Tipo dato concreto

Che cos'è?

In C++, un tipo di dato concreto è un tipo di dato che rappresenta un valore effettivo memorizzato in un'area di memoria specifica. Ciò significa che un tipo di dato concreto ha un'allocazione di memoria definita e fissa e che il valore può essere modificato tramite un puntatore o un riferimento a tale area di memoria. [[14.2-esercizio_lifetime]] Ad esempio, un tipo di dato concreto in C++ potrebbe essere un `int`, un `float` o un `char`. Questi tipi di dati hanno un'allocazione di memoria fissa e possono essere modificati direttamente utilizzando un puntatore o un riferimento.

D'altra parte, un tipo di dato astratto in C++ rappresenta un concetto astratto, come un oggetto o una collezione di oggetti, e non ha un'allocazione di memoria fissa. Ad esempio, una classe definita dall'utente è un tipo di dato astratto poiché rappresenta un'entità concettuale e non ha un'allocazione di memoria fissa. Tuttavia, un oggetto di una classe ha un'allocazione di memoria fissa e può essere considerato come un tipo di dato concreto.

In sintesi, i tipi di dato concreti in C++ rappresentano valori effettivi che sono memorizzati in una specifica area di memoria e possono essere modificati direttamente.

Torna all'indice

La classe Razionale

Un esempio di dato concreto è la classe Razionale. Prima della definizione dei metodi bisogna stabilire quali funzioni si vogliano rendere disponibili all'utente: per fare ciò si sfruttano dei *"test"* che fungeranno da linee guida per la scrittura dei metodi.

Per compilare:

```
g++ -std=c++11 -Wall -Wextra -o testRazionale.o -c testRazionale.cc
```

In seguito alla compilazione si andranno a riscontrare degli errori che fungeranno da *"todo list"* per quello che c'è da fare.

Un esempio di primo errore è l'assenza del file `Razionale.hh` in `Razionale.cc`.

> Con `-std=c++11` si intende lo standard C++ del 2011.

Torna all'indice

Razionale.hh

Definiamo l'header file, nel quale sarà presente il namespace `Numerica` contenente la classe `Razionale`:

```
#ifndef NUMERICA_RAZIONALE_HH_INCLUDE_GUARD
#define NUMERICA_RAZIONALE_HH_INCLUDE_GUARD 1

namespace Numerica {
 class Razionale {
   Razionale() = default;
   Razionale(const Razionale&) = default;
   Razionale(Razionale&&) = default;
   Razionale& operator=(const Razionale&) = default;
   Razionale& operator=(Razionale&&) = default;
   ~Razionale() = default;

   using Intero = long;
   Razionale(Intero n, Intero d = 1);
 }; // end class Razionale
} // end nameapace Numerica

#endif NUMERICA_RAZIONALE_HH_INCLUDE_GUARD
```

Maggiori informazioni nel file `./code/Razionale.hh`

Torna all'indice

---

testRazionale.cc

Definiamo ora invece il file `.cc` contenente il test. Come prima cosa dobbiamo includere l'header file:

```
#include "Razionale.hh"
```

Successivamente creiamo una funzione `test01` di tipo `void`, in cui andremo a fare i nostri test. Ora importiamo il namespace `Numerica`:

```
using Numerica::Razionale;
```

Ora inseriamo i vari tipi di costruttori:

```
Razionale r; // costruttore default
Razionale r1(r); // costruttore di copia
Razionale r2 = r; // costruttore di assegnamento
Razionale r3 { r }; // costruttore di copia (C++11)
Razionale r4 { r }; // costruttore di copia (C++11)
Razionale r5(1, 2); // costruzione diretta
Razionale r6 {1, 2}; // costruzione diretta (C++11)
Razionale r7(1); // costruzione diretta
Razionale r8{1}; // costruzione diretta (C++11)
Razionale r9 = 1234; // costruzione implicita (da evitare!), fa una conversione da int a long e da long a Razionale
Razionale r10 = true; // Fa la stessa cosa. Bisogna usare la parola chiave 'explicit' davanti al costruttore
```

I tipi di assegnamento:

```
r = r1; // assegnamento di copia
r = Razionale(1); // assegnamento per spostamento
r2 = r1 = r; // concatenazione assegnamenti (da evitare), r2 = (r1 = r) -> r2 = r1
```

Gli operatori aritmetici:

```
// Binari
r1 = r + r;
r1 = r - r;
r1 = r * r;
r1 = r / r;

// Unari
r1 = -r;
r1 = +r;
```

Gli operatori di assegnamento:

```
r += r;
r -= r;
r *= r;
r /= r;

// Incremento e decremento (pre o post)
++r;
r1 += ++r;
r++;
--r;
r--;
```

Gli operatori relazionari:

```
bool b;
b = (r == r);
b = (r != r);
b = (r < r);
b = (r <= r);
b = (r >= r);
```

```
b = (r > r);
```

E infine le operazioni di input e di output:

```
std::cin >> r;
std::cin >> r >> r1;
std::cout << r << std::endl;
```

> Maggiori informazioni nel file `./code/testRazionale.cc`

Torna all'indice

Problema: le conversioni implicite

Le conversioni implicite in C++ si verificano quando il compilatore converte automaticamente un valore di un tipo di dato in un altro tipo di dato senza che sia necessario esplicitare una conversione tramite un cast.

Un esempio:

```
void foo(int a, bool b);
void bar(){
 foo(3.7, true);
 foo(true, 3.7);
}
```

In questo caso, alla prima chiamata della funzione `foo(3.7, true)` si presenta una conversione implicita di un `float (3.7)` ad un intero. Nella seconda chiamata `foo(true, 3.7)` si presenta una conversione implicita di un booleano ad un intero, e di un float ad un booleano.

È importante notare che le conversioni implicite possono portare a comportamenti indesiderati o a errori di runtime se non si tiene conto delle regole di conversione. Pertanto, è consigliabile evitare le conversioni implicite quando possibile e utilizzare invece le conversioni esplicite tramite cast quando necessario.

Vediamo un esempio della classe Razionale:

```
void foo(const Razionale& r);
void bar(){
 foo(true);
}
```

Questo non rispetta il principio di minima sorpresa.

> Le linea guida della progettazione semplice è il dover risolvere un problema alla volta. Se bisogna risolvere più problemi, prima li risolvo indipendentemente e poi li unisco.

Torna all'indice