Comp151

Generic Programming: Iterators

Standard Sequence Containers

Here are some homogeneous container classes commonly used to represent a sequence of objects of the same type.

vector : one-dimensional array
  Fast access at any position.
  Add and remove elements only at the back.

list : doubly-linked list
  Fast access to front and back only.
  Add and remove elements at any position.

deque : double-ended array
  Fast access at any position.
  Add and remove elements only at the front and back.

Standard Sequence Containers

They differ in how quickly different access opertaions can be performed. n is the number of elements currently in the container. $O(1)$ means *constant time*.

Container	Access Control	Add/Remove
vector (1D array)	O(1) random access	O(1) at the end O(n) in front/middle
list (doubly-linked list)	O(n) in middle O(1) at front/end	O(1) at any position
deque (doubly-ended queue)	O(1) random access	O(1) at front/back O(n) in middle

```
#include <vector>
#include <deque>
#include <list>
#include <string>
#include <iostream>

main()
{ vector<double> VD;
  deque<string> DS;
  list<int> LI;

  VD.push_back(5.0); VD.push_back(10.0); VD.push_back(15.0);
  cout << VD[1] << endl;

  DS.push_back(" World "); DS.push_front(" Hello ");
  cout << DS[0] << DS[1] << endl;

  LI.push_back(1); LI.push_back(2);
  //cout << LI[0];    ERROR
    }
```

Sequence Container: Access, Add, Remove

Element access for all:

- `front()`: First element
- `back()`: Last element

Element access for `vector` and `deque`:

- `[ ]`: Subscript operator, index not checked.

Add/remove elements for all:

- `push_back()`: Append element.
- `pop_back()`: Remove last element.

Add/remove elements for `list` and `deque`:

- `push_front()`: Insert element at the front.
- `pop_front()`: Remove first element.

Sequence Container: Other Operations

"List" operations are fast for `list`, but also available for `vector` and `deque`: (what is p?)

- `insert(p, x)`: Insert an element at a given position.
- `erase(p)`: Remove an element.
- `clear()`: Erase all elements.

Miscellaneous Operations:

- `size()`: Returns the number of elements.
- `empty()`: Returns true if the sequence is empty.
- `resize(int i)`: Change size of the sequence.

Comparison operators ==, !=, < etc. are also defined.

Example: Print with an Array


```
const int LEN = 10;
int x[LEN];
int* const x_end = &x[LEN];

for (int* p = x; p != x_end; ++p)
    cout << *p;
```

We use an `int` pointer to access the elements of an `int` sequence with some basic operations:

Operation	Goal
`p = x`	initialize to the beginning of an array
`*p`	access the element being pointed to
`++p`	point to the next element
`p != x_end`	compare with the end of an array

Example: Printing a List Sequentially

To access `list<int>` elements *sequentially*, one may define an iterator `list<int>::iterator`, and add functions `begin()` and `end()` to set to the beginning and end of the container.

```
#include <list>                    // "list" class of STL

list<int> x;
list<int>::iterator p;

for (p = x.begin(); p != x.end(); ++p)
    cout << *p;
```

Example: Printing a Vector Sequentially

One can similarly define an iterator `vector<double>::iterator` to sequentially go through items in a `vector<double>`.

```
#include <vector>                    // "vector" class of STL

vector<double> x;
vector<double>::iterator p;

for (p = x.begin(); p != x.end(); ++p)
    cout << *p;
```

Iterators

- For each kind of container in the STL there is an *iterator type*.

```
list<int>::iterator lp;
vector<string>::iterator vp;
deque<double>::iterator dp;
```

- Iterators are used much like pointers: They can be used to indicate elements in the sequence, or to indicate a subsequence.
- Operations on iterators are:
  - Access element: `*p`, `p->`
  - Go to next or previous element: `++p`, `--p`
  - Compare iterators: `==`, `!=`

```
#include <list>
#include <iostream>

void Display(list<int>::iterator start, list<int>::iterator end)
{ list<int>::iterator p;
  for( p = start; p != end; p++)
      cout << *p << " ";
}

main()
{list<int> L, Small, Large;
 list<int>::iterator lp;

 for(int i = 1; i < 13; i++)
   L.push_back(i*i % 13);

 for(lp = L.begin(); lp != L.end(); lp++)
   if (*lp < 7)
      Small.push_back(*lp);
      else Large.push_back(*lp);

cout << "L:  "; Display(L.begin(), L.end()); cout << endl;
cout << "Small:   "; Display(Small.begin(), Small.end()); cout << endl;
cout << "Large:   "; Display(Large.begin(), Large.end()); cout << endl;}
```

Example: find() with an int Iterator

- Iterators provide a systematic way of looking at elements of sequence container without differentiating between different container classes.
- The same code works correctly for all sequence container classes.

```
// File: "find_int_iterator.cpp"
typedef int* Int_Iterator;

Int_Iterator find( Int_Iterator begin, Int_Iterator end, const int& value )
{
    while (begin != end && *begin != value)
        ++begin;

    return begin;
}
```

Example: find() with an int Iterator ...

```
#include "find_int_iterator.cpp"

int main() {
    const int SIZE = 100; int x[SIZE]; int num;
    Int_Iterator begin = x; Int_Iterator end = &x[SIZE];

    for (int i = 0; i < SIZE; i++)
        x[i] = 2 * i;

    while (true) {
        cout << "Enter number:  "; cin >> num;
        Int_Iterator position = find(begin, end, num);

        if (position != end) {
            ++position;
            if (position != end) cout << "Found before " << *position << '\n';
            else cout << "Found as last element\n";
        }
        else cout << "Not found\n";
    }
}
```

Why Are Iterators So Great?

Because they allow us to separate algorithms from containers.

- If we change the `find()` function as follows, it still works:

```
template <class Iterator, class T>
Iterator find( Iterator begin, Iterator end, const T & value )
{
    while (begin != end && *begin != value)
        ++begin;
    return begin;
}
```

- The new `find()` function contains no information about the implementation of the container, or how to move the iterator from one element to the next.
- The same `find()` function can be used for any container that provides a suitable iterator.

Example: find() with an Iterator

```
#include <vector>                                    // "vector" class from STL
int main()
{
    vector<int> x(SIZE); int num;
    for (int i = 0; i < SIZE; i++)
        x[i] = 2 * i;

    while (true)
    {
        cout << "Enter number:  "; cin >> num;
        vector<int>::iterator position = find(x.begin(), x.end(), num);

        if (position != x.end())
        {
            ++position;
            if (position != x.end()) cout << "Found before " << *position << '\n';
            else cout << "Found as last element.\n";
        }
        else cout << "Not found\n";
    }
}
```

```
#include <list>
#include <vector>
#include <string>
#include <iostream>

template<class Iterator>
void Display(Iterator start, Iterator end)
{ for( Iterator p = start; p != end; p++)
        cout << *p << "  ";
}

main()
{list<int> L;
 vector<string> S;

 for(int i = 1; i < 13; i++)
      L.push_back(i*i % 13);
 S.push_back("Now"); S.push_back("Is"); S.push_back("The");
 S.push_back("Time"); S.push_back("For"); S.push_back("All");

 cout << "L:  "; Display(L.begin(), L.end()); cout << endl;
 cout << "S:  "; Display(S.begin(), S.end()); cout << endl;
}
```