

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

Abbiamo visto il concetto di overloading in diversi contesti:

- **overloading dei costruttori**: diversi costruttori di una classe con sequenze di inizializzazione 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 operazioni adatte al **tipo** di oggetto a cui sono applicati

Es 1: `istogramma& istogramma::operator=(const istogramma& original);`

Es 2:

```
complesso& complesso::operator= (const complesso&
                                    original);
compleSSO& complesso::operator= (const double& original);
```

Overloading

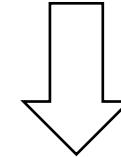
- **overloading di funzioni:** è possibile creare funzioni (o anche metodi di una classe) con lo stesso nome, ma parametri di input differenti (in numero, sequenza o tipo), che implementano istruzioni differenti. Ad esempio:

```
#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`, che per i `float`



L'overloading consente di usare lo stesso nome per funzioni che si applicano a tipi diversi, però una **medesima funzione** 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 compilatore C++ sa già come deve eseguire la moltiplicazione per 2 a seconda che il numero sia `int` o `float`
- È possibile evitare di dover implementare più volte la stessa funzione attraverso la definizione dei **TEMPLATE**

File `lib.h`

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

- Definisce un tipo generico (cioè un *template*) chiamato `genericType`
- Nello scope della funzione `genericType` prende il posto del tipo di variabile

Template: come funziona

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

File **lib.h**

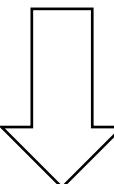
Utilizzo nel **main**

```
#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;
}
```

Durante la fase di compilazione il compilatore “si accorge” che nel `main` è stata chiamata la funzione `raddoppia` in due modi: una volta passando un `int` e una volta passando un `float`

Il compilatore costruisce così 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: un altro esempio

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

Utilizzo nel **main**

```
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;
```

File **lib.h**

- Qui non si esplicita il tipo, ma, dato che vengono passati due numeri interi, il compilatore “capisce” che si vuole fare la somma tra interi

- Qui non si esplicita il tipo, e, dato che vengono passati un **int** e un **double** c’è ambiguità → **errore**

- Qui si esplicita **somma<double>**, quindi **i1** verrà convertito a **double** (cast) prima di sommarlo a **d2**

Le classi template

Anche le classi possono essere *templetizzate*

```
template <typename T> ... }  
template <class T> ... }
```

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

- Utili 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, e che può contenere `int`, `float`, `double`, oppure anche tipi generici come una classe da noi scritta (e.g. `istogramma`, `Complesso`, ...)

La classe SimpleArray (.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;
};

```

Definizione della classe template in cui **T** è utilizzato per indicare il tipo generico di degli elementi dell'array

Il metodo **element** restituisce la referenza ad un elemento dell'array, permettendo così di leggerne e/o modificarne il contenuto

Overloading dell'**operator[]** (fa esattamente la stessa cosa di **element**) per poter accedere agli elementi dell'array con una sintassi più intuitiva e convenzionale

Se viene passato in input un indice maggiore del numero di elementi dell'array → restituisce l'ultimo elemento dell'array (si può anche aggiungere un messaggio di errore)

La classe SimpleArray (.cpp)

```
#include <iostream>
#include "SimpleArray.h" ←
#define N_ELEM 10

int main()
{
    int elementi = N_ELEM;
    // 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 e si 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)

```
#include <iostream>
#include "SimpleArray.h"
#define N_ELEM 10

int main()
{
    int elementi = N_ELEM;
    // 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
```

Template: header ed implementazione

Per questioni di ordine del codice si potrebbero separare i prototipi dalla loro implementazione nel modo seguente:

```
#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 lib.h

File lib.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];
}
```

Standard Template Library (STL)

La Standard Template Library è una libreria che contiene diverse classi ormai divenute fondamentali per i programmatori C++

Queste classi consentono di gestire diversi tipologie di oggetti in maniera **generale** e **standardizzata**:

- array di elementi: std::vector
- stringhe di caratteri: std::string
- mappe associative: std::map
- coppie di contenitori: std::pair
- operazioni bit a bit: std::bitset

...

Per vedere come si utilizzano potete andare sul sito →

www.cplusplus.com/reference/

The screenshot shows the 'Reference' section of the cplusplus.com website. The left sidebar lists categories: C library, Containers, Input/Output, Multi-threading, and Other. The main content area is titled 'C Library' and lists various C headers with their descriptions. Below this is the 'Containers' section, which lists various container classes like array, bitset, deque, etc. At the bottom is the 'Input/Output Stream Library' section, which provides a relationship map between <iostream>, <ios>, <istream>, <iostream>, <ifstream>, and <sstream>.

Header	Description
<cassert> (assert.h)	C Diagnostics Library (header)
<cctype> (ctype.h)	Character handling functions (header)
<cerrno> (errno.h)	C Errors (header)
<cenv> (env.h)	Floating-point environment (header)
<cfenv> (float.h)	Characteristics of floating-point types (header)
<cinttypes> (inttypes.h)	C Integer types (header)
<iso646> (iso646.h)	ISO 646 Alternative operator spellings (header)
<climits> (limits.h)	Sizes of integral types (header)
<clocale> (locale.h)	C localization library (header)
<cmath> (math.h)	C numerics library (header)
<ceiljmp> (setjmp.h)	Non local jumps (header)
<csignal> (signal.h)	C library to handle signals (header)
<cstdarg> (stdarg.h)	Variable argument handling (header)
<cstdlib> (stdlib.h)	Boolean type (header)
<cstddef> (stddef.h)	C Standard definitions (header)
<cstdint> (stdint.h)	Integer types (header)
<cstdio> (stdio.h)	C library to perform Input/Output operations (header)
<cstdlib> (stdlib.h)	C Standard General Utilities Library (header)
<cstring> (string.h)	C Strings (header)
<ctgmath> (tgmath.h)	Type-generic math (header)
<ctime> (time.h)	C Time Library (header)
<cuchar> (uchar.h)	Unicode characters (header)
<cwchar> (wchar.h)	Wide characters (header)
<cwctype> (wctype.h)	Wide character type (header)

Container	Description
<array>	Array header (header)
<bitset>	Bitset header (header)
<deque>	Deque header (header)
<forward_list>	Forward list (header)
<list>	List header (header)
<map>	Map header (header)
<queue>	Queue header (header)
<set>	Set header (header)
<stack>	Stack header (header)
<unordered_map>	Unordered map header (header)
<unordered_set>	Unordered set header (header)
<vector>	Vector header (header)

Input/Output Stream Library
Provides functionality to use an abstraction called *streams* specially designed to perform input and output operations on sequences of character, like files or strings.
This functionality is provided through several related classes, as shown in the following relationship map, with the corresponding header file names on top:

```

graph LR
    ios_base --> istream
    istream --> iostream
    iostream --> ifstream
    ifstream --> istringstream
    
```

std::vector

- È un array di elementi allocati dinamicamente, ma la sua dimensione, cioè il numero di elementi che contiene, non è fissato a priori. Funziona infatti come una **lista concatenata**
- Si riempie con il metodo `std::vector::push_back()`: ogni volta che si chiama 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): www.cplusplus.com/reference/vector/vector/
- La gestione della memoria è ottimizzata e sicura
- Si possono definire `std::vector` per oggetti di ogni tipo (es: `vector<complesso>`, `vector<istogramma>`, ...)

Attenzione: `std::vector` può fare uso del copy constructor e dell'`operator=` se necessario. Quindi, se si utilizzano `std::vector` per gestire oggetti di classi che abbiamo scritto noi, è necessario che queste due operazioni siano definite nelle nostre classi

std::vector

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

int main ()
{
    // Definisco 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;
}
```

La classe è inclusa come al solito

L'oggetto di tipo `std::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 l'`std::vector`

Il metodo `size()` restituisce il numero degli elementi contenuti in `std::vector`

Rilegge gli elementi dell'`std::vector` con l'`operator[]`

std::string

Gli oggetti di tipo `string` non sono semplici array di caratteri (come, e.g. `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=`, ... www.cplusplus.com/reference/string/string/

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 si vogliono 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`

std::string

```
#include <iostream>
#include <string>
int main() {
    std::string frase;
    std::cout << "Inserisci una frase: ";
    getline(std::cin, frase);

    const char* fraseC = frase.c_str();
    std::cout << "Frase: " << fraseC << std::endl;
    int lunghezza = frase.length();
    std::cout << "Lunghezza frase: " << lunghezza << 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;
}
```

Metodo della classe string per ricevere una stringa di caratteri da tastiera

Output a terminale

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

N.B.: per i `char` si usano apici singoli:

```
char a = 'A';
```

Mentre per le `std::string` si usano apici doppi:

```
std::string s = "ciao";
```

Uso della memoria dinamica e invocazione del costruttore

Fare attenzione alla seguente sintassi:

```
std::vector<int>* myVec; // Puntatore a vettore di interi  
int x = 10;  
myVec->push_back(x);
```

A questa istruzione il programma darà **errore** perché non è stato invocato il costruttore della classe

La sintassi corretta è:

```
std::vector<int>* myVec = new std::vector<int>(); // Puntatore  
a vettore di interi con chiamata del costruttore  
int x = 10;  
myVec->push_back(x);
```

Oppure senza usare
l'allocazione dinamica:

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

Attenzione: quando si definisce un puntatore ad una classe NON si sta automaticamente chiamando il suo costruttore
Se oltre ad istanziare il puntatore si vuole anche chiamare il costruttore la sintassi generare è:
`tipo* nome = new tipo(...);`

Esercizi

Esercizio 1: Completare la classe template `SimpleArray`, implementando i seguenti metodi:

- default constructor e copy constructor
- `int size()` → restituisce la dimensione dell'array
- `void push(const T& elem)` → inserisce un nuovo elemento in coda all'array
- `T pop(int i)` → 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 un array (utilizzare il metodo `size` per impostare l'istruzione di arresto del ciclo)

Esercizi

Esercizio 2: 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 3: Utilizzando la classe `complesso` implementata nella scorsa lezione scrivere un programma che chieda all'utente di inserire da tastiera \mathbb{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 sul terminale

Esercizi

Esercizio 4: Modificare la soluzione dell'esercizio 6 nella lezione 2 templetizzando la funzione di ordinamento. Utilizzarla nel main con un array di caratteri, di interi e di double

Suggerimento: per convertire `int` in `char` si può scrivere così

```
int num = 65; // Il 65 corrisponde al carattere "A"  
char a = char(num);  
  
std::cout << a << std::endl; \\ Scrive "A" sullo schermo
```