Class templates

Programação (L.EIC009)

José Proença (FCUP) & **João Bispo** (FEUP) – *slides by Eduardo R. B.*

Templates

Classes can have templates. For instance, `std::vector` is a class template.

The declaration of a class template is similar to what we have seen before for function templates and `struct` type templates. Below, `T` designates a template type argument:

```
namespace some_namespace {
  template <typename T>
  class some_template_class {
    ...
  };
}
```

Note: the code of a template class usually resides in a single header file. Separate compilation is not possible.

Examples:

- `polynomial<T>`: an incomplete sketch for the `polynomial` example, now as a template class;
- `simple_vector<T>`: complete example for container (generalisation of the `simple_vector` struct type seen in previous classes);
- `pair<T,U>`: a template class for a pair of elements;

`simple_vector` and `pair` are available online at GitHub.

```
class polynomial {
  polynomial(const std::vector<fraction>& c)
  : coeffs(c) {
    reduce();
  }
  ...
private:
  std::vector<fraction> coeff;
  ...
};
```

polynomial example where the coeffs field now is of type vector<T> instead of vector<fraction>:

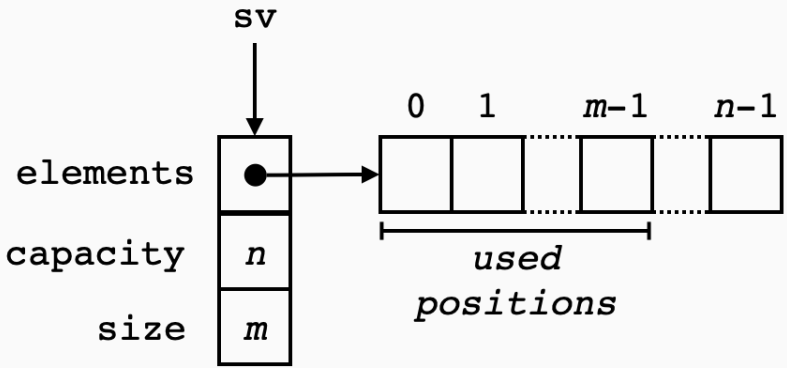
```
template <typename T>
class polynomial {
  polynomial(const std::vector<T>& c)
  : coeffs(c) {
    reduce();
  }
  ...
private:
  std::vector<T> coeff;
  ...
};
```

polynomial<T> (cont.)

U.PORTO

```
template <typename T>
class polynomial {
  ...
  polynomial(const std::vector<T>& c) { ... }
  ...
};
...
int main() {
  polynomial<fraction> p { {1, 2}, {3, 4} };
  polynomial<double> q { 0.5, 0.75 };
  ...
}
```

simple_vector<T>: a template class for a sequence of elements stored in a "growable array", like std::vector. Conceptually similar to the simple_vector example from the "Dynamic memory" slides.



simple_vector<T> (cont.)

```
template <typename T>
class simple_vector {
public:
  ...
private:
  // Capacity of the array.
  int capacity_;
  // Stored elements.
  int size_;
  // Dynamically allocated array holding elements.
  T* elements_;
};
```

simple_vector<T> (cont.)

```
template <typename T>
class simple_vector {
public:
  simple_vector(int initial_capacity = 5);
  simple_vector(const simple_vector<T>& sv);
  ~simple_vector();
  int size() const;
  int capacity() const;
  void add(const T& elem);
  T& at(int index);
  const T& at(int index) const;
private: ...
};
```

```
int main() {
  simple_vector<int> v(2);
  cout << v.size() << ' ' << v.capacity() << '\n';
  v.add(-1);
  v.add(2);
  v.add(4); // grows capacity to 4
  v.add(3);
  for (int i = 0; i < v.size(); i++)
     cout << "[" << i << "] : " << v.at(i) << '\n';
  cout << v.size() << ' ' << v.capacity() << '\n';
```

simple_vector<T> (cont.)

```
int main() {
  simple_vector<int> v(2);
  cout << v.size() << ' ' << v.capacity() << '\n';
  v.add(-1);
  v.add(2);
  v.add(4); // grows capacity to 4
  v.add(3);
  for (int i = 0; i < v.size(); i++)
     cout << "[" << i << "] : " << v.at(i) << '\n';
  cout << v.size() << ' ' << v.capacity() << '\n';
```

```
0 2
[0] : -1
[1] : 2
[2] : 4
[3] : 3
4 4
```

Constructors set `elements_` to an array allocated using `new`.

```
template <typename T>
simple_vector<T>::simple_vector(int initial_capacity) :
  capacity_(initial_capacity), size_(0) {
    elements_ = new T[capacity_];
}
```

Constructors set `elements_` to an array allocated using `new`.

```
template <typename T>
simple_vector<T>::simple_vector(const simple_vector<T>& sv) :
  capacity_(sv.capacity_), size_(sv.size_) {
    elements_ = new T[capacity_];
    for (int i = 0; i < size_; i++) {
      elements_[i] = sv.elements_[i];
    }
}
```

Destructor releases the memory for the array of elements using `delete`.

```
template <typename T>
simple_vector<T>::~simple_vector() {
  delete [] elements_;
}
```

With the exception of add the other member functions are simple.

```
template <typename T>
int simple_vector<T>::size() const { return size_; }

template <typename T>
int simple_vector<T>::capacity() const { return capacity_; }
```

Note that at has two variants: the const variant returns a const reference to an array element, while the non-const one returns a mutable reference.

```
template <typename T>
const T& simple_vector<T>::at(int index) const
{ return elements_[index]; }

template <typename T>
T& simple_vector<T>::at(int index)
{ return elements_[index]; }
```

```
template <typename T>
void simple_vector<T>::add(const T& elem)  {
  if (capacity_ == size_) {
    int new_capacity = 2 * capacity_; // Double the capacity
    T* new_array = new T[new_capacity];
    for (int i = 0; i < capacity_; i++) // Copy elements to new array
      new_array[i] = elements_[i];
    delete [] elements_; // Free memory for old array
    elements_ = new_array;     // Point to new array
    capacity_ = new_capacity;
  }
  elements_[size_] = elem; size_++;
}
```

pair<T,U>: a template class with two type arguments, representing pairs of elements (base functionality similar to std::pair).

```
template <typename T, typename U>
class pair {
public:
  pair(const T& a, const U& b) : first_(a), second_(b) {}
  T& first() { return first_; }
  const T& first() const { return first_; }
  U& second() { return second_; }
  const U& second() const { return second_; }
private:
    T first_; U second_;
};
```

```cpp
#include "pair.hpp"

int main() {
  pair<int,std::string> a{ 2024, "leic" };
  std::cout << a.first() << ' ' << a.second() << '\n';

  pair<std::string, pair<int,int>> b{ "leic", { 2023, 2024 } };
  std::cout << b.first() << ' '
            << b.second().first() << ' '
            << b.second().second() << ' ';
  return 0;
}
```

```cpp
#include "pair.hpp"

int main() {
  pair<int,std::string> a{ 2024, "leic" };
  std::cout << a.first() << ' ' << a.second() << '\n';

  pair<std::string, pair<int,int>> b{ "leic", { 2023, 2024 } };
  std::cout << b.first() << ' '
            << b.second().first() << ' '
            << b.second().second() << ' ';
  return 0;
}
```

```
2024 leic
leic 2023 2024
```