Iterators, containers and algorithms

The C++ standard Library includes the standard Template Library (STL). that is a collection of generic algorithms, data structures (containers) and iterators.

This is call the triad of the STL.

Containers

A container is a generic class which implements a certain data structure.

* A container is a holder object that stores a collection of other objects (elements). They are implemented as class templates.

  e.g. array
  generic since int array, bool array, etc.

  We are going to learn vector, list, map, etc.

* The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators.
* All types of containers in STL are programmed in a very standard way. So these container classes have same names for member functions doing the same work.

Iterators

An iterator is a pointer that, pointing to some element in a range of elements, has the ability to iterate through the elements of that range using a set of operators with at least increment operator (++), and dereference operators (*).

- An iterator hides how to iterate & access the elements. We don't need the internal details and just use

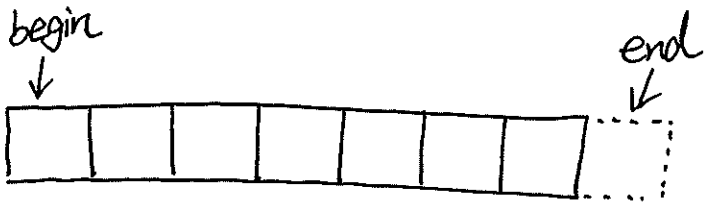
  ++ to iterate

  * to access

- In C++, we always pass two special iterators to identify a range of elements: begin & end

*** [begin, end) is a half-open range.

e.g.



A range of size 7

*** Advantage: No special case for testing empty range

Simply begin == end

So a loop termination condition could be:

begin != end

Algorithms

* In C++, an algorithm is a generic function doing a certain operation on a range of elements [begin, end).

  e.g. Copy arrays or other containers

  You can use a for/while loop, but STL provides an algorithm

  std::copy

We prefer using algorithms because it's more

- efficient
- correct
- composable : We can combine multiple algorithms in one call

* A good reference:

  cplusplus.com/reference/algorithm

* You can find definitions & examples on this website

  cplusplus.com

  e.g.

std::copy

OutputIterator copy (InputIterator first, InputIterator last, InputIterator result);

- copy the elements in the range [first, last) into the range beginning at result.
- return an iterator to the end of the destination range

e.g. int myints[] = {10, 20, 30, 40, 50, 60, 70};

std::vector<int> myvec(7);

std::copy (myints, myints+7, myvec.begin());

std::fill

void fill (iterator first, iterator last, const T& value);

- fill / assign value to all elements in the range [first, last)
- return none

Container : Vector

One disadvantage of arrays is the size of an array is fixed.
A vector is a dynamic resizing array.
Let us consider a vector in OOP (as a class):

- data members / attributes / instance variables
    1. an array holding the vector (storage)
    2. size of the current vector
    3. the max capacity of a vector
- member functions / methods
    1. begin() returns the begin iterator
    2. end() returns the end iterator
    3. You can use operator [] or at() to access any element
    4. When reach its capacity, a vector will double its capacity
    5. push_back() insert an element at the end
    6. pop_back() delete the last element

(Recall that STL naming is very standard. You will see the same names for other containers with the same purposes in the future.)

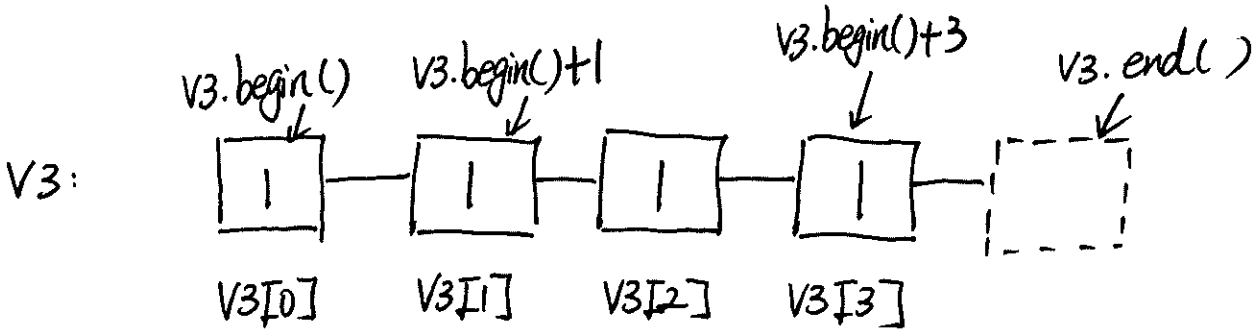
e.g. `#include <vector>`

⋮

⋮

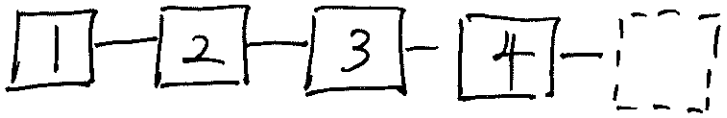
Containers are generic objects. For each object, you have to specify its value type.

```
std::vector<int> V1;   // empty vector
std::vector<int> V2(3); // A vector of three 0s.
std::vector<int> V3(4,1); // A vector of four 1s.
```

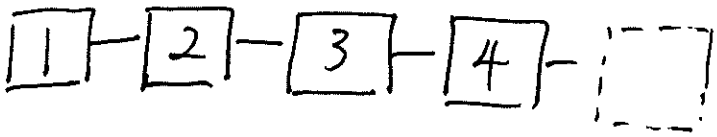


You can directly access each element using index OR using iterators and operators (++, *).

e.g.
```
std::vector<int>::iterator iter1 = V3.begin();
std::vector<int>::iterator iter2 = V3.end();
int value = 1;
while (iter1 != iter2) {
    *iter1 = value;  // dereference operator, rewrite elements
    ++value;
    ++iter1;   // iter1 points to the next element
}
```

V3: 


```
int sum = 0;
iter1 = v3.begin();
while (iter1 != iter2) {
    sum += *iter1;
    ++iter1;
}
```



-3 -2 -1 end

+1 +2

What is sum ?

```
iter1 -= 3;     // what is *iter1 ?
iter1 +=2;      // what is *iter1 ?
```

examples of using std::max_element
std::min_element
} #include <algorithm>
std::accumulate — <numeric>

check cplusplus.com

max_element & min_element

* return an iterator

accumulate returns a T value

What is T?

Every container also has a const_iterator type.

A const_iterator is read-only.

e.g. `std::vector<int>::const_iterator iter3 = v3.begin();`

`std::cout << *iter3 << std::endl;`

`// *iter3 = 0;` is illegal

★★★ const_iterator and const iterator are different.

We must define const_iterator in the container class.

Why?

`const iterator iter4 = v3.begin();`

The iterator is const. So,

`++iter4;` is illegal

But `*iter4 = 0` is fine.

More vector member functions

1. destructor

```
~vect() {
    // delete the member variable m_array
}
```

2. Assignment operator=

Assign a vector reference "a" to this object

```
vect& operator= (const vect& a) {
    if (&a != this) {
        // reserve capacity
        // copy a to this
        // reset m_size
    }
}
```

3. int& operator[] (unsigned index) OR const int& operator[] ..

return ? (one line)

More vector member functions

— size related

1. size()

read-only function

```
unsigned size() const {
    return m_size;
}
```

2. bool empty() const;

3. unsigned capacity() const; // leave to readers

4. Reserve capacity for "space" elements

```
void reserve(unsigned space) {
    if (space > capacity()) {
        // create a new array of size "space"
        // std::copy this to the new array
        // delete m_array
        // reset m_array to the new one
        // reset m_capacity
    }
}
```

5. Resize m_size to be "newSize"

if newSize < m_size , erase

if newSize > m_size , insert "value" at the end

```
void resize(unsigned newSize , const int& value ) {
    if (m_size< newSize) {
        // reserve capacity
        // std::fill_n  fill from end # of value
                                    ? newSize - m_size
    }
    // reset m_size
}
```

More vector member functions

– iterator related

1. iterator begin() OR const_iterator begin()

   return ? (hint: single line, name of an array ⟷ pointer)

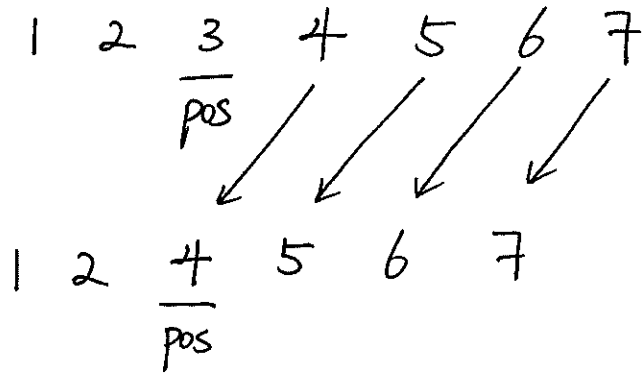
2. iterator end() OR const_iterator end()

   return ? + ? (hint: single line)

3. erase the element at iterator "pos", and return an iterator pointing to the next element

```
iterator erase (iterator pos) {
    // std::copy from pos+1 to end() to pos
```

e.g.



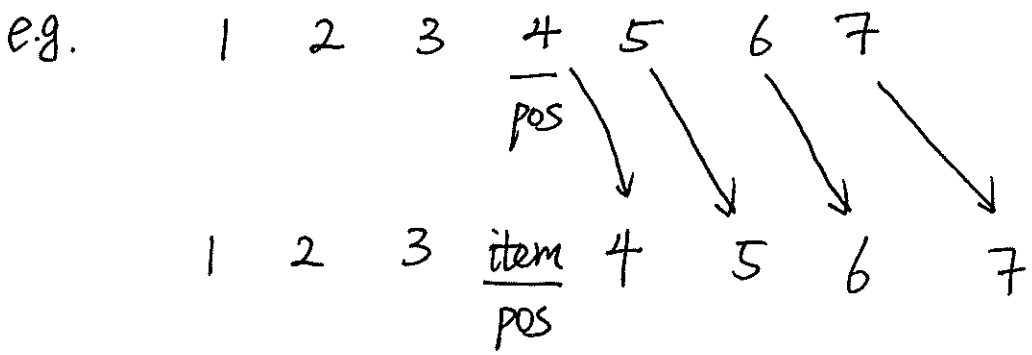

```
    // m_size should be reduced
    // return the iterator pointing to the next element
    (pos)
```

in the above example : 4

4. insert "item" before iterator "pos" and return an iterator pointing to "item"

```
iterator insert (iterator pos, const int& item) {
    if (m_size == m_capacity) {
        // reserve new space = : 1 if empty()
                               : double size() otherwise
    }
                                  pos   end()  end()+1
    // std::copy_backward (?  ,  ?  ,  ?  );
```

e.g.



Why can't we use std::copy ? try it, what is the problem

```
    // insertion:  ? = item;   (*pos)
    // reset m_size
    // return the iterator pointing to item
}
```

other member functions

check cplusplus.com/reference/vector/vector

e.g. vector::erase

- signatures: iterator erase (iterator position);

  iterator erase (iterator first, iterator last);

- remove from the vector either a single element or a range of elements.

- return an iterator pointing to the new location of the element that followed the last element erased by the function call.

How to implement erase() by ourselves?

can we use copy?

(Algorithm)

e.g. 1 2 3 4 5 6 7 8

↑

1 2 3 5 6 7 8

Iterator Invalidation

Iterators pointing into an std::vector can become invalid, but only when performing certain operations. Using invalid iterators will result in undefined behavior.

1. Any insertion operation which changes the capacity of the vector, will invalidate all iterators.

e.g.

```
std::vector<int> V(5);

V.reserve(20); // capacity is at least 20

int* p1 = &V[0];

std::vector<int>::iterator p2 = V.begin();

V.push_back(4); // p1 & p2 are not invalidated, size=6

V.insert(V.end(), 30, 9); // insert 30 elements at the end
                          // p1 & p2 are probably invalidated.

V.reserve(V.capacity()+20); // p1 & p2 are definitely
                            invalidated.
```

2. Any insertion operation, which does not change the capacity, will still invalidate iterators/pointers pointing to elements at the insertion position and past it. This include end().

e.g.
```
std::vector<int> V(5);
V.reserve(20);   // capacity is at least 20;
int* P1 = &V[0];
int* P2 = &V[3];
V.insert(V.begin()+2, 5, 0); // P1 is valid but P2 Not.
```

3. Any removal operation will invalidate iterators/pointers pointing to the removed elements and to any elements past it.

e.g.
```
std::vector<int> V(10);
int* P1 = &V[0];
int* P2 = &V[5];
V.erase(V.begin()+3, V.end());

// P1 is valid
// P2 is invalid
```