Object Oriented Programming - Lecture 6

Diana Borza - diana.borza@ubbcluj.ro

April 2021

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off." (B. Stroustrup)

Content

- Generic programming - templates - compile time polymorphism
- C++ standard library
- Containers
- Iterators
- What's new in C++ (cont'd) - range based for loop

Templates I

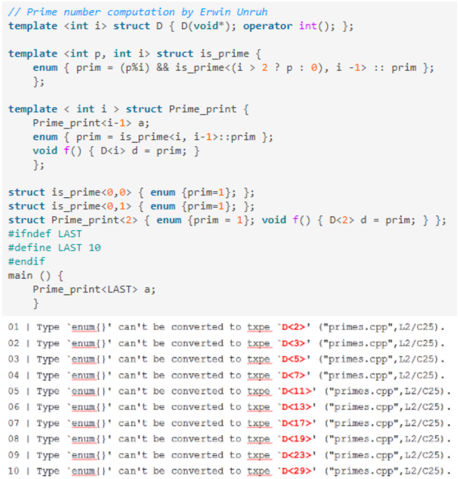
- *Generic programming* - algorithms are written with generic types, that are going to be specified later.
- Generic programming is supported by most modern programming languages.
- Templates allow working with generic types.
- Provide a way to reuse source code. The code is written once and can then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

Templates - the most famous program that didn't compile

- The first concrete demonstration templates of this was a program written by *Erwin Unruh*, which computed prime numbers although it did not actually finish compiling: **the list of prime numbers was part of an error message generated by the compiler on attempting to compile the code**.

http://aszt.inf.elte.hu/~gsd/halado_cpp/ch06s04.html#Static-metaprogramming

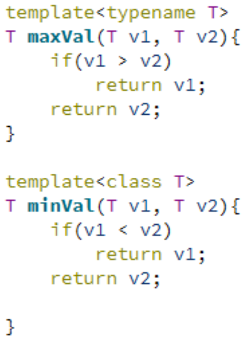
Unruh example



```cpp
// Prime number computation by Erwin Unruh
template <int i> struct D { D(void*); operator int(); };

template <int p, int i> struct is_prime {
    enum { prim = (p%i) && is_prime<(i > 2 ? p : 0), i -1> :: prim };
    };

template < int i > struct Prime_print {
    Prime_print<i-1> a;
    enum { prim = is_prime<i, i-1>::prim };
    void f() { D<i> d = prim; }
    };

struct is_prime<0,0> { enum {prim=1}; };
struct is_prime<0,1> { enum {prim=1}; };
struct Prime_print<2> { enum {prim = 1}; void f() { D<2> d = prim; } };
#ifndef LAST
#define LAST 10
#endif
main () {
    Prime_print<LAST> a;
    }
```

```
01 | Type `enum{}' can't be converted to txpe `D<2>' ("primes.cpp",L2/C25).
02 | Type `enum{}' can't be converted to txpe `D<3>' ("primes.cpp",L2/C25).
03 | Type `enum{}' can't be converted to txpe `D<5>' ("primes.cpp",L2/C25).
04 | Type `enum{}' can't be converted to txpe `D<7>' ("primes.cpp",L2/C25).
05 | Type `enum{}' can't be converted to txpe `D<11>' ("primes.cpp",L2/C25).
06 | Type `enum{}' can't be converted to txpe `D<13>' ("primes.cpp",L2/C25).
07 | Type `enum{}' can't be converted to txpe `D<17>' ("primes.cpp",L2/C25).
08 | Type `enum{}' can't be converted to txpe `D<19>' ("primes.cpp",L2/C25).
09 | Type `enum{}' can't be converted to txpe `D<23>' ("primes.cpp",L2/C25).
10 | Type `enum{}' can't be converted to txpe `D<29>' ("primes.cpp",L2/C25).
```

Function templates

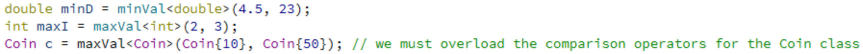


```
template<typename T>
T maxVal(T v1, T v2){
    if(v1 > v2)
        return v1;
    return v2;
}

template<class T>
T minVal(T v1, T v2){
    if(v1 < v2)
        return v1;
    return v2;

}
```

- 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

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



```
double minD = minVal<double>(4.5, 23);
int maxI = maxVal<int>(2, 3);
Coin c = maxVal<Coin>(Coin{10}, Coin{50}); // we must overload the comparison operators for the Coin class
```

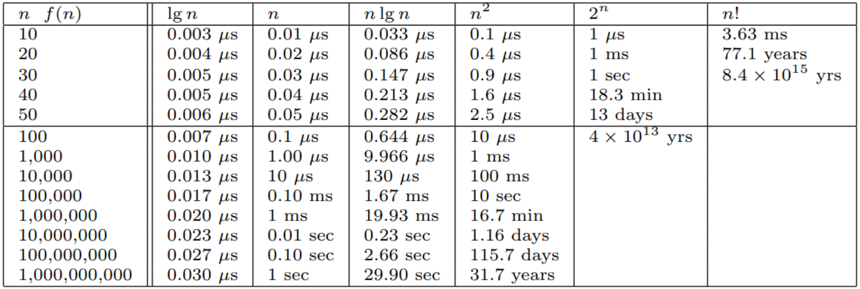
Class templates

- 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.
- Simple solution: **place the definition of a template in a header file.**

Container classes

- **A container** - is a class designed to hold and organize multiple instances of another type (either another class, or a fundamental type).
- Common operations on containers:
  - Create an empty container (via a constructor);
  - Insert a new object into the container;
  - Remove an object from the container;
  - Return the number of objects from the container;
  - Empty the container (remove all the objects from it);
  - Access to the stored objects;
  - Sort the elements (optional).

Growth rates



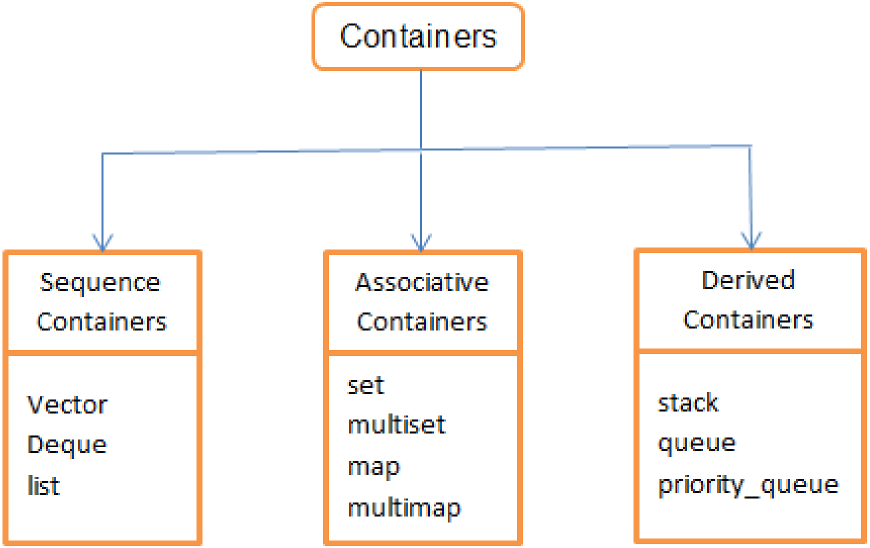
n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 2.4: Growth rates of common functions measured in nanoseconds

C++ STANDARD LIBRARY - STL

- Advantages of using STL:
  - **simplicity**: use well written existing code, instead of writing everything from scratch;
  - **correctness**: well tested, known to be correct;
  - **efficiency**: generally, structures and algorithms from STL have a better performance than the code we write;
  - **maintainability**: code is easier to understand and more straightforward.
  - Using STL your code becomes easier to read, write, maintain and enhance!

STL containers

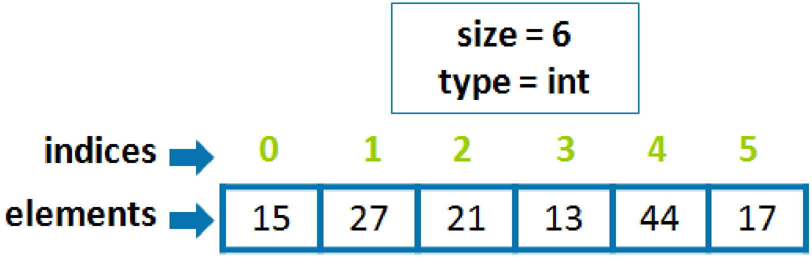


Sequence containers

- Sequence containers are container classes that **maintain the ordering** of elements in the container.
- What is the difference between ordered and sorted?
- You can choose where to insert your element by position.

std::array

- Is a container that encapsulates fixed size arrays.
- Has the same semantics as a struct holding a C-style array T[N] as its only data member.
- Combines the performance and accessibility of a C-style array with the benefits of a standard container (knowing its own size, supporting assignment, random access iterators).



std::vector

- It is a sequence container that encapsulates dynamic size arrays (what you implemented for the 3rd laboratory :)).
- The elements are stored *contiguously* $\rightarrow$ elements can be accessed not only through iterators, but also using offsets to regular pointers to elements.
- The storage of the vector is handled automatically, being expanded and contracted as needed.
- capacity() actual allocated memory; capacity() - size of the vector;
- Complexity of common operations:
  - Random access (*at, [] operator* - constant O(1);
  - Insertion or removal of elements at the end *push_back, pop_back* - amortized constant O(1);
  - Insertion or removal of elements *insert, erase* - linear in the distance to the end of the vector O(n)

std::deque

- Double-ended queue class, implemented as a dynamic array that can grow from both ends.
- Allows fast insertion and deletion at both its beginning and its end.
- As opposed to std::vector, the elements of a deque are not stored contiguously.
- The storage of a deque is automatically expanded and contracted as needed. Expansion of a deque is cheaper than the expansion of a std::vector.
- Complexity of common operations:
  - Random access - constant O(1);
  - Insertion or removal of elements at the end or beginning - constant O(1);
  - Insertion or removal of elements - linear O(n).

std::list

- List containers are implemented as doubly-linked lists;
- Adding, removing and moving the elements does not invalidate the iterators or references. An iterator is invalidated only when the corresponding element is deleted.
- Compared to other sequence containers (array, vector and deque), lists perform generally better in *inserting*, *extracting* and *moving* elements in any position within the container;
- Main drawback: that they lack direct access to the elements by their position.

Associative containers

- Associative containers are containers that automatically sort their inputs when those inputs are inserted into the container. By default, associative containers compare elements using operator $<$.

std::set

- A set is a container that stores unique elements, with **duplicate elements disallowed**.
- The elements are sorted according to their values. Sorting is done using a comparison function that you can redefine.
- **A multiset** is a set where duplicate elements are allowed.

std::map

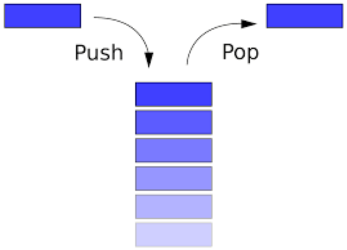
- A **map**: is a set where each element is a *key/value pair*.
- The *key* is used for *sorting* and *indexing* the data, and must be unique.
- The value is the actual data.
- A **multimap** (a dictionary) is a map that allows duplicate keys.

Derived containers

- Derived containers are special predefined containers that are adapted to specific uses.
- They provide a different interface for sequential containers.
    - stack
    - queue
    - priority_queue

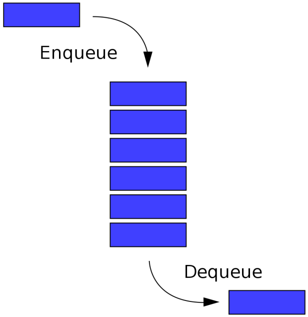
std::stack

- a container where elements operate in a **LIFO** (Last In, First Out) context;
- elements are inserted (pushed) to the front of the container;
- elements are removed (popped) from the front of the container;
- use deque as their default sequence container.

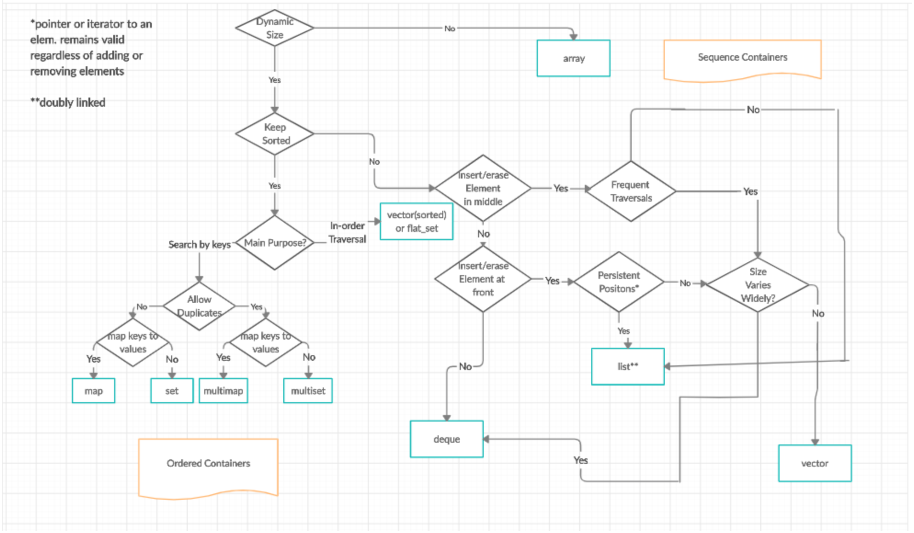


std::queue, std::priority_queue

- a container where elements operate in a **FIFO** (First In, First Out) context;
- elements are inserted (pushed) to the back of the container;
- elements are removed (popped) from the front of the container;
- use deque as their default sequence container.
- A **priority queue** is a queue where the elements are sorted (via operator<). At push the element is sorted in the queue. At pop (deque) the element with the highest priority is removed.



Decisions, decisions...



Iterators I

- An **Iterator** is an object that can traverse (iterate over) a container class without the user having to know how the container is implemented.
- ? What OOP feature is this ?
- Can be visualized as a pointer to a given element in the container.

Iterators II

- Operators defined in iterators:
  - **Operator*** : dereferences the iterator $\rightarrow$ return the element that the iterator is pointing at.
  - **Operator++**: moves the iterator to the next element in the container.
  - **Operator-** - to move to the previous element.
  - **Operator==** and **Operator!=**: basic comparison operators;
  - **Operator=**: assigns the iterator to a new position.

Iterators III

- All C++ containers provide (at least) two types of iterators:
  - container::iterator - read/write iterator
  - container::const_iterator - read-only iterator
- Each container includes four basic member functions to work with iterators:
  - **begin()**: returns an iterator representing the beginning of the elements in the container.
  - **end()**: returns an iterator representing the element just past the end of the elements.
  - **cbegin()**: returns a const (read-only) iterator representing the beginning of the elements in the container.
  - **cend()**: returns a const (read-only) iterator representing the element just past the end of the elements.

STL algorithms

- algorithms are implemented as functions that operate using iterators'
- std::min_element() and std::max_element() algorithms find the min and max element in a container class;
- std::find() algorithm to finds a value in a container; returns the end of the iterator if the element is not present in the container;
- std::sort() - sorts a container; doesn't work on list container classes!
- std::reverse() - reverses a container;

What's new in C++ (cont'd) - range based for

- Used as a more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container.
- syntax:

  ```
  for ( range_declaration : range_expression )
      loop_statement
  ```