Programming Tasks

- Understanding the problem
- Creating a design
- Selecting data structures for storing data
- Selecting algorithms to work on the data
- Coding, testing and Debugging
- Generate results [and write papers]

Reality Checks (I)

- I don’t know much about data structures
- I do know data structures pretty well, but am I implementing it correctly? (Same thing about algorithms.)

Reality Checks (II)

- I know my implementation is correct, but is it efficient?
- I know my implementation is correct, but if I change my data structure (algorithm), will my algorithm (data structure) still work?

In comes C++ Standard Template Library.

What’s in C++ STL (1)

- A collection of data structures regardless of data types
- A collection of algorithms that solve many common problems regardless of data types
- A collection of interfaces (iterators) between data structures and algorithms so that one can vary regardless of the other
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)
        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
void swap(int &a, int & b)
{
    int temp = a;
    a = b;
    b = temp;
}

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

Version 1-main

```cpp
#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";
}
```

Version 1 - find

```cpp
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)

Version 2.1 - main

```cpp
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";
}
```
Version 2.1 - find

```c
#include <iostream>

bool find2(int *begin, int *end, int value, int *&key)
{
    for (int *i = begin; i != end; ++i)
        if (*i == value)
            key = i;
    return true;
}
```

Version 2.2-find

```c
#include <iostream>

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;
}
```

Version 2.2 Facts

- 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)

Version 3-main

```c
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 << 
```
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 {
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];
  }

private:
  int size;
  T *p;
public:
  class iterator {
public:
    iterator();
    iterator(T *p):current(p){}
    iterator(const iterator & it):current(it.current){}
  
  // operators
  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;
  
};
```

vector and vector::iterator(2)

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

private:

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

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.
Further Information