Laboratorio II – 1° modulo

Lezione 7

Template e
Standard Template Library

Indice

- Funzioni e classi template:
  - Esempio di funzioni template
  - Esempio di una classe template (*SimpleArray*)
- La Standard Template Library (STL) del C++
- Introduzione all'utilizzo delle standard libraries
  - `std::vector`
  - `std::string`
- Esercizi

Overloading

Nella scorsa lezione abbiamo visto il concetto di overloading in diversi esempi:

- overloading dei costruttori: diversi costruttori di una classe con sequenze di inizializzazione (cioè parametri di input) differenti.

```
//default constructor
istogramma();

//costruttore
istogramma(int Nbin, float min, float max);

//copy constructor
istogramma(const istogramma& original);
```

- overloading degli operatori: è possibile ridefinire gli operatori (+, -, *, /, =, …) in modo che possano eseguire determinate (e differenti) istruzioni a seconda del *tipo* di oggetto su cui sono applicati

Es1:
```
//operator=
istogramma& operator=(const istogramma& original);
```

Es2:
```
complesso complesso::operator+(complesso& other const)
complesso complesso::operator+(double& other const)
```

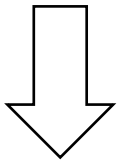
Overloading

E ancora...

- overloading di funzioni: è possibile creare funzioni (o anche metodi di una classe) con lo stesso nome, ma parametri di input differenti (in numero, sequenza o tipo), che implementano istruzioni differenti. Ad esempio:

```
#include <iostream>
int raddoppia (int a)
{
  return a*2;
}
float raddoppia (float a)
{
  return a*2;
}
int main()
{
    int A=2;
    float B=2.22;
    std::cout<<raddoppia(A)<<std::endl;
    std::cout<<raddoppia(B)<<std::endl;
    return 0;
}
```

La funzione `raddoppia` è implementata sia per gli `int`, sia per i `float`



L'overloading ci consente di usare lo stesso nome per funzioni che si applicano a *tipi* diversi, però una medesima funzione (es. raddoppia) deve essere **implementata per ogni tipo**

Template

- Spesso però le funzioni, anche se hanno parametri di tipo differente, di fatto eseguono la stessa sequenza di istruzioni.
- Ad esempio, la funzione raddoppia, prende il parametro di input (`int` o `float`) e lo moltiplica per 2. Il C++, sa già come deve eseguire la moltiplicazione per 2 a seconda che il numero sia `int` o `float`…
- Possiamo evitare di dover implementare più volte la stessa funzione, quando l'unica differenza nel codice è il tipo di variabile?
  - Sì, attraverso la definizione dei TEMPLATE

Nel file `.h`

```
template <typename GenericType>
GenericType raddoppia (GenericType a) {
  return a*2 ;
}
```

- Definisco un tipo generico (cioè un *template*) che chiamo `GenericType`
- Nel codice della funzione `GenericType` prende il posto del tipo di variabile

Template: come funziona

File `.h`

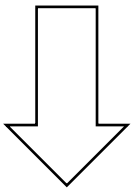
```
template <typename GenericType>
GenericType raddoppia (GenericType a) {
  return a*2 ;
}
```

File `.cpp`

```
#include <iostream>
#include "lib.h"

int main () {
  int A=2;
  float B=2.22;
  std::cout<<raddoppia(A)<<std::endl;
  std::cout<<raddoppia(B)<<std::endl;
  return 0 ;
}
```

- Quando compiliamo il codice, durante la fase del *parsing*, il compilatore analizza il codice e "*si accorge*" che nel main abbiamo chiamato la funzione `raddoppia` in due circostanze: una volta passando un `int` e una volta passando un `float`
- Il compilatore costruisce l'effettiva implementazione delle due diverse funzioni raddoppia, sostituendo a `GenericType` una volta `int` e una volta `float`



Per questo motivo le funzioni "template" devono essere implementate direttamente nel file `.h`

Template: come funziona

Per questioni di ordine del codice si potrebbe procedere anche in questo modo:

File .h

```
#ifndef simpleArr_h
#define simpleArr_h

template <class T>
class SimpleArray
{
 public:
  SimpleArray(const int& elementsNum);
  ~SimpleArray();

  T& element(const int& i);
  T& operator[](const int& i);

 private:
  int elementsNum_p;
  T* elements_p;
};

#include "SimpleArray.tcc"
#endif
```

File .tcc

```
template <typename T>
SimpleArray<T>::SimpleArray(const int& elementsNum)
{
  elementsNum_p=elementsNum;
  elements_p = new T[elementsNum];
}

template <typename T>
SimpleArray<T>::~SimpleArray()
{
  delete[] elements_p;
}

template <typename T>
T& SimpleArray<T>::element(const int& i)
{
  if(i<elementsNum_p) return elements_p[i];
  else return elements_p[elementsNum_p - 1];
}

template <typename T>
T& SimpleArray<T>::operator[](const int& i)
{
  if(i<elementsNum_p) return elements_p[i];
  else return elements_p[elementsNum_p - 1];
}
```

Template: un altro esempio

File `.h`

```
template <typename GenericType>
GenericType somma (GenericType a, GenericType b) {
  return a+b ;
}
```

File `.cpp`

```
int main () {

  double d1 = 1.2 ;
  double d2 = 3.14 ;

  std::cout << "somma di double: "<< d1 << " + " << d2 << " = ";
  std::cout << somma<double> (d1,d2) << std::endl ;

  int i1 = 1 ;
  int i2 = 5 ;

  std::cout << "somma di int:"<< i1 << " + " << i2 << " = ";
  std::cout << somma (i1,i2) << std::endl ;

/* questo da' errore!
  std::cout << "somma mista: "  << somma (i1,d2) << std::endl ;*/

// questo  invece si può fare !!!
  std::cout << "somma mista: "<< i1 << " + " << d2 << " = ";
  std::cout << somma<double> (i1,d2) << std::endl ;
```

Con questa sintassi esplicito voglio fare la somma tra `double`

Qui non esplicito il tipo, ma, dato che passo due numeri interi, il compilatore "capisce" che voglio fare la somma tra interi

Qui non esplicito il tipo, e, dato che passo un `int` e un `double` c'è ambiguità → **dà errore**

Qui esplicito `somma<double>`, quindi **`i1`** verrà convertito a **`double`** (cast) prima di sommarlo a **`d2`**

Le classi *template*

Anche le classi possono essere template

- `template <typename T> …`
- `template <class T> …`

È la stessa istruzione e l'uso di `typename` o `class` qui è equivalente

- Utile per creare “strumenti di lavoro” più complessi di quelli offerti dal C++
- Esempio: un array a dimensione variabile (come quello che si crea con `new`) che si distrugge da solo alla fine dell’esecuzione che possa contenere `int`, `float`, `double`, oppure anche tipi generici di una classe che ci siamo inventati (es: `istogramma`, `complesso` …)

La classe *SimpleArray* (.h)

```
#ifndef simpleArr_h
#define simpleArr_h

template <class T>
class SimpleArray
{
  public:
  //costruttore
  SimpleArray(const int& elementsNum)  {
    elementsNum_p=elementsNum;
    elements_p = new T[elementsNum];
  }
  //distruttore
  ~SimpleArray()  {
     delete[] elements_p;
  }
  //metodo che restituisce un elemento dell'array
  T& element(const int& i)  {
     if(i<elementsNum_p) return elements_p[i];
     else return elements_p[elementsNum_p - 1];
  }
  //overloading operator[]
  T& operator[](const int& i)  {
     if(i<elementsNum_p) return elements_p[i];
     else return elements_p[elementsNum_p - 1];
  }

  private:
  int elementsNum_p;
  T* elements_p;
};
#endif
```

Definizione della classe template in cui **T** è utilizzato per indicare il tipo generico di un semplice array di elementi

Il metodo `element` restituisce la referenza di un elemento dell'array (e permette quindi di leggerne e/o modificarne il contenuto)

Possiamo anche realizzare l'overloading dell'`operator[]` (che fa esattemente la stessa cosa di `element`) per poter accedere agli elementi dell'array con una sintassi più intuitiva e convenzionale

Se passo in input un indice maggiore del numero di elementi dell'array, per evitare errori formali, restituisco l'ultimo elemento dell'array (si può anche aggiungere un messaggio di errore)

La classe *SimpleArray* (.cpp)

```
#include <iostream>
#include "SimpleArray.h"

int main()
{
  int elementi = 10;

  //array semplice di int
  SimpleArray<int> arrayInt(elementi);
  std::cout << " Array riempito con int " << std::endl;
        //riempimento
  for(int i=0; i<elementi; ++i)
    arrayInt.element(i) = 2*i;
        //rilettura
  for(int i=0; i<elementi; ++i)
    std::cout << " elemento " << i << " = " << arrayInt.element(i) << std::endl;

  //array semplice di float
  SimpleArray<float> arrayFloat(elementi);
  std::cout << "\n Array riempito con float " << std::endl;
        //riempimento
  for(int i=0; i<elementi; ++i)
    arrayFloat.element(i) = 0.5 + arrayInt.element(i);
        //rilettura
  for(int i=0; i<elementi; ++i)
    std::cout << " elemento " << i << " = " << arrayFloat[i] << std::endl;

  return 0 ; // chiama automaticamente il distruttore
}
```

- La classe è inclusa come al solito
- Il tipo da utilizzare per ogni oggetto è indicato con `<int>` (o `<float>`)
- L'oggetto si definisce ed utilizza come un qualunque altro oggetto
- L'elemento dell'array è restituito dal metodo `element(const int& i)` oppure con `operator[]`

La classe *SimpleArray* (.cpp)

```cpp
#include <iostream>
#include "SimpleArray.h"

int main()
{
  int elementi = 10;

  //array semplice di int
  SimpleArray<int> arrayInt(elementi);
  std::cout << " Array riempito con int " << std::endl;
        //riempimento
  for(int i=0; i<elementi; ++i)
    arrayInt.element(i) = 2*i;
        //rilettura
  for(int i=0; i<elementi; ++i)
    std::cout << " elemento " << i << " = " << arrayInt.element(i) << std::endl;

  //array semplice di float
  SimpleArray<float> arrayFloat(elementi);
  std::cout << "\n Array riempito con float " << std::endl;
        //riempimento
  for(int i=0; i<elementi; ++i)
    arrayFloat.element(i) = 0.5 + arrayInt.element(i);
        //rilettura
  for(int i=0; i<elementi; ++i)
    std::cout << " elemento " << i << " = " << arrayFloat[i] << std::endl;

  return 0 ; // chiama automaticamente il distruttore
}
```

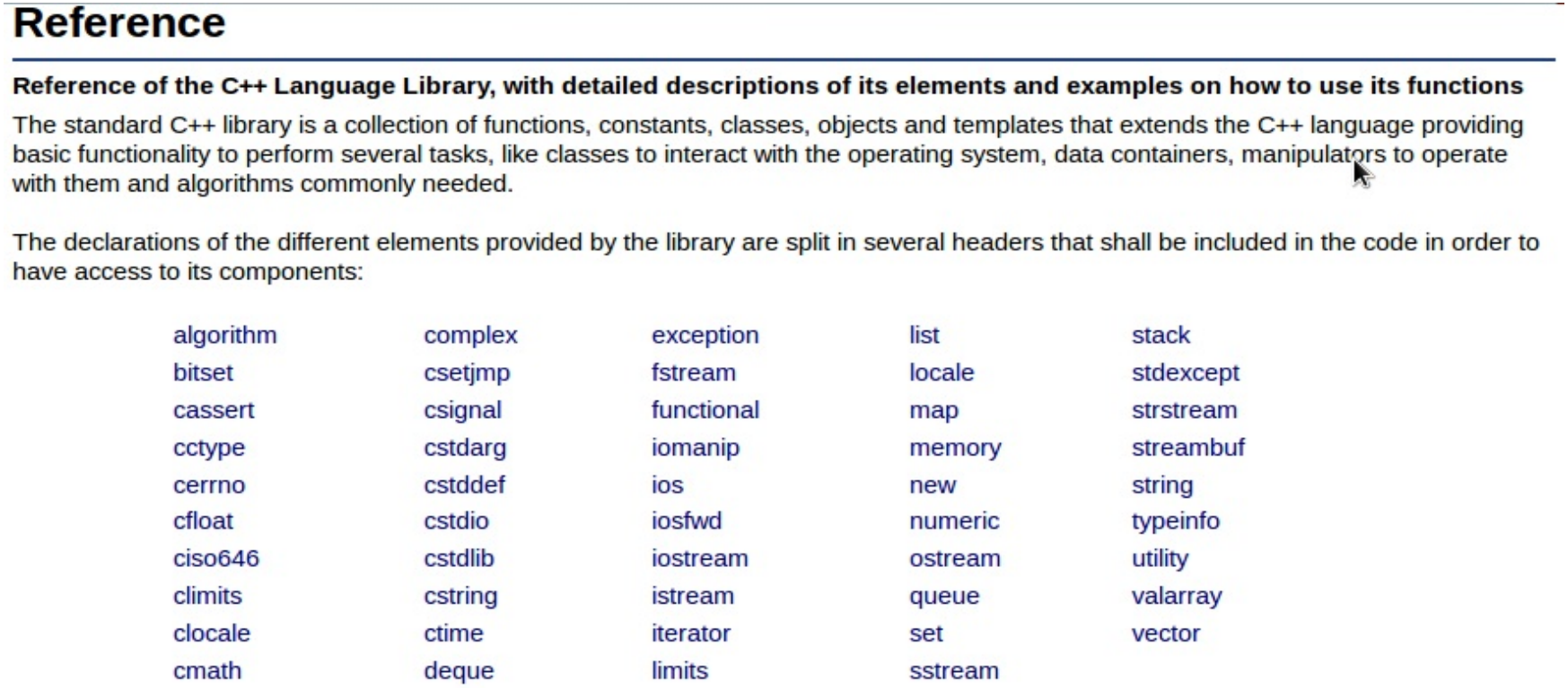
Output a terminale

```
Array riempito con int
elemento 0 = 0
elemento 1 = 2
elemento 2 = 4
elemento 3 = 6
elemento 4 = 8
elemento 5 = 10
elemento 6 = 12
elemento 7 = 14
elemento 8 = 16
elemento 9 = 18

Array riempito con float
elemento 0 = 0.5
elemento 1 = 2.5
elemento 2 = 4.5
elemento 3 = 6.5
elemento 4 = 8.5
elemento 5 = 10.5
elemento 6 = 12.5
elemento 7 = 14.5
elemento 8 = 16.5
elemento 9 = 18.5
```

Standard Template Library (STL)

- La Standard Template Library è un insieme di classi ormai divenute fondamentali per i programmatori C++
- In queste librerie sono contenute diverse classi per che consentono di gestire diversi tipologie di oggetti in maniera generale e standardizzata (ad esempio stringhe di caratteri, array di elementi, mappe associative, …)
- Come si utilizzano → www.cplusplus.com/reference/



Reference

Reference of the C++ Language Library, with detailed descriptions of its elements and examples on how to use its functions

The standard C++ library is a collection of functions, constants, classes, objects and templates that extends the C++ language providing basic functionality to perform several tasks, like classes to interact with the operating system, data containers, manipulators to operate with them and algorithms commonly needed.

The declarations of the different elements provided by the library are split in several headers that shall be included in the code in order to have access to its components:

algorithm	complex	exception	list	stack
bitset	csetjmp	fstream	locale	stdexcept
cassert	csignal	functional	map	strstream
cctype	cstdarg	iomanip	memory	streambuf
cerrno	cstddef	ios	new	string
cfloat	cstdio	iosfwd	numeric	typeinfo
ciso646	cstdlib	iostream	ostream	utility
climits	cstring	istream	queue	valarray
clocale	ctime	iterator	set	vector
cmath	deque	limits	sstream	

std::vector

- È un array di elementi allocati dinamicamente (come il nostro SimpleArray), ma la sua dimensione, cioè il numero di elementi che contiene non è fissato a priori
- Si riempie con il metodo `std::vector::push_back()`: ogni volta che chiamo questo metodo viene aggiunto un elemento in coda all'array
- Si possono leggere/modificare i suoi elementi con l'`operator[]`
- Esistono anche altri metodi, tutti documentati (anche con semplici esempi)

  http://www.cplusplus.com/reference/vector/vector/
- La gestione della memoria è ottimizzata e sicura
- Si possono definire `vector` per oggetti di ogni tipo (es: `vector<complesso>`, `vector<istogramma>` ...)

  Attenzione: `std::vector` copia gli oggetti il meno possibile, ma lo fa, quindi, se utilizziamo `vector` per gestire oggetti di classi che abbiamo scritto noi, è necessario che siano definiti il copy constructor e l'`operator=`

std::vector

```
#include <iostream>
#include <vector>

int main ()
{
  // Ddefinisco un vector
  std::vector<double> MyVector;

  // Riempio il vector
  for (int i=0 ; i<10 ; ++i)
    MyVector.push_back(i * 3.14);

  // Richiedo la dimensione del vector
  std::cout << "Numero elementi = ";
  std::cout << MyVector.size() << std::endl;

  // Posso accedere agli elementi del vector con []
  for (int i = 0; i < MyVector.size(); i++)
    {
      std::cout <<"Elemento "<<i << ": ";
      std::cout << MyVector[i] << std::endl;
    }

  return 0;
}
```

Ricordate di includere la classe `vector`

L'oggetto di tipo `vector` viene creato templetizzato sul tipo che contiene

L'oggetto `MyVector`, di tipo `vector<double>` è qui definito utilizzando il default constructor (quindi inizialmente è un array con zero elementi)

Con il metodo `push_back` riempio il `vector`

Il metodo `size()` restituisce il numero degli elementi contenuti nel `vector`

Rileggo gli elementi del `vector` con l'`operator[]`

std::string

Gli oggetti di tipo `string` non sono semplici array di caratteri (come, ad es: `char parola[]="ciao"`) ma si usano per gestire stringhe di caratteri in maniera ottimizzata, mettendo a disposizione diversi metodi per manipolarle

Esistono infatti diversi operatori che si possono utilizzare sulle stringhe: `operator+` per concatenare diverse stringhe, `operator=`, …

Alcuni metodi della classe `std::string`

- `const char* c_str()`: per convertire una string del C++ in un array di caratteri (utile se ad esempio vogliamo usare metodi che accettano in input array di caratteri)
- `int compare(const string& str)`: per confrontare se due stringhe sono uguali
- `int find(const string& str)`: per trovare caratteri o sequenze di caratteri in una stringa
- `int length()`: restituisce il numero di caratteri di una stringa
- `char& operator[] (int pos)`: restituisce il carattere in posizione "pos"

http://www.cplusplus.com/reference/string/string/

std::string

File .cpp

```
#include <string>
#include <iostream>
int main()
{
  std::string frase = "W la pizza";

  int lunghezza = frase.length();
  std::cout << "Lunghezza frase: " <<
    lunghezza << std::endl;

  const char* fraseC = frase.c_str();
  std::cout << fraseC << std::endl;

  std::string parola ("pizza");
  int posizione = frase.find(parola);
  std::cout << "La parola " << parola <<
    " si trova in posizione " << posizione << std::endl;

  std::string parola2 ("pozza");
  int confronto = parola2.compare(parola);
  if (confronto == 0) std::cout << "Le due parole sono uguali" << std::endl;
  else std::cout << "Le due parole sono diverse" << std::endl;

  for (int i = 0; i < parola2.length(); i++)
    std::cout << "Carattere " << i << ": " << parola2[i] << std::endl;

  return 0;
}
```

Output a terminale

```
Lunghezza frase: 10
W la pizza
La parola pizza si trova in posizione 5
Le due parole sono diverse
Carattere 0: p
Carattere 1: o
Carattere 2: z
Carattere 3: z
Carattere 4: a
```

Istanziazione nella memoria dinamica

- Fate attenzione alla seguente sintassi:

```
std::vector<int>* myVec; // Istanzio puntatore a vettore di interi
int x = 10;
myVec->push_back(x); // A questa istruzione il programma vi dara` un errore perche` non avete ancora chiamato il costruttore di vector
```

- La sintassi corretta e`:

```
std::vector<int> myVec = new std::vector<int>(); // Istanzio puntatore a vettore di interi e lo inizializzo con il costruttore del vector
int x = 10;
myVec->push_back(x);
```

- Oppure senza unsare la memoria dinamica:

```
std::vector<int> myVec;
int x = 10;
myVec.push_back(x);
```

Attenzione:
Se definite un puntatore ad una classe NON state automaticamente chiamando il suo costruttore. Se oltre a istanziare il puntatore ad una classe lo volete anche inizializzare la sintassi generare e`:

```
tipo* nome = new tipo(...);
```

Esercizi

- **Esercizio 1:** Completare l'esercizio sulla classe template SimpleArray, implementando i seguenti metodi:
  - default constructor e copy constructor
  - `int size ()`: che restituisce la dimensione dell'array
  - `void push (T elem)`: che inserisce un nuovo elemento in coda all'array
  - `T pop (int i)`: che estrae l'i-esimo elemento dell'array e di conseguenza ridimensiona l'array "shiftando" di una posizione gli elementi successivi per eliminare il "buco" rimasto

  Nel `main`, dopo aver definito un `SimpleArray<int>` vuoto, impostare un ciclo `while`, in cui si chieda all'utente di scegliere quale operazione svolgere (`push` o `pop`) in base al valore assegnato ad una variabile `char`: inserisci (`i`), togli (`t`), esci (`e`). Dopo aver provato a inserire e togliere alcuni numeri interi, eseguire un ciclo `for` in cui viene stampata la lista dei numeri salvata in array (utilizzare il metodo `size` per impostare l'istruzione di arresto del ciclo)

- **Esercizio 2:** Modificare il `main` dell'esercizio precedente per gestire l'inserimento/rimozione di caratteri `char` al posto che numeri `int`

Esercizi

- **Esercizio 3:** Scrivere un programma che richieda all'utente di inserire il proprio nome, lo memorizzi in una variabile `string` e quindi usi questa variabile per salutare l'utente con un messaggio “Ciao, NOME!”
- **Esercizio 4:** Scrivere un programma che, data una stringa, restituisca un conteggio delle diverse lettere in essa presenti. Si consiglia di usare un array con 26 posizioni, una per ogni lettera dell'alfabeto, inizializzato tutto a zeri. Dopodiché bisogna scorrere le lettere che compongono la stringa e incrementare il contatore della lettera corrispondente
- **Esercizio 5:** Utilizzando la classe `complesso` implementata nella scorsa lezione scrivere un programma che chieda all’utente di inserire da tastiera *N* numeri complessi (cioè coppie di numeri corrispondenti a *Re* e *Im*). Salvare i numeri complessi in un oggetto di tipo `std::vector<complesso>`. Il ciclo `while` in cui si chiede di inserire i numeri, si deve arrestare quando si inserisce la coppia (0, 0). Al termine del ciclo, calcolare la somma di tutti i numeri inseriti e stamparla a terminale