CS 32: Discussion 1D

TA: Shichang Zhang
LA: Stephanie Doan, Rish Jain

Announcements

- The second midterm on Thursday (May 13th), 6:30pm-8:00pm PDT
  - Stacks and queues
  - Inheritance and polymorphism
  - Recursion
  - **NO** templates, STL, big-O, or sorting.
- Project 3 due 11 pm Tuesday (May 18th)
- Homework 4 due 11 pm Tuesday (May 25th)

Overview

- Templates
- STL (Standard Template Library)
  - Vector
  - List
  - Iterator
- Worksheet on templates, STL, and **recursions**

Templates: motivation

- Think about the `Pair` class. The class should not work only with integers. That is we want a “generic” `Pair` class.
- Here we go: `Pair<int> p1; Pair<char> p2;`

```
class Pair {
    public:
        Pair();
        Pair(int firstValue,
             int secondValue);
        void setFirst(int newValue);
        void setSecond(int newValue);
        int getFirst() const;
        int getSecond() const;
    private:
        int m_first;
        int m_second;
};
```

```
template<typename T>
class Pair {
    public:
        Pair();
        Pair(T firstValue,
             T secondValue);
        void setFirst(T newValue);
        void setSecond(T newValue);
        T getFirst() const;
        T getSecond() const;
    private:
        T m_first;
        T m_second;
};
```

Templates: multi-type template

- What if we need pair with different types? (One with `int` value while the other with `string` value)
- Just slightly change your template class and: `Pair<int, string> p1;`

```
template<typename T>
class Pair {
    public:
        Pair();
        Pair(T firstValue,
             T secondValue);
        void setFirst(T newValue);
        void setSecond(T newValue);
        T getFirst() const;
        T getSecond() const;
    private:
        T m_first;
        T m_second;
};
```

```
template<typename T, U>
class Pair {
    public:
        Pair();
        Pair(T firstValue,
             U secondValue);
        void setFirst(T newValue);
        void setSecond(U newValue);
        T getFirst() const;
        U getSecond() const;
    private:
        T m_first;
        U m_second;
};
```

Templates: template specialization

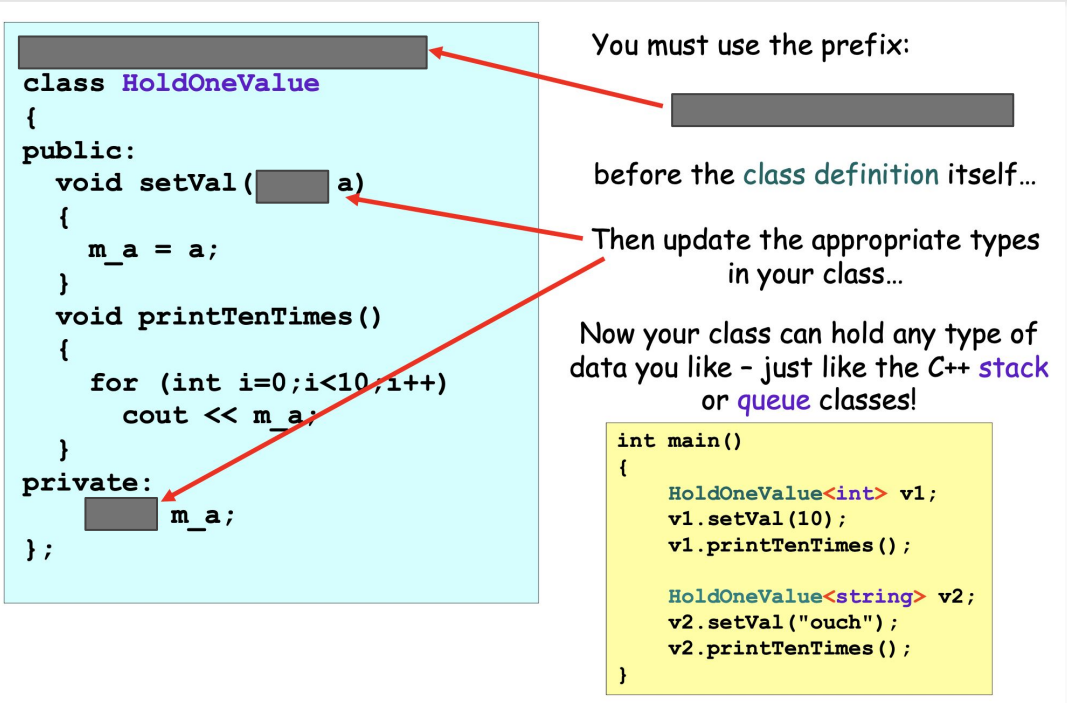
- What if we want a template class with certain data type to have its own exclusive behaviors? For example, in `Pair` class we only allow `Pair<char>` has `uppercase()` and `lowercase()` function but not for `Pair<int>`.

```
template<>
class Pair<char> {
    public:
       Pair();
       Pair(char firstValue,
            char secondValue);
       void setFirst(char newValue);
       void setSecond(char newValue);
       char getFirst() const;
       char getSecond() const;
       void uppercase();
    private:
       char m_first;
       char m_second;
};
```

```
Pair<int> p1;
Pair<char> p2;

p1.uppercase(); //error
p2.uppercase(); //correct
```

Example: a template class



Example: a template class

```
template <typename Item>
class HoldOneValue
{
public:
  void setVal(Item a)
  {
    m_a = a;
  }
  void printTenTimes()
  {
    for (int i=0;i<10;i++)
      cout << m_a;
  }
private:
    Item m_a;
};
```

You must use the prefix:

```
template <typename xxx>
```

before the class definition itself...

Then update the appropriate types in your class...

Now your class can hold any type of data you like - just like the C++ stack or queue classes!

```
int main()
{
    HoldOneValue<int> v1;
    v1.setVal(10);
    v1.printTenTimes();

    HoldOneValue<string> v2;
    v2.setVal("ouch");
    v2.printTenTimes();
}
```

Standard Template Library (STL)

- A collection of pre-written, tested classes provided by C++.
- All built using templates (adaptive with many data types).
- Provide useful data structures
  - `vector(array), set, list, map, stack, queue`
- Standard functions:
  - Common ones: `.size(), .empty()`
  - For a container that is neither stack or queue: `.insert(), .erase(), swap(), .clear()`
  - For list or vector: `.push_back(), .pop_back()`
  - For set or map: `.find(), .count()`
  - More on stacks and queues...

STL: Vector

- Works exactly like array but doesn't have fixed size
  - Grows and shrinks when adding and removing items
  - Based on dynamic arrays placed in contiguous storage
  - Fast on access but slow on insert/delete
- Some methods
  - Add item with `push_back`
  - Remove item from back of vector with `pop_back`
  - Access elements with brackets `v[0]`
  - Get size of vector with `.size()`
- Link to Reference

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

void print_vector(vector<int> v){
 for (int i=0; i<v.size(); i++) {
   cout << v[i] << " ";
 }
 cout << endl;
}

int main() {
  vector<int> vals{4,5,6};
  cout << "size: " << vals.size() << endl;  // prints ??

  vals.push_back(123);
  print_vector(vals);                        // prints ??

  vals.pop_back();
  vals[0] = 1;
  print_vector(vals);                        // prints ??
}
```

STL: Vector

- Works exactly like array but doesn't have fixed size
  - Grows and shrinks when adding and removing items
  - Based on dynamic arrays placed in contiguous storage
  - Fast on access but slow on insert/delete
- Some methods
  - Add item with `push_back`
  - Remove item from back of vector with `pop_back`
  - Access elements with brackets `v[0]`
  - Get size of vector with `.size()`
- Link to Reference

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

void print_vector(vector<int> v){
 for (int i=0; i<v.size(); i++) {
   cout << v[i] << " ";
 }
 cout << endl;
}

int main() {
  vector<int> vals{4,5,6};
  cout << "size: " << vals.size() << endl;   // size: 3
  vals.push_back(123);
  print_vector(vals);                        // 4 5 6 123

  vals.pop_back();
  vals[0] = 1;
  print_vector(vals);                        // 1 5 6
}
```

STL: List

- Lists are a linked list
  - Offer fast insertion/deletion
  - Slow to access Elements
  - Cannot access list elements by brackets
- Some popular methods
  - insert()
  - erase()
  - size()
  - push_front() & pop_front()
    - vectors don't have these two methods
- Link to Reference

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

void print_list(list<int> v); //Assume we give you one

int main() {
 int a[3] = {4,5,6};
 list<int> vals(a, a+3);
 cout << "size: " << vals.size() << endl;  // prints ??
 vals.push_front(123);
 print_list(vals);                          // prints ??
 list<int>::iterator it = vals.begin();
 for (it = vals.begin(); it != vals.end(); it++) {
  if (*it == 4)
   break;
}
 vals.erase(it);
 print_list(vals);                          // prints ??
}
```

STL: List

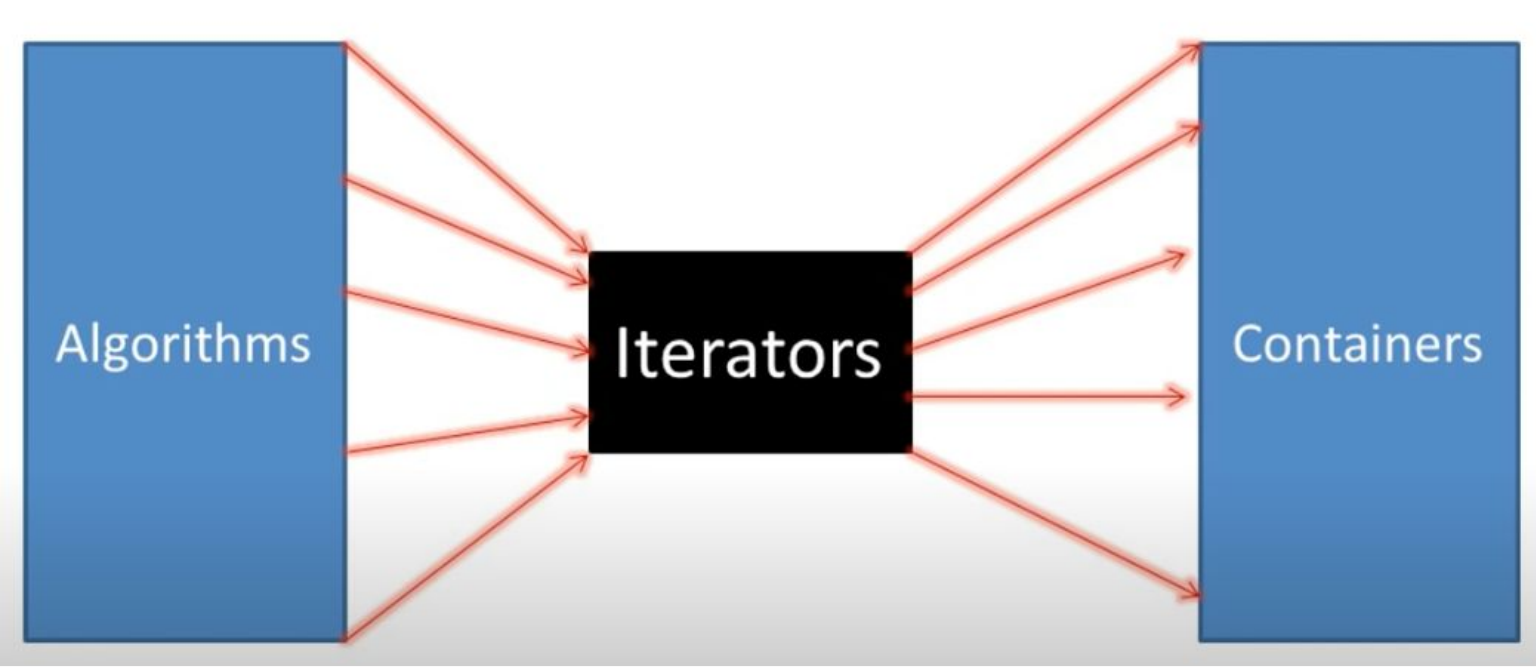
- Lists are a linked list
  - Offer fast insertion/deletion
  - Slow to access Elements
  - Cannot access list elements by brackets
- Some popular methods
  - insert()
  - erase()
  - size()
  - push_front() & pop_front()
    - vectors don't have these two methods
- Link to Reference

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

void print_list(list<int> v); //Assume we give you one

int main() {
 int a[3] = {4,5,6};
 list<int> vals(a, a+3);
 cout << "size: " << vals.size() << endl;  // prints size: 3
 vals.push_front(123);
 print_list(vals);                          // prints 123 4 5 6
 list<int>::iterator it = vals.begin();
 for (it = vals.begin(); it != vals.end(); it++) {
  if (*it == 4)
   break;
}
 vals.erase(it);
 print_list(vals);                          // prints 123 5 6
}
```

STL: iterator



Example: iterators

- STL Iterators: Use `.begin()` and `.end()`
  - `.begin()` : return an iterator that points to the first element.
  - `.end()`: return an iterator that points to the ***past-the- last*** element.

```
// First Example:
using namespace std;

vector<int> vec;
vec.push_back(4);
vec.push_back(1);
vec.push_back(8);  // vec: {4, 1, 8}

vector<int>::iterator itr1 = vec.begin();  // half-open:  [begin, end)
vector<int>::iterator itr2 = vec.end();

for (vector<int>::iterator itr = itr1; itr!=itr2; ++itr)
   cout << *itr << " ";  // Print out:  4 1 8
```

Iterator Cheat Sheet (might help with proj 3)

Iterator invalidation

Read-only methods never invalidate iterators or references. Methods which modify the contents of a container may invalidate iterators and/or references, as summarized in this table.

Category	Container	After **insertion**, are...		After **erasure**, are...		Conditionally
		iterators valid?	**references** valid?	**iterators** valid?	**references** valid?	
Sequence containers	`array`	N/A		N/A		
	`vector`	No		N/A		Insertion changed capacity
		Yes		Yes		Before modified element(s)
		No		No		At or after modified element(s)
	`deque`	No	Yes	Yes, except erased element(s)		Modified first or last element
			No	No		Modified middle only
	`list`	Yes		Yes, except erased element(s)		
	`forward_list`	Yes		Yes, except erased element(s)		
Associative containers	`set` `multiset` `map` `multimap`	Yes		Yes, except erased element(s)		
Unordered associative containers	`unordered_set` `unordered_multiset` `unordered_map` `unordered_multimap`	No	Yes	N/A		Insertion caused rehash
		Yes		Yes, except erased element(s)		No rehash

Here, **insertion** refers to any method which adds one or more elements to the container and **erasure** refers to any method which removes one or more elements from the container.

C++ in real life

```
auto it = find(v.begin(),v.end(),x);
```

auto is a keyword that can be used to create an iterator automatically

- Certain Internships might not love it since you can often have issues due to this and you don't know exactly what it refers to
  - You may be told to use iterators (T-T)
- You could theoretically use it for Project 3

C++ in real life

Advanced For Loops much make every for loop look nice.

```
int arr[] = {1,2,3,4,4};
set<int> s(arr, arr+5);

//yucky
for(set<int>::iterator it = s.begin(); it != s.end(); it++) {
     cout << (*it) << endl;
}

// pretty
 for(auto e: s) {
     cout << e << endl;
}
```

Break: 5 mins

Worksheet

Codeshare

Room 1

Room 2

Room 3

Room 4

Worksheet Solution