

The C++ Standard Template Library (STL)

- C++ Templates
- Containers Library
 - array, **vector**, list, queue, stack,...
- Iterators Library
- Algorithms Library
 - searching, sorting, reversing,...

The C++ Standard Template Library (STL)

- The C++ standard template library (STL) is a collection of tools that are commonly used in software development such as containers (variables that can store several objects), algorithms (functions that do typical tasks such as sorting, searching, etc) and iterators (variables that enable us to process elements of containers)
- They are C++ classes and functions that are already implemented and are readily available for us
- Moreover they are packaged inside the **std** namespace
- Hence we need to include the class name of the container and also use the std namespace whenever we need to use any of the tools in the STL library
- Importantly these libraries are templated libraries; that is they can work with any valid data type such as primitive data types (int, float, char, double, bool, ...) or programmer defined data types (structs and classes)

C++ Vector

- An implementation of an array that can
 - ✓ Store any data type, and
 - ✓ Whose size can expand (appending more elements to the container) and shrink (removing elements from the container)
- It comes with many member functions designed to work with vector objects such as
 - ✓ **Default constructor:** construct a vector with 0 elements in it
 - ✓ **Non-default constructor:** construct with a specified size (all elements initialized some default values)
 - ✓ **Non-default constructor:** construct with specified size and an appropriate single value of element type (all elements will be initialized to the same value)
 - ✓ **Copy constructor:** construct a deep copy of another vector object of the same data type
 - ✓ **Assignment operator:** assign a deep copy of a vector object of the same data type
 - ✓ **Destructor:** that deletes the elements of the vector and sets the size to zero
 - ✓ **size:** member function that returns the number of elements in the vector
 - ✓ **operator []:** the indexing operator
 - ✓ **push_back:** appends the value to the vector as a last element and increment size by 1
 - ✓ **pop_back:** delete the last element in the vector and reduce size by 1
 - ✓ **front:** returns the first element of the vector
 - ✓ **back:** returns the last element in the vector
 - ✓ **empty:** test if the vector is empty (that is if size is zero). Returns true if the vector is empty.
 - ✓ **erase:** remove one or more values from the vector
 - ✓ **Insert:** insert one or more values into the vector

C++ vectors construction

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main()
{
    vector<int> A1;
    vector<float> A2(5);
    vector<string> A3(3, "FIC");
    vector<int> A4(A1);
    cout << A1.size() << ", " << A2.size() << ", " << A3.size() << ", and " << A4.size() << endl;

    cout << "Elements of vector A1" << endl;
    for (int i = 0; i < A1.size(); i++)
        cout << A1[i] << endl;
    cout << endl << endl;

    cout << "Elements of vector A2" << endl;
    for (int i = 0; i < A2.size(); i++)
        cout << A2[i] << endl;
    cout << endl << endl;

    cout << "Elements of vector A3" << endl;
    for (int i = 0; i < A3.size(); i++)
        cout << A3[i] << endl;
    cout << endl << endl;

    cout << "Elements of vector A4" << endl;
    for (int i = 0; i < A4.size(); i++)
        cout << A4[i] << endl;
    cout << endl << endl;

    system("Pause");
    return 0;
}
```

Remark:- The vector class does not have the output streaming operator friend function. As such the following statement is syntactically wrong.

cout << A2 << endl;

Appending elements to a vector

- The most common operation with vectors is appending element to a vector
- Appending of a new element to a vector is performed using the **push_back** member function
- This function takes a value of the same type as the elements of the vector and appends the value to the vector (inserts the value at the end of the vector)
- See the following example...

Appending elements to a vector

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> a;
    //push_back five integers to the vector
    for (int i = 0; i < 5; i++)
        a.push_back(2*i+1);
    //print the vector
    for (int i = 0; i < a.size(); i++)
        cout << a[i] << " ";
    cout << endl;
    system("Pause");
    return 0;
}
```

Output: 1 3 5 7 9

Pointers to vectors

- We can also declare a pointer to a vector object as we do with simple data types
- Moreover we can use pointers to create vectors on the heap
- We use a de-referenced pointer together with a dot operator in order to access member functions of vectors
- Alternatively, we can use the -> operator which won't require de-referencing

Pointers to vectors

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<double> *a; //pointer to a vector of double data type
    a = new vector<double>; //default constructor on the heap
    for (int i = 0; i < 5; i++)
        a->push_back(1.0*rand()/RAND_MAX * 10 - 5);
    cout << "The vector has " << a->size() << " elements." << endl;
    for (int i = 0; i < a->size(); i++)
        cout << (*a)[i] << endl;
    delete a;
    cout << "After deleting the size of the vector is " << a->size() << endl;
    system("Pause");
    return 0;
}
```

The same as saying
cout << a->operator[](i) << endl;

This is a runtime error. Explain.

References to vectors

- Similarly, we may declare a reference variable to a vector
- Since a reference variable is essentially referring to the same object as the referenced variable, any modification made to either the variables modifies the other
- See the following example...

References to vectors

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<double> a;
    for (int i = 0; i < 5; i++)
        a.push_back(1.0*rand()/RAND_MAX * 10 - 5);
    vector<double> &b = a;
    cout << "The vector has " << b.size() << " elements." << endl;
    cout << "The elements are" << endl;
    for (int i = 0; i < a.size(); i++)
        cout << a[i] << endl;
    cout << "Using the reference variable, the elements are" << endl;
    for (int i = 0; i < b.size(); i++)
        cout << b[i] << endl;
    system("Pause");
    return 0;
}
```

Passing vectors to functions

- We can also pass vectors as arguments to functions
- The parameter passing can be by value, by pointer or by reference
- When using parameter passing by value, the parameter of the function gets a copy of the argument and therefore any modification made to the function parameter does not modify the argument
- When using parameter passing by reference or by pointer (together with de-referencing), then any modification made to the function parameter will also modify the function argument
- See the following example...

Passing vectors to functions

```
#include <iostream>
#include <vector>
using namespace std;

void populateVector1(vector<int> x, const int n)
{
    for (int i = 0; i < n; i++)
        x.push_back(rand() % 31 - 10);
}
void populateVector2(vector<int> *x, const int n)
{
    for (int i = 0; i < n; i++)
        x->push_back(rand() % 31 - 10);
}
void populateVector3(vector<int> &x, const int n)
{
    for (int i = 0; i < n; i++)
        x.push_back(rand() % 31 - 10);
}
void printVector(const vector<int> &x)
{
    for (int i = 0; i < x.size(); i++)
        cout << x[i] << " ";
    cout << endl;
}
int main()
{
    vector<int> a, b, c;
    int n;
    cout << "How many elements would you like to have in the vector? ";
    cin >> n;
    populateVector1(a, n);
    populateVector2(&b, n);
    populateVector3(c, n);
    cout << "After the function calls,..." << endl;
    cout << "Vector a = "; printVector(a);
    cout << "Vector b = "; printVector(b);
    cout << "Vector c = "; printVector(c);
    system("Pause");
    return 0;
}
```

Returning vectors from functions

- We can also return a vector from a function
- We can return the value of a vector, a reference to a vector, or a pointer to a vector
- Returning a vector by value will return a copy of the vector object to be returned
- On the other hand returning by reference or by pointer, returns the vector to be returned without making a copy
- Thus we should not return a local vector object by pointer or reference because the vector object will be destructed when we go out of the function
- We can also use the copy constructor or the assignment operator in order to copy the returned vector to a vector of the same data type
- Remember the copy constructor and the assignment operator are both overloaded in the vector class and as such a deep copy of the right hand side operand is copied or assigned to the left hand side operand
- See the following example...

Returning vectors from functions

```
#include <iostream>
#include <vector>
using namespace std;

vector<int> getPopulatedVector(const int n)
{
    vector<int> x;
    for (int i = 0; i < n; i++)
        x.push_back(rand() % 31 - 10);
    return x;
}

void printVector(const vector<int> &x)
{
    for (int i = 0; i < x.size(); i++)
        cout << x[i] << " ";
    cout << endl;
}

int main()
{
    int n;
    cout << "How many elements would you like to have in the vector? ";
    cin >> n;
    vector<int> a;

    a = getPopulatedVector(n);
    cout << "After the function call, vector a = ";
    printVector(a);
    system("Pause");
    return 0;
}
```

Deleting the last element of a vector

- In order to delete the last element of a vector, use the **pop_back()** member function
- In order to test this member function, perform the following steps:
 - ✓ Create a vector and push_back two or more values
 - ✓ Print the first element using **front()** member function
 - ✓ Print the last element using **back()** member function
 - ✓ Delete the last element using **pop_back()** member function
 - ✓ Print the first element using **front()** member function [You must get the same front element as before]
 - ✓ Print the last element using **back()** member function [You must get a different last element]

Deleting the last element of a vector

```
#include <iostream>
#include <vector>
using namespace std;

void printVector(const vector<char> &x)
{
    for (int i = 0; i < x.size(); i++)
        cout << x[i] << " ";
    cout << endl;
}

int main()
{
    vector<char> a;
    for (int i = 0; i < 5; i++)
        a.push_back(97+i);
    cout << "The elements of the vector are: ";
    printVector(a);
    cout << "The front element of the vector is " << a.front() << endl;
    cout << "The back element of the vector is " << a.back() << endl;

    //Delete the last element
    a.pop_back();

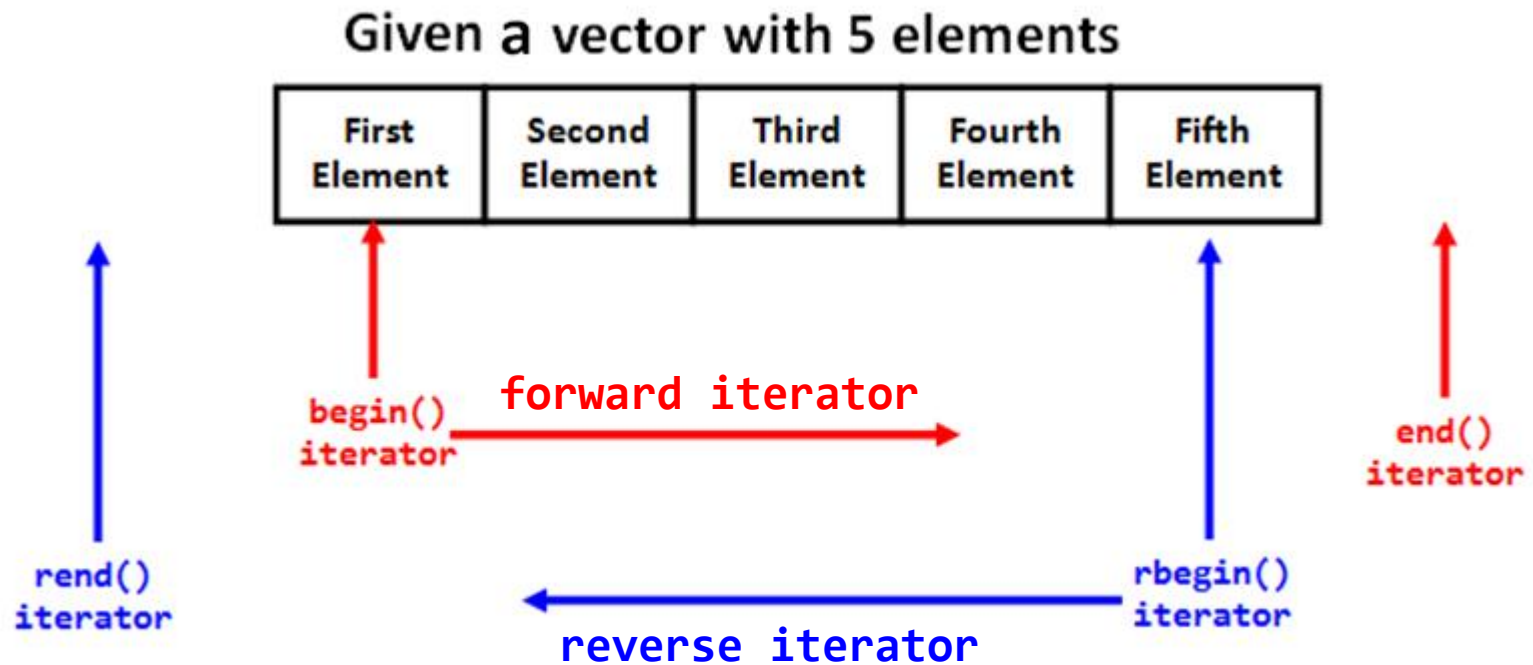
    cout << "The elements of the vector after deleting the last element are: ";
    printVector(a);
    cout << "The front element of the vector is " << a.front() << endl;
    cout << "The back element of the vector is " << a.back() << endl;

    system("Pause");
    return 0;
}
```


Iterators

- In C++, an iterator is a position specifier in a vector or other containers
- They are implemented as pointers
- There are two types of iterators: **forward iterators** and **reverse iterators**
- When a forward iterator is incremented, it moves forward
- When a reverse iterator is incremented, it moves backwards
- There are four pre-defined iterator values for any container:
 - ✓ **begin()**: forward iterator value specifying the beginning of a container
 - ✓ **end()**: forward iterator value specifying the end of a container
 - ✓ **rbegin()**: reverse iterator value specifying the beginning of a container in a reverse order
 - ✓ **rend()**: reverse iterator value specifying the end of a container in a reverse order
- Just like with pointers, we can perform arithmetic operations with iterators to locate any element of a container
- It should be noted that forward and reverse iterators are different data types and therefore can not be used interchangeably
- See the following example...

Iterators



Iterators

```
int main()
{
    //Construct a vector
    vector<int> a;
    for (int i = 0; i < 5; i++)
        a.push_back(4*i);

    //Print the elements of the vector using forward iterator
    cout << "The elements of the vector are" << endl;
    vector<int>::iterator it;
    for (it = a.begin(); it < a.end(); it++)
        cout << *it << endl;

    //Print the elements of the vector using reverse iterator
    cout << "The elements of the vector in reverse are" << endl;
    for (vector<int>::reverse_iterator itr = a.rbegin(); itr < a.rend(); itr++)
        cout << *itr << endl;

    //Modify an element using an iterator
    it = a.begin() + 3;
    *it = -1;

    //Print the elements of the vector using forward iterator
    cout << "The elements of the modified vector are" << endl;
    for (it = a.begin(); it < a.end(); it++)
        cout << *it << endl;

    system("Pause");
    return 0;
}
```

Constant Iterators

- If a container (such as a vector) is a constant, then neither a forward nor a reverse iterator can be used with such a container
- Why? Because iterators are designed to access elements of containers for read and write purposes but a constant container can not be modified
- For this reason, C++ provides constant iterators that can be used with constant containers
- These constant iterators are called **const_iterator** and **const_reverse_iterator**
- The `const_iterator` traverses the elements of a container in the forward order while the `const_reverse_iterator` traverses the elements of a container in a reverse order
- See the example below

Constant Iterators

```
void printVectorForwardReverse(const vector<double> &x)
{
    cout << "The elements of the vector in a forward order are" << endl;
    for (vector<double>::const_iterator it = x.begin(); it < x.end(); it++)
        cout << *it << " ";
    cout << endl;
    cout << "The elements of the vector in a reverse order are" << endl;
    for (vector<double>::const_reverse_iterator it = x.rbegin(); it < x.rend(); it++)
        cout << *it << " ";
    cout << endl;
}

int main()
{
    vector<double> v;
    for (int i = 0; i < 5; i++)
        v.push_back(1.0*rand()/RAND_MAX * 2 - 1);
    printVectorForwardReverse(v);
    system("Pause");
    return 0;
}
```

Iterator out of bound errors

- Extra care is required when we use iterators as it is common to get into runtime errors whenever we use iterators
- We can run into two types of runtime errors when using iterators as described below
 - ✓ If a forward iterator is assigned an iterator value less than `begin()` or greater than `end()`
 - ✓ If a reverse iterator is assigned an iterator value less than `rbegin()` or greater than `rend()`
 - ✓ If we try to dereference a forward iterator value that is less than `begin()` or greater than or equal to `end()`
 - ✓ If we try to dereference a reverse iterator value that is less than `rbegin()` or greater than or equal to `rend()`
- Analyze the following program and determine its output

Iterator out of bound errors

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<bool> a;
    for (int i = 0; i < 5; i++)
        a.push_back(rand() % 2);
    vector<bool>::iterator it;
    it = a.end();
    cout << "The last element of the vector is " << *it << endl;
    cout << "The elements of the vector in reverse scan are" << endl;
    for (it = a.end() - 1; it >= a.begin(); it--)
        cout << *it << endl;
    system("Pause");
    return 0;
}
```

Runtime Error



Runtime Error



When at last it gets assigned a.begin()-1

Using Iterators

- We can use iterators to delete an element or several elements from a vector
 - ❖ **erase(it)**: deletes an element at the position specified by **it** argument and then **returns a new (updated) iterator value pointing to the same position where it was pointing to**
 - ❖ **erase(it1, it2)**: deletes all the elements starting from **it1** (inclusive) up to **it2** (exclusive) and then **returns a new (updated) iterator value pointing to the same position where it2 was pointing to**
- We can also use iterators to insert an element or several elements at specified iterator positions as follows
 - ❖ **insert(it, value)**: inserts the **value** argument at the position specified by **it** and then **returns a new (updated) iterator value pointing to the same position where it was pointing to**
 - ❖ **insert(it, num, value)**: inserts **num** copies of the **value** argument at the position specified by **it** and then **returns a new (updated) iterator value pointing to the first element inserted (it will return the end() iterator value if num is 0)**

Using Iterators

```
#include <iostream>
#include <vector>
using namespace std;

void printVector(vector<int> a)
{
    vector<int>::iterator it;
    for (it = a.begin(); it < a.end(); it++)
        cout << *it << " ";
    cout << endl;
}

int main()
{
    vector<int> a;
    for (int i = 0; i < 10; i++)
        a.push_back(4*i);
    cout << "The vector is" << endl;
    printVector(a);

    //delete an element
    int index = 2;
    vector<int>::iterator it = a.begin() + index;
    a.erase(it);
    cout << "After deleting the element at index " << index << ", the vector is" << endl;
    printVector(a);
}
```

0	4	8	12	16	20	24	28	32	36
---	---	---	----	----	----	----	----	----	----

0	4	12	16	20	24	28	32	36
---	---	----	----	----	----	----	----	----

Using Iterators

```
//delete several elements
int startIndex = 4, lastIndex = 7;
a.erase(a.begin() + startIndex, a.begin() + lastIndex);
cout << "After deleting the elements from " << startIndex << " up to " << lastIndex << ", the vector is" << endl;
printVector(a);

//insert an element at a specified position
a.insert(a.begin(), 21); //Insert 21 as a first element
cout << "After inserting 21 at the beginning, the vector is" << endl;
printVector(a);
a.insert(a.end(), 29); //Insert 29 as a last element
cout << "After inserting 29 at the end, the vector is" << endl;
printVector(a);
a.insert(a.begin()+3, 33); //Insert 33 at index 3
cout << "After inserting 33 at index 3, the vector is" << endl;
printVector(a);
a.insert(a.begin()+5, 3, 37); //Insert 3 elements each equal to 37 starting from index 5
cout << "After inserting three elements equal to 37 starting from index 5, the vector is" << endl;
printVector(a);

system("Pause");
return 0;
```

0	4	12	16	32	36
---	---	----	----	----	----

21	0	4	12	16	32	36
----	---	---	----	----	----	----

21	0	4	12	16	32	36	29
----	---	---	----	----	----	----	----

21	0	4	33	12	16	32	36	29
----	---	---	----	----	----	----	----	----

21	0	4	33	12	37	37	37	16	32	36	29
----	---	---	----	----	----	----	----	----	----	----	----

Using Iterators

- It should be noted that every time we **insert**, **erase**, **push_back** or **pop_back** an element or several elements from a vector, then
 - ❖ All iterator values of the vector such as **begin()**, **end()**, **rbegin()**, and **rend()** will be invalidated
- This means if an iterator variable was declared and assigned some iterator value (such as **begin()**, **end()**, **rbegin()**, **rend()**, or any other iterator value such as **begin() + 2**); then
 - ❖ **Such an iterator variable should never be used after the iterator values of the vector are invalidated because of erase, insert, push_back, or pop_back operations**
- See the following example

Using Iterators

```
int main()
{
    vector<int> a;
    for (int i = 0; i < 5; i++)
        a.push_back(4*i);
    cout << "The vector elements are ";
    for (int i = 0; i < a.size(); i++)
        cout << a[i] << "    ";
    cout << endl;

    //Now let us declare an iterator and point it to the second element
    vector<int>::iterator it = a.begin() + 1;


    //Print the second element of the vector using the iterator
    cout << "The second element in the vector is " << *it << endl;

    //Delete the second element
    cout << "Deleting (erasing) the second element of the vector." << endl;
    a.erase(it);

    //Now print the new second element using indexing
    cout << "The second element is now " << a[1] << endl;

    //Now print the new second element using iterator
    cout << "The second element is now " << *it << endl;

    system("Pause");
    return 0;
}
```



Runtime error. Why?

C++ STL algorithms

- C++ also provides STL algorithms implemented for vectors
- In order to use the algorithms, we need to include the algorithms directive
- Some of the common algorithms are:
 - ✓ **sort(iterator1, iterator2)**: sort the elements of the vector starting from position iterator1 (inclusive) up to iterator2 (exclusive)
 - ✓ **reverse(iterator1, iterator2)**: reverse the elements of the vector starting from position iterator1 (inclusive) up to iterator2 (exclusive)
 - ✓ ***min_element(iterator1, iterator2)**: returns the iterator pointing to the smallest element in the vector among the elements starting from iterator1 up to iterator 2
 - ✓ ***max_element(iterator1, iterator2)** similarly defined
 - ✓ ***find(iterator1, iterator2, search_value)**: searches the search_value in the container starting from iterator1 (inclusive) up to iterator2 (exclusive) and returns an iterator to the first element in the range that matches the search_value. If the search_value is not found, the function returns iterator2.

C++ STL algorithms

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
    vector<int> a;
    for (int i = 0; i < 10; i++)
        a.push_back(rand() % 11 - 5);
    cout << "The vector is" << endl;
    printVector(a);
    //sort the vector
    sort(a.begin(), a.begin() + 8);
    cout << "After sorting the first 8 elements, the vector is " << endl;
    printVector(a);
    //reverse the vector
    reverse(a.begin(), a.end()-4);
    cout << "After reversing the vector elements except the last four, the vector is " << endl;
    printVector(a);
    //compute and print the minimum and maximum elements of the vector
    cout << "The min element of the vector is " << *min_element(a.begin(), a.end()) << endl;
    cout << "The max element of the vector is " << *max_element(a.begin(), a.end()) << endl;
    //Search for a specified value
    int searchValue = -4;
    vector<int>::iterator it = find(a.begin()+2, a.begin()+8, searchValue);
    if (it == a.begin()+8)
        cout << searchValue << " is not found in the vector between index 2 and 8." << endl;
    else
        cout << searchValue << " is found in the vector at index " << (it - a.begin()) << endl;
    system("Pause");
    return 0;
}
```

Concluding Remarks

- Of course we can not cover all the containers and algorithms provided in the STL
- However the design concept among all the containers is very similar with some specific differences in the structure of the organization of data in each container
- Also the algorithms have similar design concepts
- For more details see
<http://www.cplusplus.com/reference/stl/>