An Introduction to STL

The C++ Standard Template Libraries

- In 1990, Alex Stepanov and Meng Lee of Hewlett Packard Laboratories extended C++ with a library of class and function templates which has come to be known as the STL.
- In 1994, STL was adopted as part of ANSI/ISO Standard C++.

The C++ Standard Template Libraries

- STL had three basic components:
  - Containers
    Generic class templates for storing collection of data.
  - Algorithms
    Generic function templates for operating on containers.
  - Iterators
    Generalized 'smart' pointers that facilitate use of containers.
    They provide an interface that is needed for STL algorithms to operate on STL containers.
- **String abstraction was added during standardization.**

Why use STL?

- STL offers an assortment of containers
- STL publicizes the time and storage complexity of its containers
- STL containers grow and shrink in size automatically
- STL provides built-in algorithms for processing containers
- STL provides iterators that make the containers and algorithms flexible and efficient.
- STL is extensible which means that users can add new containers and new algorithms such that:
  - STL algorithms can process STL containers as well as user defined containers
  - User defined algorithms can process STL containers as well user defined containers

Strings

- In C we used char * to represent a string.
- The C++ standard library provides a common implementation of a string class abstraction named string.

Hello World - C

```c
#include <stdio.h>

void main()
{
    // create string 'str' = "Hello world!"
    char *str = "Hello World!";

    printf("%s\n", str);
}
```

Hello World - C++

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

int main()
{
    // create string 'str' = "Hello world!"
    string str = "Hello World!";

    cout << str << endl;
    return 0;
}
```

String

- To use the string type simply include its header file.

```
#include <string>
```

Creating strings

```
string str = "some text";
```

or

```
string str("some text");
```

other ways:

```
string s1 = str;
```

string length

The length of string is returned by its `size()` operation.

```
#include <string>

string str = "something";
cout << "The size of "
   << str
   << "is " << str.size()
   << "characters." << endl;
```

The size method

str.size() ???

In C we had structs containing only data, In C++, we have :

```
class string
{
   …
 public:
   …
   unsigned int size();
   …
};
```

String concatenation

concatenating one string to another is done by the '+' operator.

```
string str1 = "Here ";
string str2 = "comes the sun";
string concat_str = str1 + str2;
```

String comparison

To check if two strings are equal use the '==' operator.

```
string str1 = "Here ";
string str2 = "comes the sun";

if ( str1 == str2 )
   /* do something */
else
   /* do something else */
```

String assignment

To assign one string to another use the "=" operator.

```
string str1 = "Sgt. Pappers";
string str2 = "lonely hearts club bend";
str2 = str1;
```

Now : str2 equals "Sgt. Pappers"

What more ?

- Containers
- Algorithms

Containers

Data structures that hold anything (other objects).

- list: doubly linked list.
- vector: similar to a C array, but dynamic.
- map: set of ordered key/value pairs.
- Set: set of ordered keys.

Algorithms

generic functions that handle common tasks such as searching, sorting, comparing, and editing:

- `find`
- `merge`
- `reverse`
- `sort`
- `and more: count, random shuffle, remove, Nth-element, rotate.`

Vector

- Provides an alternative to the built in array.
- A vector is self grown.
- Use It instead of the built in array!

Defining a new vector

Syntax: `vector<of what>`

For example :

`vector<int>` - vector of integers.
`vector<string>` - vector of strings.
`vector<int * >` - vector of pointers to integers.
`vector<Shape>` - vector of Shape objects. Shape is a user defined class.

Using Vector

- `#include <vector>`

- Two ways to use the vector type:
  1. Array style.
  2. STL style

Using a Vector - Array Style

We mimic the use of built-in array.

```
void simple_example()
{
    const int N = 10;
    vector<int> ivec(N);
    for (int i=0; i < 10; ++i)
        cin >> ivec[i];

    int ia[N];
    for ( int j = 0; j < N; ++j)
        ia[j] = ivec[j];
}
```

Using a vector - STL style

We define an empty vector

```
vector<string> svec;
```

we insert elements into the vector using the method push_back.

```
string word;
while ( cin >> word ) //the number of
          words is unlimited.
{
   svec.push_back(word);
}
```

Insertion

```
void push_back(const T& x);
```

Inserts an element with value x at the end of the controlled sequence.

```
svec.push_back(str);
```

Size

```
unsigned int size();
```

Returns the length of the controlled sequence (how many items it contains).

```
unsigned int size = svec.size();
```

Class Exercise 1

Write a program that read integers from the user, sorts them, and print the result.

Solving the problem

- Easy way to read input.
- A "place" to store the input
- A way to sort the stored input.

Using STL

```
int main()
{
   int input;
   vector<int> ivec;

   /* rest of code */
}
```

STL - Input

```
while ( cin >> input )
   ivec.push_back(input);
```

STL - Sorting

```
sort(ivec.begin(), ivec.end());
```

Sort Prototype:

```
void  sort(Iterator first, Iterator  last);
```

STL - Output

```
for ( int i = 0; i < ivec.size(); ++i )
   cout << ivec[i] << " ";
cout << endl;
```

Or (more recommended)

```
vector<int>::iterator it;
for ( it = ivec.begin(); it != ivec.end(); ++it )
   cout << *it << " ";
cout << endl;
```

STL - Include files

```
#include <iostream>   // I/O
#include <vector>     // container
#include <algorithm> // sorting

//using namespace std;
```

Putting it all together

```
int main() {
    int input;
    vector<int> ivec;

    // input
    while (cin >> input )
        ivec.push_back(input);

    // sorting
    sort(ivec.begin(), ivec.end());

    // output
    vector<int>::iterator it;
    for ( it = ivec.begin();
          it != ivec.end(); ++it ) {
           cout << *it << " ";
    }
    cout << endl;

    return 0;
}
```

Operations on vector

- iterator **begin**();
- iterator **end**();
- bool **empty**();
- void **push_back**(const T& x);
- iterator **erase**(iterator it);
- iterator **erase**(iterator first, iterator last);
- void **clear**();
- ….

Standard C++ Library

Part II

Standard C++ Library

- Map
- pair
- copy algorithm

Employee

```
class Employee {
public:
    // Constructors ...:
  Employee () {}
  Employee (const string& name) : _name(name) {}

    // Member functions ...:
  void set_salary(int salary) {_salary = salary; }
  int salary() const { return _salary; }
  void set_name(const string& name) { _name = name; }
  const string& name() const { return _name;}
    // …
private:
    int  _salary;
    string  _name;
};
```

Locating an Employee

Save all employees in a vector.

When we need to find a specific employee:

go over all employees until you find one that its name matches the requested name.

Bad solution - not efficient!

Solution: Map – Associative Array

- Most useful when we want to store (and possibly modify) an associated value.
- We provide a key/value pair. The key serves as an index into the map, the value serves as the data to be stored.
- Insertion/find operation – O(logn)

Using Map

Have a map, where the key will be the employee name and the value – the employee object.

```
name          employee.
string     class Employee

map<string, Employee *> employees;
```

Populating a Map

```
void main()
{
 map<string, Employee *> employees;
 string name("Eti");
 Employee *employee;

 employee = new Employee(name);

 //insetrion
 employees[name] = employee;
}
```

Locating an Employee

```
map<string, Employee> employees;
```

Looking for an employee named Eti :

```
//find
Employee *eti = employees["Eti"];
//or
map<string, Employee *>::iterator iter =
employees.find("Eti");
```

The returned value is an iterator to map.
If "Eti" exists on map, it points to this value, otherwise, it returns the end() iterator of map.

Iterating Across a Map

Printing all map contents.

```
map<string,Employee *>::iterator it;
for ( it = employees.begin();
      it != employees.end(); ++it )
{
   cout << it->first;
}
```

Iterators

Provide a general way for accessing each element in sequential (vector, list) or associative (map, set) containers.

Pointer Semantics

Let `iter` be an iterator then :

- `++iter` (or `iter++`)
  Advances the iterator to the next element
- `*iter` Returns the value of the element addressed by the iterator.

Begin and End

Each container provide a `begin()` and `end()` member functions.

- `begin()` Returns an iterator that addresses the first element of the container.
- `end()` returns an iterator that addresses 1 past the last element.

Iterating Over Containers

Iterating over the elements of any container type.

```
for ( iter = container.begin();
      iter != container.end();
      ++iter )
{
   // do something with the element
}
```

Map Iterators

```
map<key, value>::iterator iter;
```

What type of element iter does addresses?

- The key ?
- The value ?

It addresses a **key/value pair**.

Pair

Stores a pair of objects, first of type T_1, and second of type T_2.

```
struct pair<T1, T2>
{
   T1 first;
   T2 second;
};
```

Our Pair

In our example iter addresses a

```
pair<string, Employee *>
```

Element.

Accessing the name (key)

```
iter->first
```

Accessing the Employee* (value)

```
iter->second
```

Printing the Salary

```
for ( iter = employees.begin();
      iter != employees.end();
      ++iter )
{
   cout << iter->first << " "
        << (iter->second)->salary();
}
```

Example Output

alon 3300

dafna 10000

eyal 5000

nurit 6750

Thank You