EE538: Computing Principles for Electrical Engineers

Discussion 5: C++ Language

University of Southern California
06/15,06/16 Summer 2022
Instructor: Arash Saifhashemi
TA and Mentors: Yijun Liu, Aditi Bodhankar, Zixu Wang

Outline

- **Class**
- Const
- Set vs unordered_set
- Map vs unordered_map
- Complexity of map
  - Map
  - Unordered_Map
- Iterator
- Algorithms

Main Parts of a Class

- Public and private
  - Default is private
    - Only class methods can access
  - Must explicitly declare the public
- Be used just like other data types
  - Get pointers/references to them
  - Pass/Return them to/ from functions
  - Dynamically allocate them

```
class Student1{
    std::string _name;
    int _age;
    Student1(){
        _name = "UNKNOW";
        _age = -1;
    }
};


int main()
{
    Student1 s1;
    std::cout << s1._age;
```

```
main.cpp:49:14: error: 'Student1::Student1()' is private within this context
     Student1 s1;
main.cpp:50:21: error: 'int Student1::_age' is private within this context
     std::cout << s1._age;
```

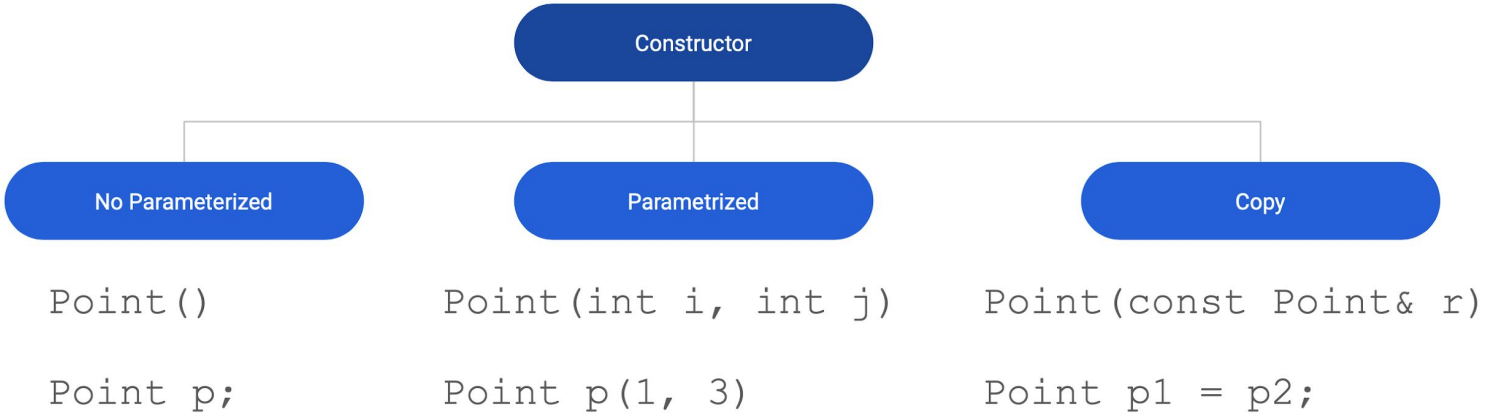
Main Parts of a Class

- Member variables
  - What data must be stored?
- Constructor(s)
  - How do you initialize an instance?
- Member functions
  - How does user need to interact
  - with the stored data?
- Destructor
  - How do you clean up an instance?

~ s1

```
class Student{
private:
    std::string _name;
    int _age;
public:
    Student(){
        _name = "UNKNOW";
        _age = -1;
    }
    Student(const std::string& name, int age){
        _name = name;
        _age = age;
    }
    int getAge(){
        return _age;
    }
    void printInfo(){
        std::cout << _name << std::endl;
        std::cout << _age << std::endl;
    }
};
```

Constructor



Default Constructor

- Any user defined class will have a “default constructor” which does nothing.
  - These two writings are the same.

```
class Student1 {
  public:
    int id;
};

class Student2 {
  public:
    int id;
    Student2(){}
};
```

Default Constructor

- But usually, you need to initialize your class.
  - Initializer list

```
class Student1 {
  public:
    int id;
    Student1(){
      id = 0;
    }
};

class Student2 {
  public:
    int id;
    Student2():id(0){}
};
```

Default Constructor

- Get random number if we use default constructor to initialize the class.

```
1851052224
32767
100
10
```

```
class Point{
public:
    int i;
    int j;
};

class Point1{
public:
    int i;
    int j;
    Point1() {
        i = 100;
        j = 10;
    };
};
```

Parameterized Constructor

- If you declared any constructor function, the “default constructor” will be disabled.
  - Compile error.

```
class Student3 {
   public:
    int id;
    Student3(int id_) : id(id_) {}
};
```

```
int main() {
    Student1 s1;
    Student2 s2;
    printf("s1: %d\n", s1.id);
    printf("s2: %d\n", s2.id);

    Student3 s3;

    return 0;
}
```

Parameterized Constructor

- Usage of default parameter.
  - Use multiple explicit parameterized constructor, e.g. Student4, is usually recommended.
  - Student5 is also okay when it is simple enough.

```
class Student4 {
    public:
     int id;
     Student4() : id(0) {}
     Student4(int id_) : id(id_) {}
};

class Student5 {
    public:
     int id;
     Student5(int id_ = 0) : id(id_) {}
};
```

Outline

- Class
- **Const**
- set vs unordered_set
- Map vs unordered_map
- Complexity of map
  - Map
  - Unordered_Map
- Iterator
- Algorithms

const keyword in C++

const can be used in following contexts in a C++ program –

1. Variables
2. Pointers
3. Function arguments and return types
4. Class Data members
5. Class Member functions
6. Objects

const variable

- Cannot change value
- Must be initialized while declared

```
int main
{
    const int i = 10;
    const int j = i + 10;     // works fine
    i++;    // this leads to Compile time error
}
```

Why const?

- Prevent **unintentional changes** that might cause bugs.
- Easier to **reason about our code**.

const function arguments

```
void t(int*)
{
    // function logic
}
```

What happens if we pass const **int*** to this function?

Error – Cannot pass a const argument

```
void g(const int*)
{
    // function logic
}
```

This function can have a **int*** and **const int*** type argument.

Const in Function Parameters

Data Type	**Feature**
Pass by Value	• Copying, so original is protected • But, copying can have high cost
Pass by Reference	• No copying • Original might be changed (can be good or bad) • No copy overhead
Pass by **Const reference**	• No copying • Original cannot changed

const Member Functions in C++

- What is the motivation behind const functions?
  - Prevent functions from modifying the object on which they are called.

  - When a function is declared as const, it can be called on any type of object, const object as well as non-const objects.

  - Non-const functions can only be called by non-const objects.

What is the Output?

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

class Test {
    int value;
public:
    Test(int v = 0) {value = v;}
    int getValue() const { return value;}
};

int main() {
    Test t(20);
    cout<<t.getValue();
    return 0;
}
```

Output: 20

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

class Test {
    int value;
public:
    Test(int v = 0) {value = v;}
    int getValue() const { value = 2000; return value;}
};

int main() {
    Test t(20);
    cout<<t.getValue();
    return 0;
}
```

Output: Compiler Error

What is the Output?

```
#include <iostream>
using namespace std;
class Point
{
    int x, y;
public:
 Point(int i = 0, int j =0)
   { x = i; y = j;  }
   int getX() const { return x; }
   int getY() {return y;}
};

int main()
{
    const Point t;
    cout << t.getX() << " ";
    cout << t.getY();
    return 0;
}
```

(A) Garbage Values
(B) 0 0
(C) Compiler Error in line cout << t.getX() << " ";
(D) Compiler Error in line cout << t.getY();

Answer:
(D) A const object can only call const functions.

Outline

- Class
- Const
- **STL data structures**
- Complexity of different data structures
  - Map
  - Unordered_Map
- Iterator
- Algorithms

std::set

- Store a set of elements
- Important methods to know
  - size()
  - insert()
  - count()
  - find()
- Things to know about std::set:
  - Internally it is sorted based on keys
  - Access, Insert, and find complexity is O(log(n))
  - Reinserting the same key will just update the data, there is no duplicate keys

std::set vs std::unordered_set

- Used for keeping a list of unique items
  - A set is really the list of keys in a map
- Both provide similar APIs

std::set	std::unordered_set
Internally sorted	Not sorted
Implemented using balanced trees (red-black trees)	Implemented using a hash table
Search, removal, and **insertion** operations have logarithmic complexity: **O(log n)**	**Search, insertion**, and **removal** of elements have average time of **O(1)**, but worst case can be **O(n)**

std::pair

- std::pair
  - Couples together a pair of things (of type T1 to T2)
  - For a pair of items p:
    - Access the first item by p.first
    - Access the second item by p.second

```
std::pair<std::string, int> p1("Ari", 3);
std::pair<std::string, int> p2("Ted", 4);



std::cout << "p1.first: " << p1.first << std::endl;
std::cout << "p1.second: " << p1.second << std::endl;
```

std::map

- Associative Array or Dictionary
- A Collection of **pairs** (key, value)
- Internally it is **sorted** based on keys
- Accessing a non-existent key using [ ], creates that key
- No duplicate keys
- Methods
  - Operator[ ]
  - insert()
  - erase()
  - size()
  - find()

Guess the complexity of each function! Note that std::map is sorted! A binary search tree(BST) is used here!

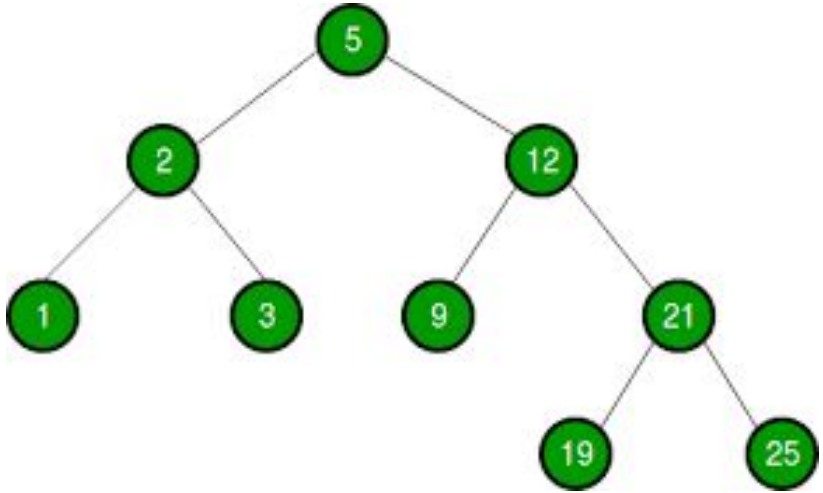
std::map vs std::unordered_map

- Similar functionality to std::map
- Used for mapping unique Keys to Values.
  - Example: Mapping SSN to Person
  - Example: Count the number of each word in a book: Map of words to numbers.
- Both provide similar APIs

std::map	std::unordered_map
Internally sorted	Not sorted
Implemented using balanced trees (red-black trees)	Implemented using a hash table
Search, removal, and **insertion** operations have logarithmic complexity: **O(log n)**	**Search, insertion**, and **removal** of elements have average time of **O(1)**, but worst case can be **O(n)**

std::map

- Methods
  - Operator[ ] O(logn)
  - insert() O(logn)
  - erase() O(logn)
  - size() O(1)
  - find() O(logn)



log n

https://www.cplusplus.com/reference/map/map/

std::unordered_map

- Methods
  - Operator[ ]
  - insert()
  - erase()
  - size()
  - find()

Guess the complexity of each function! Note that std::unordered_map is not sorted!
std::unordered_map is generally faster than std::map!
Hash map is used here.

std::unordered_map

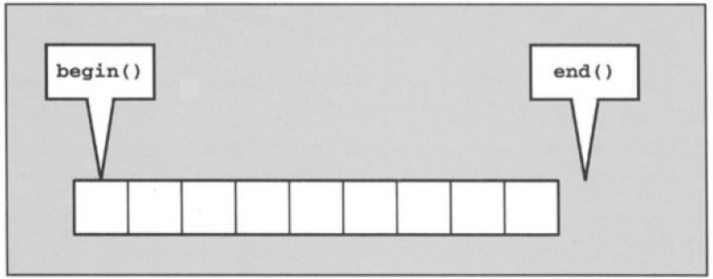
- Methods
  - Operator[ ] Average case: constant. Worst case: linear in container size.
  - insert() Average case: constant. Worst case: linear in container size.
  - erase() Average case: constant. Worst case: linear in container size.
  - size() O(1) Constant
  - find() Average case: constant. Worst case: linear in container size.

https://www.cplusplus.com/reference/unordered_map/

Outline

- Class
- Const
- STL data structures
- Complexity of map
  - Map
  - Unordered_Map
- **Iterator**
- Algorithms

Iterator




```
std::vector<int> v = {1, 2, 3, 4};
std::vector<int>::iterator it = v.begin();
int n1 = *it;                          // 1  *it=1
int n2 = *(it++);                      // 1  *it=2  n2 = *it; it = it + 1;
int n3 = *(it + 2);                    // 4  *it=2
int n4 = *(--it);                      // 1  *it=1 it = it - 1; n4 = *it;
int n5 = it - v.begin();               // 0  *it=1
auto it2 = next(it);                   // i.e. it2=it+1
auto it3 = prev(it2);                  // i.e. it3=it2-1

for (auto it4 = v.begin(); it4 != v.end(); it4++) {
  cout << *it4 << " ";
}
```

Note: Once you insert/erase an element of a vector, **all iterators** to that vector become invalid.

```cpp
#include <iostream>
#include <string>
#include <vector>
int main()
{

  std::vector<int> vec = {1,2,3,4,5};
  std::vector<int>::iterator it=vec.begin();
  vec.push_back(100);
  std::cout << *(it) << std::endl;
  std::cout << *(it+1) << std::endl;
  std::cout << *(it+2) << std::endl;
  std::cout << *(it+3) << std::endl;
  std::cout << *(it+4) << std::endl;
  std::cout << *(it+5) << std::endl;
  std::cout << *(it+6) << std::endl;
  return 0;
}
```

😔

```
0
0
3
4
5
0
49
```

```cpp
// Example program
#include <iostream>
#include <string>
#include <vector>
int main()
{

  std::vector<int> vec = {1,2,3,4,5};
  vec.push_back(100);
  std::vector<int>::iterator it=vec.begin();
  std::cout << *(it) << std::endl;
  std::cout << *(it+1) << std::endl;
  std::cout << *(it+2) << std::endl;
  std::cout << *(it+3) << std::endl;
  std::cout << *(it+4) << std::endl;
  std::cout << *(it+5) << std::endl;
  return 0;
}
```

😊

```
1
2
3
4
5
100
```

Outline

- Class
- Const
- STL Data structures
- Complexity of map
  - Map
  - Unordered_Map
- Iterator
- **Algorithms**

STL algorithms: sort()

- sort(it_first, it_last, compare_rule)
- It sorts the given list in **ascending** order by default
- The order of equal elements is not guaranteed to be preserved. (**not stable**)

```cpp
// Example program
#include <iostream>
#include <array>
#include <algorithm>
int main()
{
    std::array<int, 10> s = {5, 7, 4, 2, 2, 6, 6, 9, 0, 3};

    // sort using the default operator<
    std::sort(s.begin(), s.end());
    for (auto a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';

    // sort using a standard library compare function object
    std::sort(s.begin(), s.end(), std::greater<int>());
    for (auto a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';
}
```

options | compilation | execution

```
0 2 2 3 4 5 6 6 7 9
9 7 6 6 5 4 3 2 2 0
```

STL algorithms: reverse()

- void reverse(it first, it last)
- Reverses the order of the elements in the range [first,last).

```cpp
// reverse algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::reverse
#include <vector>        // std::vector

int main () {
  std::vector<int> myvector;

  // set some values:
  for (int i=1; i<10; ++i) myvector.push_back(i);   // 1 2 3 4 5 6 7 8 9

  std::reverse(myvector.begin(),myvector.end()-1);
  // print out content:
  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

```
myvector contains: 8 7 6 5 4 3 2 1 9
```

STL algorithms: find()

```
find(it_beg, it_end, val);
```

```
string s = "Happy Valentine's Day";

auto it1 = find(s.begin(), s.end(), 'H');
auto it2 = find(s.begin(), s.end(), 'p');
auto it3 = find(s.begin(), s.end(), 'h');
```

To which char does the iterator point?

STL algorithms: find()

```
find(it_src_beg, it_src_end, val);
```

```
string s = "Happy Valentine's Day";
             ^
auto it1 = find(s.begin(), s.end(), 'H');
auto it2 = find(s.begin(), s.end(), 'p');
auto it3 = find(s.begin(), s.end(), 'h');
```

To which char does the iterator point?

STL algorithms: find()

```
find(it_src_beg, it_src_end, val);
```

```
string s = "Happy Valentine's Day";
               ^
```

return iterator pointing to the first element found

```
auto it1 = find(s.begin(), s.end(), 'H');
auto it2 = find(s.begin(), s.end(), 'p');
auto it3 = find(s.begin(), s.end(), 'h');
```

To which char does the iterator point?

STL algorithms: find()

```
find(it_src_beg, it_src_end, val);
```

```
string s = "Happy Valentine's Day";
```

^ return iterator pointing to the next pos of the end

i.e. s.end()

```
auto it1 = find(s.begin(), s.end(), 'H');
auto it2 = find(s.begin(), s.end(), 'p');
auto it3 = find(s.begin(), s.end(), 'h');
```

To which char does the iterator point?

STL algorithms: count() & count_if()

You will learn lambda expressions in later lectures

```
count(it_beg, it_end, val);
```

```
string s = "Happy Valentine's Day";
int n1 = count(s.begin(), s.end(), 'i');
int n2 = count_if(s.begin(), s.end(), [](char c) -> bool
                  { return c >= 'A' && c <= 'Z'; });
```

How many 'i' s do we have? n1: 1

How many capital letters do we have? n2: 3

STL algorithms: copy()

```
copy(it_src_beg, it_src_end, it_dst_beg);
```

always make sure dst has the same size of src

```
int myints[] = {1, 2, 3, 4};
std::vector<int> myvector  ;
std::copy(myints, myints + 4, myvector.begin());
```

What is myvector right now? **Error**

STL algorithms: copy() & copy_if()

```
copy(it_src_beg, it_src_end, it_dst_beg);
```

always make sure dst has the same size of src

```
int myints[] = {1, 2, 3, 4};
std::vector<int> myvector (4); // indicate the size
std::copy(myints, myints + 4, myvector.begin());
```

What is myvector right now? 1, 2, 3, 4

Try copy_if() after class, following the usage of count_if()

STL algorithms: accumulate()

```
accumulate(it_beg, it_end, initial_val)


vector<int> v = {10,20,30};
std::cout << std::accumulate(v.begin(), v.end(), 0) << std::endl;       // 60
std::cout << std::accumulate(v.begin(), v.begin()+1 ,0) << std::endl;   // 10
std::cout << std::accumulate(v.begin(), v.begin()+2 ,0) << std::endl;   // 30
std::cout << std::accumulate(v.begin(), v.end(), 1) << std::endl;       // 61
```

STL algorithms: transform()

You will learn functors and lambda expressions in later lectures

When the functor / lambda expression is unary

```
// dst.resize(src.size()); always make sure dst has the same size of src
transform(it_src_beg, it_src_end, it_dst_beg, functor);
```

When the functor / lambda expression is binary

```
// if(src1.size()==src2.size()) always make sure dst has the same size of src1, src2
transform(it_src1_beg, it_src1_end, it_src2_beg, it_dst_beg, functor);
```

STL algorithms: transform()

You will learn functors and lambda expressions in later lectures

E.g.

```
char MyToLower(char c) { return tolower(c); }

int main() {
  string s = "Discussion 5 of EE538";
  transform(s.begin(), s.end(), s.begin(), ::tolower);
  cout << s << endl;                          // "discussion 5 of ee538"
  transform(s.begin(), s.end(), s.begin(), [](char c) -> char
            { return toupper(c); });
  cout << s << endl;                          // "DISCUSSION 5 OF EE538"
  transform(s.begin(), s.end(), s.begin(), MyToLower);
  cout << s << endl;                          // "discussion 5 of ee538"
  return 0;
}
```

Practice Q1 two sum

Given two inputs:

1. an **vector** containing several integers.

2. A specific **integer** n.

Try to return a vector with the **indices of** two integers (a,b) that the **summation** of them equals ‘n’.

(You may assume that each input would have exactly one solution)

- If there’s no answer, you may return an empty vector.
- Try to analyze time complexity.

Examples:

Input = {4, 2, 3, 1, 5}, n = 8; output = {2, 4}

Input = {1, 2, 3, 4, 5}, n = 10; output = {}

Solution to Practice Q1

```
std::vector<int> twoSum(std::vector<int>  &input, int n) {
      unordered_map<int, int> hashtable;
       for (int i = 0; i < input.size(); ++i) {
                  auto it = hashtable.find(n - input[i]);
                  if (it != hashtable.end()) {
                        return {it->second, i};
                  }
                  hashtable[input[i]] = i;
        }
        return {};
}
```

Practice Q2 two sum - sorted input

Given two inputs:

1. A **sorted vector** containing several integers.

2. A specific **integer** n.

Try to return a vector with two integers (a,b) that the **summation** of them equals ‘n’.

(You may assume that each input would have exactly one solution)

- If there’s no answer, you may return an empty vector.
- Try to analyze time complexity.

Examples:

Input = {1, 2, 3, 4, 5}, n = 8; output = {3, 5}

Input = {1, 2, 3, 4, 5}, n = 10; output = {}

Solution to Practice Q2

```
std::vector<int> twoSum2(std::vector<int>  &input, int n) {
    std::vector<int>::iterator start = input.begin();
    std::vector<int>::iterator end = input.end()-1;
    int sum = 0;
    while(start < end) {
        sum = *start + *end;
        if(sum == n) {
            return {*start, *end}
        }
        else if(sum < n){
           ++start;
         } else {
            --end;
         }
        }
        return {};
    }
```