U O W

CSCI251 Advanced Programming

Generic Programming IV: Containers and iterators

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Outline

- What is a container
- Why use containers
- Examples
- Iterators

What is a container?

- Containers are used to store a collection of objects.
  - the built-in container： Arrays , like int arr[3];

- We develop containers to manage collections of more complicated objects, tailored to our needs.

- We can use class templates to define our own container classes.

Advantage of container

- Think about the String vs an array of chars

- Some advantages over arrays:
  - Subscript bound checking.
  - Memory gets tidied up automatically.
  - Inserting elements anywhere may be made easy, depending on the context.
  - Pass by reference or value, arrays passed by reference only.

Container (adaptor) example

- Vector, queue, stack, and linked list
  - **Queues** are data structures in which elements are removed in the same order they were entered (FIFO).
  - **Stacks** are data structures in which elements are removed in the reverse order from which they were entered (LIFO).
  - **Linked lists** provide a method of organizing stored data based on a logical order of the data.
    - contain data and references to other nodes.

Container method

- We need to be able to:
  - Insert a new element.
  - Remove an element.
  - Reorder the list.
  - Retrieve and display the objects.

Example

- A collection of objects
  - Int
  - Double
  - Customized data type (say, book)
- Function:
  - Display all objects in it

A Book class

```
class Book {
    friend ostream& operator<<(ostream&, const Book &);
    private:
        string title;
        double price;
    public:
        void setBook(string, double);
};

void Book :: setBook(string t, double p) {
    title = t;
    price = p;
}

ostream& operator<<(ostream& out, const Book &b) {
  out << b.title << " " << b.price;
  return out;
}
```

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Container example

```
template<typename T>
class Array {
    private:
        T *data;
        int size;
    public:
        Array(T *, int);
        ~Array();
        void display();
};

template<typename T>
Array<T> :: Array(T *d, int s) {
    size = s;
    data = new T [size];
    for( int i=0; i<size; i++ )
         data[i] = d[i];
}
```

```
template<typename T>
Array<T> :: ~Array() {
    if( data != NULL )
         delete [] data;
}

template<typename T>
void Array<T> :: display() {
   for(int i=0; i<size; i++)
      cout << " " << data[i] <<
   endl;
   cout << endl;
}
```

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Container example

```
int main() {

    int intData[] = {1,2,3};
    double doubleData[] = {11.11, 22.22, 33.33};

    Book books[2];
    books[0].setBook(“ba", 8.90);
    books[1].setBook(“bb", 8.69);


    Array<int> intArray( intData, 4 );
    intArray.display();

    Array<double> dArray( doubleData, 3 );
    dArray.display();

    Array<Book> books( bookRecs, 2 );
    books.display();

    return 0;
}
```

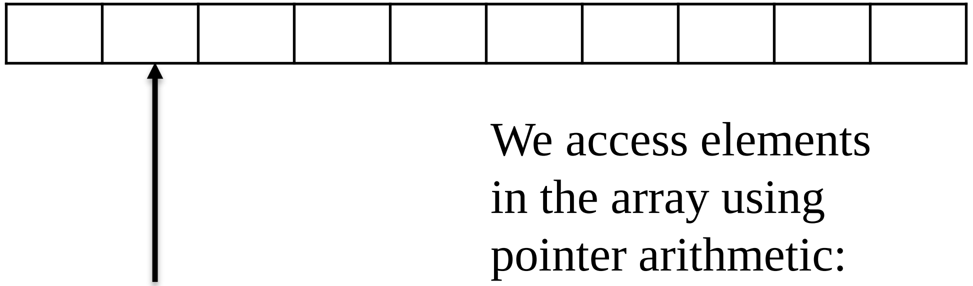
UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Container method

- We might want to add functions to the `Array` class that:
  - Displays the last element in an array.
  - Displays an element in a specified position in an array.
  - Reverses the order of elements in an array.
  - Sort the elements in an array in ascending or descending order.
  - Remove duplicate elements from an array.
  - …

Container method

- Its all about accessing elements in a container
- For an array we have something like …



We access elements in the array using pointer arithmetic:

```
*ptr
*ptr++
*ptr--
*(ptr+i)
```

Problem with pointers

- This won’t work for containers, such as linked lists, where the next element isn’t contiguous in memory.
  - And generally accessing the elements of a container is going to require tailoring to the container

- A smart mechanism (or smart pointer)
  - iterators

Iterators

- An iterator is an object that moves through a container and selects one at a time

- Iterators provide a **standard way** to access elements
  - whether or not the container provides a way to access the elements directly
  - have the same interface

- sometimes referred to as smart pointers
  - help to detect when you are past the end of your container, like the end of an array

Iterator operations

- With pointers we use the address-of operator (&) to point at something in particular.
  - For iterators we will manage it like members
  - So for an object `v` of a type with iterators …

```
auto b=v.begin();
auto e=v.end();
```

- … `begin` is an iterator denoting the first element, and `end` denoting the position of the one (after the last element).

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Iterator operations

Operation	Meaning
`*iter`	Returns a reference to the element denoted by `iter`.

`iter->mem`
`(*item).mem`
	Deferences `iter` and fetches the member named mem from the underlying element.

`++iter` `--iter`	Increments `iter` to refer to the next element. Decrements `iter` to refer to the previous element.

`iter1 == iter2`
`iter1 != iter2`
	Compares two iterators for equality or inequality. Equal if they denote the same element or are both the off-the-end iterator for the same container.

Iterator operations

- 4 types:
  - Container::iterator iter;
  - Container::const_iterator iter;
  - Container::reverse_iterator iter;
  - Container::const_reverse_iterator iter;
- Usage:
  - Iterator and reverse_iterator
    - ++ or --
  - Const and non_const
    - (for the *iter), only for reading, and not for writing

Iterator for vector

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v;
    for (int n = 0; n<5; ++n)
        v.push_back(n);  //push_back to add new elements to the end of vector
    vector<int>::iterator i;
    for (i = v.begin(); i != v.end(); ++i) {
        cout << *i << " ";
        *i *= 2;
    }
    cout << endl;

    for (vector<int>::reverse_iterator j = v.rbegin(); j != v.rend(); ++j)
        cout << *j << " ";
    return 0;
}
```

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Iterator for vector

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v;
    for (int n = 0; n<5; ++n)
        v.push_back(n);  //push_back to add new elements to the end of vector
    const vector<int>::iterator newiter=v.begin();
    *newiter=3; cout<<*newiter;
    vector<int>::const_iterator citer = v.begin();
    *citer=3; cout<<*citer;
    vector<int>::iterator newiter2=v.end();
    cout<<"\n"<<newiter2-newiter;
    newiter2 -= 3;
    cout<<"\n"<<*newiter2; cout<<"\n"<<*(newiter2+1);
    return 0;
}
```

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Iterator for string

- The `string` class is not actually a container class but it has a lot of functionality in common with containers

```
string s("this is a string");
if (s.begin() != s.end()){
    auto it = s.begin();
    *it = toupper(*it);
}
```

Iterator for string

- To step through elements in our string

```
for (auto it=s.begin(); it != s.end(); ++it)
    cout << *it << " ";
cout << endll;
```

Iterator operations

- **We can use `cbegin()` and `cend()` in place of `begin()` and `end()` to get `const` iterators.**
- advance(p, n)： move p with n elements
- distance(p, q)： so that p == q
- iter_swap(p, q)： exchange p and q’s value

Iterator operations

```
#include <list>
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
    int a[5] = { 1, 2, 3, 4, 5 };
    list <int> lst(a, a+5);
    list <int>::iterator p = lst.begin();
    advance(p, 2);
    cout << *p << endl;
    advance(p, -1);
    cout << *p << endl;
     ...
```

```
 ...
  list<int>::iterator q = lst.end();
    q--;
   cout << distance(p, q) << endl;
    iter_swap(p, q);

    for (p = lst.begin(); p != lst.end(); ++p)
        cout << *p << " ";
    return 0;

}
```

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

More on iterators later …

- We will look at iterators a bit more later.
  - There are different kinds of iterators and various other operations

- For now we are going to turn to the STL to took at how the containers and iterators work together.
  - We can take `vector` as a prototype for how the rest of the standard template library works and as indicative of how templating generally works.