Dichiarazioni e Definizioni

Dichiarazione vs Definizione

Una DICHIARAZIONE è un costrutto del linguaggio che introduce (dichiara) un nome per una entità (tipo di dato, variabile, funzione, template di classe, template di funzione, ecc.).

Una DEFINIZIONE è una dichiarazione che, oltre al nome, fornisce ulteriori elementi per caratterizzare l'entità (per esempio, la struttura interna di un tipo di dato, l'implementazione del corpo di una funzione, ecc.).

Le dichiarazioni che NON sono anche definizioni vengono talvolta chiamate dichiarazioni "*forward*" (perché rimandano la definizione ad un momento successivo) o anche dichiarazioni "*pure*". Vediamo la differenza tra dichiarazioni pure e definizioni per varie tipologie di entità.

Torna all'indice

Tipi di dato

Dichiarazione pura del tipo `S`:

```
struct S;
```

Definizione del tipo `T`:

```
struct T { int a; };
```

Nel primo caso, non conosco la struttura del tipo `S` e, di conseguenza, non posso creare oggetti del tipo `S` (per esempio, il compilatore non saprebbe quanta memoria allocare per memorizzare un tale oggetto).

Nel secondo caso, siccome `T` è definita, conosco la struttura e posso creare oggetti di tipo `T`.

Ci si potrebbe chiedere quale sia l'utilità di avere una dichiarazione pura di tipo, visto che non si possono creare valori di quel tipo.

In realtà sono utili quando occorre definire puntatori o riferimenti a valori del tipo `T`, senza dover conoscere il tipo `T`; si parla in questo caso di puntatori opachi:

```
struct T;
T* t_ptr; // un puntatore "opaco" a T
```

Ci si potrebbe anche chiedere perché il linguaggio insista, nel caso di dichiarazioni pure, a richiedere che il programmatore dichiari se il nome introdotto è un tipo o un valore (cioè, perché occorre indicare la parola *`struct`* / *`class`* per dire che è un tipo).

La risposta è che questa informazione è essenziale per poter fare il *parsing* (cioè l'analisi sintattica) del codice. Consideriamo infatti una variante dell'esempio precedente:

```
nome1 * nome2
```

Il compilatore che si trova di fronte a questo codice lo può interpretare in due modi completamente differenti:

- `nome2` è dichiarato essere un puntatore al tipo `nome1`, oppure
- si richiede di applicare l'operatore binario `*` ai due valori `nome1` e `nome2`

Se il compilatore vede che `nome1` è un *tipo*, sceglie la prima opzione; se vede che è invece un *valore*, sceglie la seconda.

Torna all'indice

Un caso speciale

Nel caso del C++ 2011 (o successivo), è possibile anche fornire una dichiarazione pura per un tipo *enumerazione* `enum`, cosa che non era possibile fare con il C++ 2003:

```
enum E : int;                   // dichiarazione pura
enum E : int { a, b, c, d};     // definizione
```

Torna all'indice

Variabili

Dichiarazione pura di variabile (globale):

La parola chiave extern vuole dire: Non creare lo spazio per inizializzare la variabile, lo farà qualcun altro

```
extern int a;
```

Definizione di variabile:

```
int b;
int c = 1;
extern int d = 2; // definizione, perchè viene inizializzata
```

Nel caso della dichiarazione pura, il compilatore viene informato dell'esistenza di una variabile di nome `a` e di tipo `int`, ma la creazione di tale variabile verrà effettuata altrove (probabilmente in un'altra unità di traduzione).

Nel caso delle definizioni, invece, il compilatore si occuperà di creare le variabili `b`, `c`, `d` e, se richiesto di inizializzarle.

Torna all'indice

Funzioni

Come identifico una funzione (in ordine di importanza)?

1. Identifico il `namespace`.
2. Identifico il nome.
3. Identifico il numero e tipo di argomenti.
4. Come sono stanziati i parametri del `template`.

Il tipo di ritorno non serve ad identificare una funzione.

Dichiarazioni pure di funzioni:

```
void foo(int a);
//poco usato
extern void foo(int a);
// Sono due ri-dichiarazioni della stessa funzione
```

Definizione di funzione:

```
void foo(int a) {
    std::cout << a;
}
```

> La parola chiave `extern` è opzionale e, in pratica, è usata raramente: *le definizioni hanno il corpo.*

Possiamo anche interfacciarci al mondo interno cambiando le regole di collegamento; in questo caso la funzione `foo` utilizzerà i metodi del C e non C++:

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

Torna all'indice

Template (di classe e di funzione)

I template di classe non sono classi; essi infatti si possono considerare come uno schema, che indica al compilatore come dichiarare e/o definire ciò che segue attorno a dei *tipi parametrici*.

Dichiarazione pura di template di classe:

```
template  struct S;
```

Definizione di template di classe:

```
template
struct S {
    T t;
};
```

Dichiarazione pura di template di funzione:

```
template
T add(T t1, T t2);
```

Definizione di template di funzione:

```
template
T add(T t1, T t2) {
    return t1 + t2;
}
// Il compilatore non sa cosa fare con l'operatore '+'.
// Lo saprà solo quando scoprirà di che tipo saranno le variabili 't1' e 't2'.
```

Torna all'indice