CS-202

C++ Templates (Pt.3)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	PASS Session		PASS Session

Your 10th Project will be announced today Thursday 4/25.

9th Project Deadline was this Wednesday 4/24.

- NO Project accepted past the 24-hrs delayed extension (@ 20% grade penalty).
- Send what you have in time!

Today’s Topics

Template(s) and Names

- Dependent-Qualified Type names – Keyword ***`typename`*** Disambiguator.
- Explicitly-Qualified names – Keyword **`template`** Disambiguator.

Template(s) Compilation process

The Standard Template Library

- Containers & Iterators.
  - `std::vector`
  - `std::list` , `std::forward_list`
  - `std::set` , `std:: multiset`
  - `std::pair`
  - `std::set` , `std::multiset`
  - `std::map` , `std::multimap`
- String Class.
  - `std::string`

Remember: **The keyword `template`**

Declares a family of classes / family of functions.

- Two alternatives:

A) Overloading the keyword `class`:

```
template < class T >
return-type tpl-func-name(parameters-list){ … }
template < class T >
class TplClassName { … };
```

B) Using the new keyword `typename`:

```
template < typename T >
return-type tpl-func-name(parameters-list){ … }
template < typename T >
class TplClassName { … };
```

Template(s)

Remember: **Syntax**

The Templated Function:

```
template < class T >
void Swap(T & v1, T & v2);

template < class T >
void Swap(T & v1, T & v2){ T temp = v1; v1 = v2; v2 = temp; };
```

Call with implicit / explicit template parameter statement:

```
int     i1=0,         i2=1;           Swap(i1, i2);
float   f1=0.1,       f2 = 99.9;      Swap< float >(f1, f2);
Car     c1("GRAY"), c2("WHITE");      Swap(c1, c2);
Date    d1(4,20),     d2(4,21);       Swap< Date >(1, d2);
```

Inferred / Declared Type
`T : int`
`T : float`
`T : Car`
`T : Date`

Remember: **Syntax**

The Templated Class:

```
template < class T >
class Buffer{
 public: …
  Buffer();
 private: …
  T *m_buffer;
};

template < class T >
Buffer< T > :: Buffer(){ m_buffer = new T[…]; … }
```

Instantiation with explicit template parameter statement:

```
Buffer < int > intBuffer;
Buffer < Car > carBuffer;
```

Declared Type

```
T : int
T : Car
```

Template(s)

Remember: **Syntax**

Member template(s).

```
template < class T >
template < class M >
void TplClass< T > :: TplClassTpl( M * t_arr)
{ /* T and M - mentioning implementation */ };
```

```
template <class T>
class TplClass{
 public: …
  template <class M>
  void TplFuncTpl(M * t_arr);
};
```

template(s) Specialization & Overloading

Base Template

```
template < class T >
T & Max(const T & v1,
      const T & v2){
 return (v1 < v2)? v2:v1;
};
```

A) Overloaded (for char*)

```
const char* Max(const char* & v1
             ,const char* & v2){
 return strcmp(v1,v2)? v2:v1;
};
```

Has priority over any Templated version !

Partially Specialized Template (for T*)

```
template < class T >
const T* & Max(const T* & v1
           ,const T* & v2){
 return (*v1 < *v2)? v2:v1;
};
```

B) Explicitly Specialized (for char*)

```
template < >
const char* Max(const char* & v1
             ,const char* & v2){
 return strcmp(v1,v2)? v2:v1;
};
```

Remember: **Syntax**

The Parameter List of `template`(s).

- Multiple Parameter Types & Non-Type Parameters:

```
template < class T, class U, class V, int N, char C >
return-type multi-tpl-func-name(parameters-list){ … }
template < class T, class U, class V, int N, char C >
class MultiTplClassName { … };
```

- Default Parameters:

```
template < class T = int, int N = MAX_ELEMENTS >
class MultiTplClassName { … };
```

Note:
Only for Class Templates.

`template`(s) as Arguments in Function Parameter List(s).

```
template < class T >
void Sort( ArrayContainer< T > & arrayContainer ){
  for (…)
    if ( arrayContainer[…] < arrayContainer[…] )
         arrayContainer[…] = …;
}
```

A Function that has a Templated Parameter.

Template(s)

Remember: **Syntax**

friend(s) of Class template(s)

A) Implementation out of Templated Class:

```
template<class T> class ArrayContainer;
template<class T> ostream & operator<< (ostream & os, const ArrayContainer<T> & a);
```

Forward Declarations

```
template < class T >
class ArrayContainer { …
 friend ostream & operator<< <> (ostream & os, const ArrayContainer<T> & a);
};
```

Class Template

```
template < class T >
ostream & operator<<(ostream & os, const ArrayContainer<T> & a){ /* function body */ }
```

Function Implementation

B) Inline implementation (inside of Templated Class Declaration):

```
template < class T >
class ArrayContainer { …
 friend ostream & operator<< (ostream & os, const ArrayContainer<T> & a)
 {  /* function body */  }
};
```

Inline Implementation in Templated Class

The keyword **`template`**

Name(s) and **`template`**(s).

Qualified / Unqualified Names

- Example of Qualified names:

```
std::cout << "Hello World!" << std::endl ;
```

Objects' Names ***`cout`*** and **`endl`** are Qualified (in the **`namespace`** ***`std`***).

The keyword `template`

Name(s) and `template`(s).

Qualified / Unqualified Names

➢ Example of Qualified names:

```
std:: cout << "Hello World! " << std:: endl ;
```

➢ Example of Unqualified names:

```
using namespace std;
cout << "Hello World!" << endl ;
```

`using` directive introduces Names into the Namespace scope that it appears in (same as `using std::cout; using std::endl;`).

➢ The Compiler has to lookup names by respecting and evaluating qualifications.

The keyword `template`

Name(s) and `template`(s).

Qualified / Unqualified Names

- Example of Qualified names:

```
std:: cout << "Hello World! " << std:: endl ;
```

- Example of Unqualified names:

```
using namespace std;
cout << "Hello World!" << endl ;
```

- The Compiler has to lookup names by respecting and evaluating qualifications.

Note: Still Qualified names, despite introducing `std` names into global scope.

```
using namespace std;
std::cout << "Hello World!" << std::endl ;
```

Template(s)

The keyword `template`

Name(s) and `template`(s).

Dependent Names

- ➤ Name(s) of constructs whose *instantiations* Depend on Template Parameters (& can't be looked-up until these are known).

```
template < class T >
class MotionPlan : public List< T >
{
    MotionPlan() : m_total( List<T>::total_num )
    { }
    void AdvanceWaypoint(List< T > * l)
    {   l -> m_wp ++;   }
    const int m_total;
    typename List< T >::Waypoint m_wp;
};
```

`List<T>` is Dependent on `T`:
(CityName, GeoReferencedCoordinate, CheckerboardSquare …)

The keyword `template`

Name(s) and `template`(s).

Dependent Names

➢ Name(s) of constructs whose *instantiations* Depend on Template Parameters (& can't be looked-up until these are known).

```
template < class T >
class MotionPlan : public List< T >
{
    MotionPlan() : m_total( List<T>::total_num )
    { }
    void AdvanceWaypoint(List< T > * l)
    {  l -> m_wp ++; }
    const int m_total;
    typename List< T >::Waypoint m_wp;
};
```

A *Member* whose instantiation is dependent on `T` : (total *miles*^{2}, number of cities in map, …)

A *Member* of the Templated Class `l` whose instantiation is dependent on `T`: (raw pointer, index, iterator…)

The keyword `template`

Name(s) and `template`(s).

Dependent Names

➢ Name(s) of constructs whose *instantiations* Depend on Template Parameters (& can't be looked-up until these are known).

```
template < class T >
class MotionPlan : public List< T >
{
    MotionPlan() : m_total( List<T>::total_num )
    { }
    void AdvanceWaypoint(List< T > * l)
    {   l -> m_wp ++;   }
    const int m_total;
    typename List< T >::Waypoint m_wp;
};
```

```
template < class T >
class List
{
  static const int total_num = … ;
  typedef T* Waypoint;
};
```

A *Type* whose instantiation is Dependent on `T`.

The (keyword) typename Disambiguator

Disambiguation of Dependent & Qualified Names:

```
template < class T >
void sort_local(){
  ArrayContainer< T > :: Accessor ac;

  ac.moveToBeginning(); T data = ac.accessData();
  ac.moveForward(); T data = ac.accessData();
  …
}
```

- … :: Accessor is a Qualified name.
- Accessor is the name of a *Type* whose instantiation is Dependent on Template Parameter T.

C++ Standard (14.6/2): "A name used in a template declaration or definition and that is dependent on a template-parameter is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword typename."

Compiler will attempt to interpret this as a *variable*.

The (keyword) `typename` Disambiguator

Disambiguation of Dependent & Qualified Names:

```
template < class T >
void sort_local(){
  ~~ArrayContainer< T > :: Accessor~~ ac;
  typename ArrayContainer< T > :: Accessor ac;
  ac.moveToBeginning(); T data = ac.accessData();
  ac.moveForward(); T data = ac.accessData();
  …
}
```

- `… :: Accessor` is a Qualified name.
- `Accessor` is the name of a Type whose instantiation is Dependent on Template Parameter `T`.

When instantiating code employing Names that are Qualified and Dependent the Compiler needs to know `T` is used as part of a Type name, as the Standard demands *early checking* to be enabled (otherwise `Accessor` can be the name of a Member *or* a nested Type, and we would have to wait until `T` is known to perform any checks.

The (keyword) `template` Disambiguator

Disambiguation of Explicitly Qualified Template Member(s):

- Explicitly-Qualified names:

```
class TplMethodClass {
  public: …
    template<class T>
    T tpl_member_func();
};

template < class U >
void func( U arg )
{
    int obj =
    arg . template tpl_member_func<int>();
}
```

ISO C++03 14.2/4 :

When the name of **a member template specialization** appears after `.` or `->` in a postfix-expression, or after nested-name-specifier in a qualified-id, and the postfix-expression or qualified-id **explicitly depends on a template-parameter** (14.6.2), the member template name must be prefixed by the keyword **`template`**. Otherwise the name is assumed to name a non-template.

Note: Otherwise the compiler can't *know early on* whether the upcoming symbol **<** is a less-than operator or the beginning of a template parameters list.

Compiling Templates

Function & Class `template`(s) compilation process.

- Keep declarations (normally placed in the `.h` header file in any case) as well as implementation (normally placed in the `.cpp` source file inside A SINGLE Header (`.h`) file.

- Include this (`.h`) Header file in all places you would normally `#include` your Function / Class Declarations.

The Standard Template Library

A set of Standard-implemented Classes & Libraries.

The STL contains many useful things, including:

- Containers
  Store Data and maintain Data Association.
- Container Adapters
  Provide Higher-level Data Structure interfaces but can rely on different Container back-ends.
- Iterators
  Access & Manipulation of Data.

All are Templated:

- They can be used with *(almost)* any type of data.

STL

The Standard Template Library

A set of Standard-implemented Classes & Libraries.

The STL provides Re-usable code:

- Exhaustively tested & optimized
- Thoroughly debugged
- Standardized
- Extensive & Comprehensive

  linked list, vector, map, multi-map, pair, set, multi-set, queue, stack, etc …

We usually have many more things to do than re-inventing the wheel ...

STL Containers

All containers implemented in the form of Templated Classes.

They all provide support for some common basic methods:

- `bool empty();` → Check if container is empty.
- `void clear();` → Mark the container as empty, and deallocate data if necessary.
- `std::size_t size();` → Return number of elements in the container.

Note:

`std::size_t` is the type of any `sizeof` expression and is guaranteed to be able to express:

- The maximum size of any object (including any array).
- The maximum number for any array index.
- Used for array sizes, indexes, and for iteration:

```
for ( std::size_t i = 0; i<…; ++i){ … }
```

STL Containers

Vector(s) (`std::vector< … >`)

Basic attributes:

- Dynamic Container (its size can change), contains elements of Type `T`.
- Sequential Container (its elements are in order).
- Random Access Container.

  Using:

```
T & operator[](std::size_t pos);
```

Access element at position `pos` (without bounds checking).

```
T & at(std::size_t pos);
```

Access element at position `pos` (throws a `std::out_of_range` Exception if `!(pos < size())`).

Both return Reference (not Pointer) to requested element.

STL Containers

Vector(s) (`std::vector< … >`)

Basic functions:

- `T & front();`
- `T & back();`

Access first and last element in the container.

- `void push_back(const T & value);`
- `void pop_back();`

Push element to last position (back) or remove last element (back).

Insert or erase element in any position within the container (iterator-based method).

- `std::vector<T>::iterator insert(std::vector<T>::const_iterator pos,`
  `const T & value);`
- `std::vector<T>::iterator erase(std::vector<T>::const_iterator pos);`

- `void resize(std::size_t size);`

Resize container to fit size number of elements.

STL Containers

Linked-List(s) (`std::list< … >`, `std::forward_list< … >`)

Basic attributes:

- Double-Linked –or– Forward-Linked List, contain elements of Type `T`.
- Constant-time insertion / removal from anywhere in the List.
- Does not support Random Access.

Basic functions:

- `T & front();` / `T & back();`
- `void push_back(const T & value);` / `void pop_back();`
- `void push_front(const T & value);` / `void pop_back();`
- `std::list<T>::iterator insert(std::list<T>::const_iterator pos, const T & value);`
- `std::list<T>::iterator erase(std::list<T>::const_iterator pos);`
- `void reverse();` / `void sort();` / `void merge(std::list<T> & other);`

STL Containers

Set(s) (`std::set< … >`)

Basic attributes – *Unique* `Keys` :

- Associative Container – contains Sorted set of elements of Type `Key`.
- Elements are sorted when added to the set, uses a `Compare` function (by default `std::less`, largely `operator<()`) to perform ordering of `Key` elements.
- Cannot change the `Key` key element value once added.
- Does not support Random Access.

Basic functions:

```
std::set<Key>::iterator find(const Key & key);
std::size_t count(const Key& key);
std::pair<std::set<Key>::iterator, bool> insert(const Key & value);
std::set<Key>::iterator erase(std::list<Key>::const_iterator pos);
```

Note:
Uses *Equivalence* checking (via the `Compare` function) to find matching `Key` key from the Set.

STL Containers

Multiset(s) (`std::multiset< … >`)

Basic attributes – *Multiple* `Key` *Entries* :

- Can contain multiple Type `Key` keys with Equivalent values (unlike `std::set<…>`).
- Elements are sorted when added via `Compare` (by default `operator<()`).
- Cannot change the `Key` key element value once added.
- Does not support Random Access.

Basic functions:

```
➢ std::list<T>::iterator find(const Key & key);
➢ std::size_t count(const Key & key);
➢ std::pair<std::multiset<Key>::iterator, bool> insert(const Key & value);
➢ std::multiset<Key>::iterator erase(std::multi<Key>::const_iterator pos);
```

Note:
Uses *Equivalence* checking.

STL Containers

Pair(s) (`std::pair< …,… >`)

Basic attributes:

- Connects two-Type items into a single Object.
- Two elements are stored in type `T1` and `T2` member variables:

```
T1 first;
T2 second;
```

- Publicly accessible (`std::pair` is a `struct` ADT).
- `std::pair`(s) are used by other containers.

```
int racePosition = 1;
Car myCar("Gandalf");
std::pair<int, Car> racecar;
racecar.first = racePosition;
racecar.second = myCar;
```

Basic functions:

- `std::pair<T1,T2> make_pair(T1 t1, T2 t2);`

```
std::pair<int, Car> racecar = std::make_pair(racePosition, myCar);
```

STL Containers

Map(s) (`std::map< …,… >`)

Basic attributes – *Unique* `Key`s :

- Associative Container – contains Sorted Pairs of Type `Key`–Type `T` (value).
- Elements are sorted by their `Key` key when added (via `Compare` –by default `operator<()`), while association to the Type `T` value is maintained in the Pair.
- Cannot change the `Key` key element value once added.
- Can change however the associated `T` value of that key-value Pair.

Basic functions:

- `std::map<Key,T>::iterator find(const Key & key);`
- `std::size_t count(const Key & key);`

Note:
Uses *Equivalence* checking (via `Compare`) to find matching `Key` key.

- `std::pair<std::map<Key,T>::iterator, bool> insert(const std::pair<const Key,T> & v);`
- `std::map<Key,T>::iterator erase(std::map<Key>::const_iterator pos);`

STL Containers

Multimap(s) (`std::multimap< …,… >`)

Basic attributes – *Multiple* **`Key`** *Entries* :

- Can contain Sorted Pairs of Type **`Key`**–Type **`T`** (value) with multiple Type **`Key`** keys with Equivalent values (unlike `std::map<…,…>`).
- Elements are sorted by their **`Key`** key when added (via **`Compare`**).
- Cannot change the **`Key`** key element value once added.
- Can change however the associated **`T`** value of that key-value Pair.

Basic functions:

```
➢ std::multimap<Key,T>::iterator find(const Key & key);
➢ std::size_t count(const Key & key);
```

Note:
Uses *Equivalence* checking.

```
➢ std::pair<std::map<Key,T>::iterator, bool> insert(const std::pair<const Key,T> & v);
➢ std::multimap<Key,T>::iterator erase(std::multimap<Key>::const_iterator pos);
```

STL Containers

Unordered Map(s) (`std::unordered_map< …,… >`)

Basic attributes – *Unique* **`Key`**s :

- Associative Container – contains Non-Sorted Pairs of Type **`Key`**–Type **`T`** (value).
- Operations such as element Search, Insertion, Removal have *Average Constant Time* complexity.
- Typically used to implement *Hash-Tables*.
- Cannot change the **`Key`** key element value once added.
- Can change however the associated **`T`** value of that key-value Pair.

Basic functions:

```
➢ std::unordered_map<Key,T>::iterator find(const Key & key);
➢ std::pair<std::unordered_map<Key,T>::iterator, bool> insert(const std::pair<const Key,T> & v);
➢ std::unordered_map<Key,T>::iterator erase(std::unordered_map<Key>::const_iterator pos);
```

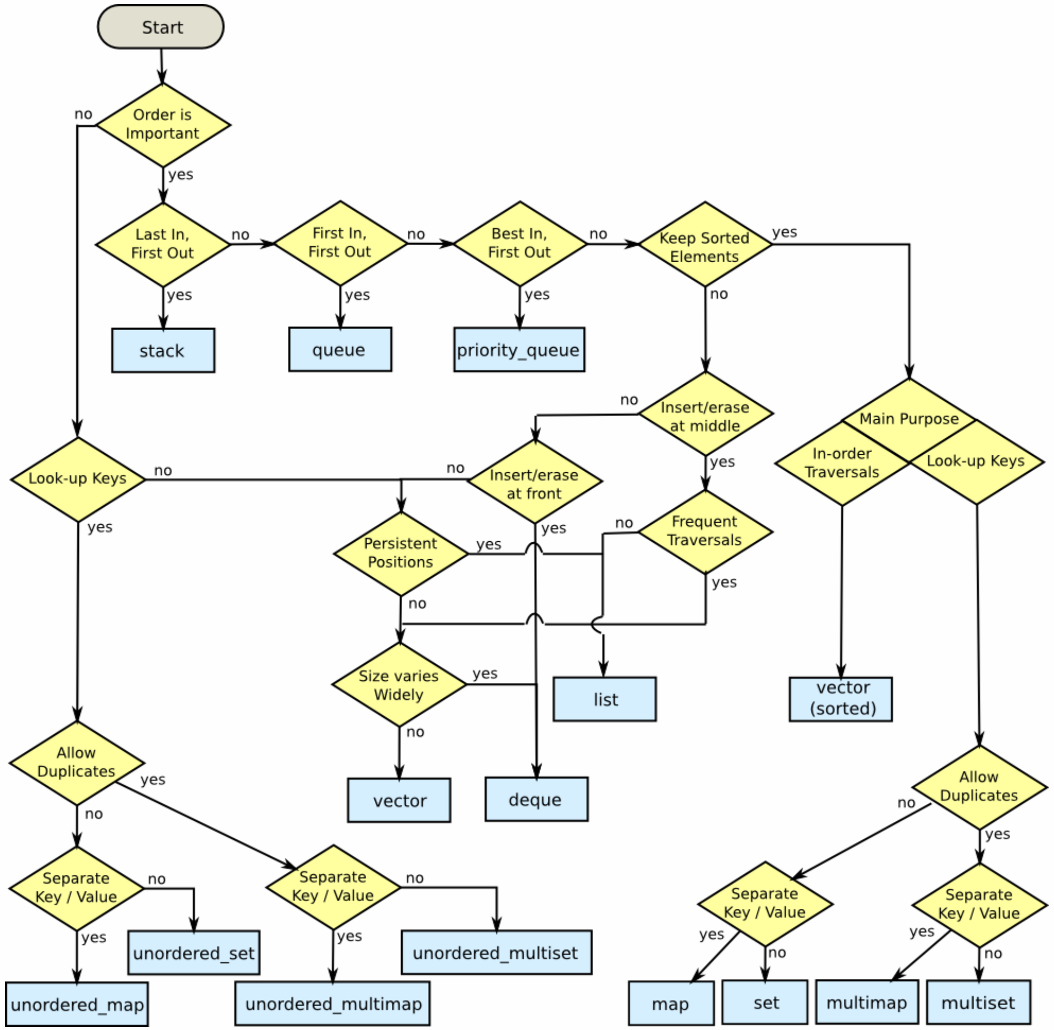
STL

STL Containers

A Cheatsheet for:

- Array(s)
- Vector(s)
- Queue(s)
  Double-ended Queue(s) – Deque(s) …
  Priority Queue(s) …
- Stack(s)
- List(s)
  Doubly-Linked List(s)
  Forward-Linked List(s)

Many more …



STL Iterators

Iteration over STL Container elements.

- Problem: Not all STL classes provide Random Access
  (*Remember*: Like `std::vector<T>::at(size_t pos);` conveniently does.)

Iterators

- Special (Container & Template-Dependent) pointers.
- Enable Iteration through each element in the STL Container.
- Abstraction → The same iteration Interface for any Container.
- Encapsulation → The user shouldn't need to know how it works.

STL Iterators

Access elements in any Data Structure using a unified interface, regardless of the internal details of the DS itself.

Any such Iterator should be able to perform:

- Moving to the container's "beginning" (first element).
- Advancing to the "next" element.
- Returning the "value" it refers (points) to.
- Check if it has reached the container's "end".

STL Iterators

Iterator Operations

a) A STL Vector Container of Type `int`

```
std::vector<int> intVec;
```

b) An Iterator for a STL Vector of Type `int`

```
std::vector<int>::iterator intVec_it;
```

- `begin()` ➡ Returns an iterator (pointing) to first element in Container.

  ```
  intVec_it = intVec.begin();
  ```

- `end()` ➡ Returns an iterator to one element past the last of the Container.

  ```
  intVec_it = intVec.end();
  ```

  Handling of empty range condition: `intVec.begin()==intVec.end()`.

- `operator++(…)` / `operator--(…)` (Post-&-Pre Increment / Decrement)

  `++intVec_it;` / `intVec_it++;`
  `--intVec_it;` / `intVec_it--;`
  ➡ Advances iterator (element it is pointed to) by one, forward or backward.

Note:
Kinds of Iterators include Forward Its (`++` works), Bidirectional Its (`++`/`--` works), Random Access Its (`++`/`--` works).

STL Iterators

Iterator Operations

a) A STL Vector Container of Type `int`

```
std::vector<int> intVec;
```

b) An Iterator for a STL Vector of Type `int`

```
std::vector<int>::iterator intVec_it;
```

➢ Dereferencing an Iterator

```
if (!intVec.empty())
    intVec_it = intVec.begin();
    int intVec_element0 = * intVec_it ;
}
```

Checking with `empty()` is usually faster than checking `size()>0`.

Set Iterator to point to first element in the Container.

Dereferencing returns a Reference-to the Container's element pointed-to by the Iterator.

Note:
Attention, the `end()` Iterator is pointed to one element past the last in the Container.

➢ It should never be Dereferenced:

```
intVec_it = intVec.end();
* intVec_it;
```

Undefined Behavior

STL Iterators

Iterator Operations

Behavior of Dereferencing an Iterator dictates if it is Constant / Mutable.

Mutable Iterator

- Can change corresponding element in Container using a Mutable Iterator.
- Can use `*it` to assign to variable or output, but as well assign to the element in the Container by-Reference (and change it).

Example:

`*it` ⟸ Returns an `lvalue`

`*it` can be on the left-(or right)-hand side of the assignment operator.

STL Iterators

Iterator Operations

Behavior of Dereferencing an Iterator dictates if it is Constant / Mutable.

Constant Iterator

- Cannot change contents of Container using a Constant Iterator.
- Dereferencing (`*`) produces a read-only version of element.
- Can use `*it` to assign to variable or output, but cannot change element in container

Example:

```
*it = <anything>;
```
⟸ Illegal

`*it` can only be on the right-hand side of the assignment operator.

STL Iterators

Iterator Operators

- `*` ⟵ Dereferences the Iterator.

- `++` ⟵ Moves Iterator forward to point to "next" element.
- `--` ⟵ Moves Iterator backward to point to "previous" element.

- `==` ⟵ True if two Iterators point to same element.
- `!=` ⟵ True if two Iterators point to different elements.

- `=` ⟵ Assignment, makes two iterators point to same element.

STL Iterators

Iterator Operations – `std::vector<T>`

Container Traversal w/ Iterator(s)

```
#include <vector>
```

➢ Include appropriate Header(s).

```
std::vector<int> intVec;
intVec.push_back( 1 );
intVec.push_back( 5 );
…
for (std::vector<int>::iterator it = intVec.begin();
                                it != intVec.end();
                                ++it)
{
    std::cout << *it << std::endl;
}
```

Declare Iterator for STL Vector of Type `int`.

➢ Set it to the first element.

➢ Check to see if container end has been reached.

➢ Advance to next element.

➢ Dereference current element.

STL Iterators

Iterator Operations – std::list<T>

Container Traversal w/ Iterator(s)

```
#include <list>
```

➢ Include appropriate Header(s).

```
std::list<int> intList;
intList.push_back( 1 );
intList.push_front( 5 );
…
for (std::list<int>::iterator it = intList.begin();
                              it != intList.end();
                              ++it)
{
    std::cout << *it << std::endl;
}
```

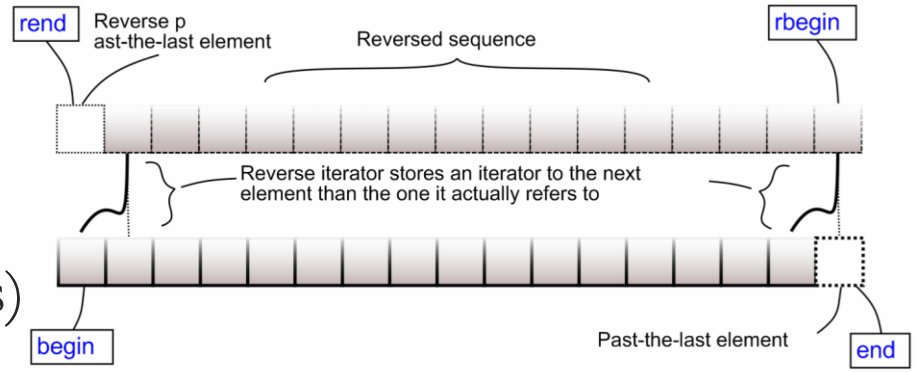
Declare Iterator for STL List of Type **int**.

- ➢ Set it to the first element.
- ➢ Check to see if container end has been reached.
- ➢ Advance to next element.

➢ Dereference current element.

STL Iterators

Reverse-Iterator(s)



Container Reverse-Traversal w/ Reverse-Iterator(s)

a) A STL Vector Container of Type `int`.

```
std::vector<int> intVec;
```

b) A Reverse-Iterator for a STL Vector of Type `int`.

```
std::vector<int>::reverse_iterator intVec_revit;
```

```
for ( intVec_revit  =  intVec.rbegin();
      intVec_revit !=  intVec.rend();
      ++intVec_revit )
{
  std::cout << *intVec_revit << std::endl;
}
```

- Working with Reverse-Iterators requires using `rbegin()` and `rend()`.
- `++` advances Reverse-Iterator reversely (so backwards in the Container order).

STL Iterators

Iterator Examples – `std::set<Key>`

```
int main ( ) {
  set<int> iSet;
  iSet.insert(4);
  iSet.insert(12);
  iSet.insert(7);
  /* the same looping construct for traversal */
  for (set<int>::const_iterator it = iSet.begin(); it != iSet.end(); ++it)
   {
      cout << *it << endl;
   }
   return 0;
}
```

Output:
```
4
7
12
```

A Constant (non-Mutable) Iterator for a STL Set of Type `int`.

STL Iterators

Iterator Examples – std::map<Key,T>

```
int main ( ) {
 map<string, float> stocks;
 stocks.insert( make_pair( string("IBM"),  42.50) );
 stocks.insert( make_pair( string("MS"),   2.50) );
 stocks.insert( make_pair( string("FB"),   0.50) );
 /* the same looping construct for traversal */
 for (map<string,float>::iterator it=stocks.begin(); it!=stocks.end(); ++it)
 {
        cout << it->first << "," << it->second << endl;
 }
 return 0;
}
```

Output:

```
FB,0.5
IBM,42.5
MS,2.5
```

Iterator for STL Map functions like a Pointer to a Pair Struct.

String

The Standard String Class – `std::string` (`std::basic_string<char>`)

Defined in library:

```
#include <string>
/* using namespace std; */
```

Standard String variables and expressions are treated much like simple types.

- Can assign, compare, add:

```
std::string s1("Hello "), s2("World!");    //c-string based String constructor
std::string s3 = s1 + s2;                  //String concatenation
s3 = "Hello Mom!"                          //String assignment
```

Note:
c-string literal `"Hello Mom!"` is automatically converted to Standard String type in assignment.

String

Standard String, by-Example:

```cpp
//Demonstrates the standard class string.
#include <iostream>
#include <string>
using namespace std;

int main( )
{
    string phrase;
    string adjective("fried"), noun("ants");
    string wish = "Bon appetite!";

    phrase = "I love " + adjective + " " + noun + "!";
    cout << phrase << endl
         << wish << endl;

    return 0;
}
```

Initialized to the empty string.

Two equivalent ways of initializing a string variable

Sample Dialogue

I love fried ants!
Bon appetite!

String

Input / Output with Standard String(s)

Treated like other types:

```
std::string s1, s2;
std::cin >> s1;
std::cin >> s2;
```

User input:

```
Today is a beautiful day!
```

Note:
Extraction still ignores whitespace.

Output:

```
std::cout << s1 << " " << s2 ;
```

s1 has received the value **"Today"**

s2 has received the value **"is"**

String

Input / Output with Standard String(s)

Usage with `getline()` for complete lines :

```
std::string line;
std::getline(std::cin, line);
std::cout << line << "END";
```

Similar to a c-string's usage of `getline()`.

Input:

```
How are you? Fine I hope!
```

Output:

```
How are you? Fine I hope!END
```

Can specify Delimiter character :

```
std::string line;
std::getline(std::cin, line, '?');
```

Receives input until `char` `'?'` encountered.

- Does not append Delimiter to String.

Input:

```
How are you? Fine I hope!
```

Output:

```
How are youEND
```

String

Conversion between c-String(s) & Standard String(s)

```
char cString[] = "My C-string";
std::string stringObj;
```

- From c-String to Standard String object:

```
stringObj = cString;
```

Legal, uses String's appropriate Assignment Operator (=) overload.

- From a Standard String object to c-String:

```
cString = stringObj;
```

Illegal

Must use appropriate c-String method:

```
strcpy( cString, stringObj.c_str() );
```

String

Member Functions of the Standard Class string

EXAMPLE	REMARKS
Constructors	
`string str;`	Default constructor; creates empty `string` object `str`.
`string str("string");`	Creates a `string` object with data `"string"`.
`string str(aString);`	Creates a `string` object `str` that is a copy of `aString`. `aString` is an object of the class `string`.
Element access	
`str[i]`	Returns read/write reference to character in `str` at index `i`.
`str.at(i)`	Returns read/write reference to character in `str` at index `i`.
`str.substr(position, length)`	Returns the substring of the calling object starting at `position` and having `length` characters.
Assignment/Modifiers	
`str1 = str2;`	Allocates space and initializes it to `str2`'s data, releases memory allocated for `str1`, and sets `str1`'s size to that of `str2`.
`str1 += str2;`	Character data of `str2` is concatenated to the end of `str1`; the size is set appropriately.
`str.empty( )`	Returns `true` if `str` is an empty `string`; returns `false` otherwise.
`str1 + str2`	Returns a `string` that has `str2`'s data concatenated to the end of `str1`'s data. The size is set appropriately.
`str.insert(pos, str2)`	Inserts `str2` into `str` beginning at position `pos`.
`str.remove(pos, length)`	Removes substring of size `length`, starting at position `pos`.
Comparisons	
`str1 == str2` `str1 != str2`	Compare for equality or inequality; returns a Boolean value.
`str1 < str2` `str1 > str2`	Four comparisons. All are lexicographical comparisons.
`str1 <= str2` `str1 >= str2`	
`str.find(str1)`	Returns index of the first occurrence of `str1` in `str`.
`str.find(str1, pos)`	Returns index of the first occurrence of string `str1` in `str`; the search starts at position `pos`.
`str.find_first_of(str1, pos)`	Returns the index of the first instance in `str` of any character in `str1`, starting the search at position `pos`.
`str.find_first_not_of (str1, pos)`	Returns the index of the first instance in `str` of any character *not* in `str1`, starting search at position `pos`.

CS-202

Time for Questions !