UEE1303(1070) S'12 Object-Oriented Programming in C++

Lecture 11: *Data Structure & Standard Template Library*

*modified from Comp-233 @ Brookdale C.C. and PGDST/PCCP

Outline

- Understand basic data structure
  - Linked List vs. Array
  - Stacks & Queues
  - Trees
- Standard Template Library (STL)
  - Iterators
  - Containers
  - Generic Algorithms

Data Structure

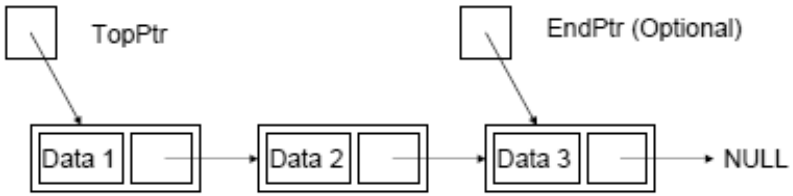
- A means of representing data to achieve efficiency in terms of
  - time complexity: to minimize runtime
  - memory complexity: to minimize memory usage
- A means of demonstrating relationships between data elements
- A means of enforcing a processing order
- A means of modeling real-world problems

Linked Lists vs. Arrays

- Placement and access of storage
  - arrays: use contiguous storage + random access
  - linked lists: use non-contiguous storage + sequential access
- Volume of storage
  - Arrays: fixed size
  - Linked lists: expand and contract as needed

Fundamental Linked List

- Made up of a series of dynamically allocated nodes, each containing a link to the next node
- Managed by a single pointer to the first node, and possibly a second pointer to the last
  - `TopPtr`
  - `EndPtr` (optional)



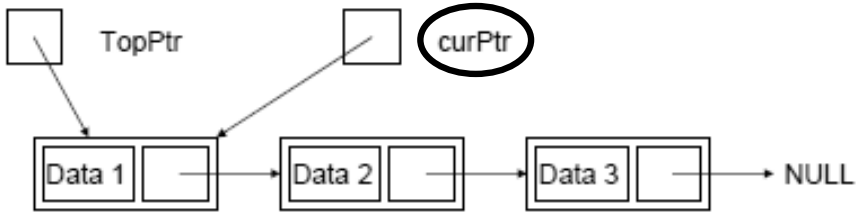
Linked List Usage

- Sequential Access
  - each access must start at first node and traverse all others until the desired node is found
  - an illusion of random access can be created by

```
dataType getElement(int index);
```

- Any process that traverses the list or affects all elements in the list must use a secondary pointer to refer to "the current element“
  - `curPtr` (current pointer)

Linked List Traversal



- `curPtr` points to the element currently being operated on
- Can traverse the list by following each node's `next` pointer
  - `curPtr = curPtr->next;`

Stacks & Queues

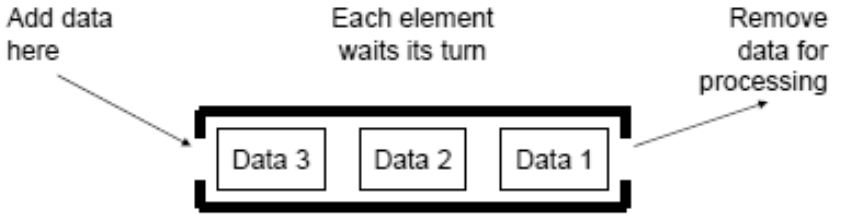
- Establish and enforce an order for processing

- Stack
  - *Last in, first out* (LIFO)
- Queue
  - *First in, first out* (FIFO)
  - Priority Queue
- Deque (pronouced ‘deck’ or ‘de-Q’)
  - *Double-ended queue*

Stacks

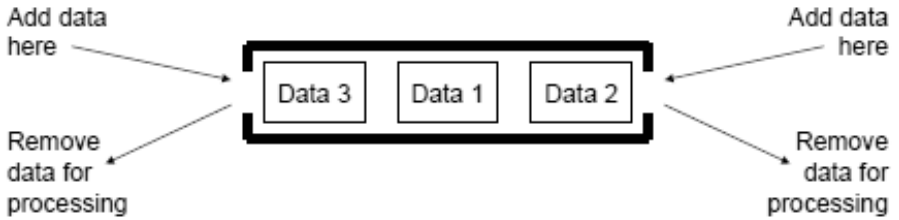
- Add element
  - Push
- Remove element
  - Pop
- Only top element can be seen
  - Can you now understand why it is called first-in last-out (FILO) ?

Queues



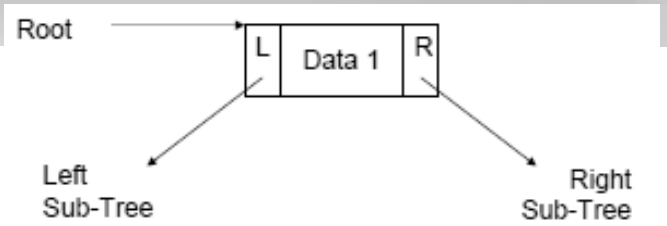
- FIFO: first-in first-out
- Can use array or linked list internally
  - internal links are *unreachable*

Deque



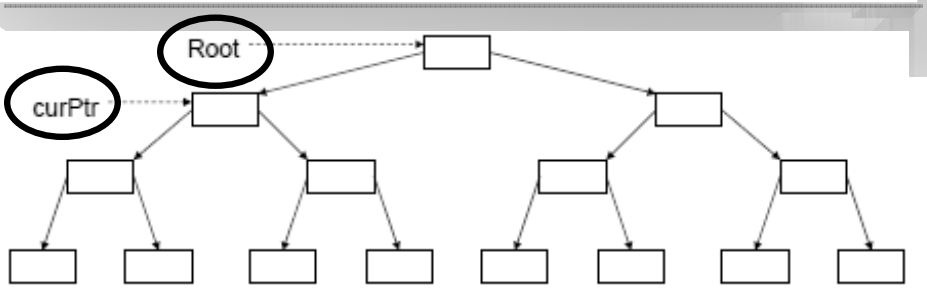
- Deque is a special type of queue
  - data can be added/deleted from either end
  - internal nodes are still inaccessible

Trees



- A "`root`" node with some number (usually 2) of links (pointers) to "`child`" nodes
- Each child node is the root of a "sub-tree"
  - inherently recursive in nature
- Many flavors of trees to choose from

Tree Traversal



- Binary trees (2 children) are the most common type
  - access by `Root`
- Use a reference pointer to access sub-nodes during traversal
  - `curPtr`

What is STL?

- **STL** = **Standard Template Library**
  - based (heavily) on template programming
  - with a guarantee of performance
- A general-purpose library of *generic* algorithms and data structures in C++
  - container classes
  - generic algorithms
  - other components (ex: iterators)
- Part of the ISO Standard C++ Library

Why Should I Use STL?

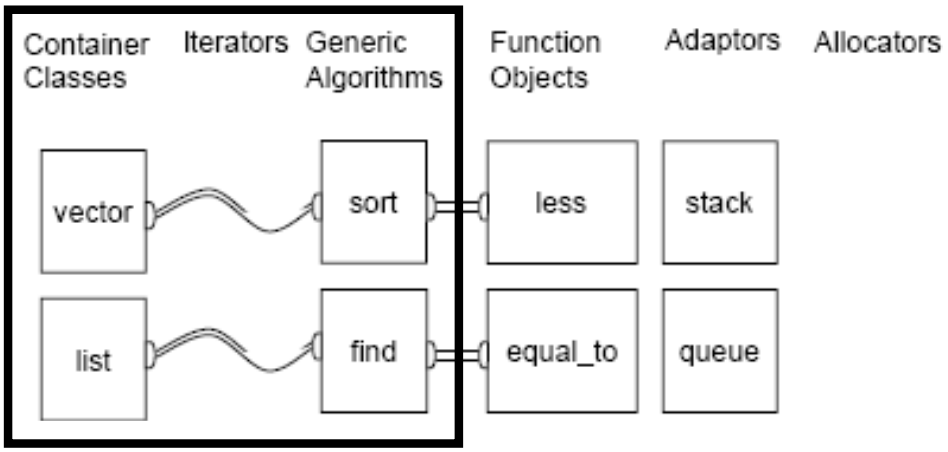
- Reduce development time
  - data-structures already well-written and thoroughly debugged
- Code readability
  - fit more meaningful stuff on one page
- Robustness
  - STL data structures *grow automatically*
- Portable code
- Maintainable code
- Easy

STL Components (1/2)

- STL has 7 kinds of components
- First-order components
  - containers
  - iterators
  - algorithms
- Second-order components
  - functional objects (*functors*)
  - adaptors
  - allocators
  - traits

STL Components (2/2)

- 7 components can be visualized as



STL Basics

- Iterators (a.k.a. *smart pointers*)
  - referencing devices similar to the pointers to the individual elements used in data structures
- Container Classes
  - class *templates* which implement many of the classic data structures
- Generic Algorithms
  - *template functions* that implement frequently used algorithms (i.e., sorting, searching, etc.)

Container `vector`

- Like arrays, *vectors* in concept are used to store a *homogeneous* sequence of data objects
- But unlike arrays, vectors have no a predefined value for *size limit*
- Therefore, **`vector`** is made into a class template
  - part of Standard Template Library (STL)

Memory Allocation for `vector`

- By default, vector objects are created with a size of 0
  - a parameter to the constructor can override this behavior
- The size of a vector object can be changed at run-time
  - `push_back(T)`: member function adds storage at the end of the sequence for one new element
  - `resize(int)`: member function sets size, adding or deleting elements as necessary

Constructors of vector

- Default Constructor
  - `vector <T> v1;`
- Allocation Constructor
  - `vector <T> v2(int);`
- Copy Constructor
  - `vector <T> v3(vector <T>);`

vector Methods (1/3)

- `size_type vector::`**`capacity`**`();`
  - return number of elements for which memory has been allocated
- `size_type vector::`**`size`**`();`
  - return number of elements physically existing in the vector
- `void vector::`**`resize`**
  `(size_type n, T x = T());`
  - reallocate memory, preserves contents if new size is larger than existing size

vector Methods (2/3)

- `void vector::`**`push_back`**`(const T& x);`
  - append (insert) an element to the end of a vector, allocating memory for it if necessary
- `void vector::`**`pop_back`**`();`
  - erase the last element of the vector
- `const T&` **`operator[](`**`size_type pos) const;`
- `T&` **`operator[]`**`(size_type pos);`
  - constant and non-constant [ ] operator

vector Methods (3/3)

- `vector::`**`erase`**`()` or `vector::`**`clear`**`()`
  - erase all elements in the vector
- `void vector::`**`erase`**`(iterator);`
  - erase the element indexed by the iterator
- `void vector::`**`erase`**
  `(beg_iterator, end_iterator);`
  - erase the elements between the begin iterator (`beg_iterator`) and the end iterator (`end_iterator`)
- `bool vector::`**`empty`**`();`
  - return true if vector has no elements

Iterators (1/2)

- Container class provides its own iterator class
  - implemented as a sub-class
  - usage and syntax is very similar to that of pointers
- The container class provides methods for creating, initializing, and controlling iterators
  - `begin()` : return an iterator that points to the first element
  - `end()` : return an iterator that point just "behind" the last element
- Does **NOT** reference the last element

Iterators (2/2)

- Prefix and postfix increment and decrement
  - `++` moves the iterator to the next element
  - `--` moves the iterator to the previous element
- Equals and not-equals
  - `==` & `!=` compares two iterators
- Dereference
  - `*` & `[]` returns the referenced data item
- Not all iterators provide all of these functions

Example of `vector`

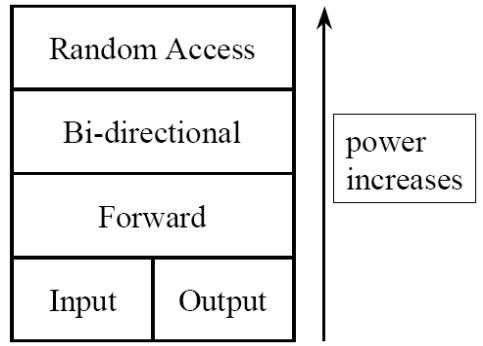
```
#include <iostream>
#include <vector>
using namespace std;
void main ()
{
    vector <int> v1 (10);
    vector <int>::iterator i;
    int n = 0;
    for (i=v1.begin(); i!=v1.end(); i++)
        *i = n++;
    for (i=v1.begin(); i!=v1.end(); i++)
        cout << *i << " ";
    cout << endl;
}
```

```
0 1 2 3 4 5 6 7 8 9
```

lec11-1.cpp

Basic Iterators

- Points to an object in a container
- Access to object by de-referencing
- Increment and decrement operators used to move forward and backward
- Category by move
  - input
  - output
  - forward
  - bi-directional
  - random access



Input & Output Iterators

- Input iterator
  - used to read value from a *sequence* container
  - support de-reference **`*iter`**, increment **`++iter`**/**`iter++`** and in-/equality **`!=`**/**`==`**
- Output iterator
  - used to write values *once* to a sequence container
  - no **`++it`** more than once between two **`*it`** or assign **`*it`** more than once without **`++it`**
  - include ostream and inserter, e.g. **`back_inserter()`**

Example of Stream Iterators

Can you guess what is happen!?

```
...
//in main()
  vector<string> vec;
  copy( istream_iterator<string>(cin),
        istream_iterator<string>(),
        back_inserter(vec)
  );
  sort( vec.begin(), vec.end() );
  unique_copy(
     vec.begin(),
     vec.end(),
     ostream_iterator<string> (cout,"\n")
  );
...
```

lec11-2.cpp

Forward Iterator

- Both input and output iterator
- Read and write in one direction
- Read and write multiple times
- All standard library containers use this iterators

```
vector<int> iv;
iv.push_back(4);
iv.push_back(53);
…
for (vector<int>::iterator it=iv.begin();
  it!=it.end(); it++)
  cout << *it << endl;
```


Bi-directional & Random Iterators

- **Bi-directional iterator**
  - quite similar to forward iterator
  - can also be decremented `--iter`/`iter--`
  - read and write forward and backward
- Random access iterator
  - allow access to any element
  - compare iterators using < and >
  - does **NOT** work with `list`
  - e.g. `vector` and `string` iterators

Examples of Iterator Usages

- Declare

```
list<int>::iterator  li;
```

- Front of container

```
list<int> L;
li = L.begin();
```

- Past the end

```
li = L.end();
```

- Increment

```
list<int>::iterator li;
list<int> L;
li=L.begin();
++li;  // second item;
```

- De-reference

```
*li = 10;
```


(Advanced) Reverse Iterators

- Behave similarly to *normal* iterators, but in reverse order
- Member functions of the container class:
  - **`rbegin():`** return an iterator that points to the last element
  - **`rend():`** return an iterator that point just "before" the first element
- Does **NOT** reference the first element

(Advanced) Constant Iterators

- **`const_iterator`**
  - used just like an iterator
- **`const_reverse_iterator`**
  - used just like a reverse_iterator

- A `const` vector (object) will always return a `const` iterator (read-only)
  - no assignments can be made through these iterators
  - `*p = <something>;` ⇨ illegal

(Advanced) Mutable Iterators

- Can assign a value through dereferencing such iterator
  - `*p` can be assigned a value
  - i.e. `*p = <something>;`
  - change the corresponding element in the container

Example of Advanced Iterators

```
#include <vector>
using namespace std;
void main ()
{
    vector <int> v2(10); //size of 10
    vector <int>::const_iterator c=v.begin();
    vector <int>::reverse_iterator r;
    int n = 0;
    for (i=v2.rbegin(); i!=v2.rend(); i++)
        *i = n++;
    for (n=0; n<10; n++)
        cout << c[n] << " ";
    cout << endl;
}
```

```
9 8 7 6 5 4 3 2 1 0
```

lec11-3.cpp

Basic Containers in STL

- **Sequential** Containers
  - elements are stored in whatever order they were added, unless sorted manually
  - ex: `list`, `vector`, and `deque`
  - some implementations also provide `slist` (not part of STL standard)
- **Associative** Containers
  - elements are sorted automatically according to some key field
  - ex: `set` and `map`
- Top: `string`, `vector` , `list` and `map`

Adapter Containers in STL

- Adapter category:
  - sequential adapters,
    ex: **`stack`** and **`queue`**
  - associative adapters,
    ex: **`multimap`** **and** **`multiset`**
- Used as wrapper for other (basic) containers
- limit or control access to elements of the inner container

`vector` Container Class

- Provides random-access iterators
  - forward and reverse
  - mutable and constant
- Member functions support:
  - iterator manipulation
  - direct manipulation

list Container Class

- Doubly-linked list
- Provides bi-directional iterators
  - forward and reverse
  - mutable and constant
- Member functions support:
  - iterator manipulation
  - direct manipulation

list Member Functions

- **Iterator** manipulation
  - `begin()`, `end()`, `rbegin()` and `rend()`
  - `insert()` and `erase()`
- **Direct** manipulation
  - `push_back()` and `push_front()`
  - `pop_back()` and `pop_front()`
  - `front()`, `back()` and `clear()`

Example of list (1/2)

```
#include <iostream>
#include <list>
using namespace std;

int FuncSquared(int a) {
    return a*a;
}

void main ()
{
    list<int> x;
    list<int>::const_iterator c=x.begin();
    list<int>::const_iterator r;
    ...;
```

lec11-4.cpp

Example of list (2/2)

```
//in main()
    for (int n=0; n<10; n+=3) {
        x.push_front(n); //direct mani.
        x.push_back(n+1);
    }
    for (r=x.rbegin(); r!=x.rend(); r++)
        *r = FuncSquared(*r); //itr mani.

    for (c=x.begin(); c!=x.end(); c++)
        cout << *c << " "; //itr. mani.
    cout << endl;

    return 0;
}
```

```
81 36 9 0 1 16 49 100
```

lec11-4.cpp

Associative Containers

- Each data item has a key
  - include `set` and `map`
  - support bi-directional iterator
  - key-based fast retrieval of objects from collection ⇨ simple database
- Store data objects based on an ordering function
  - easy to look up a object based on a given key
  - objects stored using a tree organization

Associative Class `set` (1/2)

- Simplest container possible
- Stores elements without repetition
- 1st insertion places element in the set
  - additional (later) insertions have no effect ⇨ no element appears more than once
- Capabilities:
  - add elements
  - delete elements
  - ask if element is in the set

Associative Class `set` (2/2)

- Designed to be efficient
  - store values in a sorted order
- Can specify order: `set<T, ordering> s;`
  - ***ordering*** is well-behaved ordering relation that returns bool
  - If none specified, use **<** relational operator
- Note that its insert function is different from the insert function for sequence containers, such as `list`, `vector`, or `deque`

Example for `set` (1/2)

```
#include <iostream>
#include <set>
using std::cout;
using std::endl;
using std::set;

int main(){
    set<char> s;
    s.insert('A');
    s.insert('D');
    s.insert('D'); //2nd insertion of ‘D’
    s.insert('C');
    s.insert('C'); //2nd insertion of ‘C’
    s.insert('B');
```

lec11-5.cpp

Example for set (2/2)

lec11-5.cpp

```
// in main()
    cout << "The set contains: ";
    set<char>::const_iterator p;
    for (p=s.begin( ); p!=s.end( ); p++)
        cout << *p << " "; //itr. mani.
    cout << endl << "Removing C" << endl;
    s.erase('C'); //direct mani.
    for (p=s.begin( ); p!=s.end( ); p++)
        cout << *p << " ";
    cout << endl;
```

```
The set contains: A B C D
Removing C
A B D
```


map Template Class

- A function given as set of ordered pairs
  - for each value first, at most one value second in map
- Ex: map<**string**, **int**> numMap;
  - Each **string** value known as a key
  - numMap can associate a unique **int** value
- Stores in sorted order, like set
  - *Ordering* is on key values only
  - Second value can have no ordering impact

Example of map (1/3)

lec11-6.cpp

```
#include <iostream>
#include <fstream>
#include <map>
#include <set>
#include <string>

using namespace std;

int main() {
    set<string> ignore; //words to ignore
    map<string, int> freq;
    //map of words and their frequencies
    string word; //used to hold input word
```


Example of map (2/3)

```
//in main()
    //-- read file of words to ignore.
    ifstream ignoreFile("ignore-list.txt");
    while (ignoreFile >> word) {
        ignore.insert(word);
        //insert word to set ignore
    }

    //-- count from input stream.
    while (cin >> word) {
        if (ignore.find(word)==ignore.end())
            freq[word]++; //update map freq
    }
    ...
```

lec11-6.cpp

Example of map (3/3)

```
//in main()
    //-- write word(key)/count as a pair
    map<string, int>::const_iterator iter;
    for (iter = freq.begin();
         iter != freq.end();
         ++iter) {
        cout << iter->first << " "
             << iter->second << endl;
        //itr mani. using first & second
    }
    ...
```

lec11-6.cpp

Algorithms

- Generic & rich standalone template functions
  - Operate on containers
  - Perform container access through iterators
  - generally unaware of containers
- Category of Algorithms
  - non-mutating sequence algorithm
  - mutating algorithms
  - sorting/searching algorithms
  - generalized numeric algorithms

Non-mutating Sequence Algorithms

- Apply to sequence containers
  - NOT modify container's contents
  - search for elements in sequences, check for equality and to count sequence elements
- Include:
  - `for_each()`, `count()`, `mismatch()`, `equal()`, `search()`, `find()`, `adjacent_find()`, `find_if()`, `find_end()` and more

Examples for find() and find_if()

```
…
char * str = "Hello world in C++ and STL";
int len = strlen(str);
char * pos = find(&str[0], &str[len],'S');
cout <<pos;
…
```

lec11-7.cpp

```
class gt100 {
public:
  bool operator() (int x) { return x>100; }
};
…
vector<int> iv;
for (int i = 0; i<20; i++)
  iv.push_back(i*20);
vector<int>::iterator pos;
pos=find_if(iv.begin(),iv_end,gt100()); // *pos=??
```

Mutating Sequence Algorithms

- Modify contents of containers
- Many have
  - **`if`** version: perform actions only if data member evaluates to be true
  - **`copy`** version: copy output to new containers
- Include:
  - `copy(), copy_backward(), swap(), fill(), generate(), partition(), replace(), reverse(), rotate(), swap_ranges(), transform(), unique()` and more

Example for `copy()`/`copy_backward()`

lec11-7.cpp

```
…
vector<int> v1;
v1.push_back(1);
v1.push_back(11);
v1.push_back(21);
v1.push_back(31);

vector<int> v2(v1.size());

copy(v1.begin(), v1.end(), v2.begin());
// guess what happens?
copy_backward(v1.begin(), v1.end(),
  v2.end());
...
```


Sorting/Searching Algorithms

- Used to either search or sort container contents
- Versions
  - use **<** operator for comparison
  - use user-defined comparison *functor* (not covered,; self-study)
- Inlcude
  - `sort(), stable_sort(), partial_sort(), nth_element(), merge(), equal_range(), binary_search(), lower_bound()` and more

Example of `sort()`

```
class comp {
public:
    bool operator() (int x, int y) {
        return x>y;
    }
};
…
vector<int> iv;
for (int i = 40; i>=0; --i)
     iv.push_back(i);

sort( iv.begin(), iv.end(), comp());
…
```

lec11-8.cpp

Performance issue

- STL implementation was 40% slower than hand-optimized version
  - STL: used `deque`
  - hand coded: used “circular buffer” array
- Application with STL `list` ~5% slower than custom list
- However, spending several days debugging the hand-coded version

  ⇨ usually not worth it
- Use STL for rapid prototyping

More on Generic Programming

- Often missing in basic programming courses, even in OOP courses
- The most successful example of the GP implementation is STL
  - until 1994, STL became part of C++ standard library
- STL is not merely a collection of functions and classes, but a collection of templates or patterns.
- GP is orthogonal to OOP ⇨ new language level
  - no encapsulation, (almost) no inheritance

Summary (1/2)

- STL is a powerful library
  - includes many generic containers and generic algorithms
- Iterator is ‘generalization’ of a pointer
  - used to move through elements of container
- Container classes with iterators have:
  - member functions `end()`and `begin()`to assist cycling

Summary (2/2)

- Main kinds of iterators:
  - forward, bi-directional, random-access
- A few containers, generic algorithms provided
  - some algorithms work on specific containers and some on all containers
- Iterators provide common mechanism to access elements in any container