CH-231-A

Algorithms and Data Structures

ADS

Lecture 2

Dr. Kinga Lipskoch

Spring 2020

Some Remarks

- The code is generated by the compiler, which substitutes the generic `T` with the provided type
- Templates can be used both for methods and for "algorithms", i.e., functions which work with different data types
- It is common (and necessary ...) to put both declaration and definition of the templatized classes in the same header file
  - Does not break anything
  - The compiler will not allocate space and will not run into duplicate definition problems
- `stemplate.h`
- `stemplate.cpp`
- `stemplate_main.cpp`

Additional Remarks

- Multiple definitions are merged together
- If you specify `template<class T>`, the type parameter can be either a class or a basic data type
- You can have more than one parameter (generic or not):
  `template<class T, int, double>`

One More Example

- Templates are very useful when developing general purpose containers, i.e., classes whose task is to store objects
- Most of containers use the same business logic to access data; the only difference is the data type
- A good container library can dramatically cut down your developing time
- `templatestack.cpp`

The Ownership Problem

What should a container hold? Objects or pointers to objects?

- If it contains object instances, we say it owns the objects
- If it contains pointers, we say it does not own objects
  - If it holds objects, they can be safely removed from memory during destructor execution
  - If it holds pointers to objects, other code is in charge of the destruction
    - Both seem to have their own advantages

Ownership: General Guidelines

Containers should not own objects; thus their destruction should be managed in places other than container destructors

- In general it is better to create objects on the heap and to access them via pointers
  - This opens the doors to a consistent use of polymorphism
- Management of object instances created on the heap is the source of many many bugs
  - Always double check your code involving `new` and `delete`

Are Templates Against OOP?

- Some OOP languages do not provide templates
- To write generic code, they just play with inheritance
  - For example, inherit everything from a single class and write code dealing with that class
  - Java and Smalltalk use this approach, but newer versions of Java have introduced something like templates
  - By offering both, C++ allows you to use both, at your choice

The Standard Template Library (STL)

- C++ standardization began in 1989 (until 1998)
- STL was later added in 1994
- STL is part of the Standard C++ Library
- Extends the core language by some general components
- May be reused for different purposes
- Programmers do not need to reinvent the wheel again and again
- Eases development of applications
- Makes software more maintainable

Brief Definitions

- Containers
  - Manage collections of objects
- Iterators
  - Navigate (step) through the elements of a container
- Algorithms
  - Process elements of collections
  - E.g., search, sort, modify

Standard Template Library (1)

- Data and algorithm separated rather than combined
- Every kind of container can be combined with every kind of algorithm
- All components work with arbitrary types
- Components are templates for (almost) any type
- STL good example for “generic programming”

Standard Template Library (2)

- Containers are objects used to store other objects
- Containers size changes dynamically
- Very useful when objects are created on the heap
  - In that case containers hold their pointers
- Based on templates, containers can be used to store any data type

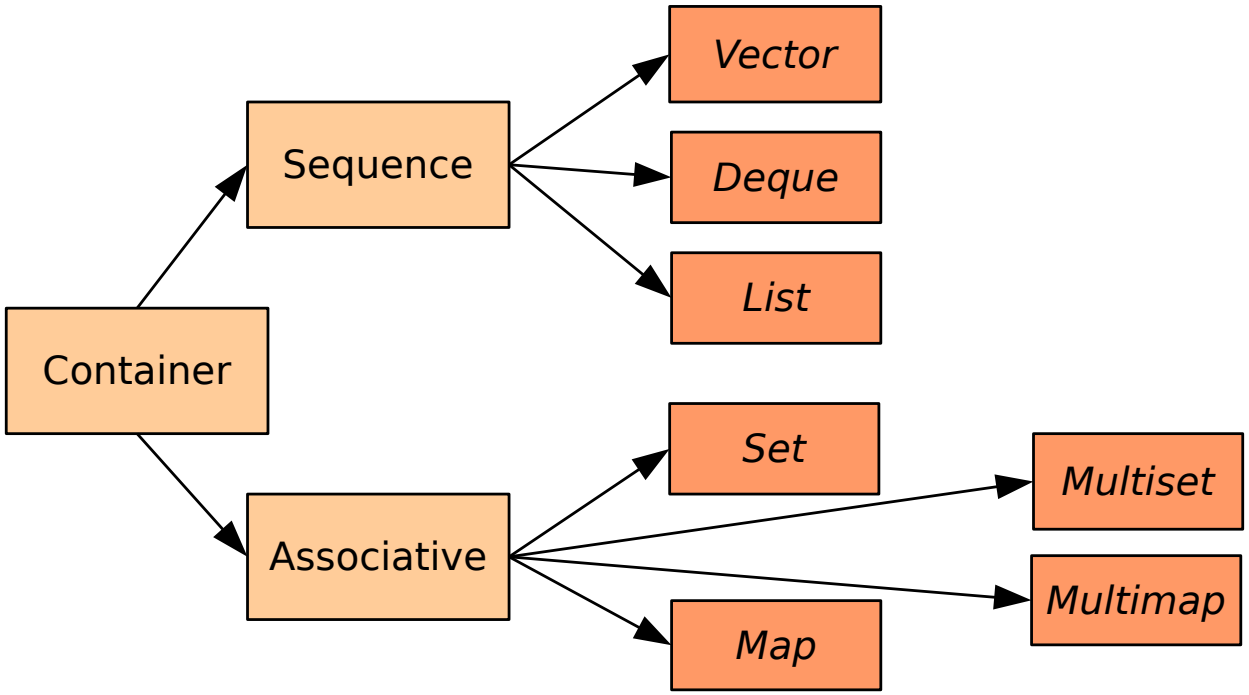
The Standard C++ Containers Library

- Derived from the STL, the two terms are often used as synonyms, but they are two different things (although pretty similar)
- Before reinventing the wheel check the standard library
  - In most of the situations, it is unlikely that you will need to develop yet another linked list class, or vector, or other widely used containers
  - Rely on widely used code developed by specialists
- Good documentation at `http://www.sgi.com/tech/stl/`

Containers Library

- Built over a restricted set of simple concepts, it can be used to quickly develop your software
- The two main concepts are containers and iterators
  - Containers hold objects, while iterators are used to move through containers to get/set objects
  - The iterator's mechanism is independent from the underlying container implementation, so you can use the same approach in many different situations
    - And of course without knowing how containers work
  - Containers dynamically grow or shrink to accommodate your storage needs

Fundamental Container Classes

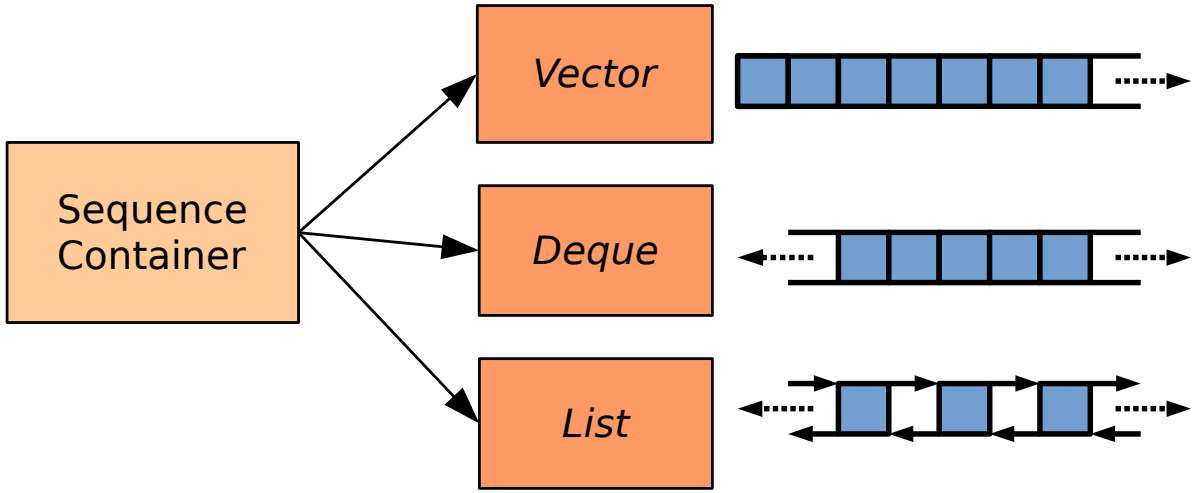


Predefined classes have different characteristics regarding insert/access speed, size, usability

Containers

- Different containers for different needs
  - Sequences:
    - `vector`, `deque`, `list`
  - Associations:
    - `set`, `multiset`, `map`, `multimap`
- Common operations:
  - Insert an object into it
  - Remove an object from it
  - Iterate over all the elements (using an iterator)

Sequence Containers



- Ordered collection where every element has certain position
- Position depends on time and place of insertion, but independent of value of element
- Predefined containers differ in speed of insertion of elements and access to elements

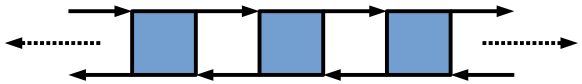
Vector

- Mimics an array
  - Provides random access
  - Fast insertion at the end, and fast indexing through overloaded `[]` operator
  - Needs more time if element is added at the middle of array
  - Not very efficient while resizing, and for what concerns memory allocation
- Constructors: `vector()`, `vector(int)`
- Basic methods: `push_back`, `pop_back`, `back`, `clear`, `size`, `max_size`, `empty`
- `vectorexample.cpp`

Deque

- Double ended queue
  - Elements are managed in dynamic array, which can grow in both directions
  - Appending and removing elements at beginning / end very fast
  - Needs more time if element is added at the middle of array
- Basic interface: very similar to vector; in addition `push_front`, `pop_front`, `front`
- Preferred to vector, unless you know exactly how many elements you will store
- `dequeexample.cpp`

List



- A double linked list
  - Element consists of
    - value
    - link to predecessor
    - link to successor
  - No random access
  - Fast insertion at both ends, slow access to intermediate elements
- Basic interface: similar to deque, but missing the `[]` operator; in addition
  - `reverse`, `sort`
- `listexample.cpp`

A Few Comments

- There are many more methods in every class
- If you use just the common methods of the containers, you will be able to change container by just changing their declaration and not the client code
- As with all containers, you can use them as black boxes
  - You do not need to care about their internal implementation

And More

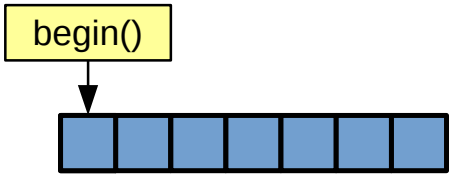
Stacks, queues, priority queues

- All implemented over the basic containers seen before
- Sometimes called adapters
- They do not provide additional capabilities, but rather reshape underlying containers

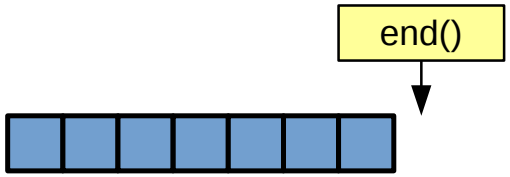
Iterators (1)

- Object that iterates over elements, which are all or part of the elements of an STL container
- Represents a certain position in a container
- Operations
  - `*` returns element at current position
  - `++` step forward to next element
  - `==` equals same position
  - `!=` not equals same position
- May iterate over complicated data structures of containers (such as binary trees)
- Internal implementation depends on container

Member functions `begin()` and `end()` return Iterators



- Returns an iterator, that points to beginning of the elements in container



- Returns an iterator, that points to end of the elements in container; this is the position behind the last element (past-the-end-iterator)
- Both functions define a half-open range (includes first, but excludes last element)

Iterators (2)

- Iterators can be dereferenced, to gain access to the element they point to
  - Think of iterators as "very smart pointers"
  - But do not push this similarity too far
- Iterators are declared as follows:

```
vector<int> vint;
vector<int>::iterator viterator;
```

  - This is because iterators are declared as container inner classes
- iteratorsexample.cpp

Using Iterators on Vectors

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main() {
6    vector<int> v; // vector container for integers
7    v.push_back(2);
8    v.push_back(5);
9
10   vector<int>::const_iterator pos;
11
12   for (pos = v.begin(); pos != v.end(); ++pos) {
13     cout << *pos << ' ';
14   }
15   cout << endl;
16   return 0;
17 }
18
19 // if using C++11, you can use cbegin() and cend() instead
20 // of begin() and end()
```

Nested Templates

- Templates can be nested
- Keyword `typename` can be needed if not the current instatiation of a type but a dependent type is used
- `templ_in_templ1.cpp`
- `templ_in_templ2.cpp`