Обектно ориентирано програмиране

STANDARD TEMPLATE LIBRARY

(STL)

STL Description

The Standard Template Libraries (STL's) are a set of C++ template classes to provide common programming data structures and functions such as doubly linked lists (list), paired arrays (map), expandable arrays (vector), large string storage and manipulation (rope), etc.

STL Description

STL can be categorized into the following groupings:

Container classes:

- Sequences:
  - **array:** Array class
  - **vector**: Dynamic array of variables, struct or objects. Insert data at the end.
  - **deque**: Array which supports insertion/removal of elements at beginning or end of array
  - **list**: Linked list of variables, struct or objects. Insert/remove anywhere.
  - **map** (unique keys), **multimap** (duplicate keys allowed): Associative key-value pair held in balanced binary tree structure.

STL Description

- Container adapters:
  - **stack** LIFO
  - **queue** FIFO
  - **priority_queue** returns element with highest priority.
- String:
  - **string**: Character strings and manipulation
  - **rope**: String storage and manipulation
- **bitset**: Contains a more intuitive method of storing and manipulating bits.

STL Description

- Operations/Utilities:
  - **iterator**: STL class to represent position in an STL container. An iterator is declared to be associated with a single container class type.
  - **algorithm**: Routines to find, count, sort, search, ... elements in container classes
  - **auto_ptr**: Class to manage memory pointers and avoid memory leaks.

std::vector

http://www.cplusplus.com/reference/vector/vector/

Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

std::vector - constructing vectors

```
// constructing vectors
#include <iostream>
#include <vector>

int main()
{
    // constructors used in the same order as described above:
    std::vector<int> first;                                // empty vector of ints
    std::vector<int> second(4, 100);                       // four ints with value 100
    std::vector<int> third(second.begin(), second.end());  // iterating through second
    std::vector<int> fourth(third);                       // a copy of third
```

std::vector - constructing vectors

```
    // the iterator constructor can also be used to construct from arrays:
    int myints[] = { 16,2,77,29 };
    std::vector<int> fifth(myints, myints + sizeof(myints) / sizeof(int));

    std::cout << "The contents of fifth are:";
    for (std::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    return 0;
}
```

std::vector - push_back

void push_back (const value_type& val);

void push_back (value_type&& val);

Add element at the end

Adds a new element at the end of the vector, after its current last element. The content of *val* is copied (or moved) to the new element.

This effectively increases the container size by one, which causes an automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity.

std::vector - push_back

```
// vector::push_back
#include <iostream>
#include <vector>
int main()
{
    std::vector<int> myvector;
    int myint;

    std::cout << "Please enter some integers (enter 0 to end):\n";
```

std::vector - push_back

```
    do {
        std::cin >> myint;
        myvector.push_back(myint);
    } while (myint);

    std::cout << "myvector stores " << int(myvector.size()) << "
numbers.\n";
    return 0;
}
```

std::vector - begin/end

```
iterator begin() noexcept;
const_iterator begin() const noexcept;
```

Return iterator to beginning

Returns an iterator pointing to the first element in the vector.

Notice that, unlike member vector::front, which returns a reference to the first element, this function returns a random access iterator pointing to it.

```
iterator end() noexcept;

const_iterator end() const noexcept;
```

Return iterator to end

Returns an iterator referring to the *past-the-end* element in the vector container.

The *past-the-end* element is the theoretical element that would follow the last element in the vector. It does not point to any element, and thus shall not be dereferenced.

std::vector - begin/end

```
// vector::begin/end
#include <iostream>
#include <vector>
int main()
{
    std::vector<int> myvector;
    for (int i = 1; i <= 5; i++) myvector.push_back(i);
    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    return 0;
}
```

std::vector - operator[]

```
// vector::operator[]
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> myvector(10);   // 10 zero-initialized elements
    std::vector<int>::size_type sz = myvector.size();
    // assign some values:
    for (unsigned i = 0; i<sz; i++) myvector[i] = i;
```

std::vector - operator[]

```
    // reverse vector using operator[]:
    for (unsigned i = 0; i<sz / 2; i++) {
        int temp;
        temp = myvector[sz - 1 - i];
        myvector[sz - 1 - i] = myvector[i];
        myvector[i] = temp;
    }
    std::cout << "myvector contains:";
    for (unsigned i = 0; i<sz; i++)
        std::cout << ' ' << myvector[i];
    std::cout << '\n';
    return 0;
}
```

std::vector - resizing vector

```
// resizing vector
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> myvector;

    // set some initial content:
    for (int i = 1; i<10; i++) myvector.push_back(i);
```

std::vector - resizing vector

```
    myvector.resize(5);

    myvector.resize(8, 100);

    myvector.resize(12);


    std::cout << "myvector contains:";

    for (int i = 0; i<myvector.size(); i++)

        std::cout << ' ' << myvector[i];

    std::cout << '\n';


    return 0;

}
```

std::vector - erasing from vector

```
// erasing from vector
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> myvector;
    // set some values (from 1 to 10)
    for (int i = 1; i <= 10; i++) myvector.push_back(i);
    // erase the 6th element
    myvector.erase(myvector.begin() + 5);
```

std::vector - erasing from vector

```
    // erase the first 3 elements:
    myvector.erase(myvector.begin(), myvector.begin() + 3);


    std::cout << "myvector contains:";
    for (unsigned i = 0; i<myvector.size(); ++i)
        std::cout << ' ' << myvector[i];
    std::cout << '\n';
    return 0;
}
```

Output:

```
myvector contains: 4 5 7 8 9 10
```