Standard Template Library (STL)

- STL is a C++ library that provides many basic algorithms and data structures to support C++ programming.
  - Container classes;
  - Generic algorithms;
  - Functional objects;
  - Iterators;


Template (generic) Functions

- **Generic functions** are functions with type variables.
- Generic function calls just like other function calls.

```cpp
#include <iostream>
using namespace std;
template<class T> T halveMe(T x){return x/2;}
int main()
{
  cout << halveMe(5.0f) << " , " << halveMe(5) << halveMe('0') << halveMe("10") << endl; // type error!!
  return 0;
}
```

Generic Functions - Continue

- Based on the argument types of a function call, compiler infers the type of the type variable:
  - `halveMe(5.0f)`: `T` is float
- Compiler then substitutes the type variable with the inferred type to generate a copy of the function (similar to cpp expands micros).
  - `float halveMe(float x){return x/2;}`
Template Class

Template classes are classes with type variables.

```cpp
//Template.h file
#ifndef _TEMPLATE_HH
#define _TEMPLATE_HH

template<class T1,
class T2>
class Pair{
public:
    Pair(T1 t1, T2 t2):
        first(t1), second(t2)
    {}    
    T1 first_v() const
    {return first;}
    T2 second_v() const
    {return second;}
private:
    T1 first;
    T2 second;
};
#endif
```

Template Classes

- Each instantiation of the a template class (Pair<int,string>) generates a distinct class.
- We can manipulate objects of the template instantiated classes (pair1) the same way as objects of non-template classes.

```
//main.cc file
#include "Template.h"
#include<iostream>
#include<string>
using namespace std;

int main()
{
    Pair<int,string> pair1(43,"Hello");
    cout << pair1.first_v() <<", " << pair1.second_v() << endl
}
```

STL Template Container Classes

- STL provides several template container classes:
  - String:
  - Sequences: vector, deque, list, and so on
  - Associative containers: set and map, hash etc.
  - Container adaptors: stack, queue, priority_queue

Using string

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

int main()
{
    string name; //create an empty string
    string stars(100,'*'); //create a string with 100 *
    string greeting="Hello"; //assignment operator
    string gretname= greeting + name; //concatenate two strings
    string::size_type i=stars.size(); //return the number of characters in stars
    //0 > int > 2^15=32767; string::size_type has no limit. Each container class has a size_type member
}
Vector Template Class

- Array (C array):
  - Implemented using pointer;
    int a[10];
    a[0]=100;
    //*a=100;
  - No member function;
  - Fixed length;
  - No out-of-bound checking;

- Vector is “smart” array.
  - Struct (not pointer) implementation
  - Member functions;
  - Flexible length;
  - Out-of-bound checking
    push_back (push element onto the back of a vector) in O(1) time.
  - insert is O(n).

Using Vector

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

int main(){
  vector<double> homework;
  //create a vector of double values
  homework.push_back(90); //add at the end
  homework.insert(homework.begin(),80); //insert at the front
  double average=(homework.at(0)+homework.at(1))/homework.size();
  cout << average << endl; //calling homework.at(3) raise a run-time exception.
}
```

Vector and Function

- Vector is treated like struct in C++:
- Vector parameters are passed by values:
  - The entire vector argument is copied over to the vector parameter.
  - The changes made in the vector argument are not visible outside the function.
- Functions can return local vector variable, which can be assigned to a vector variable.

Function with Vector Arguments

```cpp
#include<vector>
using namespace std;
vector<int> oddEven(vector<int> original)
{
  vector<int> odd; //local variable
  vector<int> even;
  for(vector<int>::size_type i=0; i < original.size(); i++)
    if(original.at(i) % 2)
      odd.push_back(original.at(i));
    else
      even.push_back(original.at(i));
  original=odd;
  return even;
}
```
```cpp
#include<iostream>
#include<vector>
using namespace std;

extern vector<int> oddEven(vector<int>);

int main()
{
    vector<int> values;
    for (int i = 0; i < 10; i++)
        values.push_back(i);

    vector<int> result = oddEven(values);
    for (vector<int>::size_type i = 0; i < result.size(); i++)
        cout << result.at(i) << endl;
    for (vector<int>::size_type i = 0; i < values.size(); i++)
        cout << values.at(i) << endl;
}
```

---

**Vector Arguments**

- To allow the changes made on vector parameter inside the function visible outside the function, pass the **reference of the vector** as function argument:
  ```cpp
  vector<int> oddEven(vector<int> & original)
  ```
- Call the function in the same way:
  ```cpp
  vector<int> result = oddEven(values);
  ```