Tipi di dati elementari

Lezione 6 di Programmazione 2

Docente: Giuseppe Scollo

Università di Catania, sede di Comiso (RG)
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Studi in Informatica applicata, AA 2006-7

Indice

1. Tipi di dati elementari
2. tipi di dati
3. tipi di dati elementari in C++
4. conversione fra tipi di numeri in C++
5. tipi di dati definiti dall'utente
6. interfaccia, implementazione e client
7. interfacce per la compilazione separata

tipi di dati

tipo di dati : un insieme di **valori** e una collezione di **operazioni** su di essi

in termini matematici, questo è il concetto di **algebra**

un tipo di dati **concreto** è la rappresentazione di un tipo di dati in una macchina o in un linguaggio di programmazione

per esso di solito si richiede che l'insieme di valori sia **finito**

un tipo di dati **astratto** è caratterizzato da proprietà logiche delle sue operazioni

in termini matematici, questo è il concetto di **classe di algebre**

tutti i linguaggi di programmazione offrono **costrutti di base** per:

tipi di dati **elementari** predefiniti

strutture dati, nelle quali ciascun valore è un aggregato di valori più elementari

alcuni linguaggi di programmazione offrono costrutti per:

tipi di dati **definiti dall'utente**

definizione di **tipi di dati astratti**

tipi di dati elementari in C++

tipi di dati elementari predefiniti in C++:

`int`	numeri interi (in un definito intervallo, v. appresso), con le operazioni aritmetiche elementari
`float`	numeri razionali in virgola mobile, con operazioni aritmetiche in tale rappresentazione
`char`	caratteri (un byte per carattere, codice ASCII)

la rappresentazione degli interi dipende dall'architettura della macchina

rappresentazione più frequente: **in complemento a 2**, l'intervallo è determinato dal numero di bit impiegati per rappresentare un intero: tipicamente 4 byte

in tal caso i tipi `short int` e `long int` rappresentano intervalli determinati dalla rappresentazione con, rispettivamente, 2 byte e 8 byte

anche la rappresentazione in virgola mobile dipende dall'architettura della macchina: il tipo `double` impiega il doppio dello spazio, rispetto al tipo `float`, per la rappresentazione in **doppia precisione**

conversione fra tipi di numeri in C++

ai simboli delle operazioni aritmetiche sui tipi **`int`**, **`float`** e rispettive varianti si applica l'**overloading**:

- il tipo di operazione e del risultato è determinato da quello degli argomenti, quando questi sono dello stesso tipo
- se gli argomenti sono di tipo diverso, si ha una **conversione di tipo (casting)** implicita al tipo più complesso (**`int`** → **`float`**, **`float`** → **`double`**, etc.)
- il **casting esplicito** è dunque talvolta necessario, ad es. quando si desidera avere il quoziente razionale di due numeri interi (altrimenti se ne otterrebbe il quoziente intero, approssimato per difetto):

```
((float) x) / N
```

l'operatore prefisso **`(float)`** converte l'argomento di tipo **`int`** nel tipo **`float`**, dopodiché anche **N** viene convertito a **`float`** per casting implicito

tipi di dati definiti dall'utente

il testo (Sedgewick, 2003) propone un programma per il calcolo della media e della varianza di **N** numeri (Programma 3.2, p. 75)

nel programma si assume che i numeri siano interi, e che l'input sia costituito solo dal valore di **N**, mentre gli **N** numeri interi vengono quindi generati pseudocasualmente dalla funzione di libreria **rand()**

per rendere il programma più facilmente adattabile al calcolo, con lo stesso algoritmo, su numeri di altro tipo, si usa il seguente costrutto per la definizione del tipo **Number** con la funzione di generazione pseudocasuale **randNum()** :

```
typedef int Number;
Number randNum()
  { return rand(); }
```

questo esempio mostra un tipo di dato definito dall'utente come **ridenominazione** di un tipo esistente (dunque con lo stesso insieme di valori) ed **estensione** delle operazioni con una nuova funzione

interfaccia, implementazione e client

la precedente definizione di tipo ne combina

- l'**interfaccia**, che contiene solo **dichiarazioni** di funzioni, e
- l'**implementazione**, che fornisce **definizioni** per le funzioni dell'interfaccia

è buona norma di metodo

- **separare l'interfaccia dalla sua implementazione**

al fine di svincolare l'**uso** delle funzioni dichiarate nell'interfaccia, da parte di programmi o moduli **client**, dalla loro implementazione: **l'uso richiede solo l'interfaccia!**

in tal modo si può sostituire un'implementazione con un'altra, ad esempio più efficiente, senza dover modificare i client

nel caso in esame, si possono codificare interfaccia e implementazione in due file distinti, ad es. **Number.h** e **NumberImpl.c++**, con i seguenti rispettivi contenuti:

```
typedef int Number;
Number randNum();
```

```
#include <stdlib.h>
#include "Number.h"
Number randNum()
  { return rand(); }
```

interfacce per la compilazione separata

la separazione di interfaccia e implementazione di un tipo di dato definito dall'utente permette di modificarne l'implementazione

- non solo senza modifiche al **sorgente** dei client del tipo di dato
- ma anche senza la loro **ricompilazione (!)**

se si effettua la **compilazione separata** di client e implementazione del tipo di dato

come si fa? supponiamo, nell'esempio visto prima, che il client sia il programma codificato nel file **MedVar.c++**, che conterrà la direttiva **`#include "Number.h"`**

con il compilatore GCC in ambiente Unix/Linux, i primi due comandi che seguono rispettivamente generano i moduli oggetto dell'implementazione e del client (con estensione **`.o`**), senza produrre alcun programma eseguibile (opzione **`"-c"`**), mentre il terzo comando assembla i moduli oggetto e genera l'eseguibile **MedVar** nella directory **bin**

```
c++ -c NumberImpl.c++
c++ -c MedVar.c++
c++ -o bin/MedVar NumberImpl.o MedVar.o
```