Data Structures

Ahsan Ijaz

Sequential Containers

A container holds a collection of objects of a specified type. The **sequential containers** let the programmer control the order in which the elements are stored and accessed. That order does not depend on the values of the elements. Instead, the order corresponds to the position at which elements are put into the container.

- Each container has different trade-offs relative to:
  - The costs to add or delete elements to the container
  - The costs to perform nonsequential access to elements of the container

Types of Sequential Containers

`vector`	Flexible-size array. Supports fast random access. Inserting or deleting elements other than at the back may be slow.
`deque`	Double-ended queue. Supports fast random access. Fast insert/delete at front or back.
`list`	Doubly linked list. Supports only bidirectional sequential access. Fast insert/delete at any point in the `list`.
`forward_list`	Singly linked list. Supports only sequential access in one direction. Fast insert/delete at any point in the list.
`array`	Fixed-size array. Supports fast random access. Cannot add or remove elements.
`string`	A specialized container, similar to `vector`, that contains characters. Fast random access. Fast insert/delete at the back.

Figure: Types of Sequential Containers

Sequential Containers

Containers provide:

- Efficient, flexible memory management.
- We can add and remove elements
- Size of containers can grow and shrink.

Strings and Vectors

String and vector hold their elements in contiguous memory.
Pros:

- Since elements are contiguous, it is fast to compute the address of an element from its index.

Cons:

- Adding or removing elements in the middle of one of these containers takes time:

  All the elements after the one inserted or removed have to be moved to maintain contiguity.
  adding an element can sometimes require that additional storage be allocated. In that case, every element must be moved into the new storage.

list and forward_List

Pro: The list and forward_list containers are designed to make it fast to add or remove an element anywhere in the container.
Con:

- In exchange, these types do not support random access to elements: We can access an element only by iterating through the container.
- The memory overhead for these containers is often substantial, when compared to vector, deque, and array.

Rule of thumb for selection of Container

- Unless you have a reason to use another container, use a vector.
- If your program has lots of small elements and space overhead matters, dont use list or forward_list.
- If the program requires random access to elements, use a vector or a deque.
- If the program needs to insert or delete elements in the middle of the container, use a list or forward_list.
- If the program needs to insert or delete elements at the front and the back, but not in the middle, use a deque.

Rule of thumb for selection of Container

- Unless you have a reason to use another container, use a vector.
- If your program has lots of small elements and space overhead matters, dont use list or forward_list.
- If the program requires random access to elements, use a vector or a deque.
- If the program needs to insert or delete elements in the middle of the container, use a list or forward_list.
- If the program needs to insert or delete elements at the front and the back, but not in the middle, use a deque.

Rule of thumb for selection of Container

- Unless you have a reason to use another container, use a vector.
- If your program has lots of small elements and space overhead matters, dont use list or forward_list.
- If the program requires random access to elements, use a vector or a deque.
- If the program needs to insert or delete elements in the middle of the container, use a list or forward_list.
- If the program needs to insert or delete elements at the front and the back, but not in the middle, use a deque.

Rule of thumb for selection of Container

- Unless you have a reason to use another container, use a vector.
- If your program has lots of small elements and space overhead matters, dont use list or forward_list.
- If the program requires random access to elements, use a vector or a deque.
- If the program needs to insert or delete elements in the middle of the container, use a list or forward_list.
- If the program needs to insert or delete elements at the front and the back, but not in the middle, use a deque.

Rule of thumb for selection of Container

- Unless you have a reason to use another container, use a vector.
- If your program has lots of small elements and space overhead matters, dont use list or forward_list.
- If the program requires random access to elements, use a vector or a deque.
- If the program needs to insert or delete elements in the middle of the container, use a list or forward_list.
- If the program needs to insert or delete elements at the front and the back, but not in the middle, use a deque.

Container Operations

Each container provides a set of common operations (methods) for using the data stored in them.

Construction of Containers

Construction	
`C c;`	Default constructor, empty container (`array`; see p. 336)
`C c1(c2);`	Construct `c1` as a copy of `c2`
`C c(b, e);`	Copy elements from the range denoted by iterators `b` and `e`; **(not valid for `array`)**
`C c{a,b,c...};`	List initialize `c`

Figure: Construction

Assignment and swap operations

Assignment and swap	
`c1 = c2`	Replace elements in `c1` with those in `c2`
`c1 = {a,b,c...}`	Replace elements in `c1` with those in the list **(not valid for `array`)**
`a.swap(b)`	Swap elements in `a` with those in `b`
`swap(a, b)`	Equivalent to `a.swap(b)`
Size	
`c.size()`	Number of elements in `c` **(not valid for `forward_list`)**
`c.max_size()`	Maximum number of elements `c` can hold
`c.empty()`	`false` if `c` has any elements, `true` otherwise

Figure: Assignment and Swap Operations

Addition and removal of elements

Add/Remove Elements (*not valid for* `array`)

Note: the interface to these operations varies by container type

`c.insert(`*args*`)`	Copy element(s) as specified by *args* into `c`
`c.emplace(`*inits*`)`	Use *inits* to construct an element in `c`
`c.erase(`*args*`)`	Remove element(s) specified by *args*
`c.clear()`	Remove all elements from `c`; returns `void`

Figure: Adding and removing elements

Defining Containers

```
1    list<Sales_data> // list that holds Sales_data
2    deque<double> // deque that holds doubles
```

```
1    vector<vector<string>> lines; // vector of vector
```

Here lines is a vector containing vectors of strings.

Iterators:motivation

- Need a way to navigate through the items in a container. An example: navigating over vector v.

```
1  for (int i = 0; i != v.size(); i++)
2  cout << v[i] << endl;
```

- However, doubly-linked list would need a different form to fetch data. We want a general approach to navigate elements for different implementations of a container.

Iterators

Definition

Like Pointers, iterators gives indirect access to objects in a container.

We can use an iterator to fetch an element and iterators have operations to move from one element to another.

```
1 vector<int>::iterator it; // it can read and write
2 vector<int>::const_iterator it2; // it2 can read
3 // but not write elements
```

Iterators job

A generalized type that helps in navigating a container

- A way to initialize at the front and back of a list
- A way to move to the next or previous position
- A way to detect the end of an iteration
- A way to retrieve the current value

Getting an iterator

Two methods in all STL containers.

- iterator begin ( ) Returns an iterator to the first item in the container
- iterator end ( ) Returns an iterator representing end marker in the container (that is, the position after the last item)

```
1 for (int i = 0; i != v.size(); i++ )
2 cout << v[i] << endl;
```

can be written as:

```
1 for(vector<int>::iterator itr=v.begin(); itr!=v.end(); itr++)
2 {
3 cout << *itr++ << endl;
4 }
```

auto keyword

auto keyword automatically checks and assigns the appropriate type to the variable.

```
1  // type is explicitly specified
2  list<string>::iterator it5 = a.begin();
3  list<string>::const_iterator it6 = a.begin();
4  // iterator or const_iterator depending on a's type
5
6  auto it7 = a.begin(); // const_iterator only if a i
7  auto it8 = a.cbegin(); // it8 is const_iterator
```

When we use auto with begin or end, the iterator type we get depends on the container type.

Initialize container

```
1  // ten int elements, each initialized to -1
2  vector<int> ivec(10, -1);
3   // ten strings; each element is "hi!"
4  list<string> svec(10, "hi!");
5  // ten elements, each initialized to 0
6  forward_list<int> ivec(10);
7  // ten elements, each an empty string
8  deque<string> svec(10);
```

Initializing a container from another container

```
1    /* each container has three elements, initialized
2  from the given initializers*/
3  list<string> authors = {"Milton", "Shakespeare", "Austen"};
4  vector<const char*> articles = {"a", "an", "the"};
5  list<string> list2(authors); // ok: types match
6  deque<string> authList(authors); //error: container types mismatch
7  vector<string> words(articles); // ok: converts const char*
8  // elements to string
```

Relational Operators

- The right- and left-hand operands must be the same kind of container and must hold elements of the same type. That is, we can compare a vector<int> only with another vector<int>. We cannot compare a vector<int> with a list<int> or a vector<double>.

Comparing two containers performs a pairwise comparison of the elements.

- If both containers are the same size and all the elements are equal, then the two containers are equal; otherwise, they are unequal.
- If the containers have different sizes but every element of the smaller one is equal to the corresponding element of the larger one, then the smaller one is less than the other.

Relational Operators-Example

```
1  vector<int> v1 = { 1, 3, 5, 7, 9, 12 };
2  vector<int> v2 = { 1, 3, 9 };
3  vector<int> v3 = { 1, 3, 5, 7 };
4  vector<int> v4 = { 1, 3, 5, 7, 9, 12 };
5  v1 < v2
6  /* true; v1 and v2 differ at
7  element [2]: v1[2] is less than v2[2] */
8  v1 < v3
9  /* false; all elements are equal, but
10 v3 has fewer of them; */
11 v1 == v4
12 /* true; each element is equal and v1 and v4
13  have the same size() */
14 v1 == v2 // false; v2 has fewer elements than v1
```

Element type Dependency

- We can use a relational operator to compare two containers only if the appropriate comparison operator is defined for the element type.

```
1  vector<Sales_data> storeA, storeB;
2  if (storeA < storeB) // error: Sales_data has
3                       // no less-than operator
```