Templates

- La possibilità di progettare classi e funzioni che lavorano su tipi generici è molto utile in pratica
- C++ lo fa mediante i template
- Permettono di implementare strutture e algoritmi indipendentemente dal tipo di oggetti su cui operano

genericità

- permette di definire una classe (o una funzione) senza specificare il tipo di dato di uno o più dei suoi membri (o parametri)
- sfrutta il fatto che gli algoritmi di risoluzione di numerosi problemi non dipendono dal tipo di dato da elaborare;
  - es: un algoritmo che gestisca una pila di caratteri è lo stesso di uno che gestisce una pila di interi
- In linguaggi come Pascal, C e COBOL, è necessario sviluppare un programma diverso per ogni tipo di dato da inserire nella pila
- Nei nuovi linguaggi OOP esistono i *templates*
- Permettono di definire *classi generiche* (o *parametriche*) che implementano strutture e classi indipendenti dal tipo di elemento da processare
- Dovranno poi essere istanziati dall'utente per produrre sottoprogrammi o classi che lavorino con specifici tipi di dato

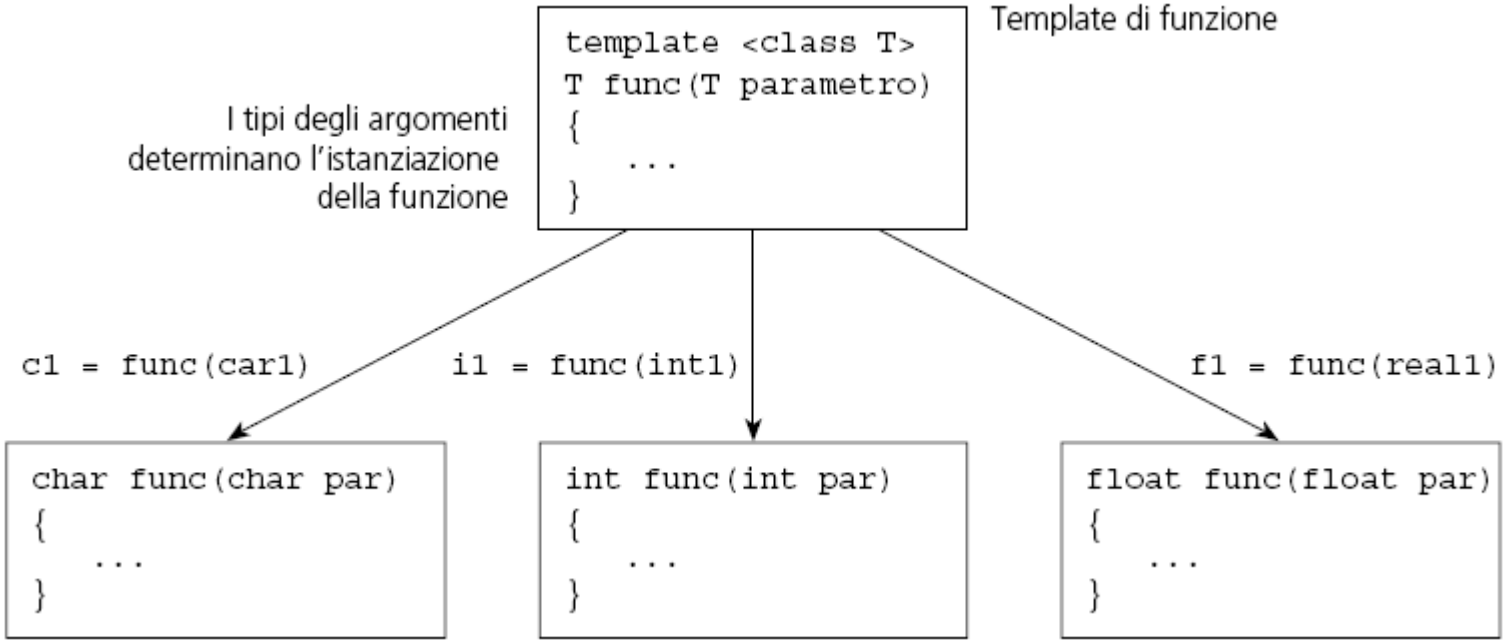
templates

- costruzione per scrivere *funzioni* e *classi* molto generali che possono applicarsi a dati di tipo diverso

Proprietà importante: questa generalità non implica perdita di rendimento e non obbliga a sacrificare i vantaggi del C++ in tema di controllo stretto dei tipi di dato

- una classe è un modello per istanziare oggetti (della classe) a tempo d'esecuzione,
- un *template* è un modello per istanziare classi o funzioni (del template) a tempo di compilazione
-
- i templates sono quindi funzioni e classi generiche, implementate per un tipo di dato da definirsi in seguito
- per utilizzarli il programmatore deve solo specificare i tipi con i quali essi debbono lavorare

templates di funzioni



- ANSI/ISO C++ scrive `typename` al posto di `class`
- La funzione viene istanziata sulla base dei parametri utilizzati nella chiamata di funzione:

```
template <typename T>
    const T& max(const T& a, const T& b)
    { return a > b ? a : b;}
```


templates di funzioni

- Occorre specificare almeno un tipo parametrico
- La funzione può avere più di un parametro formale e restituire un valore di tipo parametrico

```
template <class T> T f(int a, T b)
{
 corpo della funzione
}

template <class T1, class T2> T1 f(T1 a, T2 b)
{
 corpo della funzione
}
```

- `T1` e `T2` devono essere distinti

templates di classi

- permettono di definire classi parametriche che possono gestire differenti tipi di dato

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

- `T` è il nome del tipo utilizzato dal template (`T` non limitato a tipi di dato predefiniti)
- `tipopar` (es. *Pila*) è il nome del tipo parametrizzato del template;
- il codice viene sempre preceduto da un'istruzione nella quale si dichiara `T` come parametro di tipo, e possono esserci più parametri tipo

```
template <typename T>
struct Punto {
   T x, y;
};
...
Punto<int> pt = {45, 15};
```

Per usare il template di classe si deve fornire un argomento per ogni tipo parametrico (in questo caso solo 1)

esempio: template per una classe Pila


```
template <class T> class Pila
{   T dati[50];
    int elementi;
public:
    Pila(): elementi(0) {}
    void inserire(T el);
    T estrarre();
    int numero(); //Numero di elementi nella pila
    bool vuota();
};
```

```
Pila <int> pila_interi;
Pila <float> pila_reali;
```


esempio: template per una classe Pila (II)


```
template <class elem, int dimensione> class Pila
{   int dimensione;
    …
public:
    Pila(int n);
    int numero(); //Numero di elementi nella pila
    bool vuota();
    …
};
```

Specifica il tipo

```
Pila <int,100> pila_interi;
Pila <float,20> pila_reali;
```


Templates di classi

- L'implementazione di un template di classe richiede le funzioni
  - costruttore
  - distruttore
  - membri

Costruttore di templates

```
template <dichiarazioni-parametri-template T>
   nome_classe <parametri-template> :: nome_classe
{
   ...
}
```

Esempio: Corpo del costruttore del template `pila`

```
template <class elem, int dimensione>
   pila <elem, dimensione> :: pila(int n)
{
…
}
```


modelli di complazione di templates

- Quando definiamo un oggetto di una classe, la definzione della classe deve essere presente, ma non sono necessarie le definizioni delle funzioni membro

- Per questo possiamo mettere le definizioni delle classi negli header files (e i metodi nei corrispondenti files sorgenti)

- Per i templates le cose sono diverse

modelli di complazione di templates

- Quando il compilatore vede una definizione di template, non genera codice immediatamente
- Produce istanze specifiche di tipi del template solo quando vede una chiamata al template (di funzione o di classe che sia)
- per generare un'*istanziazione* il compilatore deve accedere al codice sorgente che definisce il template
- C++ standard definisce due modelli per la compilazione del codice dei templates:
  - "compilazione per inclusione" (supportato da tutti i compilatori)
  - “compilazione separata" (supportato solo da alcuni)
- In ogni caso, le definizioni delle classi e le dichiarazioni delle funzioni vanno in header files, mentre le definizioni di membri e metodi vanno in files sorgenti
- I due modelli differiscono però nel modo in cui si rendono disponibili al compilatore le definizioni dei files sorgenti

compilazione per inclusione

- il compilatore deve vedere la definizione di qualunque template
- Soluzione: includere nell'header file non solo le dichiarazioni, ma anche le definizioni
- permette di mantenere la separazione tra header files e files d'implementazione, anche se s'inserisce una direttiva `#include` nel header file perché inserisca le definizioni del file `.ccp`

```
//header file demo.h
#ifndef DEMO_H
#define DEMO_H
template<class T> int confrontare(const T&, const T&);
//altre dichiarazioni

#include "demo.cc"   //definizioni di confrontare
#endif
//implementazione del file demo.cc
template<class T> int confrontare(const T &a, const T &b)
{
   if(a < b) return -1;
   if(b < a) return 1;
   return 0;
}
//altre definizioni
```


compilazione separata

- permette di scrivere le dichiarazioni e funzioni in due files (estensioni `.h` e `.cpp`)
- si deve utilizzare la parola riservata `export` per ottenere la compilazione separata di definizioni di templates e dichiarazioni di funzioni di templates
- la dichiarazione del template di funzione si mette in un header file, ma la dichiarazione non deve specificare `export`

```
//definizione del template in un file compilato separatamente
export template<typename T>
T somma(T t1, T t2)
```

- l'uso di `export` in un template di classe è un po' più complicato

```
//intestazione del template di classe sta nel file
//di intestazione condiviso
template <class T> class Pila {...};
//File pila.ccp dichiara Pila come esportata
export template <class T> class Pila;
#include "Pila.h"
//definizioni di funzioni membro di Pila
```


templates e polimorfismo: differenze formali

- 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

templates e polimorfismo

- 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

In pratica:

- 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

