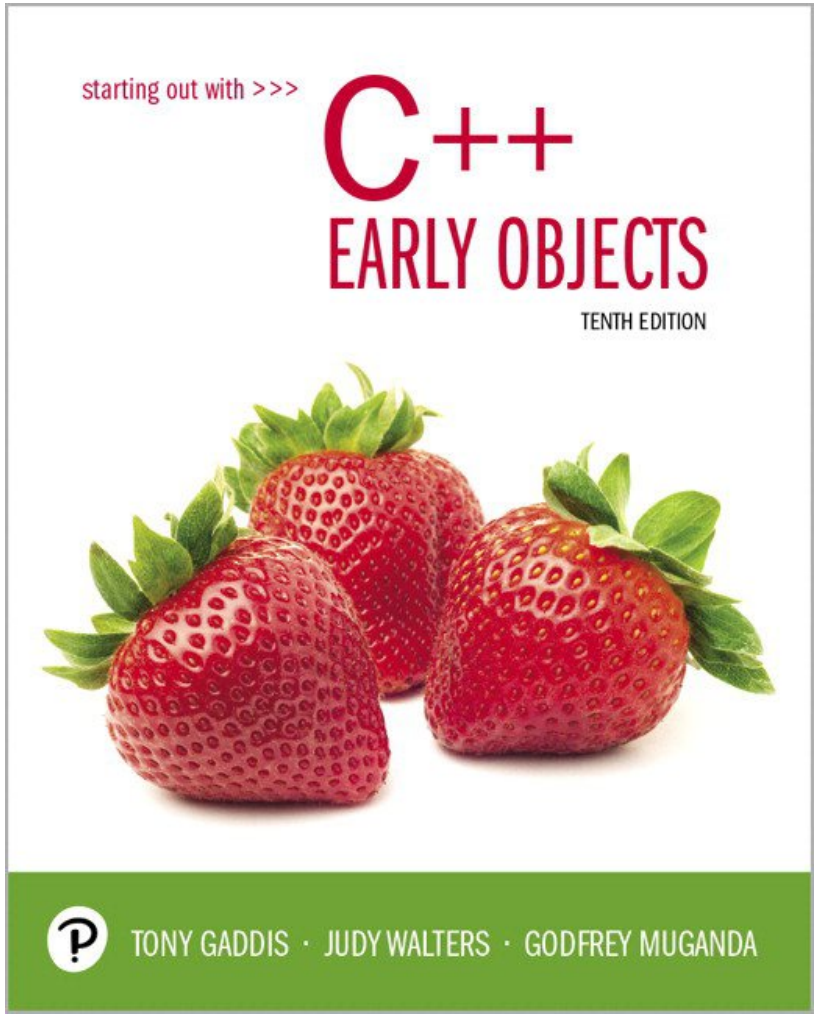
Starting Out with C++ Early Objects

Tenth Edition



Chapter 17

The Standard Template Library

Topics

17.1 Introduction to the Standard Template Library

17.2 STL Container and Iterator Fundamentals

17.3 The **`vector`** class

17.4 The **`map`**, **`multimap`**, and **`unordered_map`** classes

17.5 The **`set`**, **`multiset`**, and **`unordered_set`** classes

17.6 Algorithms

17.7 Introduction to Function Objects and Lambda Expressions

17.1 Introduction to the Standard Template Library

- Standard Template Library (STL): a library containing templates for frequently used data structures and algorithms
- The `vector` class, introduced in Chapter 8, is a data type defined in the STL

17.2 STL Container and Iterator Fundamentals

Two important types of data structures in the STL:

- containers: classes that store collections of data and impose some organization on it
- iterators: like pointers; provide mechanisms for accessing elements in a container

Containers 1 of 2

Two types of container classes in STL:

1) sequential containers: organize and access data sequentially, as in an array. These include **`array`**, **`vector`**, **`deque`**, and **`list`** containers.

Values are stored by position (index) in a sequential container

Containers 2 of 2

2) associative containers: use keys to store and to allow data elements to be quickly accessed. These include **`set`**, **`multiset`**, **`map`**, and **`multimap`** containers and their unordered equivalents.

The key of an associative container plays the role of an index of a sequence container.

Container Adapter Classes

A container adapter class implements rules for how its values are stored and retrieved.

It is not a container itself, but is a class that uses a container in its underlying implementation.

Classes: **`stack`**, **`queue`**, and **`priority_queue`**. These will be covered in Chapter 19.

Containers and Header Files

Programs must **`#include`** specific files in order to use containers.

Some files (ex: **`<array>`**, **`<deque>`**, **`<list>`**, **`<stack>`**, **`<vector>`**) are also the name of the container.

Other files have multiple containers:

`<map>` has **`map`** and **`multimap`**

`<queue>` has **`queue`** and **`priority_queue`**

The `array` Class 1 of 2

- Introduced in C++ 11.
- Fixed size
- All elements are of the same data type.
- Provide data type and size on declaration:

```
array<int, 20> tests;
```

- Can provide initialization list:

```
array<int, 3> primes = {2,3,5};
```

The `array` Class 2 of 2

- Overloaded `[]` operator allows access to elements using an index. It does not perform bounds checking.
- The size() member function returns the declared size of the array:

```
for(int i=0; i<primes/size(); i++)
  cout << primes[i] << endl;
```

The `array` Class – Member Functions

- Member functions provide capabilities beyond ordinary arrays.
- Functions include:
  - tests (**`empty()`**, **`size()`**)
  - data retrieval (**`front()`**, **`back()`**, **`at()`**, **`data()`**)
  - traversal (**`begin()`**, **`end()`**)
  - reverse traversal (**`rbegin()`**, **`rend()`**)

Iterators

- Generalization of pointers. They are used to access information in containers
- Many types:
  - forward (uses `++`)
  - bidirectional (uses `++` and `--`)
  - random-access
  - input (can be used with **`cin`** and **`istream`** objects)
  - output (can be used with **`cout`** and **`ostream`** objects)

Defining an Iterator

- Each container class defines an iterator type, used to access its contents
- The type of an iterator is determined by the type of the container:

```
array<int>::iterator x;
array<string, 3>::iterator y;
```

`x` is an iterator for a container of type `array<int>`

Containers and Iterators

Each container class defines functions that return iterators:

- **`begin()`:** points to the item at the start of the container
- **`end()`:** points to the location just past the end of the container

```
array<int> intArray;
array<int>::iterator aInt =
                    intArray.begin();
```

Defining an Iterator using `auto`

When declaring and initializing an iterator, **`auto`** can be used to simplify the declaration:

```
array<int> intArray;
auto aInt = intArray.begin();
```

Mutable Iterators

- Iterators allow modification of array contents.
- Iterators support pointer-like operations. If **`iter`** is an iterator, then
  - **`*iter`** is the item it points to: this dereferences the iterator
  - **`iter++`** advances to the next item in the container
  - **`iter--`** backs up in the container
- The **`end()`** iterator points to past the end: it should never be dereferenced

Traversing a Container

Given an array:

```
array<int, 5> a = {1, 4, 9, 16, 25};
```

Traverse it using an iterators:

```
auto iter = a.begin();
while (iter != a.end())
   { cout << *iter << " "; iter++}
```

Prints `1 4 9 16 25`

17.3 The `vector` Class

- A **`vector`** is similar to an **`array`**, in that they are both sequence containers.
- Primary differences:
  - A **`vector`** does not have a fixed size. Elements are added and removed, and the **`vector`** adjusts as needed.
  - A **`vector`** has functions unique to its class.

`vector` Class Constructors

Declaration	Description
`vector<int> ivec;`	Creates an empty vector named ivec object that will hold ints
`vector<int> ivec(20);`	Creates an empty vector with initial space for 20 ints
`vector<int> ivec(20,0);`	Creates an empty vector with initial space for 20 ints and initializes each space to 0.
`vector<int> ivec2(ivec);`	Create a vector named ivec2 and initialize it with the values of the elements found in vector ivec. This is a copy constructor. The intial size of ivec2 is taken from the size of the vector that is passed as a parameter.

Some `vector` Class Member Functions

Function	Description
front(), back()	Returns a reference to the first, last element in a vector
size()	Returns the number of elements in a vector
capacity()	Returns the number of elements that a vector can hold
clear()	Removes all elements from a vector
push_back(value)	Adds element containing value as the last element in the vector
pop_back()	Removes the last element from the vector
insert(iter, value)	Inserts new element containing value just before element pointed at by iter
insert(iter, n, value)	Inserts n new elements containing value just before element pointed at by iter

Storing Class Objects in a `vector` Class Object

- A **`vector`** can hold objects of a class as elements as well as primitive data types.
  - ex: `vector<string> names;`
- Use the -> operator with an iterator to refer to member functions of the elements

Inserting Class Objects in a `vector`: `emplace()` and `emplace_back()`

- Traditional ways to add an object to a vector: `insert()`, `push_back()`
- Problem: temporary objects are created, taking time and using space.
- Solution: `emplace()` and `emplace_back()`, introduced in C++ 11.
- Pass the parameters for the constructor of the object to `emplace()` or `emplace_back()`. The function will pass the parameters to the needed constructor.

`vector` Space Management

- `insert()` – number of elements that a vector can currently hold
- `size()` – number of elements currently in a vector
- `max_size()` – maximum nunber of elements that the vector could ever hold
- `reserve()` – request additional capacity to be added to a vector.
- `shrink_to_fit()` – reduce the capacity of a vector to be the same as the size, freeing up unused space.

17.4 The `map`, `multimap`, and `unordered_map` classes

- The **`map`** associative container stores two parts for each element:
  - key: a unique entry associated with the element
  - value: the data/element associated with the key
- Key-value pairs are also called mappings. Each key is mapped to an value.
- Keys must be unique within a map.

`map` Class Methods

- The **`map`** class is generic. The constructors require data types for the key and the value.
- The default constructor creates an empty map. The copy constructor creates a map and populates it with the contents of the map that is passed as a parameter.
- A map can be initialized via an initialization list. Each entry in the list must be a key-value pair in { }.

Adding Elements to a map

```
map<int,double> stuData;  // the student id
// (key) is the int, the GPA is the double
```

- The `[]` operator is overloaded. To assign a GPA to the student whose ID is 12345: `stuData[12345] = 3.25;`
- The **`insert()`** method adds a pair to a map. The pair must be created by **`make_pair`**:

```
stuData.insert(make_pair(54321, 3.05));
```

- The emplace() method will create objects and add them:

```
stuData.emplace(13579, 3.50);
```

Adding Object Elements to a `map`

- Like sequential containers, maps can contain objects of a class.
- Objects must be of classes that have a default constructor.

The `unordered_map` and `multimap` Classes

`unordered_map`:

- Keys are not sorted.
- Has better performance for searching when the map is large.

`multimap`:

- Keys do not have to be unique. Duplicate keys allowed.
- The **`equal_range()`** method supports iterating over all pairs that have the same key value.
- The count() method determines how many pairs there are.

The `unordered_multimap` Class

- Like unordered_map, the keys are not sorted.
- Like multimap, keys do not have to be unique.

17.5 The `set`, `multiset`, and `unordered_set` Classes

- Set: collection of unique values. Internally stored in ascending order.
- Multiset: the values do not have to be unique.

Introduced in C++ 11:

- Unordered_set: Like a set, but no order to the elements
- Unordered_multiset: Like a multiset, but no order to the elements.

Algorithms

- STL contains algorithms that are implemented as function templates to perform operations on containers.
- Requires **`algorithm`** header file
- Some of the algorithms are:

`binary_search`	**`count`**
`for_each`	**`find`**
`max_element`	**`min_element`**
`random_shuffle`	**`sort`**

and others

Using STL algorithms

- Many STL algorithms manipulate portions of STL containers specified by a begin and end iterator
- **`sort(iter1, iter2)`** sort the elements between **`iter1`** and **`iter2`** into ascending order
- **`binary_search(iter1, iter2, value)`** determine if value is present in the container between the two iterators.

Function Pointers and Related Functions

- The executable code for a function can be accessed via a function pointer.
- Like an array, the name of a function serves as its address.
- The STL has functions that can pass container elements to functions.
- Ex: `for_each()`, `count_if()`

Set Operations on Containers

- The STL includes functions to perform set operations: union, intersection, difference, symmetric difference, and inclusion.
- The functions take iterators as parameters that mark the beginning and end of the ranges to use for each set and for the location to store the results.
- For inclusion, the function returns true or false to indicate if all of the elements are in the set.

17.7 Function Objects and Lambda Expressions

- The function operator, `()`, can be overloaded.
- An object of a class that overloads the function operator is a function object, or a functor.

Function Object Example 1 of 2

A class that overloads `()`

```
class Multiply
{
  public:
     int operator() (int x, int y)
          {
             return x * y;
          }
} // end class Multiply
```

Function Object Example 2 of 2

Create and use objects:

```
Multiply prod;
int product = prod(4,3);
```

You can create and use in the same step:

```
int product = Multiply()(4,3);
```

The object created in the previous example is anonymous.

Predicates

- Predicate: A function that returns a Boolean value
- Function objects can be created to perform tasks associated with predicates.
- Unary Predicate: takes one argument
- Binary Predicate: takes two arguments

Unary Predicate

- Unary Predicate: A predicate that takes a single argument

```
class IsPositive
{
  public:
     bool operator() (int x)
          {
             return x > 0;
          }
} // end class IsPositive
```

Binary Predicate

- Binary Predicate: A predicate that takes two arguments

```
class GreaterThan
{
  public:
     bool operator() (int x, int y)
          {
             return x > y;
          }
} // end class GreaterThan
```

Lambda Expression 1 of 2

- A lambda expression provides a simplified way to create a function object.
- Examples:

```
// Multiply
[] (int x, int y){ return x * y; }
// IsPositive
[] (int x){ return x > 0; }
```

- Assigning a name to a lambda expression:

```
auto isPositive = [](auto x) {return x>0;};
```

Lambda Expression 2 of 2

In the previous slide,

- `[]` is the lambda introducer. The lambda expression begins here.
- The `[]` may include a capture list of variables in the surrounding scope of the lambda expression that can be used in the function body.
- The parenthesized list following `[]` is the parameter list. The parameters are used by the operator() member function.
- The code in the `{}` is the function body.

Functional Classes in the STL

- Requires the **`<functional>`** header file
- Some examples:

Function Object	Description
`less<T>`	`less<T>()(T a, T b)` is true if and only if `a < b`
`less_equal<T>`	`less_equal()(T a, T b)` is true if an only if `a <= b`
`greater<T>`	`greater<T>()(T a, T b)` is true if and only if `a > b`
`greater_equal<T>`	`greater_equal<T>()(T a, T b)` is true if and only if `a >= b`

Passing Function Objects to Functions

- Function objects can be passed as parameters to functions.
- Examples:
  - **`sort()`**, used for sorting an array or vector. The function parameter is used to determine how to compare two values in the array and determine how to order them
  - **`remove_if()`**, used for removing elements from an array or vector that meet a criteria. The function parameter is used to define the criteria.

Copyright