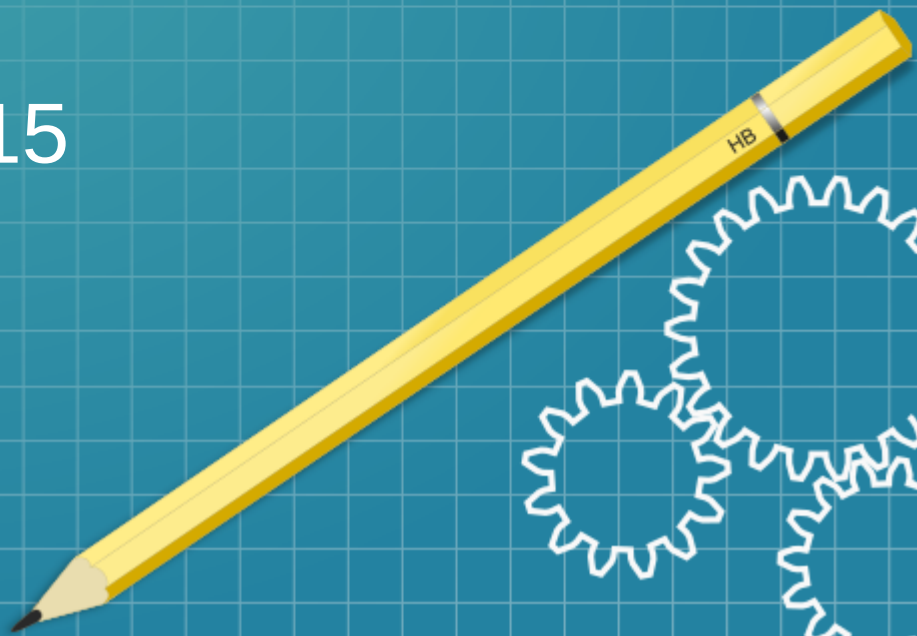
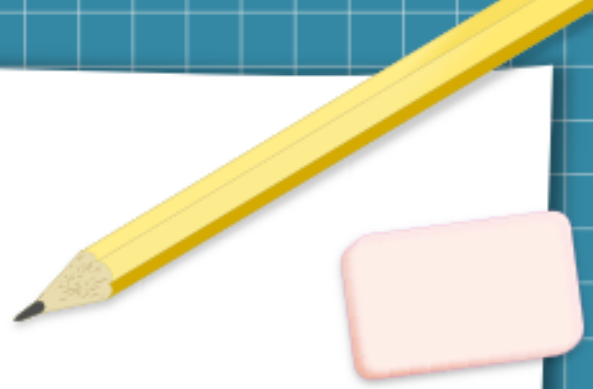
Standard Library Containers and Iterators

Chapter 15



Today we will discuss



- Containers
- Iterators
- Algorithms

Containers

The *Standard Template Library* (*STL* or *Standard Library* for short) has a number of *templatized data structures* which are called *containers*.

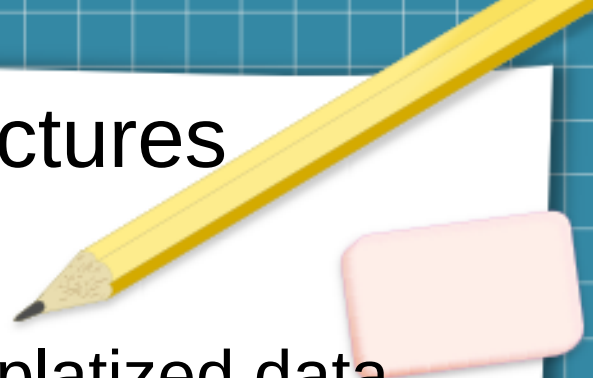
Containers are data structures capable of storing objects of almost any data type.

There are three *styles* of container classes:

- First-class containers
- Container adapters
- Near containers

Each container has associated member functions, with a subset of these defined in all containers.

Custom Templatized Data Structures



In CSI 33 you will build your own custom templatized data structures, like:

- Lists
- Linked Lists
- Stacks
- Queues
- Binary Trees

etc.

Iterators

Iterators have properties similar to those of pointers, and are used to *manipulate container elements*.

We will discuss them in more details later today.

Algorithms

The *Standard Library algorithms* are *function templates* that perform some common data manipulation.

Examples: searching, sorting, comparing containers, etc.

Each algorithm has minimum requirement for the kinds of iterators that can be used with it.

Some Container Classes

Here is a list of container classes we worked with so far:

- `array`
- `vector`

Some Container Classes

Here is a list of container classes we worked with so far:

- `array`
- `vector`

 - these are *sequence containers*, they represent *linear data*.

Some Container Classes

Here is a list of container classes we worked with so far:

- `array`
- `vector`

 - these are *sequence containers*, they represent *linear data*.

Other containers of the same type:

- `deque`
- `forward_list`
- `list`

Some Container Classes

Here is a list of container classes we worked with so far:

- `array`
- `vector`

 - these are *sequence containers*, they represent *linear data*.

Other containers of the same type:

- `deque`

  Rapid insertions and deletions at front or back.
  Direct access to any element.

- `forward_list`

- `list`

Some Container Classes

Here is a list of container classes we worked with so far:

- `array`
- `vector`

 - these are *sequence containers*, they represent *linear data*.

Other containers of the same type:

- `deque`

  Rapid insertions and deletions at front or back.
  Direct access to any element.

- `forward_list`

  Singly linked list, rapid insertion and deletion anywhere.
  Added in C++ 11.

- `list`

Some Container Classes

Here is a list of container classes we worked with so far:

- `array`
- `vector`

 - these are *sequence containers*, they represent *linear data*.

Other containers of the same type:

- `deque`

  Rapid insertions and deletions at front or back.
  Direct access to any element.

- `forward_list`

  Singly linked list, rapid insertion and deletion anywhere.
  Added in C++ 11.

- `list`

  Doubly linked list, rapid insertion and deletion anywhere.

Some Container Classes

Here is a list of container classes we worked with so far:

- `array`
- `vector`

 - these are *sequence containers*, they represent *linear data*.

Other containers of the same type:

- `deque`

  Rapid insertions and deletions at front or back.
  Direct access to any element. see dequeUse.cpp

- `forward_list`

  Singly linked list, rapid insertion and deletion anywhere.
  Added in C++ 11.

- `list`

  Doubly linked list, rapid insertion and deletion anywhere.
  see listUse.cpp

Some Container Classes

Fig. 15.2 has a list of common member functions for most Standard Library containers.

4 major *categories* of container types:

- *sequence containers*
- *ordered associative containers*
- *unordered associative containers*
- *container adapters*

Associative Container Classes

are *nonlinear data structures* that typically can quickly locate elements stored in it.

Such containers store *key – value* pairs/associations, where each *key* must be unique and immutable, and it is associated with a value (sometimes multiple values).

In *ordered associative containers* the keys are maintained in sorted order.

Associative Container Classes

In *ordered associative containers* the keys are maintained in sorted order.

- `set`
  rapid lookup, no duplicates allowed
- `multiset`
  rapid lookup, duplicates allowed
- `map`
  one-to-one mapping, no duplicates,
  rapid key-based lookup
- `multimap`
  one-to-many mapping, duplicates allowed,
  rapid key-based lookup

Associative Container Classes

In *unordered associative containers* the keys are unsorted.

- `unordered_set`
  rapid lookup, no duplicates allowed
- `unordered_multiset`
  rapid lookup, duplicates allowed
- `unordered_map`
  one-to-one mapping, no duplicates,
  rapid key-based lookup
- `unordered_multimap`
  one-to-many mapping, duplicates allowed,
  rapid key-based lookup

Some Container Classes

4 major *categories* of container types:

- *sequence containers*
- *ordered associative containers*
- *unordered associative containers*

} *first-class containers*

- *container adapters :* `stacks, queues,` `priority queues` (both category and style)

they are typically constrained versions of sequence containers

Near containers: exhibit some, but not all, capabilities of the first-class containers

built-in arrays,

`bitsets`, for maintaining sets of flag values

`valarrays`, for performing high-speed math. vector operations

Some Container Classes

4 major *categories* of container types:

- *sequence containers*
- *ordered associative containers*
- *unordered associative containers*

} *first-class containers*

- *container adapters*

Some Container Classes

4 major *categories* of container types:

- *sequence containers*
- *ordered associative containers*
- *unordered associative containers*

} *first-class containers*

- *container adapters :* `stacks, queues,` `priority queues` (both category and style)

  they are typically constrained versions of sequence containers

Some Container Classes

4 major *categories* of container types:

- *sequence containers*
- *ordered associative containers*
- *unordered associative containers*

} *first-class containers*

- *container adapters :* `stacks, queues,` `priority queues` (both category and style)

they are typically constrained versions of sequence containers

Near containers: exhibit some, but not all, capabilities of the first-class containers

built-in arrays,

`bitsets`, for maintaining sets of flag values

`valarrays`, for performing high-speed math. vector operations

First-Class Container Common Nested Types

Fig. 15.3 in the book shows a list of common first-class container *types that are defined inside each container class definition* and are used in declarations of variables, parameters to functions, and return values from functions.

First-Class Container Common Nested Types

Fig. 15.3 in the book shows a list of common first-class container *types that are defined inside each container class definition* and are used in declarations of variables, parameters to functions, and return values from functions.

Here are some of them:

`allocator_type` the type of the object used to allocate the container's memory (not used in array class template)

`value_type` the type of the element stored in the container

`reference` a reference for the container's element type

`const reference` a reference for the container's element type that can be used only to perform const operations

`pointer` a pointer to the container class element type

Look up the rest in the textbook.

Requirements for Container Elements

Before using a Standard Library container, it is important to ensure that *the type of objects being stored in the container supports the minimum set of functionality*.

For example,
The object type should provide a *copy constructor* and *copy assignment operator*, because when an object is inserted into a container, a *copy of the object is made*.

Objects must be *comparable* for *ordered associative containers*.

Iterators

Iterators have properties similar to those of pointers, and are used to point to *first-class container* elements.

They hold the state information sensitive to the particular containers on which they operate. Hence, iterators are implemented for each type of the container.

Some iterator operations are uniform across containers. For example, increment, decrement, dereferencing, etc.

First-class containers provide member functions
`begin()` returns an iterator pointing to the first element
`end()` returns an iterator pointing to the end of the container (past the last element, to non-existing element)

see dequeWithIterators.cpp and listWithIterators.cpp

Using `istream_iterator` and `ostream_iterator`

We can use the `istream_iterator` and `ostream_iterator` iterators for input and display.

see inputOutputWithIterators.cpp

Later on, when reading **Section 15.5.1** you will see their powerful application in the vector example (Fig 5.11)

Associative Containers

Let's take a look at the `map` container.

It is an *ordered associative container*, i.e. keys are maintained in sorted order.

It performs fast storage and retrieval of of *unique key* and *associated values*.

It is called one-to-one mapping.

Example: StudentID ⟷ StudentRecord

see mapUse.cpp

HW assignment

1) given in the previous class

2) Write the program that will read the ages of the people from a given file (file name should be requested from the user), store them (choose between three data types: `vector`, `deque`, `map`), and then output the count of each age that was read from the file to display.

For example, given the file `data.txt`:

```
23 67 1 4 7
67 4 1
7 4 23 1
```

The output will be:

```
Age  Count
23     1
67     2
1      3
 ...
```

Self-Study:

Section 15.5.1,

Suggested Practice:

Chapter 15, Self-Review Exercises and other exercises: 15.1 (all, except i, l, o), 15.2 (all except c, g, n, s), 15.6, 15.8, 15.9, 15.13