Lezione2

Table of contents

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) tre 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.

> **Caso `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; // pure declaration
enum E : int { a, b, c, d }; // definition
```

Variabili

- dichiarazione pura

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

- definizione

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

Funzioni

- dichiarazione pura

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

Commentando, `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 (che non importa al compilatore). Le dichiarazioni pure possono ripetersi tante volte, mentre le definizioni uniche (ODR).

- definizione

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

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) {
  t1 + t2;
}
```

Scope

Campo d'azione, è il punto del programma in cui una variabile è visibile.
Sapere dove il nome è visibile è importante, 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:

```
namespace N {
        void foo() { // foo() è visibile da quì fino alla fine di N
                bar(a); // ERRORI: 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 quì visibile
} else {
  // ptr anche quì 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.

```
void foo() { // scope di funzione
        { // scope di blocco
        }
}
```

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.
Con lo standard vecchio, per ovviare al problema, si usava fare

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

Una variante introdotta nel 2011 risolve il problema senza rimuovere il supporto per la precedente: usiamo un cast esplicito su `red`

```
enum class Colors { red, blue, green };
void foo() {
        std::cout << static_cast<int>(Colors::red);
}
```

Hiding

TO:DO

22-02-2023