Object-Oriented Programming
Introduction to STL (containers and iterators)

CSIE Department, NTUT
Woei-Kae Chen
Example: A slideshow

- We have a few images. We want to do a slideshow using these images.
- Starting from the first image, the slideshow will show each image for a duration of $t$ seconds.
- The slideshow ends when all images are shown.
**Example: A slideshow**

Solution 1

<table>
<thead>
<tr>
<th>Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Protection</td>
</tr>
</tbody>
</table>

![Diagram](image_url)
Example: A slideshow
Solution 2: Put images in a container

What if you want list instead of vector as a container? (Bad protection)
– The fact that vector is used as a container should be hidden.
Example: A slideshow
Solution 3: Use an iterator interface
Example: A slideshow
Is it better? (1)

- Protection: slideshow does not need to change in case a list is used instead of a vector.
  - To do this, replace vector with list and VectorIterator with ListIterator.
  - ListIterator must be compliant with the Iterator interface.
Example: A slideshow
Is it better? (2)
Example: A slideshow
Is it better? (3)

Reuse: an object enumerator
STL Iterators
STL Programming Paradigms

- STL programming is called *generic programming* that encourages decoupling of algorithms and data structures.
- In addition, it can easily blend with
  - Procedure-based paradigm
  - Object-based paradigm
  - Object-oriented paradigm
Ways STL Can Help

- Ready solution: how to put them together? (example 1)
- Develop your algorithms so that they work on different types of data structures: how to work with iterators? (example 2)
- Develop your data structures so that they work with different algorithms: how to create the right iterators? (example 2)
The Problem

- **Problem:** You have a number of objects stored in a container. You want to write a find function to find certain elements.
- **Objective:** write a find function that works on various type of container regardless of data types
Candidate Containers

- Fixed size, random access: vector

- Variable size, sequential access: list
Example 1: Eating Out Of The Can

```cpp
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int a[]={0,2,4,6,8,10,12,14,16,18};
    int size = sizeof(a)/sizeof(int);
    vector<int> va(a, size);
    vector<int>::iterator key;

    for (int i=0; i<size; ++i) // output: 0,2,4,6,8
        if ((key=find(va.begin(),va.end(),i)) != va.end())
            cout << "va[" << key-va.begin() << "] = " << *key << "\n";
}
```
C++ Language Feature: Template

- Functions (and classes) that are otherwise identical except the type of data they process

```cpp
template<class T>
void swap (T &a, T & b) {
    T temp t=a;
    a=b;
    b=t;
}

void swap (int &a, int & b) {
    int temp t=a;
    a=b;
    b=t;
}
```
Example 2

● OOD walk-through for developing a generic find function
A Development Scenario

- Start with a find function using int array as the container. (version 1)
- Eliminate find’s dependency on array with pointers (version 2.1)
- Convert find into template (version 2.2)
- Encapsulate pointer with iterator (version 3)
- Make container responsible for generating its own iterator (version 3)
#include <iostream>
bool find1(int a[ ], int size, int value, int &key);
int main()
{
    int a[ ]={0,2,4,6,8,10,12,14,16,18};
    int size = sizeof(a)/sizeof(int);
    int key;

    for (int i=0; i<size; ++i)
        if (find1(a,size,i,key))
            cout << "a[" <<key<<"] = " << a[key] << "\n";
bool find1(int a[], int size, int value, int &key) {
    for (int i = 0; i < size; ++i) {
        if (a[i] == value) {
            key = i;
            return true;
        }
    }
    return false;
}
**Version 1 Facts**

- `find()` explicitly knows that it is using an integer array. (Knowledge implies dependency.)

- To achieve our objective:
  - Replace indexing operator `[ ]` and range of array with pointers (Version 2.1)
  - Make `find` a template function to remove data type dependency (Version 2.2)
int main()
{
    int a[]={0,2,4,6,8,10,12,14,16,18};
    int size = sizeof(a)/sizeof(int);
    int *key, *begin, *end;

    begin=a;
    end=a+size;

    for (int i=0; i<size; ++i)
        if (find2(begin,end,i,key))
            cout << "a[" <<key-begin<<"] = " << *key << "\n";
}
bool find2(int *begin, int *end, int value, int *key)
{
    for (int *i=begin; i != end; ++i)
        if (*i==value) {
            key = i;
            return true;
        }
    return false;
}
template <class T>
bool find3(T *begin, T *end, T value, T *&key)
{
    for (T *i=begin; i != end; ++i)
        if (*i==value) {
            key = i;
            return true;
        }
    return false;
}
The following statement implies that find3 knows that the contiguous memory is allocated to the container:

```c
for (T *i=begin; i != end; ++i)
```

This is OK for array type containers, but what about list and tree type containers? Encapsulate so that find() doesn’t operate directly on raw pointers. (Version 3)
int main()
{
    int a[]={0,2,4,6,8,10,12,14,16,18};
    vector<int> va(a, sizeof(a)/sizeof(int));
    int size = sizeof(a)/sizeof(int);
    vector<int>::iterator key;

    for (int i=0; i<size; ++i)
        if (find5(va.begin(), va.end(), i, key))
            cout << "va[" << key - va.begin() << "] = " << *key << "\n";
}
template <class IT, class T>
bool find5(IT begin, IT end, T value, IT &key)
{
    for (IT i=begin; i != end; ++i)
        if (*i==value) {
            key = i;
            return true;
        }
    return false;
}
Version 3 Facts

- What are iterator’s services?
- Who should be responsible for generating iterators? A container knows how best an iterator works on it.
Iterator Services

- Operators:
  - inequality operator !=
  - prefix increment operator ++
  - de-reference operator *
  - assignment operator =

- Functions
  - constructors and destructors
Containers and Iterators

- Array and raw pointer
- STL containers and iterators
  - vector and vector::iterator
  - list and list::iterator
  - map and map::iterator
  - set and set::iterator
  - etc.
vector and vector::iterator(1)

Vector is responsible for creating vector::iterator.

```
template <class T>
class vector {
    class iterator;

public:
    vector(int sz): size(sz) { p = new T [size]; }
    vector(T a[], int sz): size(sz) {
        p = new T [size];
        for (int i = 0; i < size; ++i)
            p[i] = a[i];
    }
```
vector and vector::iterator(2)

iterator begin() {return iterator(p);}
iterator end() {return iterator(p+size);}

private:
int size;
T *p;

public:
class iterator {
public:

iterator() {}
iterator(T *p):current(p){}
iterator(const iterator & it):current(it.current){}
bool operator != (iterator x) {return current != x.current;}
T & operator*() {return *current;}
iterator & operator++() { ++current; return *this;}
int operator-(iterator it) {return current-it.current;}
iterator & operator=(iterator it){current=it.current;}

private:
    T *current;
};

};
Summary

- STL provides with great number of containers and algorithms that can be readily reused.
- STL provide a reference model for developing your data structures and algorithms.
- STL can easily work with procedure and object paradigms.