

Lezione2

Table of contents

- [Dichiarazione vs Definizione](#)
 1. [Tipi di dato](#)
 2. [Variabili](#)
 3. [Funzioni](#)
 4. [Template](#)
- [Scope](#)
 1. [Scope di `namespace`](#)
 2. [Scope di blocco](#)
 3. [Scope di classe](#)
 4. [Scope di funzione](#)
 5. [Scope delle costanti di enumerazione](#)
 6. [Hiding](#)
 1. [Estensioni della visibilità del nome](#)
 2. [Direttive di `using`](#)

Dichiarazione vs Definizione

Tipi di dato

- **Dichiarazione** intesa come costrutto del linguaggio che introduce un nome per una entità (abbiamo visto l'esempio di `cout` in `HelloWorld.cpp`). La struttura non la conosco, non posso creare oggetti di tipo `S`.

```
// dichiarazione pura del tipo S
struct S;
```

- **Definizione** sottintesa come dichiarazione (siccome introduce un nome), per fornire ulteriore elemento per caratterizzare l'entità (implementazione di `foo()`). Di `T` conosco la struttura e posso creare oggetti di quel tipo.

```
// definizione del tipo T
struct T { int a; };
```

Essendo linguaggio denso, C++ insiste sul fatto di usare gli stessi caratteri per indicare situazioni totalmente diverse. Nel caso sotto, il compilatore si può chiedere:

```
nome1 * nome2
```

1. `nome1` è puntato dal puntatore `nome2`?
2. operatore binario (moltiplicazione) tra i due valori?

Al compilatore servono indizi sui tipi delle variabili dichiarate (per default assume sia valore), altrimenti ambiguità sussistono alla compilazione, che non andrebbe a buon fine.

☰ enum del C++ 2011

Con lo standard introdotto nel 2011, il C++ implementa la possibilità di *dichiarare puramente* un tipo di enumerazione che prima necessitava obbligatoriamente la definizione

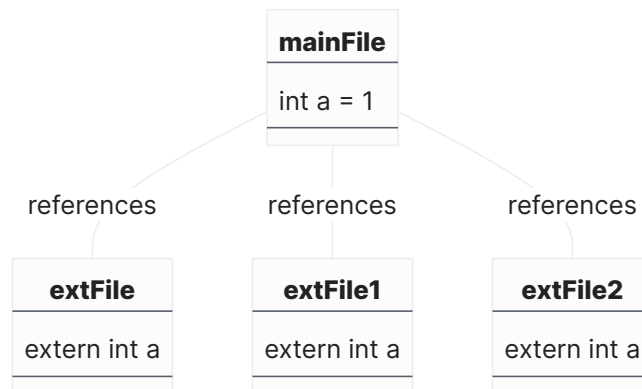
```
enum E : int; // opaque enum declaration
enum E : int { a, b, c, d }; // definition
```

Variabili

- dichiarazione pura

```
extern int a; // dichiarazione pura di variabile (globale)
```

Il compilatore sa che esiste una variabile di tipo `int` a nome `a`, che tuttavia ha forma (verrà creata) in qualche altra parte del codice (unità di traduzione)



```
extern "C"
```

Una situazione simile con l'uso della parola chiave `extern` è il caso in cui volessimo riferirci a una variabile/funzione, dichiarata in un'altra unità di traduzione, che usa la convenzione del linguaggio `"C"`

```
// dichiara variabile globale con C linkage
extern "C" int errno;
```

- definizione

```
int b; // zero-inizialization
int c = 1;
extern int d = 2; // definizione, perché inizializzata
```

Vengono create le variabili accordate, inizializzate se richiesto

Funzioni

- dichiarazione pura

```
void foo(int a);
extern void foo(int a);
```

Commentiamo il fatto che `foo()` ha due dichiarazioni.

Il **numero** e **tipo** di argomenti della funzione, sono identificanti la stessa: questo permette di distinguere univocamente funzioni che hanno lo stesso nome (cosa a cui al compilatore non importa). Detto "bruttamente": le dichiarazioni pure possono ripetersi tante volte, mentre le definizioni sono uniche (**One Definition Rule**).

- definizione

```
void foo(int a) { // presente il corpo, quindi definizione
    std::cout << a;
}
```

Hanno un corpo

Template

- dichiarazione pura

```
template <typename T> struct S;
```

- definizione di template di classe

```
template <typename T>
struct S {
    T t;
};
```

- dichiarazione pura di template di funzione

```
template <typename T>
T add(T t1, T t2);
```

- definizione di template di funzione

```
template <typename T>
T add(T t1, T t2) {
    return t1 + t2;
}
```

Scope

Lo **scope**, o campo d'azione, è il punto del programma in cui una variabile è visibile.

Sapere dove il nome è visibile è importante, ci sono certi casi in cui vogliamo che nomi uguali siano visibili in uno stesso punto (come **overloading**), altri altrimenti.

Scope di namespace

Una dichiarazione di nome, che non sta in classe/struct/parametri di funzione, resta a scope di namespace.

Quando diciamo che il namespace globale esiste, stiamo dicendo qualcosa del tipo:

```
namespace N {
    void foo() { // foo() è visibile da qui fino alla fine di N
        bar(a); // ERROR: chi sono bar() e a ? Non li conosco
    }
    int a; // a è visibile fino alla fine di N
    void bar(int n) {
        a += n; // CORRECT
    }
}
```

Scope di blocco

Porzione di codice racchiusa in parentesi graffe.

- per funzione

```
void foo()
    // codice
    {
        // blocco1
    } // fine scope di blocco
```

- per blocco iterativo

```
for (int i=0; i < 10; ++i) {
    // i in scope del blocco for()
}
```

- per parametro di funzione

```
if (T* ptr = foo()) {
    // ptr qui visibile
} else {
    // ptr anche qui visibile
}
```

Scope di classe

⚠ struct vs class

In C++ possiamo usare degli *specificatori di accesso* (`private`, `public`, `protected`). L'unica differenza tra i due è che lo `struct` è di default `public`, mentre `class` ha accesso `private`.

I membri di una classe (entità dichiarate all'interno della stessa) sono visibili all'interno dell'intera classe, indipendentemente da dove vengano dichiarati.

```
struct S {
    void foo() {
        bar(a); // CORRECT: bar() e a sono visibili ovunque nella classe
    }
    int a;
    void bar(int n) { a += n; }
};
```

🔗 Modi di referenziare oggetti nella classe

- usando l'operatore punto
`s.foo()`
- usando puntatore →
`ps → foo()`
- usando l'operatore di scope
`S::foo()`

Scope di funzione

Lo scope di funzione è diverso dallo scope di blocco. Un esempio è l'operatore `goto`, visibile ovunque nella funzione.

```

void foo() { // scope di funzione
    { // scope di blocco
        inizio:
            // codice
            goto fine;
    }
    fine:
        // altro codice
        goto inizio;
}

```

Scope delle costanti di enumerazione

Come dichiarate per C++ 2003

```
enum Colors { red, blue, green };
```

hanno come scope quello del corrispondente tipo di enumerazione `Colors` (sono visibili fuori dalle parentesi graffe). Questo potrebbe dare problema di *name clash*: 2 oggetti si chiamano `red`, uno con valore diverso dall'altro, che rende la leggibilità pessima.

```

enum Colors { red, blue, green };
enum StopLight { red, green, blue };

void foo() { std::cout << red; } // ERROR: a quale 'red' mi devo riferire?

```

Con lo standard vecchio, per ovviare al problema, si usava introdurre `namespace` sulla nostra costante di enumerazione:

```
namespace Colors { enum Colors_Enum { red, blue, gree }; }
```

Una variante introdotta con C++ 2011, risolve il problema senza rimuovere il supporto per la precedente usando un cast esplicito su `red`:

```

enum class Colors { red, blue, green };

void foo() {
    std::cout << static_cast<int>(Colors::red);
}

```

Hiding

Anche a nome "mascheramento", se ne parla quando una dichiarazione nello scope interno nasconde un'altra dichiarazione con nome uguale dello scope esterno.

```

int a = 1; // globale esterna
int main() {
    std::cout << a << std::endl; // 1
    int a = 5; // interna
    std::cout << a << std::endl; // 5
}

```

Si può avere anche per i membri ereditati da una classe, perché lo scope della classe derivata è considerato essere incluso nello scope della classe base:

```

struct Base {
    int a;
    void foo(int);
}

```

```
};
struct Derived : public Base {
    double a; // hiding di Base::a
    void foo(double d); // hiding del metodo Base::foo()
};
```

Estensioni della visibilità del nome

Dichiarare funzioni con lo stesso nome non sempre è un problema, ma lo può essere

```
// quali di queste funzioni usare in uno scope?
struct Base {
    void foo(int);
    void foo(float);
};
```

La nostra `struct` ha 2 metodi che possono essere utilizzati, ma che con la derivazione della classe non vengono presi in considerazione (niente overloading ma hiding presente). Se volessimo prenderle, e tenerle nel caso ci servissero in secondo momento, usiamo:

```
struct Derived : public Base {
    using Base::foo; // ora visibili tutti i metodi foo()
    void foo(double d); // overloading con foo(int) e foo(float)
}
```

Direttive di `using`

```
void foo() {
    using namespace std;
    cout << "Hello" << endl;
}
```

Nella definizione di `foo()`, il compilatore ha la possibilità di guardare dentro il `namespace std`: quando trova il nome, se dichiarato, lo usa, altrimenti usa la direttiva di `using`. Nell'esempio sotto, viene stampato anche 42 anziché "Hello" e basta.

```
#include <iostream>
void foo() {
    int endl = 42;
    using namespace std;
    cout << "Hello" << endl;
}
```