita:** dal momento della definizione fino alla fine del blocco (senza eccezioni)

ECCEZIONE

- **Se nel blocco compare un blocco innestato in cui è dichiarato un identificatore con lo stesso nome, la variabile del blocco esterno rimane in vita, ma non è visibile nel blocco innestato**

Esempio 1

```
#include <iostream>

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

funzione2()
   { int abc=0;
     abc=abc*10;
     cout<<abc;
   }
```

La variabile **bz** ha *tempo di vita* e *scope* (visibilità) limitato a tutto e solo il blocco della funzione1()

Esempio 2

```
#include <iostream>
funzione1()
  { int bz=5;
    float ak=3.2;
    int ps=9;
      ...
          {
            int ps=bz+5;
            cout<<ps ;
          }
     cout<<ps;
  ...
  }
```

?

?

Le variabili **bz**, **ak**, **ps** hanno *tempo di vita* relativo a tutto blocco della funzione1()

Le variabili **bz**, **ak** hanno *scope* relativo a tutto il blocco della funzione1().
La variabile **ps** definita nel blocco esterno ha *scope* relativo a tutto il blocco della funzione1(), **fatta eccezione per il blocco interno.**

Unicità e ordine di definizione degli identificatori

- Definizioni di costanti e variabili associano ad un **identificatore** una costante o una variabile
  - L'associazione dell'identificatore all'oggetto deve essere **unica** per tutto il suo ***campo di azione***

Esempi

```
{ int     intervallo;
  double  Y, Z, intervallo;
}
```

Sono scorrette perché il simbolo **intervallo** è definito due volte nello stesso campo di azione (*blocco*)

Caso 2: Variabili del *main*()

- In C/C++, le variabili dichiarate nella funzione main()
  - Se si tratta di definizioni, 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 main() è una funzione come le altre

    → **visibilità delle variabili *limitata al blocco* in cui sono dichiarate**

    NON SONO VARIABILI CON SCOPE GLOBALE!

Esempio 3

```
#include <iostream>

funzione1()
   { int bz=5;
     bz++;
   }

main()
   { int as=10;
     int xt=5;
     xt=xt*as;
     funzione1();
     cout<<xt;
    }
```

La variabile **bz** ha *tempo di vita* e *scope* limitato al blocco della funzione1()

La variabile **as** e la variabile **xt** hanno *tempo di vita* pari alla durata del programma, ma *scope* limitato al blocco del main().
NON SONO VISIBILI IN funzione1()

Caso 3: Variabili esterne alle funzioni

- Sono **visibili** dal momento della dichiarazione (non prima!) fino alla fine del file (con un'eccezione)
- Se si tratta di definizioni, le relative variabili hanno **tempo di vita** pari alla durata dell'intero programma

ECCEZIONE

- **Se nel blocco di altre funzioni è dichiarata una variabile con lo stesso nome, la variabile esterna rimane in vita, ma non è visibile all'interno di quel blocco**

Struttura generale di un programma C/C++ all'interno di un unico file

Esempio

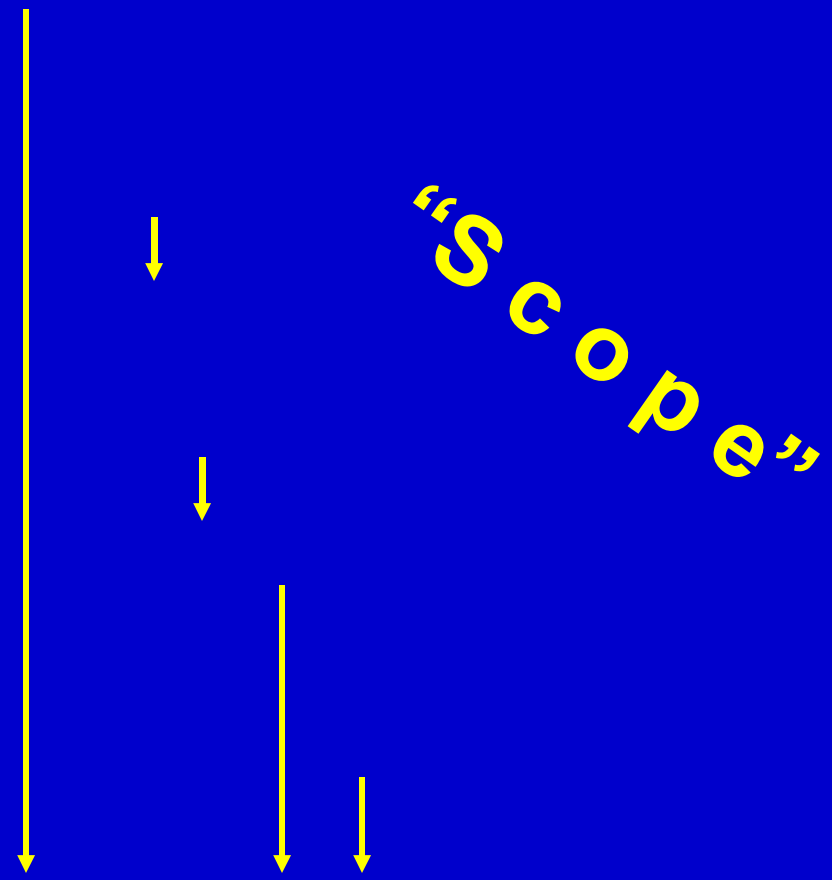
```
#include <iostream>
dichiarazioni;

funzione1()
   { <dichiarazioni>
     <istruzioni>  }

funzione2()
   { <dichiarazioni>
      <istruzioni>  }

dichiarazioni;

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



Esempio 4

```
#include <iostream>
int  x=0;
funct1()
   { x = x + 3; }
funct2()
 {
   int x = 10;            /* è un'altra x rispetto
   funct1();                  alla x globale! */
   cout<<x;  /* x = ?? */   10
 }
main()
 {
   x++;
   cout<<x;  /* x = ?? */   1
   funct2();
   cout<<x;  /* x = ?? */   4
 }
```

Nomenclatura

- Gli identificatori dichiarati nei blocchi sono comunemente denominati **locali** (al blocco)
  - Se si tratta di definizioni di variabili, si parla di **variabili locali** (al blocco)
- Gli identificatori dichiarati al di fuori dei blocchi, se il programma sta su un solo file, sono denominati **globali**
  - Se si tratta di definizioni di variabili ed il programma sta su un solo file, si parla di **variabili globali**

Struttura generale di un programma C/C++ all'interno di un unico file

Esempio

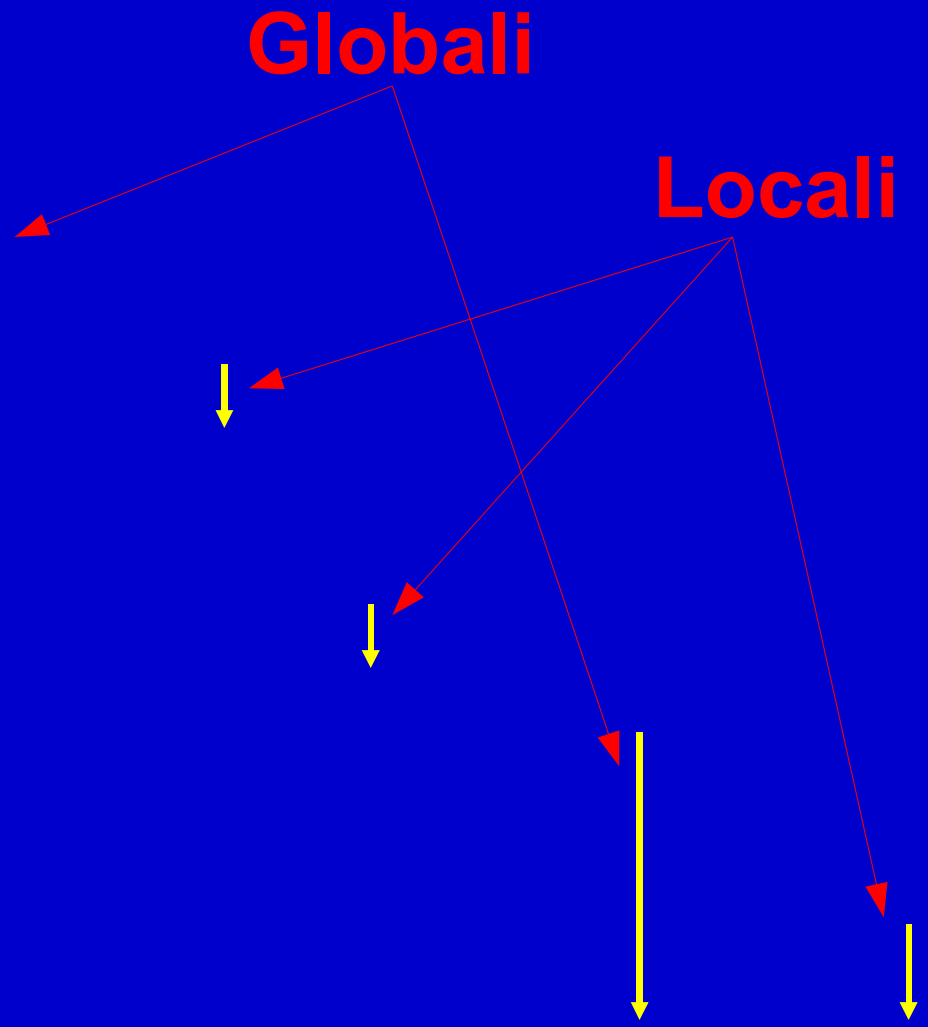
```
#include <iostream>
dichiarazioni;

funzione1()
   { <dichiarazioni>
     <istruzioni>  }

funzione2()
   { <dichiarazioni>
      <istruzioni>  }

dichiarazioni;

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



Parole chiave della lezione

- Direttive al pre-processore
- Funzione main()
- Concetto di
  - Blocco
  - Funzione

- Visibilità (*scope*) degli identificatori
- Tempo di vita delle variabili

- Tre tipologie di identificatori/variabili (dipendenti dalla posizione delle dichiarazioni/definizioni)