Università degli Studi di Bergamo
Facoltà di Ingegneria dell'Informazione

Informatica III

Elaborato 3: C++

Revisione	Data	Redatto
1.0	2006/02/09	Silvio Moioli

Indice dei contenuti

Storia delle revisioni

Rev. n	Data	Redatto	Descrizione
1.0	2006/02/08	Silvio Moioli	Versione iniziale.

Introduzione

Scopo del documento

Presentazione del terzo elaborato del corso, piccolo programma C++ con uso dei costrutti avanzati visti a lezione.

Definizioni, abbreviazioni e sigle

- UML: Unified Modeling Language;
- STL: Standard Template Library;

Riferimenti bibliografici e Web

- http://www.cplusplus.com/reference/, sito con approfondita documentazione sulle librerie standard;
- http://www.parashift.com/c++-faq-lite/ , sito con approfondimenti sull'uso avanzato del C++;

Approccio seguito in questo elaborato

L'elaborato consiste in una libreria di oggetti per il calcolo numerico scritta in linguaggio C++, creata per provare i costrutti di C++ visti a lezione: ereditarietà, programmazione generica, STL, operator overload e così via.

Presentazione del programma

Funzionalità

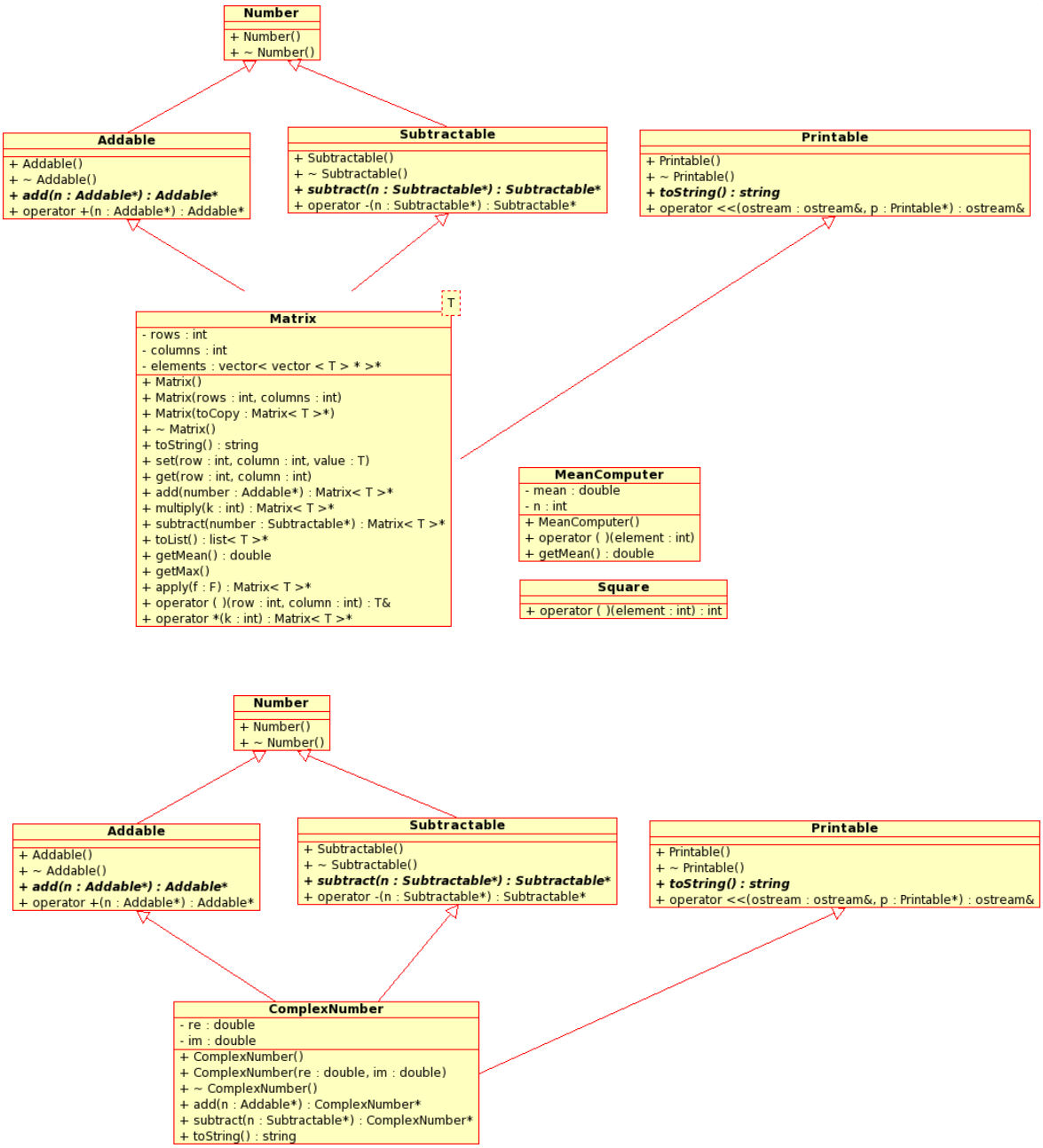
L'oggetto di questo elaborato è una libreria per il calcolo numerico che attualmente consiste in tre moduli: matrici, numeri complessi e un programma dimostrativo. La libreria utilizza i template e l'ereditarietà come meccanismi chiave per l'espandibilità futura.

Classi principali

Il programma è suddiviso nelle seguenti classi:

- Number (file `Number.cpp`, `Number.h`): la classe base di tutti i numeri;
- Addable (file `Addable.cpp`, `Addable.h`): classe dei numeri sommabili. Questa classe ha un metodo virtuale puro, `add()`, e implementa l'operatore `+`;
- Subtractable (file `Subtractable.cpp`, `Subtractable.h`) classe dei numeri sottraibili. Questa classe ha un metodo virtuale puro, `subtract()`, e implementa l'operatore `-`;
- Printable (file `Printable.h`, `Printable.cpp`) classe degli oggetti stampabili. Questa classe ha un metodo virtuale puro, `toString()`, e implementa l'operatore `<<`;
- ComplexNumber (file `ComplexNumber.cpp`, `ComplexNumber.h`) classe dei numeri complessi, una possibile implementazione di Number;
- Matrix (file `Matrix.h`, `Matrix.cpp`) classe delle matrici, ammette qualunque tipo di elemento grazie al template. E' sottoclasse di Addable, Subtractable e Printable;
- MeanComputer (file `MeanComputer.h`, `MeanComputer.cpp`) una classe per calcolare la media di più elementi, nella forma utile per l'algoritmo `for_each`, è utilizzata da Matrix;
- Square (file `Square.h`, `Square.cpp`) una classe che calcola il quadrato di un numero, nella forma utile per il metodo apply() di Matrix che applica una funzione a tutti gli elementi della matrice.

Seguono due diagrammi UML che mostrano le relazioni tra le classi.



Esempi dei costrutti utilizzati

Vengono qui di seguito riportati alcuni esempi dei costrutti utilizzati nel programma.

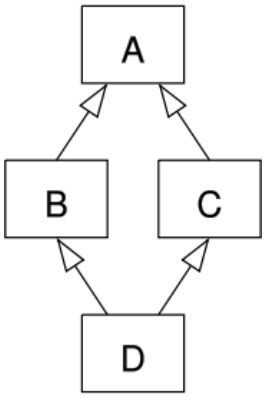
Ereditarietà multipla

Rispetto a Java, il linguaggio C++ è più permissivo sull'ereditarietà fra classi. Innanzitutto, è possibile ereditare da più classi contemporaneamente, come è ben visibile dai diagrammi UML riportati nel paragrafo precedente. Inoltre, ad ogni relazione di ereditarietà si associa anche un qualificatore di visibilità, che stabilisce la visibilità dei metodi e delle variabili della classe padre nella classe figlio. Ad esempio, Matrix eredita in modo pubblico da tre classi:

```
/**
 * Rappresenta una matrice di elementi a priori generici.
 */
...
class Matrix: public Addable, public Subtractable, public Printable{...
```

Ereditarietà virtuale – diamond problem

Uno dei problemi che sorgono con l'introduzione dell'ereditarietà multipla è il cosiddetto "diamond problem" che si verifica quando una classe D eredita da due classi B e C che hanno un antenato in comune A. In questo caso c'è un'ambiguità su quale "versione" dell'antenato deve essere ereditata da D: quella di B o quella di C?



Un modo di risolvere il problema in C++ è dichiarare che le classi B e C ereditano da A "virtualmente", specificando il fatto nella dichiarazione nel modo seguente (esempio tratto dalla classe Addable):

```
/**
 * Rappresenta oggetti che possono essere sommati a numeri.
 * E' necessario implementare il metodo add().
 */
class Addable: virtual public Number {...
```

In questo modo, al momento dell'istanziazione di un oggetto D il compilatore sa che dovrà mantenere un'istanza di A, non due copie, che dovrà essere condivisa.

Metodi virtuali

In C++, una sottoclasse può sempre ridefinire i metodi della sopraclasse, ma il dynamic binding non avviene se non si usa la parola chiave virtual (a differenza di Java dove avviene sempre). Pertanto, ogni metodo che viene (o potrebbe essere) ridefinito in una sottoclasse va dichiarato come virtuale. Un caso particolare è il distruttore, che se non fosse virtuale non permetterebbe la distruzione di tutti gli oggetti appartenenti alle sottoclassi.

Segue un esempio tratto dalla classe Number.

```
class Number{
public:
    /** Distruttore di default. */
      virtual ~Number();
...
```

Metodi virtuali puri

Similmente a quanto accade in Java con le interfacce, anche in C++ è possibile dichiarare metodi e lasciare la loro implementazione alle sottoclassi. Questi metodi devono essere, ovviamente, virtuali e prendono il nome di "virtuali puri". La sintassi per dichiararli è insolita e prevede un "assegnamento al metodo" di cui qui sotto è riportato un esempio da Addable:

```
/**
 * Rappresenta oggetti che possono essere sommati a numeri.
```

```
 * E' necessario implementare il metodo add().
 */
class Addable: virtual public Number {
public:
    ...
    /** Implementa la somma tra due oggetti. */
      virtual Addable* add(Addable* n)=0; ...
```

I metodi virtuali puri possono essere usati esattamente come gli altri, le classi contenenti metodi virtuali puri non possono però essere istanziate (si può, al più, dichiarare dei puntatori).

Operator overloading

Il C++, a differenza del C, mette a disposizione l'operator overloading, un costrutto che permette di modificare il significato degli operatori con metodi arbitrari. Le classi Addable e Subtractable sono state pensate per rendere semplice questa operazione, esse infatti implementano l'operator overloading utilizzando un metodo virtuale puro. Seguono le parti di codice rilevanti della classe Subtractable:

```
/**
 * Rappresenta oggetti che possono essere sottratti a numeri.
 * E' necessario implementare il metodo add().
 */
class Subtractable: virtual public Number {
public:
    /** Implementa la differenza tra due oggetti. */
      virtual Subtractable* subtract(Subtractable* n)=0;
      /** Ridefinizione dell'operatore. */
      virtual Subtractable* operator-(Subtractable* n);
};

Subtractable* Subtractable::operator-(Subtractable* n){
    return this->subtract(n);
}
```

Overloading di `cout<<`

Tra gli altri operatori che è utile sovraccaricare c'è `<<`, che è utilizzato nella libreria standard per inviare dati agli oggetti `ostream` (flussi in output) come il ben noto `cout`. La sintassi è leggermente diversa dal caso precedente (esempio tratto da `Printable.cpp`):

```
ostream& operator<<(ostream &ostream, Printable* p){
     return ostream<<p->toString();
}
```

Overloading di `()` e `[]`

Gli operatori `()` e `[]` sono leggermente diversi dagli altri, in quanto è possibile interpretarli in due modi a seconda che compaiano in un l-value o un r-value. Ad esempio, il significato usuale di `[]` è

l'indicizzazione degli array, ma questa può significare un assegnamento o una lettura da una cella dell'array:

```
int[] a = {1, 2, 3};
a[1] = 0; //l'operatore [] indica un assegnamento ad array
int c = a[0]; //l'operatore [] indica una lettura da array
```

In C++ è possibile sovraccaricare gli operatori in modo che si comportino in maniera diversa a seconda della posizione esattamente come accade con gli array. La sintassi, in questo caso, è un po' più criptica[1]: ecco un esempio tratto da Matrix che ridefinisce `()`:

```
T& operator() (int row, int column) const;
T& operator() (int row, int column);
```

Il simbolo `&` posto dopo il tipo ritornato indica che ci sono due definizioni dell'operatore, e il modificatore const indica che la prima è quella da usare se l'operatore sta a sinistra del simbolo di assegnamento. Questi operatori permettono scritture del tipo:

```
Matrix m;
m(0,0) = 3;
cout<<m(0,0);
```

Note:

- in Matrix si è deciso di ridefinire `()` anziché `[]` perchè quest'ultimo è limitato ad un solo parametro; in altre parole in C++ la scrittura `m[1,2]` non è valida mentre `m(1,2)` lo è;
- le implementazioni dei due metodi è identica, entrambi ritornano l'indirizzo dell'elemento voluto.

Covarianza dei metodi

In C++, come in Java dalla versione 5 in poi, è permesso che un metodo di una sottoclasse possa ridefinire il corrispettivo della classe padre cambiando il tipo ritornato con un sottotipo (i valori ritornati sono covarianti). Questo meccanismo è utilizzato nella classe `ComplexNumber`, che ridefinisce i metodi `add()` e `subtract()`:

```
class Addable: virtual public Number {
public:
    ...
      virtual Addable* add(Addable* n)=0; ...
}
class Subtractable: virtual public Number {
public:
    ...
      virtual Subtractable* subtract(Subtractable* n)=0; ...
}
class ComplexNumber : public Addable, public Printable, public Subtractable{
public:
    ...
      virtual ComplexNumber* add(Addable* n);
      virtual ComplexNumber* subtract(Subtractable* n);
```

[1] http://www.parashift.com/c++-faq-lite/operator-overloading.html#faq-13.10

```
}
```

Template - definizione

Il C++ implementa la "programmazione generica" attraverso il meccanismo dei template. I template permettono di definire funzioni, classi e metodi che possono operare su dati di tipo diverso e non specificato al momento della loro stesura. Questo permette di riusare lo stesso codice con dati di diversi tipi primitivi o classi, anche senza ricorrere all'ereditarietà.

Ad esempio, è possibile definire un metodo (o funzione) con tipo parametrico T in questo modo (esempio tratto da `Matrix.h`):

```
/**
 * Rappresenta una matrice di elementi a priori generici.
 */
template<class T> class Matrix: public Addable, public Subtractable, public Printable{
    public:
        /** Modifica un elemento. */
        void set(int row, int column, T value);
```

Il tipo T viene definito solo al momento dell'uso della classe (istanziazione) con la seguente sintassi (`main.cpp`):

```
Matrix<int>* m = new Matrix<int>(5,5);
```

Template - limitazioni del compilatore

La maggior parte dei compilatori C++[2], tra cui il g++, non permettono classica divisione di definizioni e implementazioni in file diversi (con estensioni `.cpp` e `.h`, rispettivamente). Questo è dovuto a una carenza architetturale del compilatore, che processa soltanto un file alla volta: il risultato pratico sono errori del linker anche se la sintassi è corretta. Si è costretti quindi a mettere tutto il codice in un unico file `.h`, che ha almeno due svantaggi:

1. viene copiato ad ogni inclusione, spesso accrescendo la dimensione dell'eseguibile;
2. è meno chiaro e ordinato della soluzione su due file.

Purtroppo non esiste una vera soluzione per il primo punto, mentre è comunque possibile risolvere il secondo con alcuni accorgimenti nelle direttive al preprocessore.

In pratica, con lo stesso meccanismo che impedisce che il file `.h` contenente le definizioni venga incluso più volte, è possibile fare in modo che il file `.cpp` contenente le implementazioni venga incluso una sola volta, al termine delle definizioni stesse. Questo approccio è stato seguito nell'implementazione della classe Matrix.

Il file `Matrix.h` è strutturato come segue:

```
#ifndef MATRIX_H_
#define MATRIX_H_

#include <sstream>
...
using namespace std;
```

[2] http://www.parashift.com/c++-faq-lite/templates.html#faq-35.12

```
//dichiarazione della classe template
template<class T> class Matrix:..{
    ....
}
//è come se le implementazioni fossero qui
#include "Matrix.cpp"
#endif /*MATRIX_H_*/
```

Il file `Matrix.cpp`, invece:

```
#ifndef MATRIX_CPP_
#define MATRIX_CPP_

#include "Matrix.h"

//implementazioni dei metodi qui
template<class T> Matrix<T>::Matrix(){ ...

#endif /*MATRIX_CPP_*/
```

STL – vector

La libreria standard del C++ include una parte, chiamata STL, che fornisce "contenitori" di oggetti generici e algoritmi per operare su di essi utilizzando pesantemente i template. Uno dei contenitori più semplici è `vector`, una classe che permette di gestire collezioni di oggetti ordinati come un array. I vantaggi del `vector` sul semplice array sono diversi, tra cui la gestione automatica della memoria e la presenza di diversi metodi per le operazioni più comuni quali l'inserimento di un nuovo valore.

La classe Matrix utilizza un `vector` per memorizzare gli elementi della matrice ed essendo parametrica di parametro T, utilizza un `vector< vector <T> >`.

Dichiarazione:

```
template<class T> class Matrix: public Addable, public Subtractable, public
Printable{
    public:
        ...
    private:
        /** Numero di righe della matrice. */
        int rows;
        /** Numero di colonne della matrice. */
        int columns;
        /** Contenitore degli elementi. */
        vector< vector<T>* >* elements;
}
```

Utilizzo: inizializzazione di una matrice 1x1:

```
template<class T> Matrix<T>::Matrix(){
    this->rows = 1;
```

```
    this->columns = 1;
    this->elements = new vector< vector<T>* >(1);
    (*this->elements)[0] = new vector<T>(1);
    (*this)(0,0) = 1;
}
```

STL – iteratori

Tra le altre funzionalità messe a disposizione da `vector` e dagli altri contenitori in STL ci sono gli iteratori, che costituiscono il modo standard di accedere agli elementi nel contenitore stesso. Gli iteratori si comportano in modo simile a semplici puntatori (ad esempio ammettono l'operatore `++`), anche se in realtà sono più flessibili e possono essere utilizzati anche con le liste.

Ecco un esempio di come è possibile scorrere gli elementi di un vettore utilizzando gli iteratori:

```
typename vector<vector<T>*>::iterator rowsIterator = this->elements->begin();
while (rowsIterator != this->elements->end()){
    cout<<*rowsIterator;
}
```

Si noti l'utilizzo della parola chiave `typename` che è necessaria per non introdurre ambiguità per il compilatore.

STL – list

Un altro contenitore, concettualmente simile a vector, è `list`. In STL, list è implementato come una lista concatenata anche se l'utilizzo è del tutto simile a vector grazie agli iteratori. Il metodo `toList()` di Matrix, ad esempio, converte gli elementi della matrice in una `list`:

```
template<class T> list<T>* Matrix<T>::toList(){
    list<T>* result = new list<T>();
        typename vector<vector<T>*>::iterator rowsIterator = this->elements->begin();
    while (rowsIterator != this->elements->end()){
        typename vector<T>::iterator columnsIterator = (*rowsIterator)->begin();
        while (columnsIterator != (*rowsIterator)->end()){
            result->push_back(*columnsIterator);
            columnsIterator++;
        }
        rowsIterator++;
    }
    return result;
}
```

STL – algoritmi

Come scritto nel paragrafo precedente, la libreria STL mette a disposizione anche degli algoritmi generici e riutilizzabili per operare sui contenitori. Tali algoritmi sono implementati tramite funzioni template che si possono includere nei propri file tramite la direttiva `#include <algorithm>`. Un esempio di algoritmo è la funzione `for_each()`, che applica una funzione parametro a tutti gli elementi di un contenitore. In Matrix `for_each()` è utilizzato per calcolare la media degli elementi nel metodo `getMean()` e il codice che esegue il calcolo è incapsulato nella classe MeanComputer.

Questo è necessario poiché, per calcolare la media, è necessario mantenere lo stato della somma e il conteggio degli elementi. MeanComputer ridefinisce l'operatore `()` per poter essere utilizzato in `for_each()`:

```
/**
 * Una classe che calcola la media di un'insieme di elementi, creata
 * per l'uso con l'algoritmo for_each della STL.
 */
class MeanComputer{
public:
    /** Costruttore di default. */
    MeanComputer();
    /** Ridefinizione dell'operatore, aggiorna
     * la media con un nuovo elemento. */
      void operator() (int element);
      /** Ritorna la media corrente. */
      double getMean();
private:
    /** La media attualmente calcolata. */
    double mean;
    /** Il numero di elementi su cui è calcolata (utile per l'aggiornamento). */
    int n;
};
```

Il metodo `getMean()` è implementato come segue:

```
template<class T> double Matrix<T>::getMean(){
    MeanComputer mc;
        typename vector<vector<T>*>::iterator rowsIterator = this->elements->begin();
    while (rowsIterator != this->elements->end()){
        mc = for_each((*rowsIterator)->begin(), (*rowsIterator)->end(), mc);
        rowsIterator++;
    }
    return mc.getMean();
}
```

Un altro esempio è l'implementazione di `getMax()` che usa l'algoritmo `max_element`:

```
template<class T> T Matrix<T>::getMax(){
    list<T>* elements = this->toList();
    T result = *max_element(elements->begin(), elements->end());
    delete elements;
    return result;
}
```