9.1 – Template

Libro di testo:

- Capitoli 19.3, 19.3.1, 19.3.2, 19.3.3*, 19.3.5 19.3.6

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

- Concetto di template
- Template applicato a una classe
- Template applicato a una funzione
- 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

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

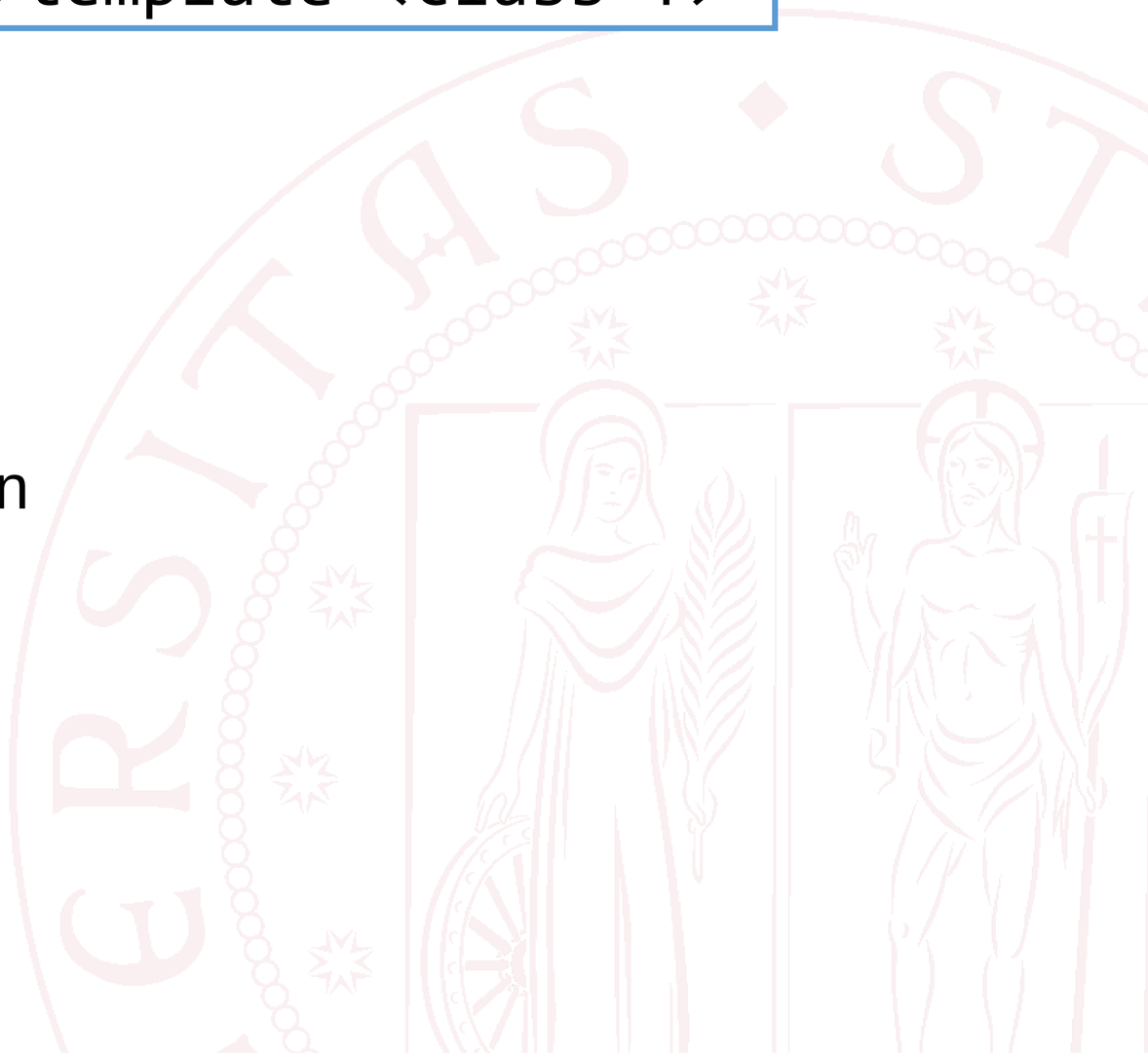
Class template

- 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.: `std::vector<char>` è una specializzazione di `std::vector`
- Questo processo avviene a tempo di compilazione
  - Che vantaggi/svantaggi ci sono?

Notazione

`template <typename T>` oppure `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
- Sostituire un tipo generico `T` a `double` nella nostra classe `vector`
- Notazione C++: 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;
    }
};
```

allocazione dinamica di s oggetti di tipo T

Cosa significa?

Class template

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

```
void vct(vector<std::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:

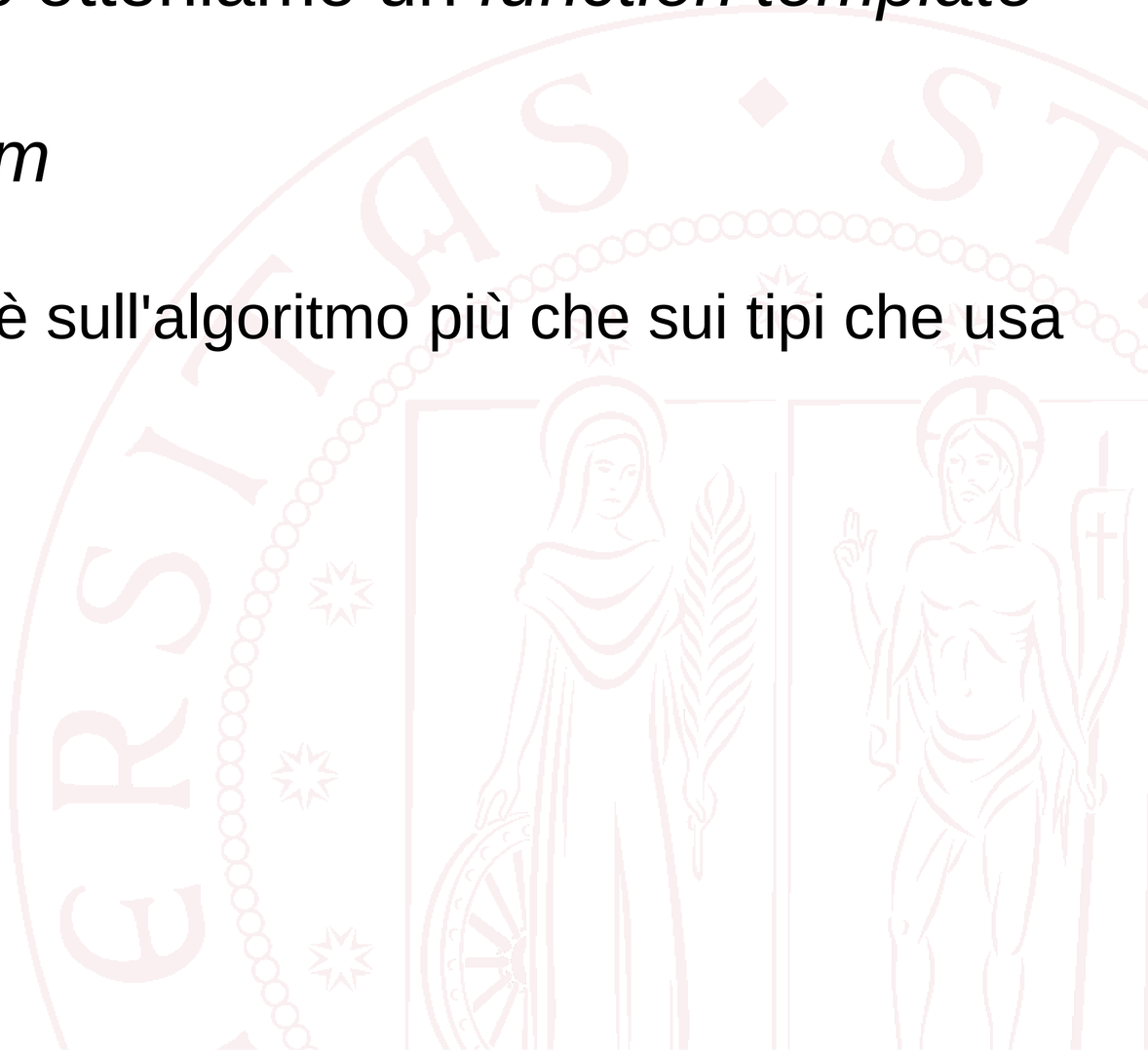
```
template <typename T>
void vector<T>::push_back(const T& d) {/* ... */}
```

- A partire da questa, il compilatore genera:

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

Function template

- 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, 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?

Function template

- Esempio, function template:

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

- Per chiamare la funzione:

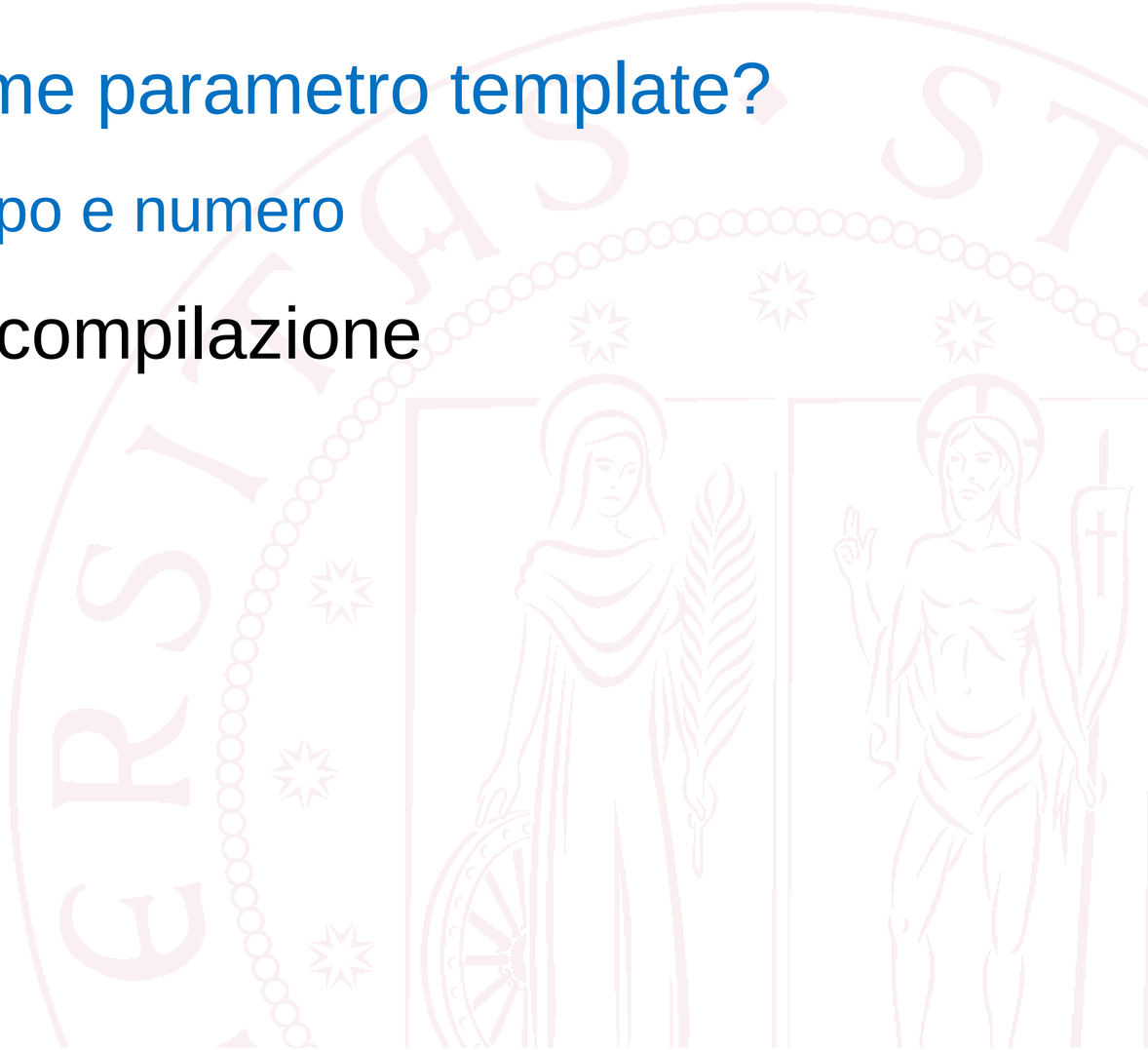
```
int x = i, j = 5;

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

Interi come parametri template

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

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



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; }
};
```

Esempio di utilizzo

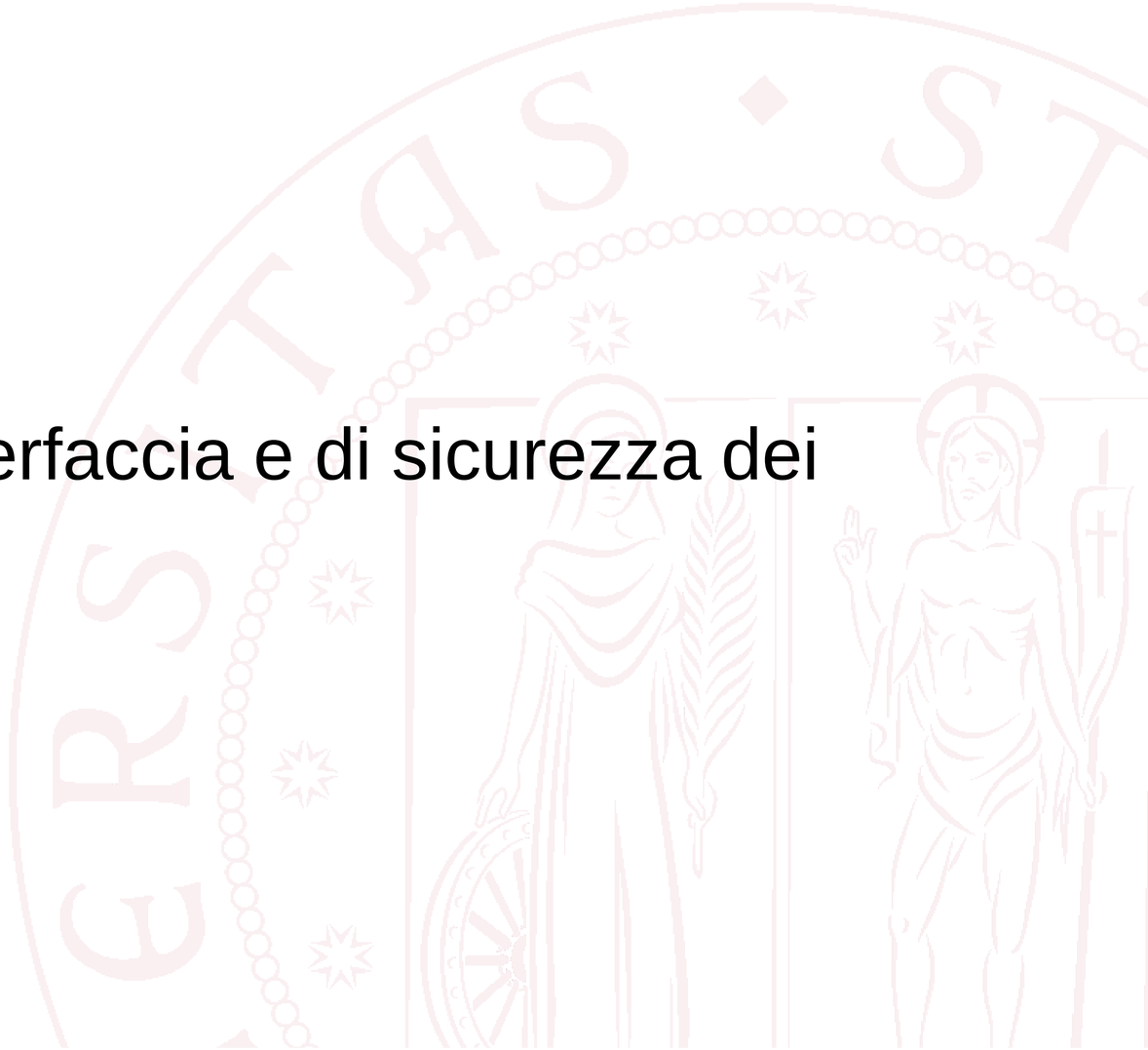
```
array<int, 256> gb;
array<double, 6> ad = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5 };
const int max = 1024;

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

Motivazione

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

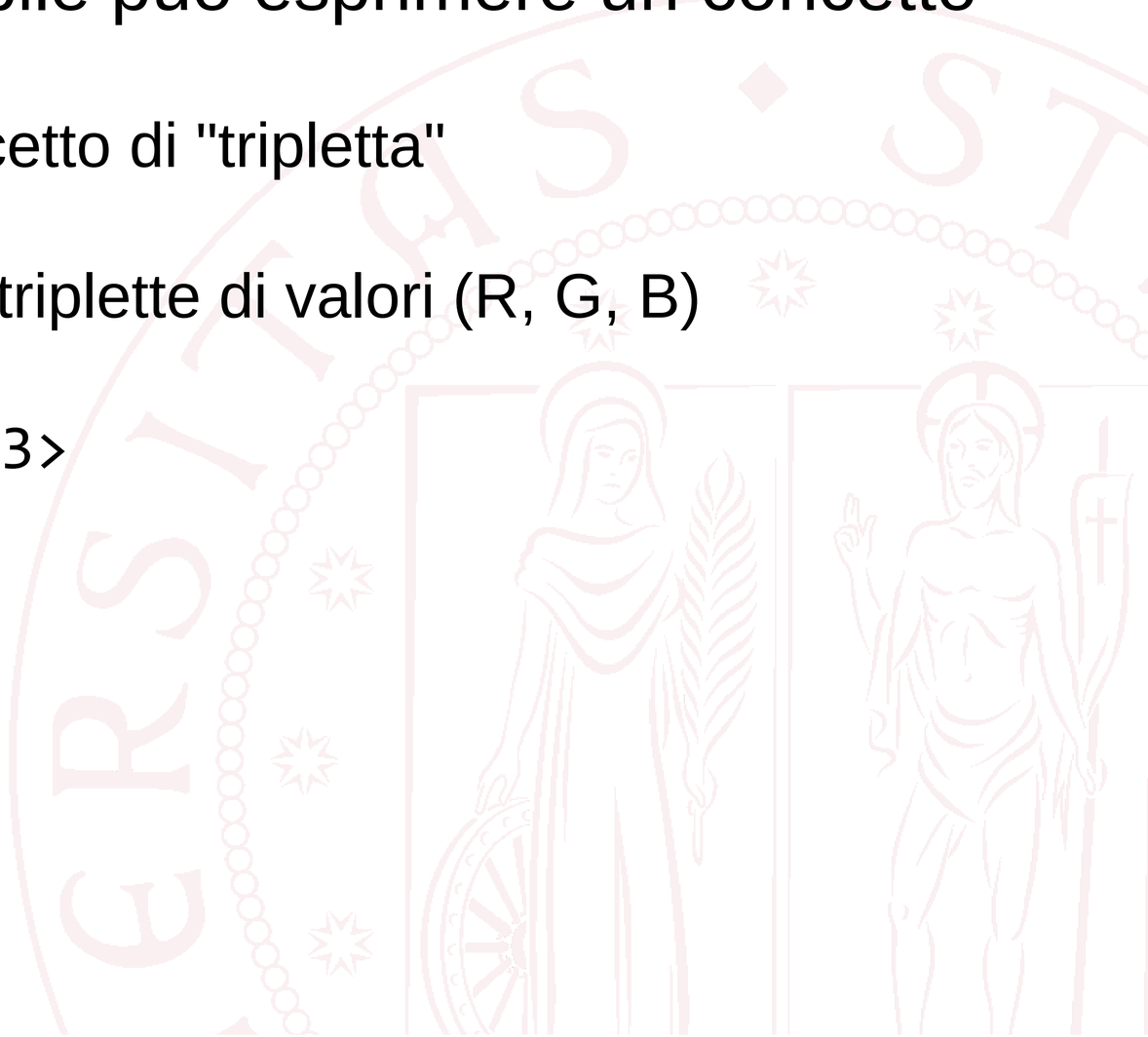
Motivazione

- 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`



Motivazione

- 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 triplette 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;
}
```

Deduzione degli argomenti template

- 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);
}
```

Deduzione degli argomenti template

- Analogamente, per l'esempio precedente:

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

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

- Equivalente a:

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

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

Recap

- Concetto di template
- Notazione per i template
- Function e class template
- Template con interi come parametri
- Deduzione degli argomenti template