CIS-255 Home http://www.c-jump.com/bcc/c255c/c255syllabus.htm

C++ Standard Library Part I

---

1. More templates: the STL

- Avoid reinventing the wheel, *use the library!*
- STL is a conventional name of the **C++ Standard Library**.
- The abbreviation **STL** originated in 1994 and stands for *Standard Template Library*
- STL keeps code portable.
- STL gives you the functionality you need,

  while preserving the efficiency you want.
- It helps to know a bit about data structures to conquer the STL quickly.

---

2. C++ Standard Library

STL includes:

  - C Standard Library
  - String support
  - Stream I/O support for files and devices
  - Numerical computation support:
    - Complex numbers
    - Vectors with arithmetic operators
  - Support for *containers* (data structures) and *algorithms* (functions)

---

3. STL components for generic programming

- Major STL components are
  - **Containers** are objects such as *vector, list, map*
  - **Iterators** are objects behaving like pointers that define ranges inside containers
  - **Algorithms** are fundamental data manipulation tasks: *sort, count, copy, reverse,* etc.
- Containers and iterators are C++ objects
- Algorithms are C++ functions
- STL facilities work correctly with *any data type*, because they are template -based.
- An animation of STL components helps to visualize how STL components interact at runtime.

Animation: reverse, copy

---

4. std::vector

```
std::vector< typename T >
```

- Supports random access
- Provides fast appending and truncation,

but slow for internal inserting and erasing.

- Provides dynamic resizing.
- Owns and manages its own memory.

---

5. std::vector example

Using prefabricated templates from the STL library is easy:

```
#include <vector>
#include <string>

using namespace std;

int main (int argc, char* argv[])
{
    vector< double > vd; // vd elements are floating point numbers
    vector< int > vi;    // vi elements are integer numbers
    vector< string > vs; // vs elements are string objects

    return 0;
}
```

The typenames which appear inside angled brackets are *template parameters*.

---

6. std::vector construction

```
// Default constructor
explicit vector( A const& al = A() ); // (*)

// Initial size and values
explicit vector( size_type n, T const& v = T(), A const& al = A() );

// Copy constructor
vector( vector const& x );

// Initial range of values
vector( const_iterator first, const_iterator last, A const& al = A() );
```

(*) where

- **A** is the *memory allocator type*, and

- **T** is the container's *data type*.

Note that using the second constructor requires either

1. a default constructor for T
2. an explicit value for the second argument.

---

7. std::vector access

- Access is limited to the defined range of the vector.
- Otherwise the access is either
    - an exception* thrown by **at( )**, or
    - undefined with operator[ ].
- operator[ ] can be used on either side of assignment.

```
const_reference operator[]( size_type pos ) const;

reference operator[]( size_type pos );

const_reference at( size_type pos ) const;

reference at( size_type pos );
```

* if position is out of range, **at( )** signals by throwing the `out_of_range` exception .

---

8. more std::vector access

- Access to elements at both ends of the vector is made very easy
- Again, both can be used on either side of assignment:

```
reference front();

const_reference front() const;

reference back();

const_reference back() const;
```

Animation: STL Containers / Element access

9. std::vector as a stack

- Adding and removing elements at the end is also made very easy
- Note that **push_back( )** makes a copy of the argument to be put into the vector

```
void push_back( T const& x );

void pop_back();

void clear(); //remove everything
```

Animation: STL Containers / Stack operations

10. Getting information

```
size_type size() const;

bool empty() const;

size_type capacity() const;
```

You can use **resize( )** to change the size of a vector.

If it gets longer, the new elements are initialized with the second argument.

```
void resize( size_type n, T x = T() );
```

reserve( ) does the same thing for capacity,

but without any initialization. Current content is preserved.

```
void reserve( size_type n );
```

Animation: STL Containers / Vector constructors

11. STL Iterators

An iterator is an object that provides access to objects stored in STL containers.

Iterators are designed to behave *like* C++ pointers.

```
#include <iostream>
#include <vector>
using namespace std;
int main (int argc, char* argv[])
{
    vector< int > vint( 3 ); // vector with 3 integer numbers
    vint[ 0 ] = 10;
    vint[ 1 ] = 20;
    vint[ 2 ] = 30;

    // Display elements of the vector:
    vector< int >::iterator it;
    for ( it = vint.begin(); it != vint.end(); ++it ) {
        // Like pointer, iterator is dereferenced to
        // access the value of the element pointed by it:
        cout << " " << *it;
    }
    return 0;
}
// Output: 10 20 30
```

12. Iterators are like pointers

- Minimally, iterators can be:
  - incremented
  - compared
  - dereferenced
  - used with `->`
- Iterators come in const and non-const varieties, just like pointers.
- Two iterators on the same structure define a *range*.

13. Declaring iterators

---

std::vector declares two typedefs:

```
using std::vector;

vector<int> v1;

vector<int>::const_iterator iter1;

vector<int>::iterator iter2;
```

---

14. Using const iterator

---

begin( ) and **end( )** provide special iterators for a **vector**:

```
using std::vector;
vector< int > v1;
//...
int total = 0;

vector< int >::const_iterator iter = v1.begin();

while ( iter != v1.end() ) {
    total += *iter;
    ++iter;
}
```

---

15. Container functions returning iterators

- Two iterators form a *half-open range*, closed on the left but open on the right.
- **insert(** ***value*** **)**
  - uses an iterator to say where to insert a new item
  - returns iterator pointing to new element:

    ```
    iterator insert( iterator it, T const& x = T() );
    ```

- **erase( )**
  - uses an iterator to tell it where to erase a new item, or

- erases a range of items;
- returns an iterator that points to the next item *after* the deleted ones.

```
iterator erase( iterator it );
iterator erase( iterator first, iterator last );
```

Animation: STL Containers / List operations

---

16. Mistakes when using iterators

- **end( )** does not point to anything...
- So you can't dereference iterator returned by **end( )**!
- You can't use `->` on it either.
- Iterators can become *stale* and hence *unsafe* to use.

---

17. Kinds of iterators

- Iterators on vectors are *random-access*.
- Other types include *forward* and *bidirectional* iterators.
- Random-access iterators can be
    - incremented ++
    - decremented --
    - added with integers to give new iterators:

```
vector<int>::const_iterator iter = v1.begin();

iter += 7; //Now iter points to the 7th element!

// This could be expensive, maybe use a list instead?
iter = v1.insert( iter, 3 );
iter = v1.insert( iter, 8 );
```

---

18. The iterator abstraction

- Iterators are a widely used abstraction in the STL
- The idea is to have something that can be used with *arrays, strings, lists,* and even *trees*.
- Combined with liberal typedefing, it is possible to switch representations with a minimum of other changes:

```
class Container {
    typedef std::vector< int > StorageType;

    typedef StorageType::const_iterator ConstStorageIter;
    typedef StorageType::iterator StorageIter;

    StorageType m_buffer;
    //...
};
```

---

19. Advice

- Use STL - it's fast, efficient, and simple to use!
- It helps to know about data structures to understand the STL containers.
- Use iterators where possible.
- Use typedefs privately or publicly for classes that include STL components.
- STL internet resources:
  - sgi STL Programmer's Guide
  - `cplusplus.com` STL Containers
