



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Template

Stefano Ghidoni



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE



Agenda

- Concetto di template
- Class & function template
- Esempi



Concetto di template

- Un template è un meccanismo che permette al programmatore di usare un tipo come parametro per una classe o una funzione
- Il compilatore genera il codice necessario per **ogni tipo** per cui il template è specializzato

```
vector<double>
vector<int>
vector<Month>
vector<char>
vector<vector<Record>>
vector<Window*>
```

Le funzioni sono state scritte generalmente nel tipo, senza quindi specificarlo.
Quando viene creato un oggetto di double, allora viene generato il codice per double.



Class template

- Un *class template* è anche chiamato *type generator*, *parametrized type* o *parametrized class*
- Il processo di generazione di una classe su un tipo specifico è chiamato *specializzazione* (*specialization*) o *template instantiation*
 - Es.: `vector<char>` è una specializzazione di `vector`
- Questo processo avviene **a tempo di compilazione**
 - Che vantaggi/svantaggi ci sono?

Scrivendo codice con i template, il tempo di compilazione è più lungo, in esecuzione invece non c'è alcun rallentamento



Notazione

- `template <typename T>` vs `template <class T>`
- Perfettamente equivalenti
- typename
 - Più chiaro
 - Evidente che possiamo usare tipi built-in
- class
 - Più corto
 - class significa "tipo" (BS)



Template per vector

- Riprendiamo ora la classe vector sviluppata precedentemente
- Sostituiamo un tipo generico T a double
- Dobbiamo aggiungere un prefisso che introduce il template

```
template<typename T>
class vector {
    int sz;
    T* elem;
public:
    vector() : sz{0}, elem{nullptr} {}
};
```



Template per vector

```
template<typename T>
class vector {

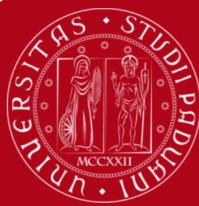
    int sz;
    T* elem;

public:
    vector() : sz{0}, elem{nullptr} {}

    explicit vector(int s) : sz{s}, elem{ new T[s] } {
        for (int i = 0; i < sz; ++i) elem[i] = 0;
    }
};
```

Cosa significa?

Elimina l'automatica conversione tra il tipo passato al tipo costruito (ad esempio, non si potrebbe convertire un intero in un vector).
Esempio: fornire un intero al costruttore di Rational permette di fare una conversione automatica da intero a Rational.
Non sempre ha senso permettere di fare le conversioni.



Class template

- Le funzioni membro della class template seguono la stessa specializzazione
- Es: supponiamo esista la funzione membro `push_back`

```
void vct(vector<string>& v)
{
    int n = v.size();
    v.push_back("Norah");
    // ...
}
```



Class template

- È usata la funzione membro `push_back()` su `v`
- La funzione è definita come:

```
template <typename T>
void vector<T>::push_back(const T& d) /* ... */
```

- Da questa, il compilatore genera:

```
void vector<string>::push_back(const string& d) /* ... */
```



Function template

- Abbiamo visto i template applicati alle classi
- È possibile parametrizzare anche le funzioni
- Quando parametrizziamo una funzione otteniamo un *function template*
 - AKA *parametrized function* o *algorithm*
 - *Algorithm* perché il focus del programma è sull'algoritmo più che sui tipi che usa



Function template

- Esempio di function template:

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y) ? x : y;
}
```

- Perché è necessario usare un template in questo caso?
- Che assunzione stiamo facendo su T?

Risposte

Se non si usasse un template, bisognerebbe scrivere una funzione `myMax` per ogni tipo per cui si vorrebbe permettere ciò, ma allora non valrebbe per gli UDT (è impossibile prevedere ogni UDT, sono virtualmente infiniti).

Stiamo assumendo che `T` abbia l'operator `>` definito. Questo è vero per i tipi standard, ma non è detto che lo sia per gli UDT.

In genere l'operator `<` è quello considerato per l'ordine (in questo caso è `>`).

Se l'operator `>` non fosse definito, il compilatore (che dovrebbe costruire la funzione per il tipo dato) darebbe errore, in quanto non potrebbe costruirla.



Function template

- Esempio di function template:

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y) ? x : y;
}
```

- Per chiamare la funzione:

```
int x = i, j = 5;

int max = myMax<int>(i, j)
```

Interi come parametri
template



Interi come parametri template

- Fino ad ora: template sui tipi
- Cosa succede se usiamo un intero come parametro template?
 - Per esempio: un vector con template su tipo e numero
- Numero di elementi è noto a tempo di compilazione
 - Non è variabile

L'eventuale intero inserito diventerebbe non modificabile.

Se creassimo un vettore di due interi e uno di tre interi, allora sarebbero due tipi diversi, mentre due std::vector di char, anche se di dimensione diversa, sono dello stesso tipo.



Esempio

```
template<typename T, int N> struct array {  
    T elem[N];  
  
    T& operator[] (int n);  
    const T& operator[] (int n) const;  
  
    T* data() { return elem; }  
    const T* data() const { return elem; }  
  
    int size() const { return N; }  
};
```

Il fatto di non allocare dinamicamente la memoria, permette di non avere problemi circa l'immutabilità di N.



Esempio di utilizzo

```
array<int, 256> gb;
array<double, 6> ad = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5 };
const int max = 1024;

void some_fct(int n)
{
    array<char, max> loc;
    array<char, n> oops;      // errore se n non è noto a
                               // tempo di compilazione
    // ...
    array<char, max> loc2 = loc; // copia di backup
    // ...
    loc = loc2;                // restore
    // ...
}
```



Motivazioni

- Questi array sono concettualmente simili ai precedenti, ma molto meno flessibili
- Perché usarli?



Motivazioni

- Ragione principale: efficienza
 - Il compilatore può ottimizzare meglio
- Non necessita di free store
 - Alcuni programmi non possono usarlo
- Offrono, però, gli stessi vantaggi di interfaccia e di sicurezza dei vector

Ricordo che allocando dinamicamente uso il freestore, altrimenti viene usato lo stack.



Motivazioni

- Un array di dimensione non modificabile può esprimere un concetto
 - Un array di 3 elementi fissi esprime il concetto di "tripletta"
 - Es: le immagini a colori sono composte da triplete di valori (R, G, B)
 - Esprimibili come `array<unsigned char, 3>`



Deduzione degli argomenti template

- Il compilatore è in grado di dedurre gli argomenti template dagli argomenti di una funzione
- Ad esempio, si consideri:

```
array<char, 1024> buf;
array<double, 10> b2;

template<class T, int N>
void fill(array<T, N>& b, const T& val)
{
    for (int i = 0; i < N; ++i) b[i] = val;
}
```



- Le chiamate a fill sono interpretate così:

```
void f()
{
    fill(buf, 'x');    // T è char e N è 1024, dedotto da buf

    fill(b2, 0.0);    // T è double e N è 10, dedotto da b2
}
```

- Equivalenti a:

```
void f()
{
    fill<char, 1024>(buf, 'x');

    fill<double, 10>(b2, 0.0);
}
```

```
array<char, 1024> buf;
array<double, 10> b2;

template<class T, int N>
void fill(array<T, N>& b, const T& val)
{
    for (int i = 0; i < N; ++i) b[i] = val;
}
```



Deduzione degli argomenti template

- Analogamente, per l'esempio precedente:

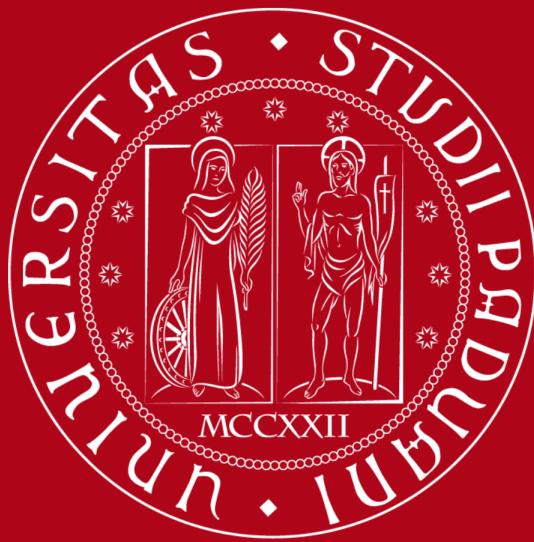
```
int x = i, j = 5;
```

```
int max = myMax<int>(i, j)
```

- È equivalente a:

```
int x = i, j = 5;
```

```
int max = myMax(i, j)
```



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Template

Stefano Ghidoni