
NJIT
THE EDGE IN KNOWLEDGE

CS 280
Programming Language Concepts

Spring 2023

Standard Template Library (STL)

(Containers)

vector, list, queue, and stack

Standard Template Library (STL)

- C++ provides templated types for collections
- The templates come with methods to perform operations on the collection
- The type of what is in the collection is part of the declaration. That means that it is a compile-time parameter
- The collections are thus strongly typed and type-safe for what they contain
- Powerful, template-based components
  - Containers: template data structures
  - Iterators: like pointers, access elements of containers
  - Algorithms: data manipulation, searching, sorting, etc.
- Object- oriented programming: reuse, reuse, reuse

Introduction to Containers

- Three types of containers
  - Sequence containers
    - Linear data structures (vectors, linked lists)
    - First-class container
  - Associative containers
    - Non-linear, can find elements quickly (map, multimap)
    - Key/value pairs
    - First-class container
  - Container adapters
- Containers have some common functions

Common STL Member Functions

- Member functions for all containers
  - Default constructor, copy constructor, destructor
  - **`empty`**
  - **`max_size`**, **`size`**
  - **`= < <= > >= == !=`**
  - **`swap`**
- Functions for first-class containers
  - **`begin`**, **`end`**
  - **`rbegin`**, **`rend`**
  - **`erase`**, **`clear`**

Sequence Containers

- Three sequence containers
  - **`vector`** - based on arrays
  - **`deque`** - based on arrays
  - **`list`** - robust linked list

Iterators

- Definition:
  - An *iterator* is any object that points to some element in a range of elements (such as an array or a container such as map),
  - It has the ability to iterate through the elements of that range using a set of operators (with at least the increment (++) and dereference (*) operators).
  - The most obvious form of iterator is a *pointer*: A pointer can point to elements in an array, and can iterate through them using the increment operator (++).
  - Iterators in the STL come in two flavors: forward and reverse
    - Typically, a container has methods called **begin** and **end** that return forward iterators pointing to its beginning and its end, respectively.
    - Methods called **rbegin** and **rend** return reverse iterators.

Iterators

- Each iterator must be declared for the specific type of container with which it will be used.
- For example, each container type (such as a map or list) has a specific *iterator* type designed to iterate through its elements. For example, if we need iterators for a list of strings, we would declare them as follows:

  list<string>:: iterator position;

  list<string>:: reverse_iterator r_position;

- For more details see:
  https://www.cplusplus.com/reference/iterator/

Vectors

- Available via #include <vector>
- A vector is an array of adjustable size
- A vector keeps track of “size” and “capacity”. It tries to be efficient in how it uses memory
- Constant time access to entries in the vector

Vectors

- **`vector`**
  - **`Header <vector>`**
  - Data structure with contiguous memory locations
    - Access elements with **`[]`**
  - Use when data must be sorted and easily accessible
- When memory exhausted
  - Allocates larger, contiguous area of memory
  - Copies itself there
  - Deallocates old memory
- Has random access iterators

Vectors

- Declarations
  - **`std::vector <`*`type`*`> v;`**
    - ***`type`***: **`int`**, **`float`**, etc.
- Iterators
  - **`std::vector<`*`type`*`>::const_iterator iterVar;`**
    - **`const_iterator`** cannot modify elements
  - **`std::vector<`*`type`*`>::reverse_iterator iterVar;`**
    - Visits elements in reverse order (end to beginning)
    - Uses **`rbegin`** to get starting point
    - Uses **`rend`** to get ending point

Vectors

- **`vector`** functions
  - **`v.push_back(value)`**
    - Adds element to end (found in all sequence containers).
  - **`v.size()`**
    - Current size of vector
  - **`v.capacity()`**
    - How much vector can hold before reallocating memory
    - Reallocation doubles size
  - **`vector<`*`type`*`> v(a, a + SIZE)`**
    - Creates **`vector v`** with elements from array **`a`** up to (not including) **`a + SIZE`**

Vectors

- **`vector`** functions
  - **`v.insert(`***iterator*`,` *value* **`)`**
    - Inserts *value* before location of *iterator*
  - **`v.insert(`***iterator*`,` *array* `,` *array* + *SIZE*`)`**
    - Inserts array elements (up to, but not including *array* + *SIZE)* into vector
  - **`v.erase( iterator )`**
    - Removes element from container
  - **`v.erase( iter1, iter2 )`**
    - Removes elements starting from **`iter1`** and up to (not including) **`iter2`**
  - **`v.clear()`**
    - Erases entire container

Vectors

- **`vector`** functions operations
  - **`v.front(), v.back()`**
    - Returns first and last element
  - **`v[elementNumber] = value;`**
    - Assigns **`value`** to an element
  - **`v.at[elementNumber] =value;`**
    - As above, with range checking
    - **`out_of_bounds`** exception

deque Sequence Container

- **`deque`** ("deek"): double-ended queue
  - Header **`<deque>`**
  - Indexed access using **`[]`**
  - Efficient insertion/deletion in front and back
  - Non-contiguous memory: has "smarter" iterators
- Same basic operations as **`vector`**
  - Also has
    - **`push_front`** (insert at front of **`deque`**)
    - **`pop_front`** (delete from front)

Container Adapters

- **`stack`**, **`queue`** and **`priority_queue`**
- Not first class containers
  - Do not support iterators
  - Do not provide actual data structure
- Programmer can select implementation
- Member functions **`push`** and **`pop`**

stack

- Header **<stack>**
- Insertions and deletions at one end (top of stack)
- Last-in, first-out (LIFO) data structure
- Can use **vector**, **list**, or **deque** (default)
- Declarations

```
stack<type, vector<type> > myStack;
stack<type, list<type> > myOtherStack;
stack<type> anotherStack; // default deque
```

  - **Implementation of a stack using vector, list, or deque**
    - Implementation of **stack** (default **deque**)
    - Does not change behavior, just performance (**deque** and **vector** fastest)

```
#include <iostream>
using std::cout
using std::endl;
#include <stack>   // stack adapter definition
int main(){
  // stack with default underlying deque
  std::stack< int > intStack;
  // push the values 0-9 onto stack
  for ( int i = 0; i < 10; ++i ) {
      intStack.push( i );
  } // end for
  // display and remove elements from each stack
  cout << "Popping from inttack: ";
  while ( !intStack.empty() ) {
    cout << intStack.top() << ' ';  // view top element
    intStack.pop();                 // remove top element
  } // end while
  cout << endl;
  return 0;
} // end main
```

```
Popping from intStack: 9 8 7 6 5 4 3 2 1 0
```

queue

- Header **<queue>**
- Insertions at back, deletions at front
- First-in-first-out (FIFO) data structure
- Implemented with **list** or **deque** (default)
  - **std::queue<double> values;**
- Functions
  - **push(element)**
    - Same as **push_back**, add to end
  - **pop(element)**
    - Implemented with **pop_front**, remove from front
  - **empty()**
  - **size()**

```
#include <iostream>
using std::cout;
using std::endl;
#include <queue>  // queue adapter definition
int main(){
   std::queue< double > values;
   // push elements onto queue values
   values.push( 3.2 );
   values.push( 9.8 );
   values.push( 5.4 );
   cout << "Popping from values: ";
   while ( !values.empty() ) {
     cout << values.front() << ' ';  // view front element
     values.pop();                   // remove element
   } // end while
   cout << endl;
   return 0;
} // end main
```

RA 7: Postfix Expression Evaluator

- A postfix expression is defined as an arithmetic expression where the operator occurs after the operands. For example, using infix notation we would add **4** and **5** using **4 + 5**. In postfix notation we would arrange the operator to occur at the end: **4 5 +**. A postfix expression can become arbitrarily complex such as **3 4 5 + − 2 ***. This expression would be expressed using infix notation like this: **((4 + 5) − 3) * 2**. As you can see, the postfix form does not require parentheses because the order of operations is made explicit. It turns out that evaluating postfix expressions is quite easy when combined with a stack data structure.

RA 7: Postfix Expression Evaluator

- We read a postfix expression from left to right pushing values onto the stack when a number is encountered and popping numbers from the stack when an operator is encountered. For example, the expression **4 5 +** would require the following operations:
  1. **Push 4 onto the stack**.
  2. **Push 5 onto the stack**.
  3. **Pop the top two values of the stack, add them, and then push the result back onto the stack**

RA 7: Postfix Expression Evaluator

- Write a C++ function that allows a user to evaluate postfix expressions. In this assignment, postfix expressions are described by using integer constants and variables (restricted to one letter identifiers), as operands, and supports the basic arithmetic operators such as +, −, *, and /, for addition, subtraction, multiplication, and division, respectively. To allow using variables, you should use the **map** container from the STL to map variable names (e.g., X) to integer values. The postfix expression evaluator supports variables as follows. When a variable is encountered from the input, it should pop the current value off the top of the stack and associate it with that variable in the map. If the variable is preceded by a '$', the postfix evaluator should simply retrieve the variable's associated value from the map container and push it onto the stack.

RA 7: Postfix Expression Evaluator

- The function scans the input string left to right. When the scanning is completed, the function pops the top of the stack if it is not empty. If the stack is empty, the function displays the following message, then returns.

```
Error: Incomplete input postfix expression.
```

- The popped value from the stack is the final result, and can be printed out if the stack becomes empty afterwards. If the stack does not become empty after popping the final result, then there is an error and the function should display the following message, and return back.

```
The evaluation is incomplete, missing input operators.
```

- For example, the evaluation of the following simple postfix expression produces the result 27 to be displayed on the console.

```
4 5 + X $X 3 *
```

RA 7: Postfix Expression Evaluator

- Given the above expression, the function should print out the result as follows:

```
The result of evaluating the postfix expression "4 5 + X $X 3 *" is the value: 27
```

- For an incomplete/invalid postfix expression similar to the following expression: `X 4 5 +`
  - the function should display the error message:

```
Error: Incomplete input postfix expression.
```

- Implement the C++ postfix expression evaluator based on the following header definition:

```
void PostfixEval(string instr);
```

  - Where, the instr is the passed input string to the function. The postfix expression evaluator should check for errors due to illegal operator symbol, incomplete input postfix expression, or invalid string/identifier. and print out messages as described in the examples shown above and in the slides.

RA 7: Examples

- Example 1:

```
Input string: 5 12 4 - 15 * +
Result:
The result of evaluating the postfix expression "5 12 4 -
15 * +" is the value: 125
```

- Example 2:

```
Input string: 137 45 21 % -
Result:
Error: Invalid string "137 45 21 % -”
```

RA 7: Examples

- Example 3:

```
Input string: 152 45 x 4 + y 12 $y $x - *
Result:
The result of evaluating the postfix expression "152 45
x 4 + y 12 $y $x - *" is the value: 1332
```

- Example 4:

```
Input string: 5 12 4 - 15 * 5
Result:
The evaluation is incomplete, missing input operators.
```


NJIT
THE EDGE IN KNOWLEDGE