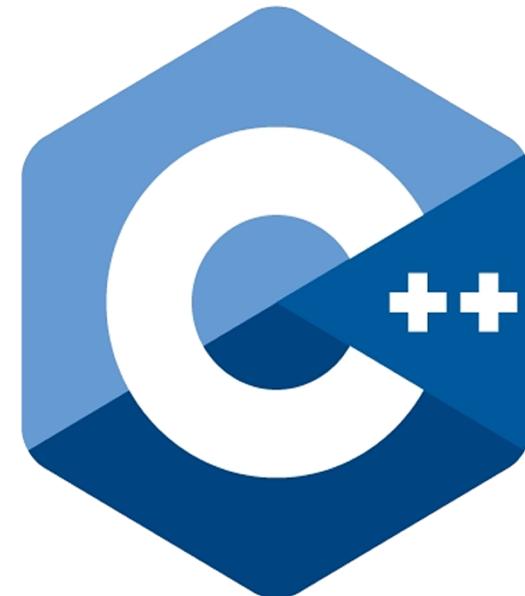




# Programmazione generica in C++

Templates

Giulio Angiani  
I.I.S. "Blaise Pascal" - Reggio Emilia





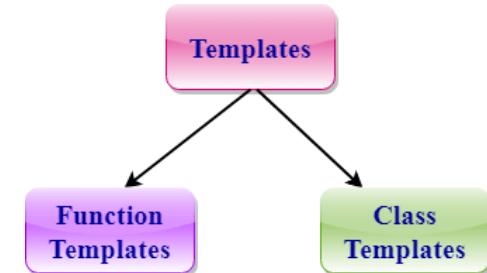
# Template functions



# Cos'è una "funzione template"?

Scopriamolo con un esercizio...

- Scrivere una funzione **void scambia(int, int)** che riceve due variabili intere e ne scambia il valore



```
void scambia_int(int &a, int &b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main() {  
    int x=1;  
    int y=2;  
    scambia_int(x,y);  
    cout << x << ":" << y << endl;  
}
```

C++

2:1

OUTPUT



# Cos'è una "funzione template"?

E un altro...

- Scrivere una funzione **void scambia(float, float)** che riceve due variabili float e ne scambia il valore

```
void scambia_float(float &a, float &b) {  
    float tmp = a;  
    a = b;  
    b = tmp;  
}
```

C++

E un altro...

- Scrivere una funzione **void scambia(string, string)** che riceve due variabili string e ne scambia il valore

```
void scambia_string(string &a, string &b) {  
    string tmp = a;  
    a = b;  
    b = tmp;  
}
```

C++



# Cos'è una "funzione template"?

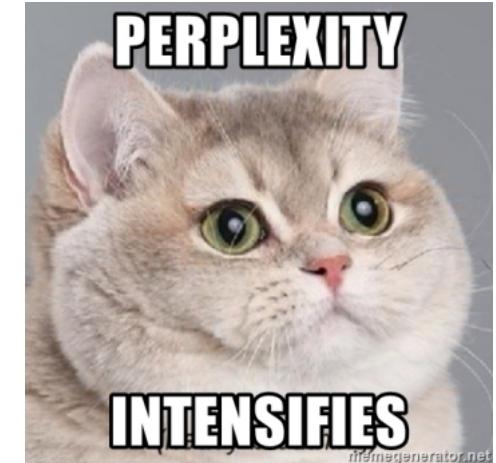
e un altro...

- Scrivere una funzione **void scambia(double, double)** che riceve due variabili double e ne scambia il valore

Sono tutte uguali!!

Cambia solo il **tipo** !

ne servirebbe una **generica**...



```
void scambia_double(double &a, double &b) {  
    double tmp = a;  
    a = b;  
    b = tmp;  
}
```

C++

Possiamo fare di meglio....



# Programmazione generica

C++ mette a disposizione la sintassi per creare funzioni **generiche**

Per queste si deve specificare il comportamento **senza** preoccuparsi del tipo.



```
template<typename T>
return_type function_name(T var_name, <var_list>) {
    code...
}
```

C++

Unica attenzione:

- tutte le operazioni all'interno della funzione devono essere legali rispetto al tipo



# Programmazione generica : scambio

Riscriviamo la nostra funzione **scambio** con la programmazione generica

```
template<typename T>
void scambia(T& x, T& y) { // per riferimento
    T tmp = x;
    x = y;
    y = tmp;
}

int main() {
    int a = 2; int b = 3;
    scambia(a, b);
    cout << " A = " << a << " B = " << b << endl;
    float c = 2.5; float d = 3.6;
    scambia(c, d);
    cout << " C = " << c << " D = " << d << endl;
}
```

C++

A = 3 B = 2  
C = 3.6 D = 2.5

OUTPUT



# Programmazione generica : esercizio

Scrivere una funzione generica **occorrenze** (template function) che riceve - un array A di elementi di tipo T - un elemento E di tipo T - un intero pari alla dimensione dell'array A e restituisce un numero intero delle occorrenze di E in T

La funzione deve essere utilizzabile con il seguente codice

```
int vettint[ ] {1,2,3,3,2,3,4,3};  
int ocl = occorrenze(vettint, 3, 8);  
cout << "3 presente " << ocl << " volte\n";  
  
string s[ ] {"ciao", "come", "state", "come", "oggi", "come", "ieri"};  
string s1 = "come";  
int oc2 = occorrenze(s, s1, 7);  
cout << "come presente " << oc2 << " volte\n";
```

C++

3 presente 4 volte  
come presente 3 volte

OUTPUT



# Programmazione generica : soluzione

```
template<typename T>
int occorrenze(T x[], T elem, int size) {    // il passaggio per array è sempre per riferimento
    int cont = 0;
    for (int i = 0; i<size; i++) {
        if (x[i] == elem) cont++;
    }
    return cont;
}
```

C++

La programmazione **generica** ci consente di passare un array di elementi di tipo qualsiasi e un elemento dello stesso tipo da confrontare.



# Programmazione generica : esercizio

Progettare una **lista** di elementi generici e relativa funzione di inserimento in testa e di visualizzazione

Ricordiamo che in C++ il concetto di **struct** è diverso da quello di **funzione**

Una struct è una **classe** di elementi simili.

Usiamo il costrutto **template<class T>** invece di "template<typename T>"

```
template<class T>
struct nodo
{
    T info;
    nodo<T>* next;
};
```

C++

*NOTA: nel campo "next" va subito specificato il tipo "generico" del nodo*



# Programmazione generica : soluzione

Scriviamo la funzione **generica** per l' inserimento in testa

```
template<typename T>
void inTesta(nodo<T>* &p, T elem) {

    nodo<T>* tmp = new nodo<T>;
    tmp->info = elem;
    tmp->next = p;
    p = tmp;
}
```

C++

e per la lettura

```
template<typename T>
void leggi(nodo<T>* p) {
    while(p) {
        cout << p->info << " ";
        p = p->next;
    }
}
```

C++



# Programmazione generica : soluzione

Possiamo utilizzare adesso una lista contenente qualsiasi tipo con la sintassi in esempio

```
nodo<int>* intptr;
nodo<string>* strptr;

int main()
{
    // inizializzazione
    intptr = nullptr;
    inTesta(intptr, 10);
    inTesta(intptr, 20);
    leggi(intptr);

    strptr = nullptr;
    inTesta(strptr, (string)"molto");
    inTesta(strptr, (string)"figo");
    leggi(strptr);

}
```

C++

20 10 figo molto

OUTPUT



# Programmazione generica

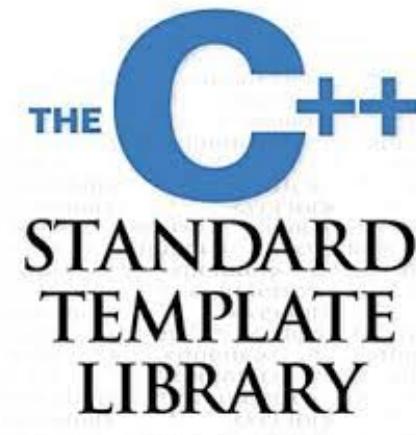
Per fortuna in C++ esiste già un insieme di strutture dati e funzioni che permette la programmazione generica e l'uso delle strutture dinamiche generiche

"La **Standard Template Library** (STL) è una libreria software inclusa nella libreria standard del linguaggio C++ e definisce strutture dati generiche, iteratori e algoritmi generici"<sup>1</sup>

STL mette a disposizione il concetto di **contenitore** che si sviluppa in

- Insiemi
- Liste
- Code
- Vector
- Alberi
- Mappe

e molto altro fra algoritmi e funzioni per lavorare su queste strutture dati



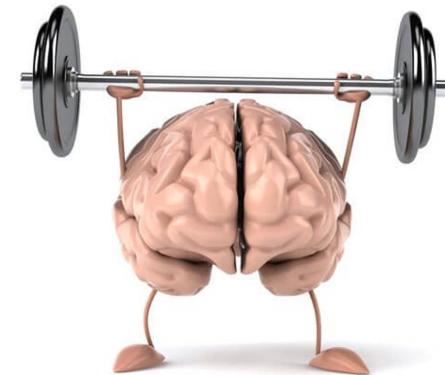
1) Rif. [https://it.wikipedia.org/wiki/Standard\\_Template\\_Library](https://it.wikipedia.org/wiki/Standard_Template_Library)



# Programmazione generica

Un po' di esercizi sulla **template programming**

- Scrivere la funzione generica **void stampaArray()** che
  - riceve un array di elementi di tipo e un intero che indica la dimensione dell'array
  - stampa il contenuto (è permesso il **cout** in questa funzione)
- Scrivere la funzione generica **void ordina()** che
  - riceve un array di elementi di tipo , un intero che indica la dimensione dell'array, un carattere **tipo** che può valere 'A' o 'D'
  - Se il parametro 'tipo' vale 'A' ordina l'array in senso **crescente** altrimenti in senso **decrescente**
  - Testare il funzionamento del programma con la funzione **stampaArray** sopra definita
- Scrivere la funzione generica **somma()** che
  - riceve in ingresso un array di elementi di tipo qualsiasi
  - restituisce il risultato
  - suggerimento: la somma di due interi è intera... la somma di due stringhe è stringa...





Giulio Angiani  
I.I.S. "Blaise Pascal" - Reggio Emilia