



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Scope e namespace

Stefano Ghidoni



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Agenda

- Scope
- Namespace



Scope

- Lo scope è una regione del testo di un programma
- Ciascuna variabile è dichiarata in uno scope e valida (*in scope*) in esso
 - Dal punto della dichiarazione fino alla fine dello scope
 - Nomi di variabili: tanto più descrittivi quanto più lo scope è grande
- Obiettivo: rendere i nomi locali
 - "Locality is good" (BS)
 - Evitare interferenze (clash)



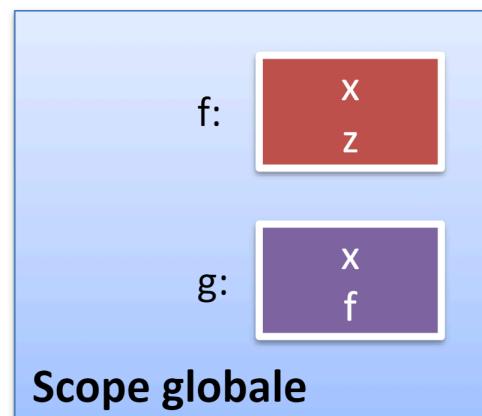
- Tipi di scope
 - Globale: al di fuori di ogni altro scope
 - Da usare con attenzione: vedi prossime slide
 - Scope di classe: il testo all'interno di una classe
 - Scope locale: all'interno di un blocco {} o nella lista degli argomenti di una funzione
 - Scope di statement: es., in un for-statement
 - Namespace: uno scope con un nome inserito nello scope globale o in un altro namespace
 - Nuovo strumento



Scope: esempio

```
void f(int x)           // f is global; x is local to f
{
    int z = x+7;        // z is local
}

int g(int x) {          // g is global; x is local to g
{
    int f = x+2;        // f is local
    return 2*f;
}
```



Qui "f" può essere una variabile o il nome della funzione.
Non vi sono conflitti perché chiamare una variabile e chiamare una funzione avviene in maniera diversa.
Se si desse lo stesso nome di una variabile globale a una locale, quella locale coprirebbe temporaneamente quella globale (senza sovrascriverla).



Scope globale

- Le **funzioni** sono spesso dichiarate e definite nello scope globale
- Per le **variabili** lo scope globale è problematico
 - Globale significa che la variabile è accessibile ovunque nel file sorgente (scope di file)
 - Ogni funzione può modificare variabili globali
 - Nessun tipo di protezione / encapsulamento
 - Il debug è quasi impossibile in software reali
 - Codice meno espressivo
 - Spesso usato per "semplificare" il passaggio di dati



Scope annidati

- Scope annidati: scope definito dentro un altro scope
 - Situazione abbastanza frequente
- Possibile combinando insieme:
 - Varie istanze dello stesso strumento di scoping
 - Es: due for-loop annidati
 - Strumenti di scoping diversi
 - Es: funzioni in classi, ...



Scope annidati

- Blocchi annidati
 - Inevitabili, ma attenti ai blocchi troppo complessi
 - L'indentazione è importante per la leggibilità

```
void f(int x, int y)
{
    if (x > y){
        // ...
    }
    else {
        // ...
    }
    // ...
}
```



- Funzioni nello scope di classi
 - Caso più frequente e più utile

```
class C {  
  
public:  
    void f();  
    void g()  
    {  
        // ...  
    }  
};  
  
void C::f()  
{  
    // ...  
}
```



Scope annidati

- Classi in classi (classi annidate)
 - Raramente necessario

```
class C {  
  
public:  
    class M {  
    };  
};
```

Esempio: per le eccezioni.

L'eccezione prende un prefisso che la lega alla classe a cui si riferisce



- Classi in funzioni (classi locali)
 - Da evitare!
 - Se sembra necessario, probabilmente la funzione f è troppo lunga

```
void f()
{
    class L {
        // ...
    };
    // ...
}
```



Scope annidati

- Funzioni in funzioni (funzioni locali o annidate)
 - Non sono legali in C++

```
void f()
{
    void g()          // illegale
    {
        // ...
    }
    // ...
}
```

Strumenti di scoping



Strumenti di scoping

- Alcuni strumenti di scoping sono già noti:
 - Funzioni
 - Classi
- Permettono di definire "entità" senza preoccuparci di conflitti di nomi (clash)
 - Usiamo le funzioni per raggruppare linee di codice in un blocco
 - Usiamo le classi per organizzare funzioni, dati e tipi in un tipo



Namespace

- Introduciamo ora i namespace
 - Costrutti **dedicati allo scoping**
- Permettono di organizzare classi, funzioni, dati e tipi dentro a uno spazio di nomi **senza definire un tipo**
- Molto utili per nomi potenzialmente usati in molte librerie
 - Mat, Matrix, Color, Shape, ...

Se due librerie usano lo stesso nome senza usare il namespace, non si possono usare le due librerie contemporaneamente.



Namespace

```
namespace Graph_lib {  
  
    struct Color { /* ... */ };  
    struct Shape { /* ... */ };  
    struct Line : Shape { /* ... */ };  
    struct Function : Shape { /* ... */ };  
    struct Text : Shape { /* ... */ };  
    // ...  
    int gui_main() { /* ... */ }  
  
}
```

Nessun conflitto
per Text e Line

```
namespace TextLib {  
  
    class Text { /* ... */ };  
    class Glyph { /* ... */ };  
    class Line { /* ... */ };  
    // ...  
}
```



Namespace

- Ora le classi hanno un nuovo nome:
 - Graph_lib::Text
 - TextLib::Text
- namespace::membro -> fully qualified name
(nome completamente specificato)



Namespace

- L'uso di namespace può appesantire il codice

```
#include <iostream>
#include <string>

int main()
{
    std::string name;
    std::cout << "Please enter your first name\n";
    std::cin >> name;
    std::cout << "Hello, " << name << '\n';

    return 0;
}
```



Dichiarazioni e direttive using

- Using declaration
 - Associa un namespace a un nome

```
using std::string;  
using std::cout;
```

- Using directive
 - Da usare solo per namespace molto noti (es., std)

```
using namespace std;
```

Permette di eliminare "std" da tutto il codice

- **Da evitare negli header! Perché?**

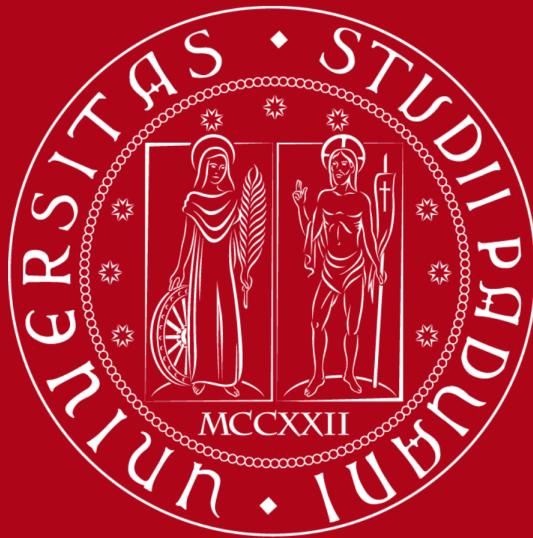
L'header è gestito dal preprocessore. Se viene inserito nell'header, chiunque lo includa è obbligato ad accettare la nostra decisione, che magari non è desiderata.

In una struttura professionale, obblighiamo chi usa i nostri header, quindi la libreria, a fare associazioni che potrebbero non voler fare.



Recap

- Scope: dove è valido un nome?
- Namespace: strumento per gestire gli scope (non l'unico)



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Scope e namespace

Stefano Ghidoni