

Comp2014 Object Oriented Programming

Lecture 12

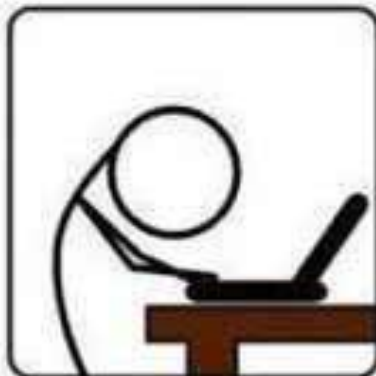
Standard Template Library

Topic covered in last lecture

- ◆ Function template
- ◆ Class template
- ◆ General concept of data structures
- ◆ Linked list

The Programmers Life

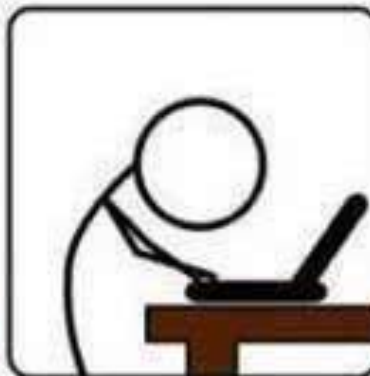
WORK



HOME



PLAY



SLEEP



The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules,

Although practicality beats purity.

Errors should never pass silently,

Unless explicitly silenced.



The Zen of Python, by Tim Peters

In the face of ambiguity, refuse the temptation to guess.

There should be one - and preferably only one - obvious way to do it

Although that way may not be obvious at first unless you're Dutch.

Now is better than never,

*Although never is often better than *right* now.*

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea - let's do more of those!



Complex is better than complicated

```
#include "Constants.h"
#include "Ticket.h"
#include "EventTicket.h"
#include "FlightTicket.h"
#include "HotelVoucher.h"
#include "ClientRequest.h"
#include "RequestGenerator.h"
#include "Package.h"
#include "TravelAgent.h"
#include "SmartTravelAgent.h"
```

Complex is better than complicated

```
cout <<"Choose a class to test: " << endl;
cout <<"1. Test ClientRequest class" << endl;
cout <<"2. Test FlightTicket class" << endl;
cout <<"3. Test HotelVoucher class" << endl;
cout <<"4. Test EventTicket class" << endl;
cout <<"5. Test RequestGenerator class"<< endl;
cout <<"6. Test Package class" << endl;
cout <<"7. Run Travel Agent" << endl;
cout <<"8. Run Smart Travel Agent" << endl;
cout <<"9. Quit" << endl;
```

Complex is better than complicated

```
void runTravelAgent() {  
    TravelAgent agent;  
    agent.readClientRequests();  
    agent.generatePackages();  
    agent.printSuccessfulPackages();  
}
```


Complex is better than complicated

```
void runSmartTravelAgent() {  
    SmartTravelAgent smartagent;  
    smartagent.readClientRequests();  
    smartagent.generatePackages();  
    smartagent.printSuccessfulPackages();  
    smartagent.vacancy();  
}
```

Complex is better than complicated

```
void TravelAgent::generatePackages() {
    for (int i = 0; i < counter; i++) {
        Package* p = generatePackage(requests[i]);
        if (p != NULL && p->validayPackage()) {
            successfulPackages.push_back(p);
        } else {
            cout << "Client " << requests[i].cId
                << ": No sufficient budget or resources. " << endl;
        }
    }
}
```

Complex is better than complicated

```
Package* TravelAgent::generatePackage(ClientRequest cr) {
    Package* p = new Package;
    p->setRequest(cr);
    int flyinDay = cr.earliestEventDay();
    int flyoutDay = cr.latestEventDay();
    p->addFlightTicket(0, flyinDay);
    p->addFlightTicket(1, flyoutDay);
    for (int i = 0; i < NUMEROFEVENTS; i++) {
        if (cr.events[i])
            p->addEvenTicket(i);
    }
    for (int i = flyinDay; i < flyoutDay; i++)
        p->addHotelVoucher(cr.hotelType, i, 0.0);
    return p;
}
```

Topic covered in the lecture

- ◆ Standard Template Library (STL)
 - Sequential containers
 - Associative containers
 - Container adaptors
 - Iterators
 - Algorithms

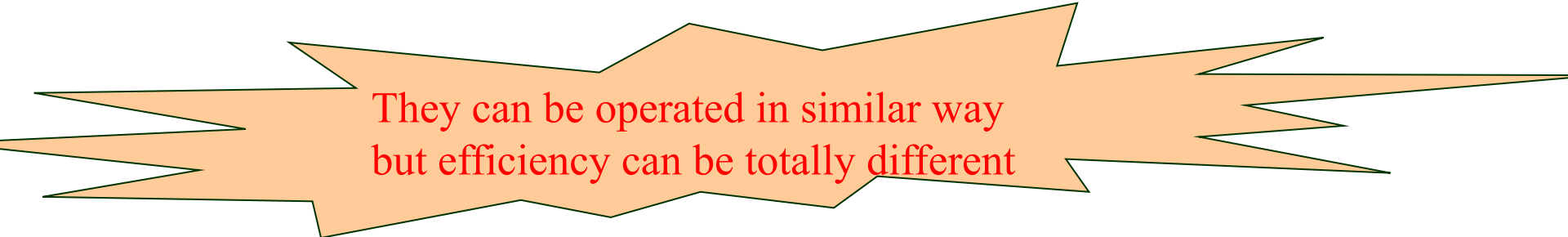
Standard Template Library

- ◆ To make programming simpler and easier, most C++ compilers provide class library for users to use. ANSI/ISO recommended a certain list of such classes as Standard Library. Most of these classes are in template form so that they are **independent** to data types. These class templates become the basis of the **Standard Template Library** (STL).
- ◆ STL includes:
 - **Containers**: are *template classes from which individual data structures can be constructed*.
 - **Iterators**: are pointers *keeping track of positions in a data structure and establishing the boundaries of sequences of elements*.
 - **Algorithms**: are *template functions that provide useful search, sort, location, and other numeric functions*.

Container types in STL

Container classes are class templates which provide a possibility to use higher abstraction level in the programming. There are three types of containers:

- ◆ **Sequential container:** *Each element has a certain position that is determined when the element is inserted. Typical sequential containers are:*
 - **vector:** *similar to an array but the size of vector is variable.*
 - **deque:** *double ended queue, elements added at front or back while allow random access via an iterator.*
 - **list:** *doubly linked list, insertions & deletions anywhere in list.*



They can be operated in similar way
but efficiency can be totally different

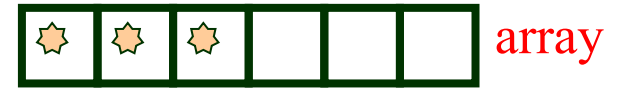
Container types in STL

- ◆ **Associate container**: *an associative container contains a list of elements with variable size that supports efficient retrieval of elements based on **keys**. STL has four associative containers:*
 - **set**: *a collection of elements with no duplicates (sorted)*
 - **multiset**: *a collection of elements with duplicates (sorted)*
 - **map**: *store and retrieve data with unique keys (hashtable or balanced binary tree).*
 - **multimap**: *store and retrieve data with possible multiple keys (hashtable or balanced binary tree).*

Container types in STL

- ◆ **Container adapters** *are class templates which provide member functions push and pop that properly insert an element into each adapter data structure and properly remove an element from each adapter data structure. STL has three container adapters:*
 - **Stack Adapter:** LIFO
 - **Queue Adapter:** FIFO
 - **Priority_queue Adapter:** *highest priority element first out*

Vector and iterator



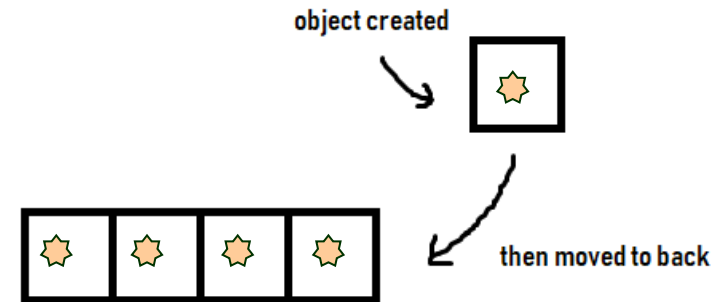
Vector: *an array-like container that stores a varying number of elements and provides **random** access to them.*

Header file:

```
#include <vector>
```

Typical operations which are effective:

- `[index]` : *random access with index*
- `push_back()`: *appending a new element value at the end.*
- `pop_back()`: *removing an existing value from the end.*



Operations that are not effective:

- `insert(iterator position, const T& x)`: inserting a new element into the vector.

Iterator will be explained later

- `erase(iterator position)`: removing an element from the vector.

Iterator

An iterator is like a pointer. It points to an element in a container.

Mostly we use an iterator as a loop variable.

Given a vector: `vector<ClientRequest> requests;`

we can declare an iterator as follows:

```
vector< ClientRequest>::iterator itor;
```

Then to use it, we can have

```
for (itor = requests.begin(); itor != requests.end(); itor++)  
    cout << itor->budget;
```

Alternative ways:

```
for (int i=0; i< requests.size(); i++)  
    cout << requests[i].budget;
```

or

```
for (ClientRequest cr : requests) // C++11  
    cout << cr.budget;
```

See [iterator.cpp](#)

deque

Deque (*double ended queue*): a double-ended queue which elements can be added to or removed from the front or the back.

Header file:

```
#include<deque>
```

Typical operations:

- `push_back()`: appending a new element value at the end.
- `push_front()`: appending a new element value at the beginning.
- `pop_back()`: removing an existing value form the end.
- `pop_front()`: removing an existing value form the beginning.
- `insert(iterator, value)`: insert a value into the deque at the position (not efficient)

Check out other methods of deque via

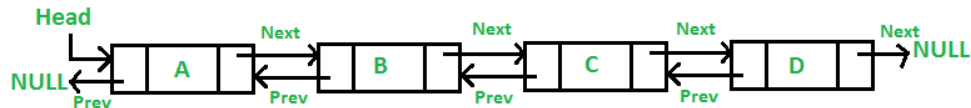
<http://www.cplusplus.com/reference/deque/deque/>

list

List: *an implementation of doubly linked list template. Each node of a linked list contains two pointers, pointing to the previous and the next nodes, respectively.*

Header file:

```
#include<list>
```



Typical operations:

- `insert(iterator position, const T& x)`: inserting a new element into the list.
- `erase(iterator position)`: removing an element from the list.

Remark 1. *The operations `push_back`, `pop_back`, `push_front` and `pop_front` are defined also for `list`.*

Remark 2. *If you want to iterate over all elements you need **iterator** to do it. You can't use index.*

Typical methods for sequential containers

- ◆ Member function `size()` indicates the current number of elements stored in the container (it has nothing to do with the size of allocated space).
- ◆ Member function `capacity()` indicates how many elements can be stored to the container without the need to allocate more memory space.
- ◆ Function `push_back()` and `push_front()` allocates automatically more memory space when needed.
- ◆ Function `reserve()` can be used to allocate a certain amount of memory. This is often useful to prevent unnecessary memory allocations in `push_xxx` operations.
- ◆ Function `resize()` can also be used to resize the underlying array.

set

Set is used for efficient management of object collections with **sortable keys** so that insertion, deletion, and search operations can be performed with logarithmic runtimes. Sets are implemented internally with **binary search trees**. Set will automatically sort elements and eliminate duplicated elements.

Operator “<” for **elements** determines the sorting order by default (users have to overload the `operator<` for user defined data type).

The new elements are put in a set using the function *insert()* (*push_back()* and *push_front()* are not available).

The best way to access the elements is to use **iterators**.

Header file `<set>` for set.

See set.cpp

map

Unordered_map and **Map** store pairs of sorted keys and objects using **hashing technology** and **binary search trees**, respectively. The key of each object is used to identify the object but is stored separately from the object. The comparison criterion is applied to the keys.

Example:

```
map<string, double> stringDoubleMap;  
stringDoubleMap["e"] = 2.71828;  
stringDoubleMap["pi"] = 3.14159;  
stringDoubleMap.insert(pair<string, double>("key", 10.0));  
  
map<int, ClientRequest> requests;  
map<int, Record> studentRecords;
```

See example map.cpp

Algorithms

STL provides a great number of algorithms which can be used across the various containers. Examples include *searching*, *sorting*, *inserting* and *deleting*. There are approximately 70 algorithms included.

Eg1. *copying all elements to a stream object:*

```
vector<int> v;  
copy( v.begin(), v.end(), output);
```

this copies all elements of a vector v from the beginning to the end to *output*.

Eg2. *insert an element into a vector:*

```
v.insert(v.begin()+1, 22);
```

this inserts 22 at the second position.

Eg3. *count elements:*

```
int result = count( v.begin(), v.end(), 8);
```

this will count the number of occurrences of 8 between the beginning and the end of the vector.

Mathematical functions

Eg4. *find minimum element*

```
cout << "The minimum element is " << *(min_element(v.begin(), v.end()));
```

this uses the function `min_element` to locate the smallest element in the vector and returns a pointer to that smallest element.

Eg5. *calculate total*

```
cout << "The total of vector v is " << accumulate(v.begin(), v.end(), 0);
```

this uses the function `accumulate` (in the `<numeric>` header file) to total all elements in vector `v`.

Example

```
#include <vector> #include <iterator> #include <algorithm> #include <iostream>
using namespace std;
void main() {
    float c_array[5] = {3.0, 4.0, 5.0, 2.0, 1.0};
    vector<float> stl_array(c_array, c_array + 5);
    sort(stl_array.begin(), stl_array.end()); //sort is STL algorithm
    vector<float>::iterator pos;
    pos = find(stl_array.begin(), stl_array.end(), 2.0);
    if (pos == stl_array.end())
        cout << "2.0 was not found " << endl;
    else
        cout << "2.0 was found in the position " << pos - stl_array.begin() << endl;
}
```

See example algorithm.cpp

Homework

- ◆ Read textbook Chapter 19.
- ◆ Demonstrate your code of assignment 2 during your practical session. **No demonstration, no marks.**
- ◆ There would be a significant delay for the assignment checking. If you have any urgency, please let your tutor know. You may be granted to show your work in week 14.
- ◆ Online tutorial 3 should have been available.
- ◆ The lecture in next week will be a review of this unit. I will show you the format of the final examination and some example questions. No previous examination papers are available in the library.