Templates

Iuliana Bocicor
maria.bocicor@ubbcluj.ro

Babes-Bolyai University

2025

Overview

1. Code analysis
2. Templates
3. C++ Standard Template Library

Code analysis I

- *Linter/Code analyser* - an automated tool that analyses the source code and signals programming errors, bugs, stylistic errors, suspicious code.
- They are used to help us improve our code.
- Advantages:
  - Fewer errors in the final code (especially useful for industrial applications, where fewer defects arrive in production).
  - Readable, maintainable, more consistent code, of better quality.
  - Learning about best practices in writing modern C++ code.
- For Visual Studio: Project → Properties → Code Analysis → Enable Code Analysis on Build

Code analysis II

- We can select the sets of rules to be verified: Project → Properties → Code Analysis → Microsoft → Choose multiple rule set: select all sets of rules starting with "C++ Core" (for rules from C++ core Guidelines: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines).
- Code analysis will be made at compilation, warnings will be reported in case guideline rules are broken.
- See also: https://docs.microsoft.com/en-us/cpp/code-using-the-cpp-core-guidelines-checkers?view=msvc-
- For other platforms, you may use **clang-tidy**: http://clang.llvm.org/extra/clang-tidy/.

Templates

- *Generic programming* - algorithms are written with generic types, that are going to be specified later.
- Generic programming is supported by most modern programming languages.
- In C++ templates allow working with generic types.
- Provide a way to reuse source code. The code is written once and can then be used with many types.
- Allow defining a function or a class that operates on different kinds of types (is parametrized with different types).

Function templates I

Declaration

template <typename identifier> function_declaration;

```
template <typename T>
T add(T a, T b)
{
    return a + b;
}
```

- **T** is the *template parameter*, a type argument for the template;
- The template parameter can be introduced with any of the two keywords: typename, class.

Function templates II

- The process of generating an actual function from a template function is called **instantiation**:

```
int resInt = add<int>(3, 4);
double resDouble = add<double>(-1.2, 2.6);
```

DEMO

Function template. (*Lecture_4 - FunctionTemplate.cpp*).

Class templates I

- A template can be seen as a skeleton or macro.
- When specific types are added to this skeleton (e.g. double), then the result is an actual C++ class.
- When instantiating a template, the compiler creates a new class with the given template argument.
- The compiler needs to have access to the implementation of the methods, to instantiate them with the template argument.
- **Place the definition of a template in a header file.**

DEMO

Template (*Lecture4 - DynamicVector.h, DynamicVector_demo.cpp*).

Class templates II

- Templates can be also defined for more types:

```
template <typename T, typename U>
class Pair
{
private:
    T first;
    U second;
// ...
};
```

DEMO

Template (*Lecture4 - Pair.h*).

Templates - differences from void* implementation

void*	**Template**
A container with void* elements can only hold addresses.	A container can hold any type (both addresses and simple objects).
Casting to a specific pointer is required, in certain situations.	No casting needed.
Memory management is required.	Memory management is required only in certain situations.
The container may include pointers to different types.	All elements will have the same type.

Templates - conclusions

- Templates are a compile-time mechanism.
- They are most commonly used in generic programming (implementation of general algorithms).
- Useful for writing compact and efficient code.
- The definition (not just the declaration) must be in scope (usually in the header file).

Standard Template Library (STL)

- Is a software library for C++.
- Is a generic library, meaning that its components are heavily parametrized: almost every component in the STL is a template.
- Is designed such that programmers create components that can be composed easily without losing any performance.
- The primary designer and implementer of STL is Alexander Alexandrovich Stepanov.

Containers in STL I

- A container is a holder object that stores a collection of other objects (its elements).
- Containers are implemented as class templates.
- Containers:
    - manage the storage space for their elements;
    - provide member functions to access the elements, either directly or through iterators (reference objects with similar properties to pointers);
    - provide functions to modify the elements.

Containers in STL II

- Container class templates:
  - **Sequence containers** (elements are ordered in a linear sequence):
    - array<T>;
    - vector<T>;
    - deque<T>;
    - forward_list<T>;
    - list<T>.
  - **Associative containers** (elements are referenced by their keys and not by their absolute positions in the container):
    - set<T, CompareT>;
    - multiset<T, CompareT>;
    - map<KeyT,ValueT,CompareT>;
    - multimap<KeyT, ValueT,CompareT>.

Containers in STL III

- **Container adapters** (created by limiting functionality in a pre-existing container):
  - stack<T, ContainerT>;
  - queue<T, ContainerT>;
  - priority_queue<T,ContainerT, CompareT>.

Iterators I

- Provide a generic (abstract) way to access the elements of a container.
- Allow access to the elements of a container without exposing the internal representation (implementation hiding).
- Make a separation between how data is stored and how we operate on data.
- An iterator will contain:
  - a reference to the current element;
  - a reference to the container.

Iterators II

- An iterator keeps track of a location within an associated STL container object, providing support for traversal (increment/decrement), dereferencing and container bounds detection.
- In C++, iterators are not pointers, but act similar to pointers in certain situations (can be incremented with **++**, dereferenced with *****, and compared against another iterator with **!=**).
- Containers expose 2 member functions: begin() and end(), which provide iterators towards the begin (first element) and the end (past the last element) of the containers.

std::vector

- Is a container that stores elements of the same type.
- Is a sequence container: its elements are ordered in a linear sequence.
- Resizes automatically when needed.
- Uses a dynamically allocated array to store the elements.
- Is very efficient in terms of element accessing (constant time).
- Works with range-based for loop.

DEMO

std::vector (*Lecture4 - stl_demo.cpp*).

std::deque

- Double ended queue, with dynamic size, that can be expanded or contracted at both ends.
- Elements are stored in chunks of storage, not in contiguous locations.
- Elements can be accessed through random access iterators.
- Insertion and deletion of elements are efficient at both ends (not just at the end, as in the case of vectors).
- Deques are a little more complex internally than vectors, but this allows them to grow more efficiently especially with very long sequences, where re-allocations become more expensive.

DEMO

std::vector (*Lecture4 - stl_demo.cpp*).

std::list

- Implemented as a doubly linked list.
- Has constant time for inserting and erasing elements on any position (only if we have a valid iterator at the position).
- Can be iterated in both directions.
- Compared to vectors and deques, lists perform better in inserting, extracting and moving elements on positions for which an iterator has already been obtained.

DEMO

std::vector (*Lecture4 - stl_demo.cpp*).

STL Algorithms I

- Algorithms are function templates that can operate on ranges of elements, ranges defined by iterators.
- The iterators returned by the functions begin() and end() of a container can be fed to an algorithm to enable using the algorithm with the container.
- Iterators are the mechanism that make possible the decoupling of algorithms from containers.
- Exempt us from writing the same functions (find, sort, count) for different individual containers.

STL Algorithms II

- Headers: <algorithm>, <numeric> - define a collection of functions especially designed to be used on ranges of elements.
- A list of available algorithms can be found at: Algorithms library.

DEMO

STL Algorithms (*Lecture4 - stl_demo.cpp*).

Lambda expressions I

- Provide a mechanism to define anonymous functions (locally, within other functions).
- The anonymous function is defined in the code where it is called.
- Are very useful for certain algorithms of the STL (find_if, count_if, transform, sort).
- The return type of lambdas can be deduced, but it can also be specified.

Lambda expressions II

Syntax

[capture list] (parameter list) {function body}
[capture list] (parameter list) $\rightarrow$ return_type {function body}

E.g.

```
// ...
vector<int> oddNumbers(5);
copy_if(integers.begin(), integers.end(),
    oddNumbers.begin(), [](int x) { return x % 2
    == 1; });
```

Lambda expressions III

- A lambda can store information about variables that are in the local block scope.
- The lambda function body can refer to those variables using the same name as in the surrounding scope.
- This is possible using the capture list.

DEMO

STL Algorithms (*Lecture4 - stl_demo.cpp*).

Advantages of STL algorithms

- **simplicity**: use existing code instead of writing the code from scratch;
- **correctness**: known to be correct, tested;
- **performance**: generally perform better than hand written code;
- **clarity**: you can immediately tell that a call to **sort** sorts the elements in a range;
- **maintainability**: code is clearer and more straightforward $\Rightarrow$ easier to write, read, enhance and maintain.