Templates

Jiun-Long Huang
National Chiao Tung University

Standard Template Library (STL)

- Containers
  - Containers are classes storing objects
  - Sequences
  - Associations
- Iterators
  - Iterators are classes used to manipulate objects stored in containers
- Algorithms
  - find, sort, binary_search…

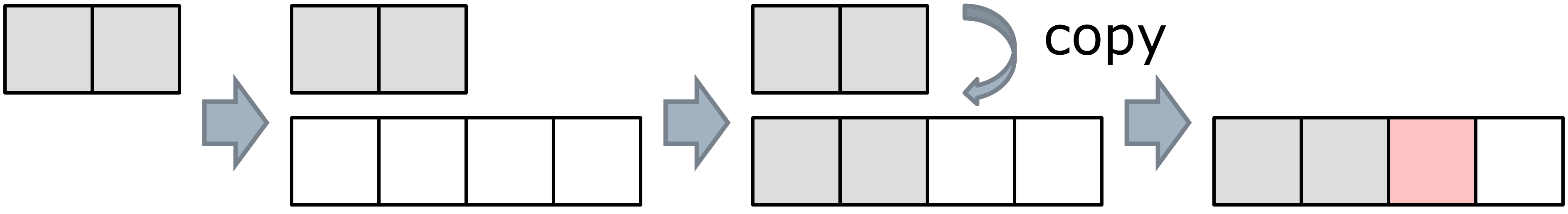
Containers (Collections)

- Sequences:
  - Basic sequences: vector, list, deque
  - stack, queue and priority_queue are implemented on top of basic sequences
- Associations:
  - set, multiset, map and multimap

vector

- The vector is intentionally made to look like an enhanced array, since it has array-style indexing but also can expand dynamically.
- To achieve maximally-fast indexing and iteration, the vector maintains its storage as a single contiguous array of objects.
  - Indexing and iteration are lighting-fast, being basically the same as indexing and iterating over an array of objects.

- Inserting an object anywhere but at the end (that is, appending) is not really an acceptable operation for a vector.
- When a vector runs out of pre-allocated storage, in order to maintain its contiguous array it must allocate a whole new (larger) chunk of storage elsewhere and copy the objects to the new storage.



- Header file
  - #include <vector>
- Constructors:
  - vector (const Allocator& = Allocator());
    - Constructs an empty vector, with no content and a size of zero.
  - vector (size_type n, const T& value= T(), const Allocator& = Allocator());
    - Initializes the vector with its content set to a repetition, n times, of copies of value.

- Member functions:
  - size_type size()
    - Returns the number of elements in the vector container.
  - size_type capacity()
    - Returns the size of the allocated storage space for the elements of the vector container.
  - reference operator[] (size_type n)
    - Returns a reference to the element at position n in the vector container.

- void push_back (const T& x);
  - This effectively increases the vector size by one, which causes a reallocation of the internal allocated storage if the vector was full before the call.
- void pop_back ( );
  - Removes the last element in the vector, effectively reducing the vector size by one
- iterator insert (iterator position, const T& x)
  - The vector is extended by inserting new elements before the element at position

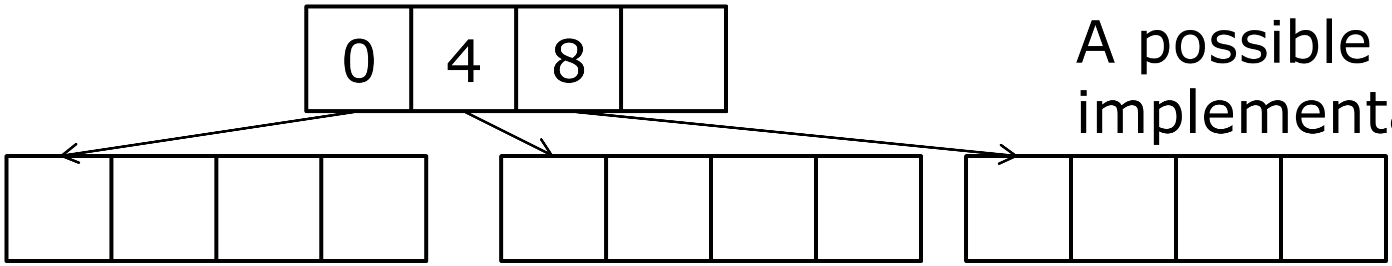
- iterator erase (iterator first, iterator last)
- iterator erase (iterator position)
  - Removes from the vector container either a single element (position) or a range of elements ([first,last)).
- iterator begin ()
  - Returns an iterator referring to the first element in the vector container.
- iterator end ()
  - Returns an iterator referring to the *past-the-end* element in the vector container.

```cpp
#include <vector>
#include <iostream>
using namespace std;
int main()
{
  vector<int> v(5);
  vector<int>::iterator it;
  cout<<v.size()<<" "<<v.capacity()<<endl;
  for(int i=0;i<v.size();i++)
    v[i]=i*i;
  for(int i=0;i<v.size();i++)
    cout<<v[i]<<" ";
  v.push_back(6);
  cout<<endl;
  cout<<v.size()<<" "<<v.capacity()<<endl;
  v.erase(v.begin()+3);
  cout<<v.size()<<" "<<v.capacity()<<endl;
  for(it=v.begin();it<v.end();it++)
    cout<<*it<<" ";
};
```

```
5 5
0 1 4 9 16
6 10
5 10
0 1 4 16 6
```

deque

- The deque (double-ended-queue, pronounced “deck”) is the basic sequence container optimized for adding and removing elements from either end.
- It also allows for reasonably fast random access – it has an operator[ ] like vector.
- It does not have vector’s constraint of keeping everything in a single sequential block of memory.



A possible implementation of deque

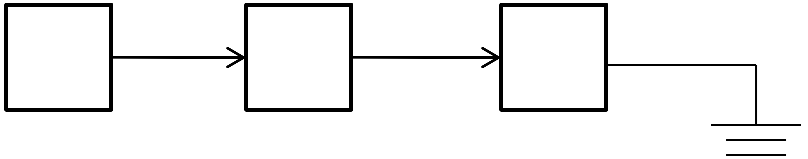
- Instead, deque uses multiple blocks of sequential storage.
  - The overhead for a deque to add or remove elements at either end is very low.
  - It never needs to copy and destroy contained objects during a new storage allocation (like vector does) so it is far more efficient than vector if you are adding an unknown quantity of objects.
- The usages of deque and vector are very similar
  - deque does not have member function capacity

```
#include <deque>
#include <iostream>
using namespace std;
int main()
{
  deque<int> v(5);
  deque<int>::iterator it;
  cout<<v.size()<<endl;
  for(int i=0;i<v.size();i++)
    v[i]=i*i;
  for(int i=0;i<v.size();i++)
    cout<<v[i]<<" ";
  v.push_back(6);
  cout<<endl;
  cout<<v.size()<<endl;
  v.erase(v.begin()+3);
  cout<<v.size()<<endl;
  for(it=v.begin();it<v.end();it++)
    cout<<*it<<" ";
};
```

```
5
0 1 4 9 16
6
5
0 1 4 16 6
```

list

- A list is implemented as a doubly-linked list and is thus designed for rapid insertion and removal of elements in the middle of the sequence (whereas for vector and deque this is a much more costly operation).
- A list is so slow when randomly accessing elements that it does not have an operator[ ]



- It’s best used when you’re traversing a sequence, in order, from beginning to end (or end to beginning) rather than choosing elements randomly from the middle.
- The usages of list and deque are very similar
  - list does not support operator[]

```
#include <list>
#include <iostream>
using namespace std;
int main()
{
  list<int> v(5);
  list<int>::iterator it;
  int i;
  cout<<v.size()<<endl;
  for(i=0,it=v.begin();it!=v.end();it++,i++)
    *it=i*i;
  for(it=v.begin();it!=v.end();it++)
    cout<<*it<<" ";
  v.push_back(6);
  cout<<endl;
  cout<<v.size()<<endl;
  it=v.begin();
  //v.erase(it+3); // ERROR
  advance(it,3);
  v.erase(it);
  cout<<v.size()<<endl;
  for(it=v.begin();it!=v.end();it++)
    cout<<*it<<" ";
};
```

```
5
0 1 4 9 16
6
5
0 1 4 16 6
```

map

- Maps are a kind of associative containers that stores elements formed by the combination of a key value and a mapped value.
- Main characteristics of a map as an associative container are:
  - Unique key values: no two elements in the map have keys that compare equal to each other.

- For a similar associative container allowing for multiple elements with equivalent keys, see multimap.

- Each element is composed of a key and a mapped value.
- Elements follow a strict weak ordering at all times.
  - Unordered associative arrays, like unordered_map, are available in implementations following TR1 (C++ Technical Report 1).

- unordered_map will replace the various incompatible implementations of the hash table (called hash_map by GCC and MSVC).
- Member functions:
  - T& operator[] (const key_type& x);
    - If x matches the key of an element in the container, the function returns a reference to its mapped value.
  - iterator insert (iterator position, const value_type& x);
    - The map container is extended by inserting a single new element

- void erase (iterator position);
- size_type erase (const key_type& x);
  - Removes from the map container a single element
- T& operator[] (const key_type& x);
  - If x matches the key of an element in the container, the function returns a reference to its mapped value.
  - If not, the function inserts a new element with that key and returns a reference to its mapped value.

- iterator find (const key_type& x)
  - Searches the container for an element with a value of x and returns an iterator to it if found, otherwise it returns an iterator to end() (the element past the end of the container).

```
#include <map>
#include <iostream>
#include <string>
using namespace std;
int main()
{
  map<string,int> m;
  map<string,int>::iterator it;
  m["Alice"]=100;
  m["Bill"]=50;
  m["Charles"]=70;
  cout<<m["Alice"]<<endl;
  cout<<m.size()<<endl;
  m.erase("Alice");
  cout<<m.size()<<endl;
  cout<<m["Alice"]<<endl;
  cout<<m.size()<<endl;
  m.erase("Alice");
  cout<<m.size()<<endl;
  if (m.find("Alice")==m.end())
    cout<<"No record";
  else
    cout<<m["Alice"];
};
```

```
100
3
2
0
3
2
No record
```

Algorithms

- Function templates:
  - for_each
  - find
  - sort
  - binary search

```cpp
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
int main()
{
  vector<int> v1(3);
  vector<string> v2(3);
  v1[0]=100;
  v1[1]=50;
  v1[2]=70;
  v2[0]="TA1";
  v2[1]="jlhuang";
  v2[2]="TA2";
  sort(v1.begin(),v1.end());
  sort(v2.begin(),v2.end());
  for(int i=0;i<3;i++)
    cout<<v2[i]<<" "<<v1[i]<<endl;
};
```

```
TA1 50
TA2 70
jlhuang 100
```

Templates

- A class template defines a family of classes.
- Template serves as a class outline, from which specific classes are generated at compile time.
  - One template can be used to generate many classes.
- Class templates promote code reusability and reduce a program's development time.

Define Templates

- To define a class template, the template keyword followed by a template parameter list must precede a class declaration.

```
template <template_parameter_list>
class class_template_name
{
  //body of the class template
};
```

```
template<class T>
class Array
{
  T * pt;
  int n;
public:
  Array();
  ~Array();
  void getValues();
  void print();
};
Array<int> intArrary;
Array<float> floatArray;
```

```
class Array<int> {
  int * pt;
  int n;
public:
  Array();
  ~Array();
  void getValues();
  void print();
};
```

```
class Array<float> {
  float * pt;
  int n;
public:
  Array();
  ~Array();
  void getValues();
  void print();
};
```

Template Parameters

- There are three forms of template parameters:
  - Type parameters
  - Non-type parameters
  - Template parameters
- When instantiating a template class, a specific data type listed in the template argument list will substitute for the type identifier.

- Either the class keyword, or the typename keyword must precede a template type parameter in a template parameter list.

```
template<class T1, class T2, class T3> class X {...};
template<typename A, typename B> class Y {...};
```

- When instantiating objects from the X and Y class templates, type identifiers (T1, T2, T3, A, and B) will be substituted with specific data types.

```
X<int, float, int> xi;  //T1=int, T2=float, T3=int
Y<char, int> yl;  //A=char, B=int
Y<int, double*> y2;      //A=int, B=double*
```

```cpp
#include <iostream>
using namespace std;
template<class T>                        //template header
class Array {
private:
   T *pt;        //pointer to array
   int n;        //number of array elements
public:
   Array(int x=20);
   ~Array() { delete [] pt; }
   void getValues();
   void print();
};
template<class T>                     //template header
Array<T>::Array(int x) {
  n=x>0 ? x : 20;   //Initializes size of the array
  pt=new T[n];      //Allocates memory dynamically
  if(!pt) {
    cout<<"Memory Allocation Error!";
    exit(1);
  }
  for(int i=0;i<n;i++)    //Initializes array
    pt[i]=0;
}
```

```
template<class T>                           //template header
void Array<T>::getValues() {
  for(int i=0;i<n;i++) {
    cout<<"\tEnter value "<<(i+1)<<": ";
    cin>>pt[i];
  }
}
template<class T>                           //template header
void Array<T>::print() {
  cout<<"\nArray elements =>"<<endl;
  for(int i=0;i<n;i++)
    cout<<"\tArray["<<i<<"]="<<pt[i]<<endl;
}
int main()
{
  Array<int> intArr(4);
  Array<char> chArr(5);
  cout<<"Integer values =>\n";
  intArr.getValues();
  intArr.print();
  cout<<"\nCharacter values =>\n";
  chArr.getValues();
  chArr.print();
  return 0;
}
```

Array<int> and Array<char> are types generated by template

```
Integer values =>
        Enter value 1: 1
        Enter value 2: 2
        Enter value 3: 3
        Enter value 4: 4

Array elements =>
        Array[0]=1
        Array[1]=2
        Array[2]=3
        Array[3]=4

Character values =>
        Enter value 1: a
        Enter value 2: b
        Enter value 3: c
        Enter value 4: d
        Enter value 5: e

Array elements =>
        Array[0]=a
        Array[1]=b
        Array[2]=c
        Array[3]=d
        Array[4]=e
```

Non-type parameters

- A non-type parameter can be one of the following types:
  - Integral type (char, int, bool)
  - Enumeration type
  - Reference to object or function
  - Pointer to object, function, or member
- A non-type parameter cannot be one of the following types:
  - Floating point type (float, double)
  - Class type
  - void

Non-type parameters

- Example

```
template<int i, char c, bool b> class X;  //CORRECT
template<float *fp, double &dr> class Y; //CORRECT
//ERROR; cannot be a class type
template<Circuit cr> class Z;
//ERROR; cannot be floating point type
template<double d> class O;

template<class T, int i>
class Array {
private:
   T pt[i]; //pointer to array
};
Array<int, 5> intArr;
```

- A non-type parameter is treated and processed as a constant.
- A non-type template argument must therefore be a constant expression

```
const int a = 4;
//Non type template arguments are constants.
X<a,'C',true> obj;
```

- A template parameter may have a default argument.
- The default template argument is specified after the = operator in the template parameter declaration.

```
template<class T=float, int n=10>
class Array { /* Body */};
```

Non-type parameters

- When using this template to generate specific classes, one or both arguments can be optional.

```
Array< > ar1; //Valid; same as: Array<float, 10> arl;
Array ar2;    //Syntax Error; missing < >
Array<int, 50> ar3;  //Valid
Array<char> ar4;  //Valid; same as: Array<char,10> ar4;
//Invalid; missing type template argument
Array<20> ar5;
```

Using Friends and Static Members with Class

- The following functions/classes can be used as friends of a template class:
  - Global functions
  - Member function of a non template class
  - Member function of a template class
  - Non template class
  - Template class

```
template<class T>
class Probe {
  friend void funl();   //friendship #1
  friend void Test1::fun2();  //friendship #2
  friend void fun3(Probe<T> &);//friendship #3
  friend void Test2<T>::fun4(Probe<T> &); //friendship #4
  friend class Test3;   //friendship #5
  friend class Test4<T>;      //friendship #6
  //members of the Probe class template
};
```

```
class Probe<int> {
  friend void funl();
  friend void Test1::fun2();
  friend void fun3(Probe<int> &);
  friend void Test2<int>::fun4(Probe<int> &);
  friend class Test3;
  friend class Test4<T>;
};


class Probe<float> {
  friend void funl();
  friend void Test1::fun2();
  friend void fun3(Probe<float> &);
  friend void Test2<float>::fun4(Probe<float> &);
  friend class Test3;
  friend class Test4<T>;
};
```

Friend Relationship

- The Probe class template has four friend functions and two friend classes with the following relationships according to their declarations:
  - fun1() is a friend function of every template class that is instantiated from the Probe class template.
  - fun2 () is a member function of the Test1 class and also a friend function of every template class that is instantiated from Probe.

Friend Relationship (contd.)

- fun3 () is a friend function of a template class that is instantiated from Probe for a particular type.
  - fun3(Probe<int> &) is a friend of Probe<int> and is not a friend of Probe<float>, Probe<double>, or Probe<char>. (T=int)
- fun 4() is a member function of the Test 2 template class and also a friend function of a template class that is instantiated from Probe for a particular type.
  - Test2<int>:: fun4(Probe<int>&) is a member function of the Test2<int> template class and a friend function of Probe<int>.

Friend Relationship (contd.)

- Test3 is a friend class of every template class generated from Probe.
- A template class instantiated from the Test4 class template for a particular type is a friend class of a template class generated from Probe for this type.
  - Test4<double> is a friend class of Probe<double> (T=double)

Static Data

- A class template can contain static data members and static member functions.

```
template<class Ttype>
class Test {
public:
   static Ttype tot;
   static void fun();
};
//static data member definition
template<class Ttype> Test<Ttype>::tot=0;
//static member function definition
template<class Ttype> void
Test<Ttype>::fun() {
     cout<<tot<<endl ;
}
```

```
class Test<int> {
public:
   static int tot;
   static void fun();
};

class Test<float> {
public:
   static float tot;
   static void fun();
};
```

Static Data (contd.)

- Every template class instantiated from a class template that contains static members will have its own copy of all static members.
- Each template class instantiated from Test, therefore, will have its own copies of tot and fun().

```
Test<int>::tot=13;
Test<float>::tot=7.9;
Test<int>::fun();   //prints 13
Test<float>::fun(); //prints 7.9
```

Function Templates

- Format of definition:

```
template <template-parameters>
return-type function-name(parameter-list)
{
  //Body
}
```

- Format of invoking

```
function-name(parameters);
```

```
#include<iostream>
using namespace std;
const int size=5;
void sort(int[]);
void sort(float[]);
int main()
{
  int nums1[size]={3,9,1,-5,0};
  float nums2[size]={9.1,-0.7,4.6,0.3,9.9};
  sort(nums1); //Calls overloaded function
  sort(nums2); //Calls overloaded function
  cout<<" Sorted arrays:"<<endl;
  for(int j=0; j<size; j++)
    cout<<setw(5)<<nums1[j]<<setw(8)<<nums2[j]<<endl;
  return 0;
}
```

```
void sort(int arr[]) {
  int temp;
  for(int j =1; j<size; j++)
    for(int k =0; k<size-1; k++)
      if(arr[k]>arr[k+1]) {
        temp = arr[k];
        arr[k] = arr[k+1];
        arr[k+1] = temp;
      }
}
void sort(float arr[]) {
  float temp;
  for(int j =1; j<size; j++)
    for(int k =0; k<size-1; k++)
      if(arr[k]>arr[k+1]) {
        temp = arr[k];
        arr[k] = arr[k+1];
        arr[k+1] = temp;
      }
}
```

```
#include <iostream>
#include <iomanip>
using namespace std;
const int size = 5;
template <class T>
void sort(T arr[])
{
  T temp;
  for(int j =1; j<size; j++)
    for(int k =0; k<size-1; k++)
      if(arr[k]>arr[k+1]) {
        temp = arr[k];
        arr[k] = arr[k+1];
        arr[k+1] = temp;
      }
}
```

```
class TwoDVector
{
private:
  float x, y;
public:
  TwoDVector(float x=0, float y=0)
    { this->x=x;this->y=y; }
  friend int operator>(TwoDVector&, TwoDVector&);
  friend ostream& operator<<(ostream&, TwoDVector&);
};
int operator>(TwoDVector& v1, TwoDVector& v2) {
  if ( (v1.x*v1.x+v1.y*v1.y) > (v2.x*v2.x+v2.y*v2.y) )
    return 1;
  else
    return 0;
}
ostream & operator<<(ostream & os, TwoDVector & v) {
  os<<"("<<v.x<<","<<v.y<<")";
  return os;
}
```

```
int main()
{
  int nums1[size]={3,9,1,-5,0};
  float nums2[size]={9.1,-0.7,4.6,0.3,9.9};
  TwoDVector v[size]={TwoDVector(2,2),TwoDVector(3,3),
    TwoDVector(1,1),TwoDVector(0,0),TwoDVector(5,5)};
  sort(nums1);
  sort(nums2);
  sort(v);
  cout<<" Sorted arrays:"<<endl;
  for(int j=0; j<size; j++)
    cout<<setw(5)<<nums1[j]<<setw(8)<<nums2[j]
      <<setw(8)<<v[j]<<endl;
  return 0;
}
```

```
 Sorted arrays:
   -5    -0.7       (0,0)
    0     0.3       (1,1)
    1     4.6       (2,2)
    3     9.1       (3,3)
    9     9.9       (5,5)
```