Capitolo 13

Template

pag. 341-357

Presenta: Prof. Misael Mongiovì

paradigmi di programmazione

- **programmazione procedurale** (efficienza)
  - tipi, variabili, puntatori, struct, funzione come "scatola nera"
- **programmazione modulare** (dalle procedure ai dati)
  - moduli, in cui i dati sono "occultati" (*data hiding*)
  - namespace
- **programmazione a oggetti** (dai moduli agli oggetti)
  - *data hiding* dentro gli oggetti
  - overload degli operatori
  - eredità e polimorfismo
- **programmazione generica**

algoritmi indipendenti dal tipo

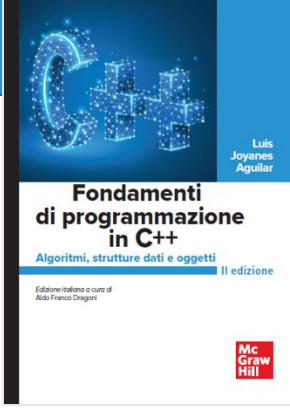
```
void scambia(int &x, int &y)
{
    int t = x;
    x = y;
    y = t;
}
```

```
int x, y;
scambia(x,y)
```

```
void scambia(tipovar &x, tipovar &y)
{
    tipovar t = x;
    x = y;
    y = t;
}
```

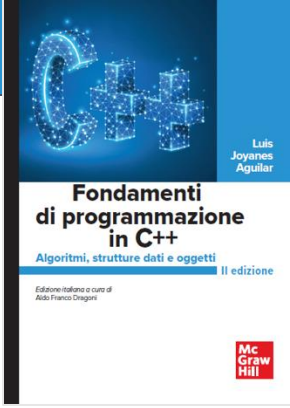
```
void scambia(char &x, char &y)
{
    char t = x;
    x = y;
    y = t;
}
```

```
char x, y;
scambia(x,y)
```



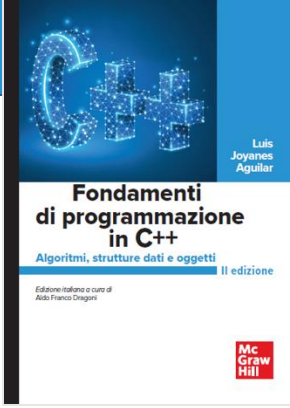
programmazione generica

- algoritmi astratti: indipendenti dal tipo di dato
- definire classi/funzioni senza specificarne tipo dei membri/parametri
- C++, Java, Ada ... forniscono i ***templates***:
  - classi *generiche* (o *parametriche*) indipendenti dal tipo degli elementi da processare
  - tipi istanziati dall'utente (staticamente, alla chiamata)
- Libreria Standard del C++ mette a disposizione strutture precostituite di *classi contenitore*
  - liste concatenate
  - mappe
  - vettori
  - ...

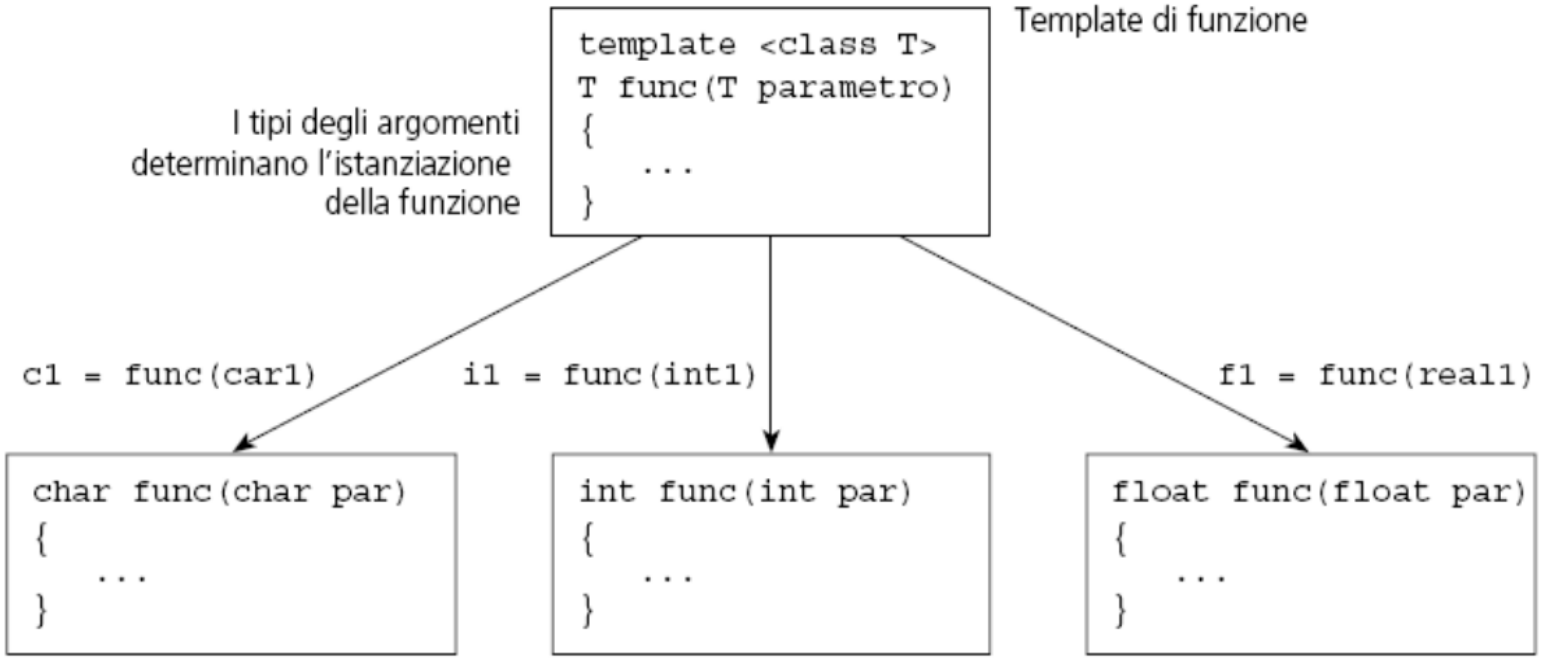


template di classi e funzioni

- non implicano perdita di rendimento e non obbligano a sacrificare i vantaggi del C++ in tema di controllo dei tipi di dato
- ***funzioni*** e ***classi generiche***, implementate per un tipo di dato da definirsi in seguito
- così come una classe è un modello per istanziare oggetti a tempo d'esecuzione, un template è un modello per istanziare classi o funzioni (del template) *a tempo di compilazione*
- il programmatore utente deve solo specificare i tipi con i quali essi debbono lavorare



C++: template di funzioni



- ANSI/ISO C++ scrive `typename` al posto di `class`
- possono avere più di un parametro di tipo:

template di funzioni

```
void scambia(int &x, int &y) {
    int t = x;
    x = y;
    y = t;
}
```

```
void scambia(float &x, float &y) {
    float t = x;
    x = y;
    y = t;
}
```

```
void scambia(char &x, char &y) {
    char t = x;
    x = y;
    y = t;
}
```

```
void scambia(Razionale &x, Razionale &y) {
    Razionale t = x;
    x = y;
    y = t;
}
```

```
template <typename T> void scambia(T &x, T &y) {
    T t = x;
    x = y;
    y = t;
}
```

```
class Razionale {
    public:
    void stampa() {cout << num << '/' << den << endl;}
    int num, den;
};

int main() {
    int i1=5, i2=8;
    scambia(i1,i2);
    cout << i1 << ' ' << i2 << endl;
    float f1=5.1, f2=8.1;
    scambia(f1,f2);
    cout << f1 << ' ' << f2 << endl;
    char c1='a', c2='b';
    scambia(c1,c2);
    cout << c1 << ' ' << c2 << endl;
    Razionale r1, r2;
    r1.num=1; r1.den=1; r2.num=2; r2.den=1;
    scambia(r1,r2);
    r1.stampa(); r2.stampa();
    return 0;
}
```

```
8 5
8.1 5.1
b a
2/1
1/1
```

template di funzioni

```
template <typename T> void scambia(T &x, T &y) {
    T t = x;
    x = y;
    y = t;
}

template <class T> void stampa(T* v, int n) {
    for(int i=0; i<n; i++) cout << v[i] << ' ';
    cout << endl;
}

template <class T> void ordina(T* v, int n) {
    for(int split=n/2; split>0; split/=2)
        for(int i = split; i < n; i++)
            for(int j=i-split; j >= 0 && v[j+split] < v[j]; j -= split)
                scambia(v[j], v[j+split]);
}

int main()
{
    char a[]={'s','d','f','g','h','j','k','l','z','x','c','v','b','n','m','q','w','e','r','t','y','u','i','o','p'};
    int b[]={1,3,2,4,3,5,4,6,5,7,6,8,7,9,11,13,24,35,46,75,68,97,80,44,22};
    float c[]={1.1,4.4,2.4,5.3,3.3,6.2,0.1,7.7,8.8,9.9,1.2,4.3,2.5,5.6,3.7,6.8,0.9,7.0,8.1,9.2,6.3,0.3,7.3,8.3,9.3};
    ordina(a,25); stampa(a,25);
    ordina(b,25); stampa(b,25);
    ordina(c,25); stampa(c,25);
    return 0;
}
```

```
b c d e f g h i j k l m n o p q r s t u v w x y z
1 2 3 3 4 4 5 5 6 6 7 7 8 9 11 13 22 24 35 44 46 68 75 80 97
0.1 0.3 0.9 1.1 1.2 2.4 2.5 3.3 3.7 4.3 4.4 5.3 5.6 6.2 6.3 6.8 7 7.3 7.7 8.1 8.3 8.8 9.2 9.3 9.9
```

template di classi

- definizione

```
template <typename T> class nome_classe
{...};
```

  - T nome del tipo (qualunque) utilizzato dal template
  - costruttore esterno:

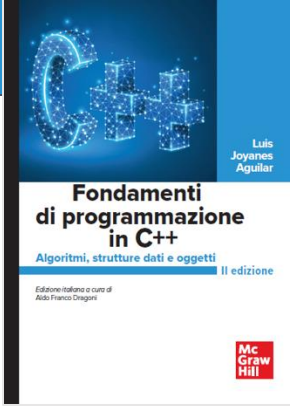
```
template <dichia_param_templ> nome_classe <param_templ> :: nome_classe
{...}
```

  - metodo esterno:

```
template <dichia_param_templ> tipo_risultato nome_classe
<param_templ> :: nome_metodo(dichia_param)
{...}
```

- istanziazione

```
nome_classe <tipo> oggetto;
```



Pila.cpp

```cpp
#include "Pila.h"

template <typename T, int n> void Pila <T, n> ::Push(T elemento) {
    dati[nElementi] = elemento;
    nElementi++;
}

template <typename T, int n> T Pila <T, n> ::Pop() {
    nElementi--;
    return dati[nElementi];
}

template <typename T, int n> int Pila <T, n> ::Quanti() {
    return nElementi;
}

template <typename T, int n> bool Pila <T, n> ::Vuota() {
    return (nElementi == 0);
}

template <typename T, int n> bool Pila <T, n> ::Piena() {
    return (nElementi == n);
}
```

header file e intestazioni di template di classe

Pila.h

```cpp
template <typename T, int n=100> class Pila {
    T dati[n];
    int nElementi;
    public:
    Pila() : nElementi(0) {}
    void Push(T elemento);  // push di un elemento
    T Pop();                // pop di un elemento
    int Quanti();           // numero di elementi
    bool Vuota();           // Pila vuota?
    bool Piena();           // Pila piena?
};
```

usaPila.cpp

```cpp
#include <iostream>
#include "Pila.cpp"

int main()
{   // definisce pila di interi (dimensione default)
    Pila <int> PilaInt;
    // definisce Pila di max 5 double
    Pila <double, 5> miniPila;
    return 0;
}
```

header file e intestazioni di classe

MiaStringa.cpp

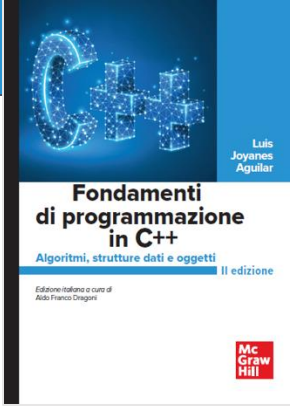
```
#include <string>
#include "MiaStringa.h"
MiaStringa::MiaStringa(const char* Stringa="")
    { Lunghezza = strlen(Stringa) + 1;
      miaStringa = new char[Lunghezza];
      strncpy(miaStringa, Stringa, Lunghezza);
      miaStringa[Lunghezza-1] = '\0';
    }
MiaStringa::~MiaStringa()
    { delete[] miaStringa;
      miaStringa = 0;
    }
char* MiaStringa::RestituisciStringa()
    { return miaStringa; }
int MiaStringa::RestituisciLunghezza()
    { return Lunghezza; }
```

MiaStringa.h

```
class MiaStringa
{
 public:
    MiaStringa(const char*);
    ~MiaStringa();
    char* RestituisciStringa();
    int RestituisciLunghezza();
 private:
    char* miaStringa;
    int Lunghezza;
};
```

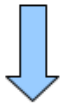
cliente.cpp

```
#include <iostream>
#include "MiaStringa.h"
int main()
{ MiaStringa MioNome("Aldo");
    cout << "Il mio nome è: " <<
    MioNome.RestituisciStringa() <<
    endl;
    return 0;
}
```

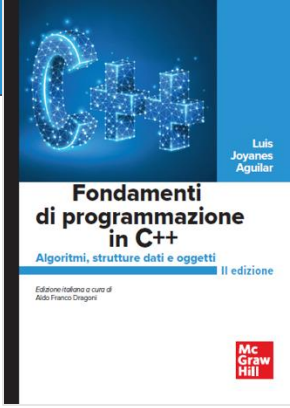


modelli di compilazione di template

- il compilatore produce istanze specifiche di tipi solo quando vede la chiamata al template (di funzione o di classe)
- per generare un'*istanziazione* il compilatore deve accedere al codice sorgente che definisce il template



- non sono ammesse librerie di template in codice binario, ma solo header-files che includano anche il codice di implementazione in forma sorgente (*compilazione per inclusione*)



Pila.cpp

```
template <typename T, int n> void Pila<T, n>::Push(T elemento) {
    dati[nElementi] = elemento;
    nElementi++;
}

template <typename T, int n> T Pila<T, n>::Pop() {
    nElementi--;
    return dati[nElementi];
}

template <typename T, int n> int Pila<T, n>::Quanti() {
    return nElementi;
}

template <typename T, int n> bool Pila<T, n>::Vuota() {
    return (nElementi == 0);
}

template <typename T, int n> bool Pila<T, n>::Piena() {
    return (nElementi == n);
}
```

compilazione per inclusione

Pila.h

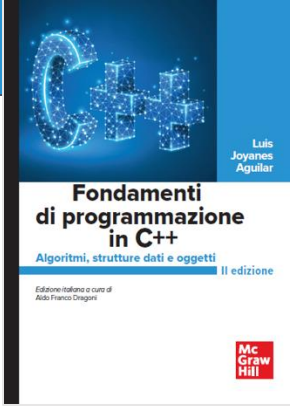
```
#ifndef PILA_H
#define PILA_H

template <typename T, int n=100> class Pila {
    T dati[n];
    int nElementi;
    public:
    Pila() : nElementi(0) {};
    void Push(T elemento);  // effettua la push di un elemento
    T Pop();                // effettua la pop di un elemento
    int Quanti();           // restituisce il numero di elementi
    bool Vuota();           // Pila vuota?
    bool Piena();           // Pila piena?
};
#include "Pila.cpp"
#endif
```

usaPila.cpp

```
#include <iostream>
#include "Pila.h"
using namespace std;

int main()
{
    Pila <int> PilaInt; // definisce pila di interi (dimensione default)
    PilaInt.Push(12);
    cout << PilaInt.Quanti() << endl;
    cout << PilaInt.Pop() << endl;
    cout << PilaInt.Quanti() << endl;
    Pila <double, 5> MiniPila; // definisce Pila di max 5 double
    MiniPila.Push(12.24);
    cout << MiniPila.Quanti() << endl;
    cout << MiniPila.Pop() << endl;
    cout << MiniPila.Quanti() << endl;
    return 0;
}
```



compilazione per inclusione

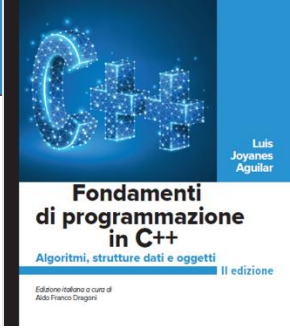
- inserisce una direttiva `#include` nell'*header file* perché inserisca le definizioni del file `.cpp`

confronta.h

```
#ifndef DEMO_H
#define DEMO_H
template<typename T> int confronta(const T&, const T&);
//altre dichiarazioni
#include "demo.cpp"
#endif
```

confronta.cpp

```
template<typename T> int confronta(const T &a, const T &b){
    if(a < b) return -1;
    if(b < a) return 1;
    return 0;
}
```



compilazione separata

- si usa `export` per ottenere la compilazione separata di definizioni di templates e dichiarazioni di funzioni di templates

somma.h

```
export template<typename T> T somma(T t1, T t2)
```

- **`export`**: usata come prefisso nella definizione di una funzione template, indicherebbe che la stessa definizione è accessibile anche da altre *translation units*
- spetterebbe poi al ***linker*** e non al compilatore generare le istanze richieste dall'utente
  - non più supportata in C++11

C++ keywords: export

Usage

Used to mark a template definition *exported*, which allows the same template to be declared, but not defined, in other translation units.	(until C++11)
The keyword is unused and reserved.	(since C++11)

template e polimorfismo

- funzione polimorfica: supporta tipi di dato differenti
  - es: parametro come puntatore ad una classe
- funzione template: preceduta da clausola `template`
  - è solo un modello e non una vera funzione
  - `template` è un generatore automatico di funzioni sovraccaricate
  - pensare in astratto, evitando qualunque dipendenza da tipi di dato, costanti numeriche, ecc.
- la genericità polimorfica si limita a gerarchie
- i templates tendono a generare un codice eseguibile grande, poiché duplicano le funzioni

template e polimorfismo

- una funzione è polimorfica se almeno uno dei suoi parametri può supportare tipi di dato differenti
- qualunque funzione che abbia un parametro come puntatore ad una classe può essere una funzione polimorfica e si può utilizzare con tipi di dato diversi
- una funzione è una funzione template solo se è preceduta da un'appropriata clausola template
- scrivere una funzione template implica pensare in astratto, evitando qualunque dipendenza da tipi di dato, costanti numeriche, ecc.
- una funzione template è solo un modello e non una vera funzione la clausola template è un generatore automatico di funzioni sovraccaricate
- le funzioni templates lavorano anche con tipi aritmetici
- le funzioni polimorfiche debbono utilizzare puntatori
- la genericità polimorfica si limita a gerarchie
- i templates tendono a generare un codice eseguibile grande, poiché duplicano le funzioni