STL Associative Containers: map, unordered map

VGP131 – OOP II

Instructor: Ivaldo Tributino

STL Associative Containers

Divided in two ways:

Keys are unique:

- set is a collection of unique keys, sorted by keys
- map is a collection of key-value pairs, sorted by keys

Multiple entries:

- multiset is a collection of keys, sorted by keys
- multimap is a collection of key-value pairs, sorted by keys [2]

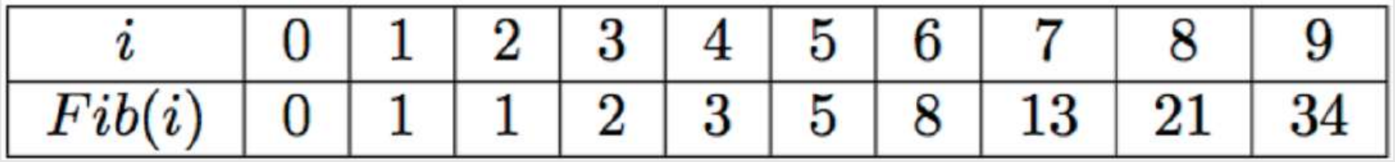
The **associative containers** are similar to the **unordered associative** containers in C++ standard library, the only difference is that the unordered associative containers, as their name implies, do not order their elements. Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched (O(1) amortized, O(n) worst-case complexity).

Map in C++ Standard Template Library (STL)

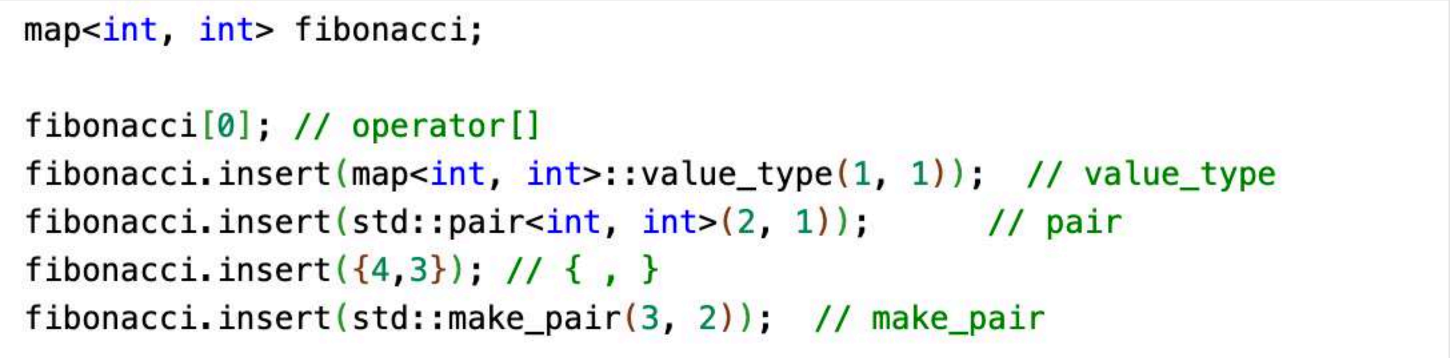
Each element has a key value and a mapped value. No two mapped values can have same key values. Some basic functions associated with Map:[4]

Expression	Description
map.begin()	Returns an iterator to the first element in the map
map.end()	Returns an iterator to the theoretical element that follows last element in the map.
map.size()	Returns the number of elements in the map
map.max_size()	Returns the maximum number of elements that the map can hold
map.erase(key)	Removes the key value from the map
map.clear()	Removes all the elements from the map

#include map<T>
std::map::insert



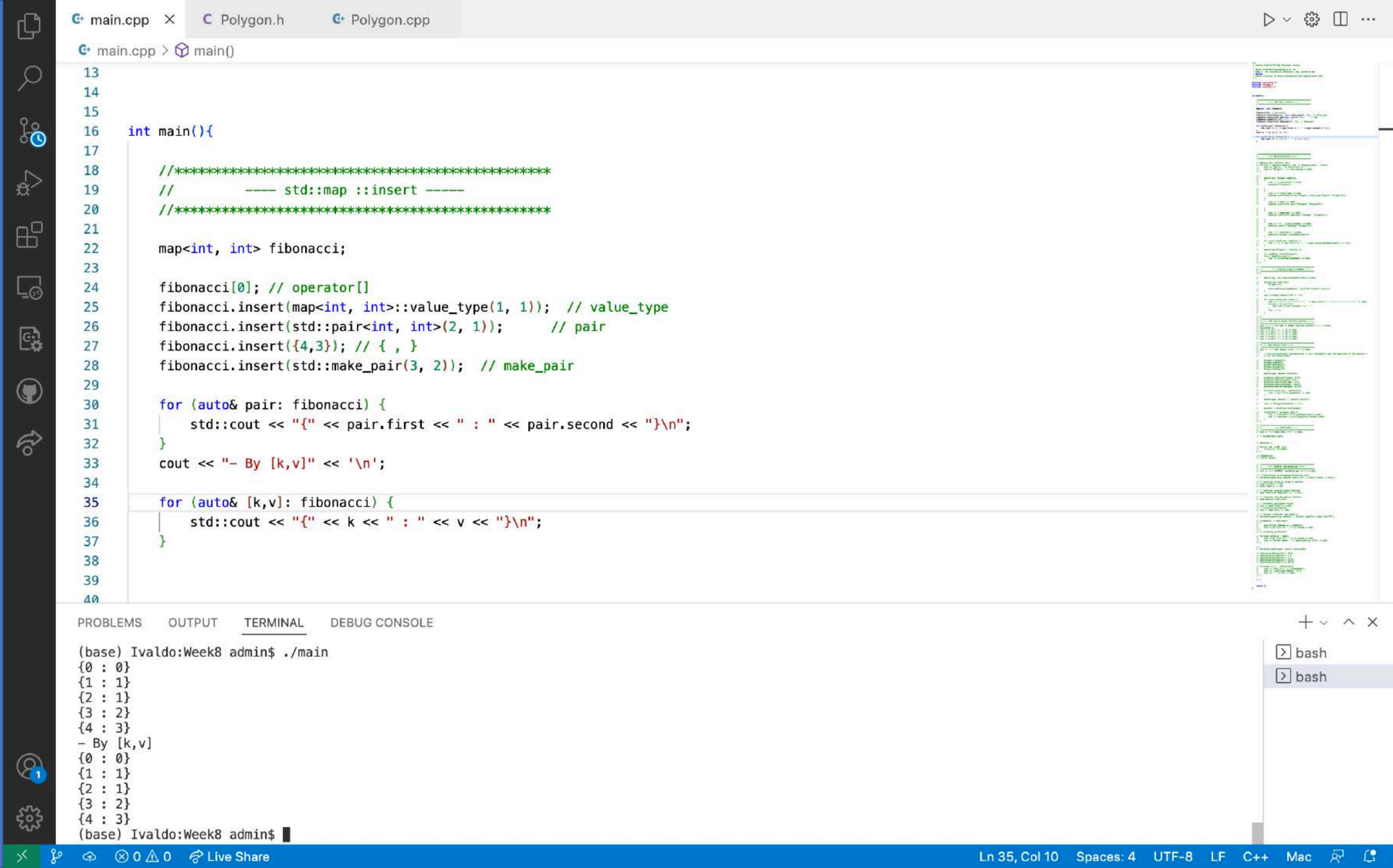
i	0	1	2	3	4	5	6	7	8	9
$Fib(i)$	0	1	1	2	3	5	8	13	21	34



```
map<int, int> fibonacci;

fibonacci[0]; // operator[]
fibonacci.insert(map<int, int>::value_type(1, 1));  // value_type
fibonacci.insert(std::pair<int, int>(2, 1));      // pair
fibonacci.insert({4,3}); // { , }
fibonacci.insert(std::make_pair(3, 2));  // make_pair
```

- **map::insert()** function is an inbuilt function in C++ STL, which is defined in header file.
- **insert()** is used to insert new values to the map container and increases the size of container by the number of elements inserted.
- The map container maintains all their elements via their respective key in the ascending order. So, whenever we insert an element it goes to its respective position according to its key.

Map insert



```cpp



int main(){

    //**********************************************
    //          ---- std::map ::insert -----
    //**********************************************

    map<int, int> fibonacci;

    fibonacci[0]; // operator[]
    fibonacci.insert(map<int, int>::value_type(1, 1));  // value_type
    fibonacci.insert(std::pair<int, int>(2, 1));      // pair
    fibonacci.insert({4,3}); // { , }
    fibonacci.insert(std::make_pair(3, 2));  // make_pair

    for (auto& pair: fibonacci) {
        std::cout << "{" << pair.first << " : " << pair.second << "}\n";
    }
    cout << "- By [k,v]" << '\n';

    for (auto& [k,v]: fibonacci) {
        std::cout << "{" << k << " : " << v << "}\n";
    }


```

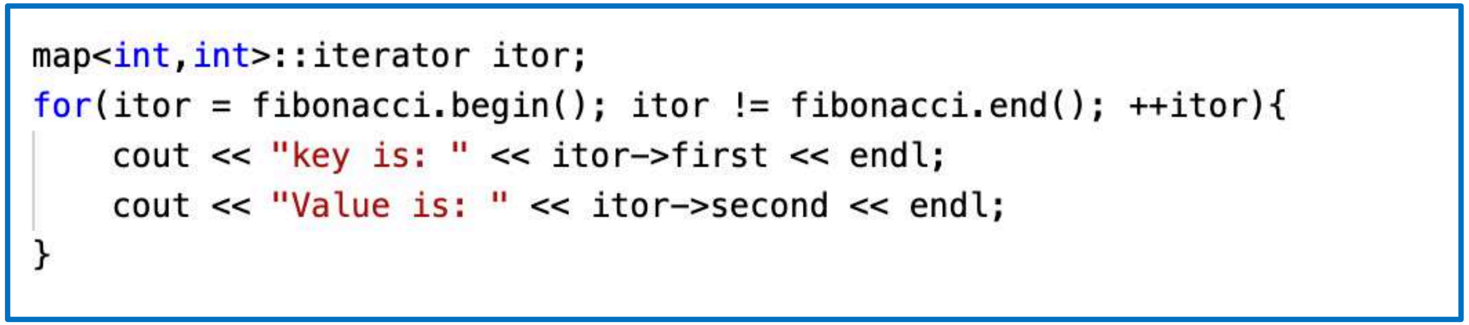
```
(base) Ivaldo:Week8 admin$ ./main
{0 : 0}
{1 : 1}
{2 : 1}
{3 : 2}
{4 : 3}
- By [k,v]
{0 : 0}
{1 : 1}
{2 : 1}
{3 : 2}
{4 : 3}
(base) Ivaldo:Week8 admin$
```

Maps & Iterators

If you still want to process each element in a map you can still iterate over all elements in the map using an iterator object for that map type.

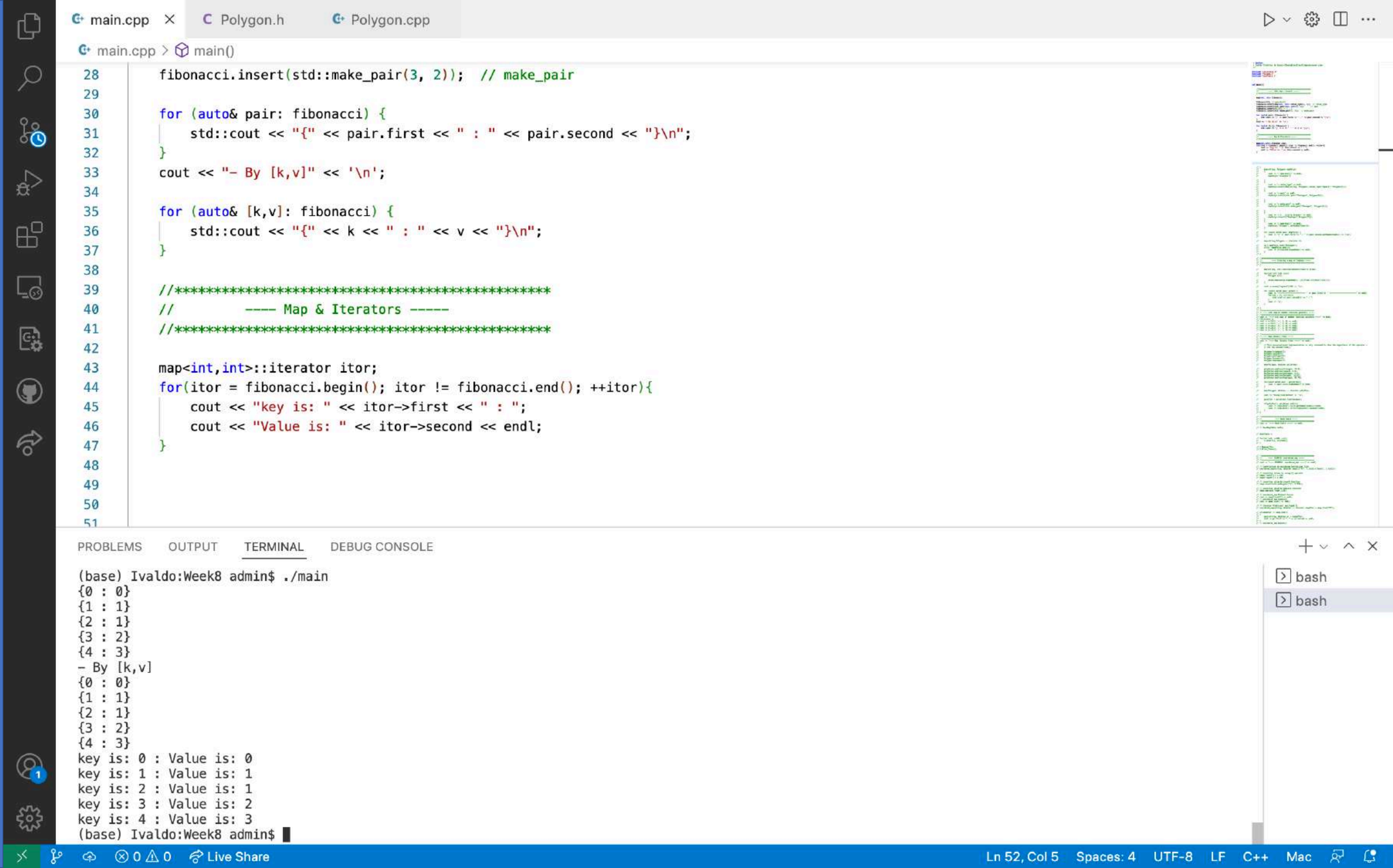
Recall: Iterator is a “pointer”/iterator to a pair struct

- it->first is the key
- it->second is the value [3]



```
map<int,int>::iterator itor;
for(itor = fibonacci.begin(); itor != fibonacci.end(); ++itor){
    cout << "key is: " << itor->first << endl;
    cout << "Value is: " << itor->second << endl;
}
```

Map & Iterator



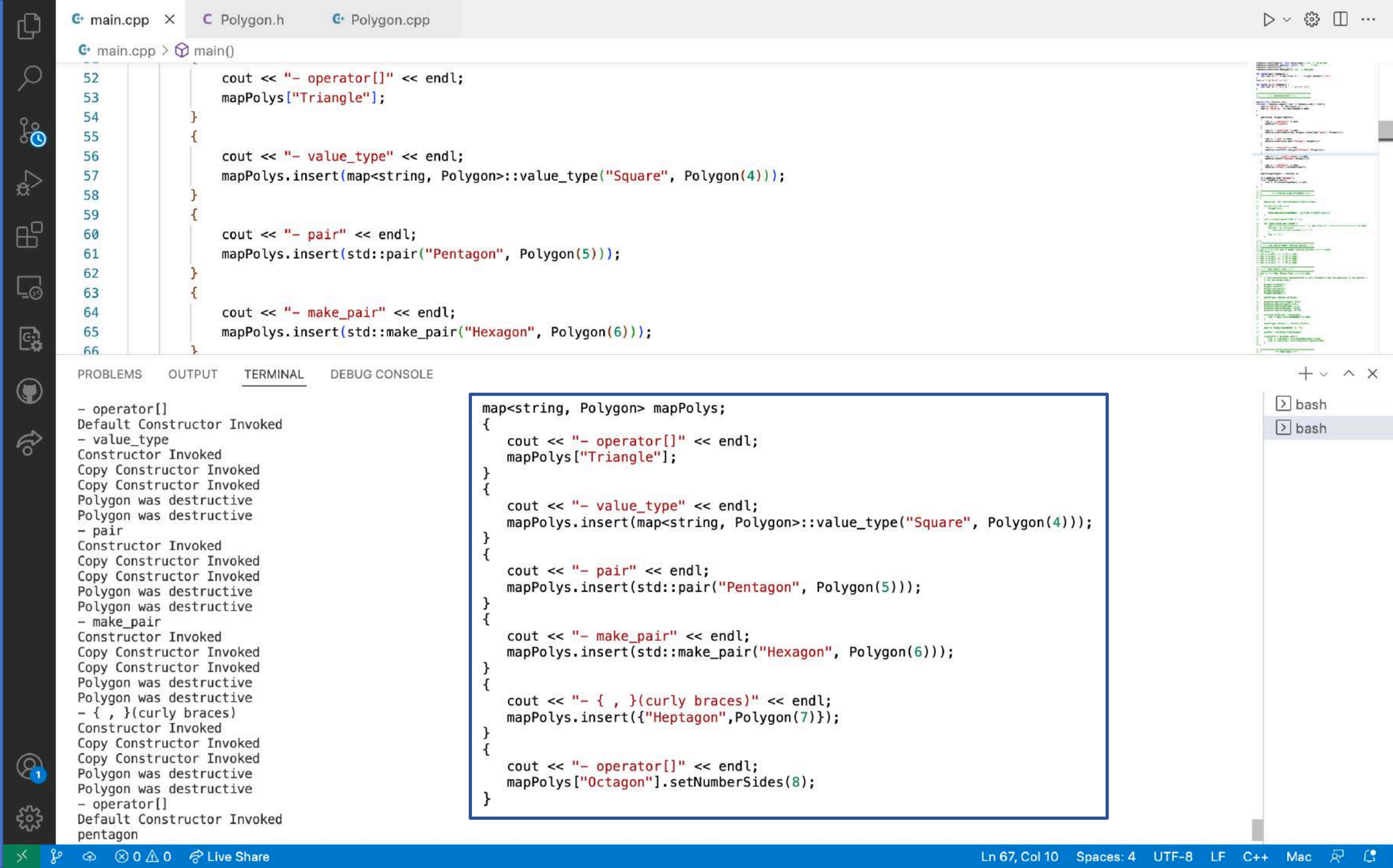
```cpp
    fibonacci.insert(std::make_pair(3, 2));  // make_pair

    for (auto& pair: fibonacci) {
        std::cout << "{" << pair.first << " : " << pair.second << "}\n";
    }
    cout << "- By [k,v]" << '\n';

    for (auto& [k,v]: fibonacci) {
        std::cout << "{" << k << " : " << v << "}\n";
    }

    //*********************************************
    //          ---- Map & Iterators -----
    //*********************************************

    map<int,int>::iterator itor;
    for(itor = fibonacci.begin(); itor != fibonacci.end(); ++itor){
        cout << "key is: " << itor->first << " : ";
        cout << "Value is: " << itor->second << endl;
    }
```

```
(base) Ivaldo:Week8 admin$ ./main
{0 : 0}
{1 : 1}
{2 : 1}
{3 : 2}
{4 : 3}
- By [k,v]
{0 : 0}
{1 : 1}
{2 : 1}
{3 : 2}
{4 : 3}
key is: 0 : Value is: 0
key is: 1 : Value is: 1
key is: 2 : Value is: 1
key is: 3 : Value is: 2
key is: 4 : Value is: 3
(base) Ivaldo:Week8 admin$
```

Map insert

```
map<string, Polygon> mapPolys;
{
   cout << "- operator[]" << endl;
   mapPolys["Triangle"];
}
{
   cout << "- value_type" << endl;
   mapPolys.insert(map<string, Polygon>::value_type("Square", Polygon(4)));
}
{
   cout << "- pair" << endl;
   mapPolys.insert(std::pair("Pentagon", Polygon(5)));
}
{
   cout << "- make_pair" << endl;
   mapPolys.insert(std::make_pair("Hexagon", Polygon(6)));
}
{
   cout << "- { , }(curly braces)" << endl;
   mapPolys.insert({"Heptagon",Polygon(7)});
}
{
   cout << "- operator[]" << endl;
   mapPolys["Octagon"].setNumberSides(8);
}
```

Map insert



```cpp
        cout << "- operator[]" << endl;
        mapPolys["Triangle"];
    }
    {
        cout << "- value_type" << endl;
        mapPolys.insert(map<string, Polygon>::value_type("Square", Polygon(4)));
    }
    {
        cout << "- pair" << endl;
        mapPolys.insert(std::pair("Pentagon", Polygon(5)));
    }
    {
        cout << "- make_pair" << endl;
        mapPolys.insert(std::make_pair("Hexagon", Polygon(6)));
    }
```

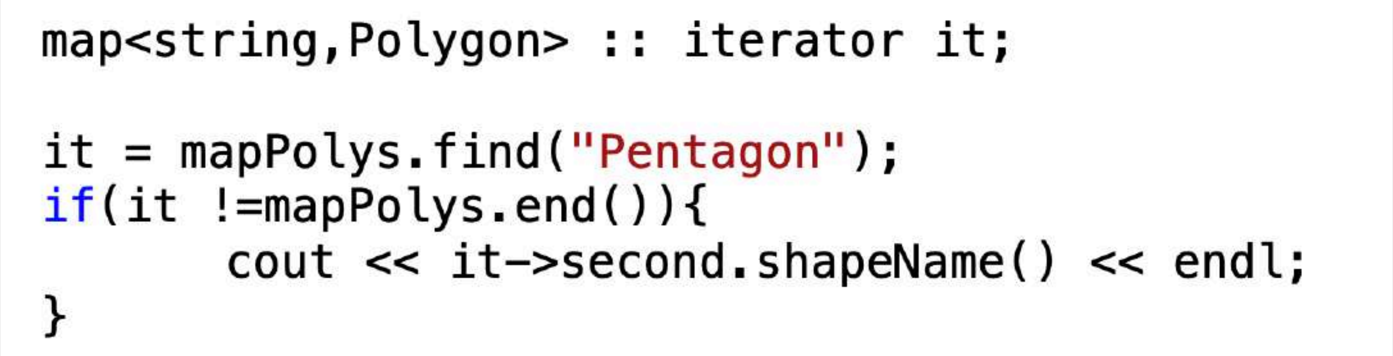
```
- operator[]
Default Constructor Invoked
- value_type
Constructor Invoked
Copy Constructor Invoked
Copy Constructor Invoked
Polygon was destructive
Polygon was destructive
- pair
Constructor Invoked
Copy Constructor Invoked
Copy Constructor Invoked
Polygon was destructive
Polygon was destructive
- make_pair
Constructor Invoked
Copy Constructor Invoked
Copy Constructor Invoked
Polygon was destructive
Polygon was destructive
- { , }(curly braces)
Constructor Invoked
Copy Constructor Invoked
Copy Constructor Invoked
Polygon was destructive
Polygon was destructive
- operator[]
Default Constructor Invoked
pentagon
```

```cpp
map<string, Polygon> mapPolys;
{
    cout << "- operator[]" << endl;
    mapPolys["Triangle"];
}
{
    cout << "- value_type" << endl;
    mapPolys.insert(map<string, Polygon>::value_type("Square", Polygon(4)));
}
{
    cout << "- pair" << endl;
    mapPolys.insert(std::pair("Pentagon", Polygon(5)));
}
{
    cout << "- make_pair" << endl;
    mapPolys.insert(std::make_pair("Hexagon", Polygon(6)));
}
{
    cout << "- { , }(curly braces)" << endl;
    mapPolys.insert({"Heptagon",Polygon(7)});
}
{
    cout << "- operator[]" << endl;
    mapPolys["Octagon"].setNumberSides(8);
}
```

More functions of Map:

Expression	Description
map find()	Returns an iterator to the element with key value ‘g’ in the map if found, else returns the iterator to end.
map emplace()	Inserts the key and its element in the map container.
map upper_bound()	Returns an iterator to the first element that is equivalent to mapped value with key value ‘g’ or definitely will go after the element with key value ‘g’ in the map
map value_comp()	Returns the object that determines how the elements in the map are ordered (‘<' by default).
map rbegin()	Returns a reverse iterator which points to the last element of the map.
map swap()	function is used to exchange the contents of two maps but the maps must be of same type, although sizes may differ.

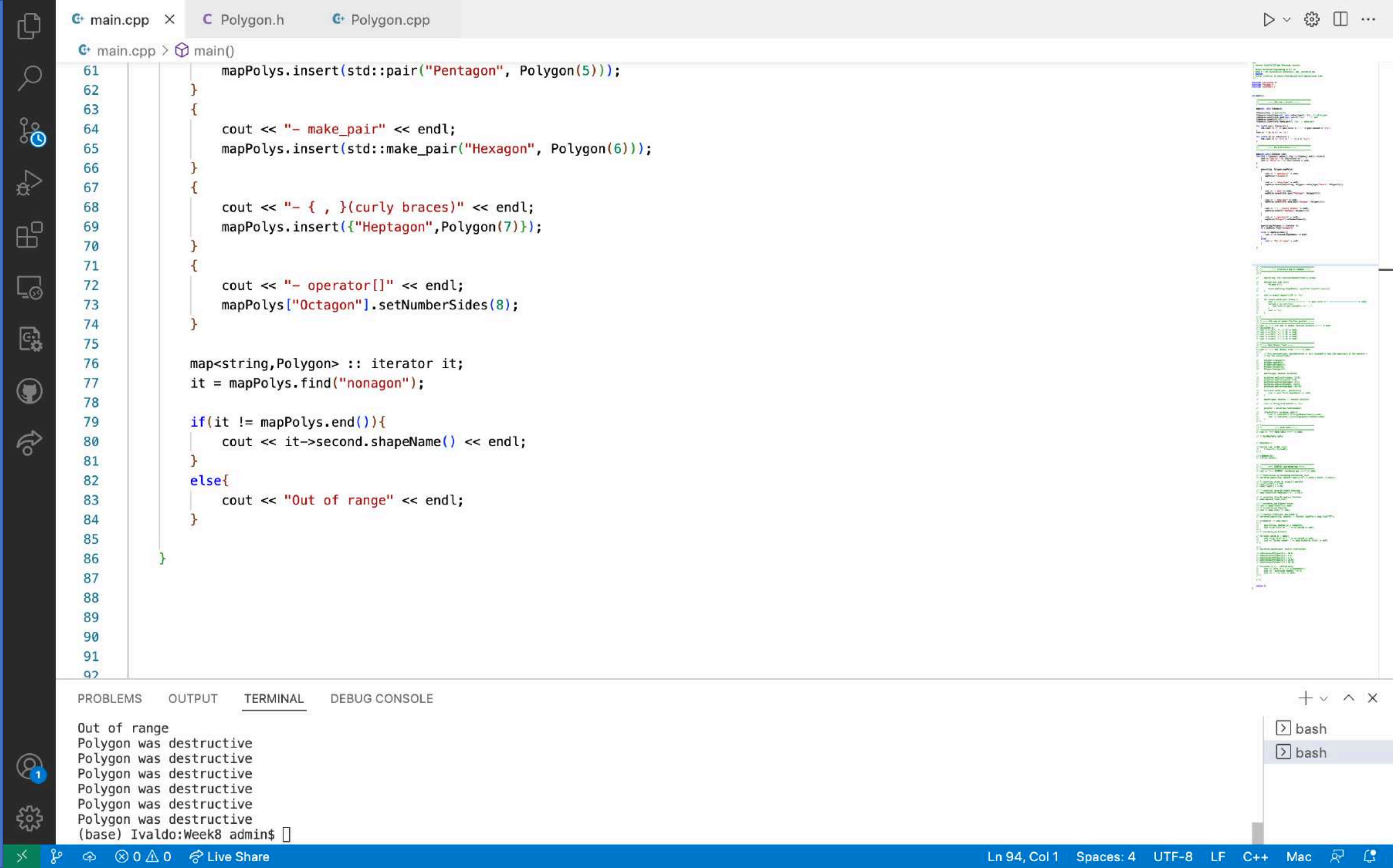
std::map::find



```
map<string,Polygon> :: iterator it;

it = mapPolys.find("Pentagon");
if(it !=mapPolys.end()){
        cout << it->second.shapeName() << endl;
}
```

Searches the container for an element with a *key* equivalent to *k* and returns an iterator to it if found, otherwise it returns an iterator to map::end.

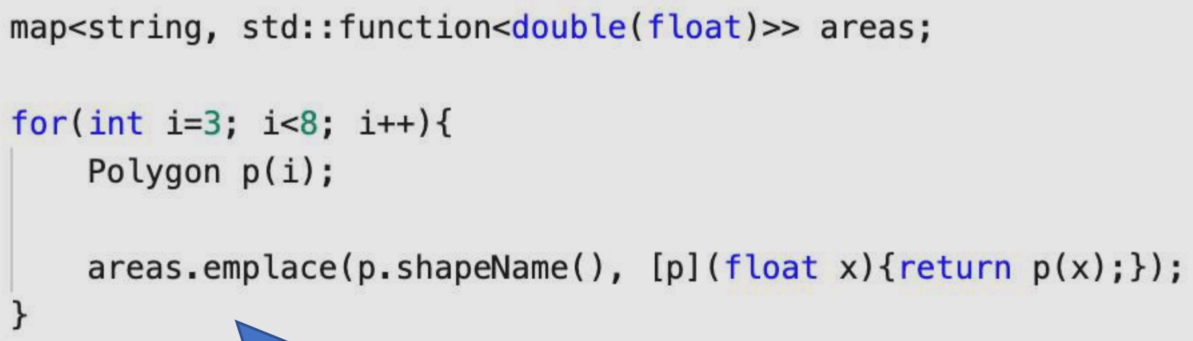
Map::find



```cpp
        mapPolys.insert(std::pair("Pentagon", Polygon(5)));
    }
    {

        cout << "- make_pair" << endl;
        mapPolys.insert(std::make_pair("Hexagon", Polygon(6)));
    }
    {

        cout << "- { , }(curly braces)" << endl;
        mapPolys.insert({"Heptagon",Polygon(7)});
    }
    {

        cout << "- operator[]" << endl;
        mapPolys["Octagon"].setNumberSides(8);
    }

    map<string,Polygon> :: iterator it;
    it = mapPolys.find("nonagon");

    if(it != mapPolys.end()){
        cout << it->second.shapeName() << endl;
    }
    else{
        cout << "Out of range" << endl;
    }

}
```

```
Out of range
Polygon was destructive
Polygon was destructive
Polygon was destructive
Polygon was destructive
Polygon was destructive
Polygon was destructive
(base) Ivaldo:Week8 admin$
```

Creating a map of lambda functions

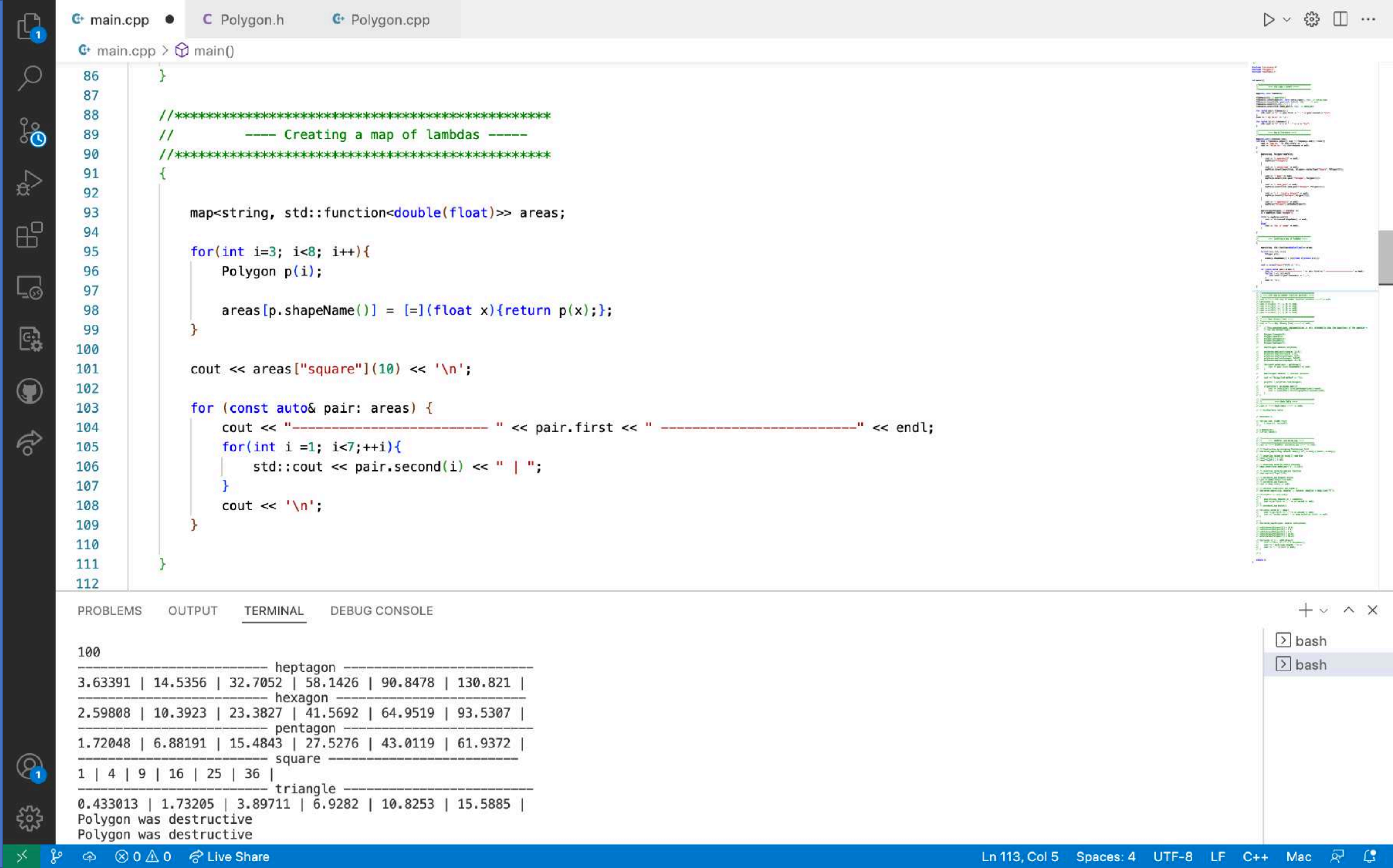


```
map<string, std::function<double(float)>> areas;

for(int i=3; i<8; i++){
    Polygon p(i);

    areas.emplace(p.shapeName(), [p](float x){return p(x);});
}
```

Inserts a new element in the map if its key is unique. This new element is constructed in place as the arguments for the construction of a value_type (which is an object of a pair type).

- Functions mush have compatible signatures: Using function wrapper (std::function).

Creating a map of lambdas



```cpp
    }

    //*************************************************
    //          ---- Creating a map of lambdas -----
    //*************************************************
    {

        map<string, std::function<double(float)>> areas;

        for(int i=3; i<8; i++){
            Polygon p(i);

            areas[p.shapeName()] = [=](float x){return p(x);};
        }

        cout << areas["square"](10) << '\n';

        for (const auto& pair: areas) {
            cout << "------------------------ " << pair.first << " ------------------------" << endl;
            for(int i =1; i<7;++i){
                std::cout << pair.second(i) << " | ";
            }
            cout << '\n';
        }

    }

```

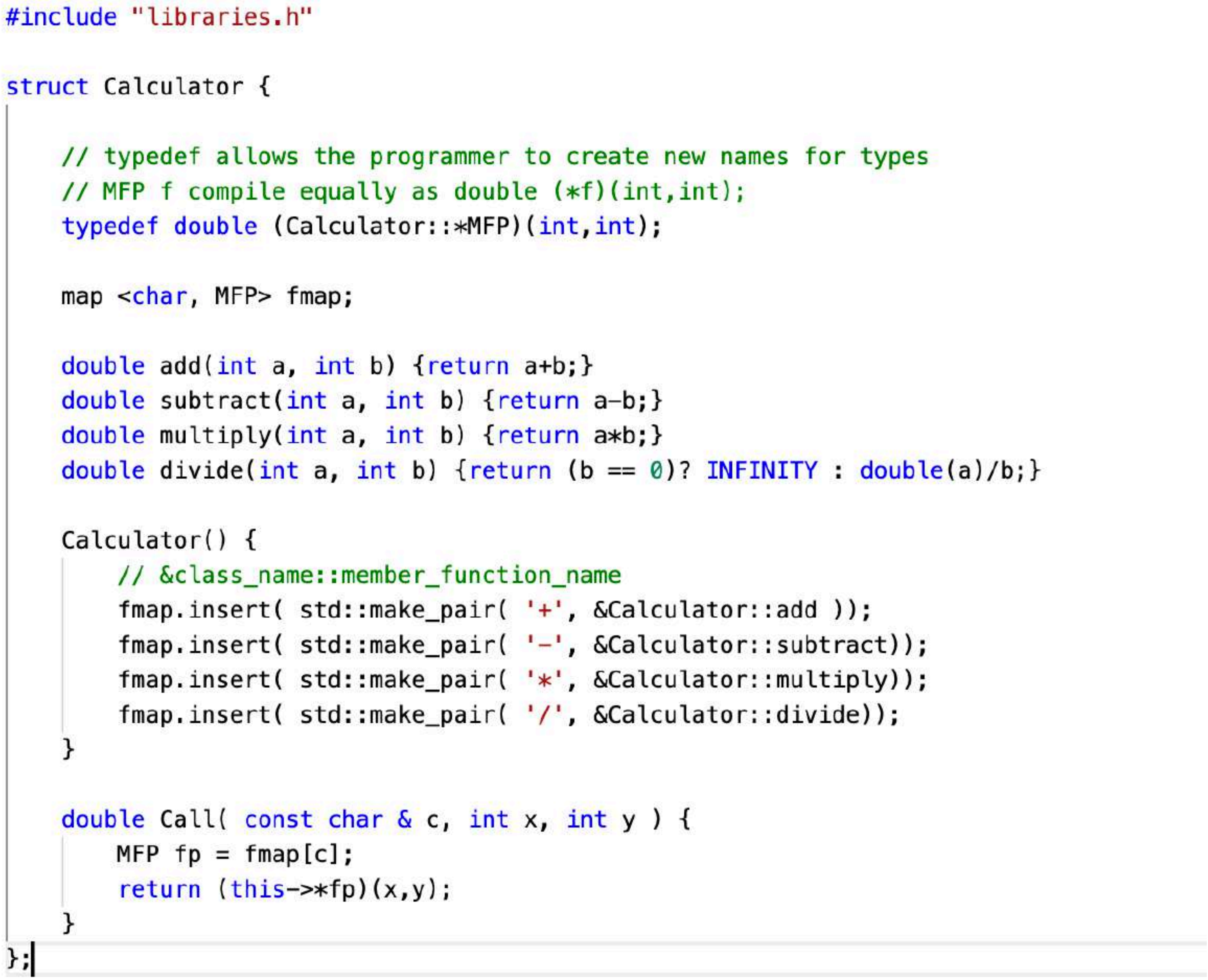
```
100
------------------------- heptagon -------------------------
3.63391 | 14.5536 | 32.7052 | 58.1426 | 90.8478 | 130.821 |
------------------------- hexagon -------------------------
2.59808 | 10.3923 | 23.3827 | 41.5692 | 64.9519 | 93.5307 |
------------------------- pentagon -------------------------
1.72048 | 6.88191 | 15.4843 | 27.5276 | 43.0119 | 61.9372 |
------------------------- square -------------------------
1 | 4 | 9 | 16 | 25 | 36 |
------------------------- triangle -------------------------
0.433013 | 1.73205 | 3.89711 | 6.9282 | 10.8253 | 15.5885 |
Polygon was destructive
Polygon was destructive
```

Call a class member function via function pointer in map.

Member functions are operators and functions that are declared as members of a class. You can declare a member function as static; this is called a *static member function*. A member function that is not declared as static is called a *nonstatic member function*.

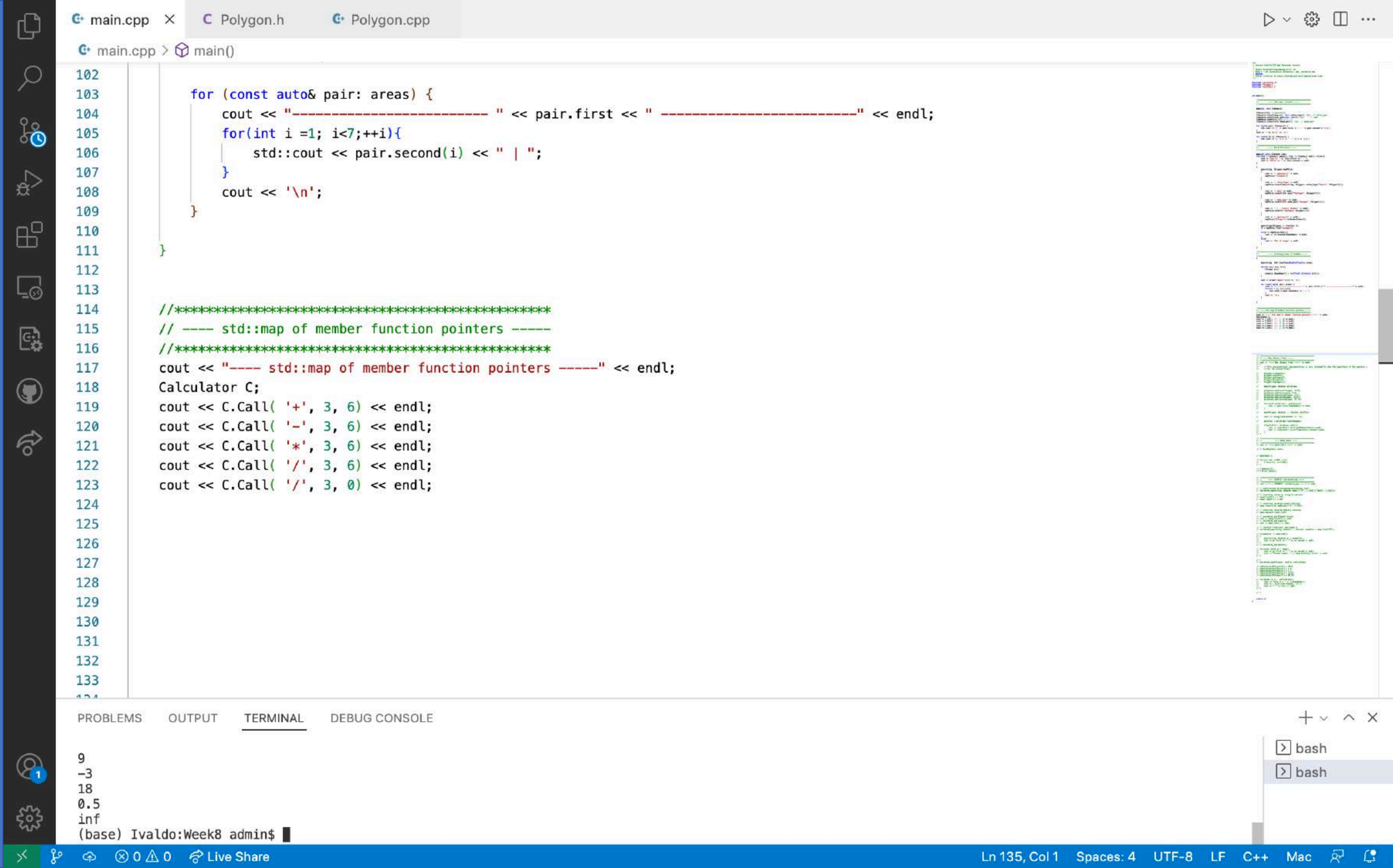
Pointers to members allow you to refer to nonstatic members of class objects. You declare a pointer-to-member-function just like a pointer-to-function, except that the syntax is a tad different (to learn more about static member functions, see [5]

function pointer in map



```
#include "libraries.h"

struct Calculator {

    // typedef allows the programmer to create new names for types
    // MFP f compile equally as double (*f)(int,int);
    typedef double (Calculator::*MFP)(int,int);

    map <char, MFP> fmap;

    double add(int a, int b) {return a+b;}
    double subtract(int a, int b) {return a-b;}
    double multiply(int a, int b) {return a*b;}
    double divide(int a, int b) {return (b == 0)? INFINITY : double(a)/b;}

    Calculator() {
        // &class_name::member_function_name
        fmap.insert( std::make_pair( '+', &Calculator::add ));
        fmap.insert( std::make_pair( '-', &Calculator::subtract));
        fmap.insert( std::make_pair( '*', &Calculator::multiply));
        fmap.insert( std::make_pair( '/', &Calculator::divide));
    }

    double Call( const char & c, int x, int y ) {
        MFP fp = fmap[c];
        return (this->*fp)(x,y);
    }
};
```

function pointer in map



```cpp

        for (const auto& pair: areas) {
            cout << "------------------------- " << pair.first << " -------------------------" << endl;
            for(int i =1; i<7;++i){
                std::cout << pair.second(i) << " | ";
            }
            cout << '\n';
        }

    }


    //************************************************
    // ---- std::map of member function pointers -----
    //************************************************
    cout << "---- std::map of member function pointers -----" << endl;
    Calculator C;
    cout << C.Call( '+', 3, 6) << endl;
    cout << C.Call( '-', 3, 6) << endl;
    cout << C.Call( '*', 3, 6) << endl;
    cout << C.Call( '/', 3, 6) << endl;
    cout << C.Call( '/', 3, 0) << endl;
```

```
9
-3
18
0.5
inf
(base) Ivaldo:Week8 admin$ 
```

Binary Tree

- **Binary trees** is a special case of trees where each node can have at most 2 children.
- Efficient search & insertion/deletion in *logarithmic* time O($\log_2 n$)

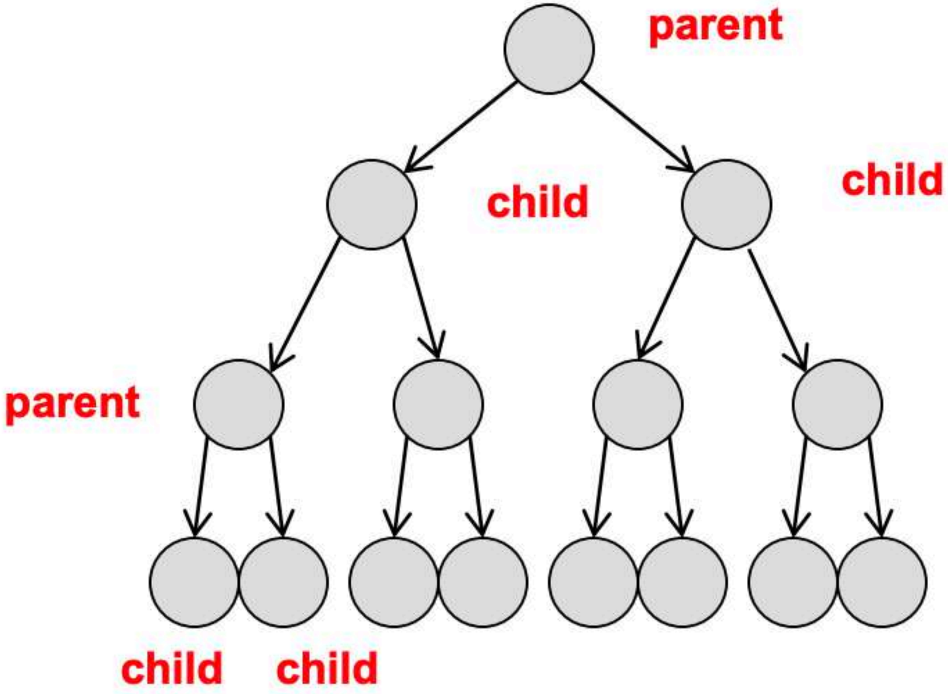


Figure from: https://ee.usc.edu/~redekopp/cs104/slides/L09_STL.pdf

Binary Search Tree

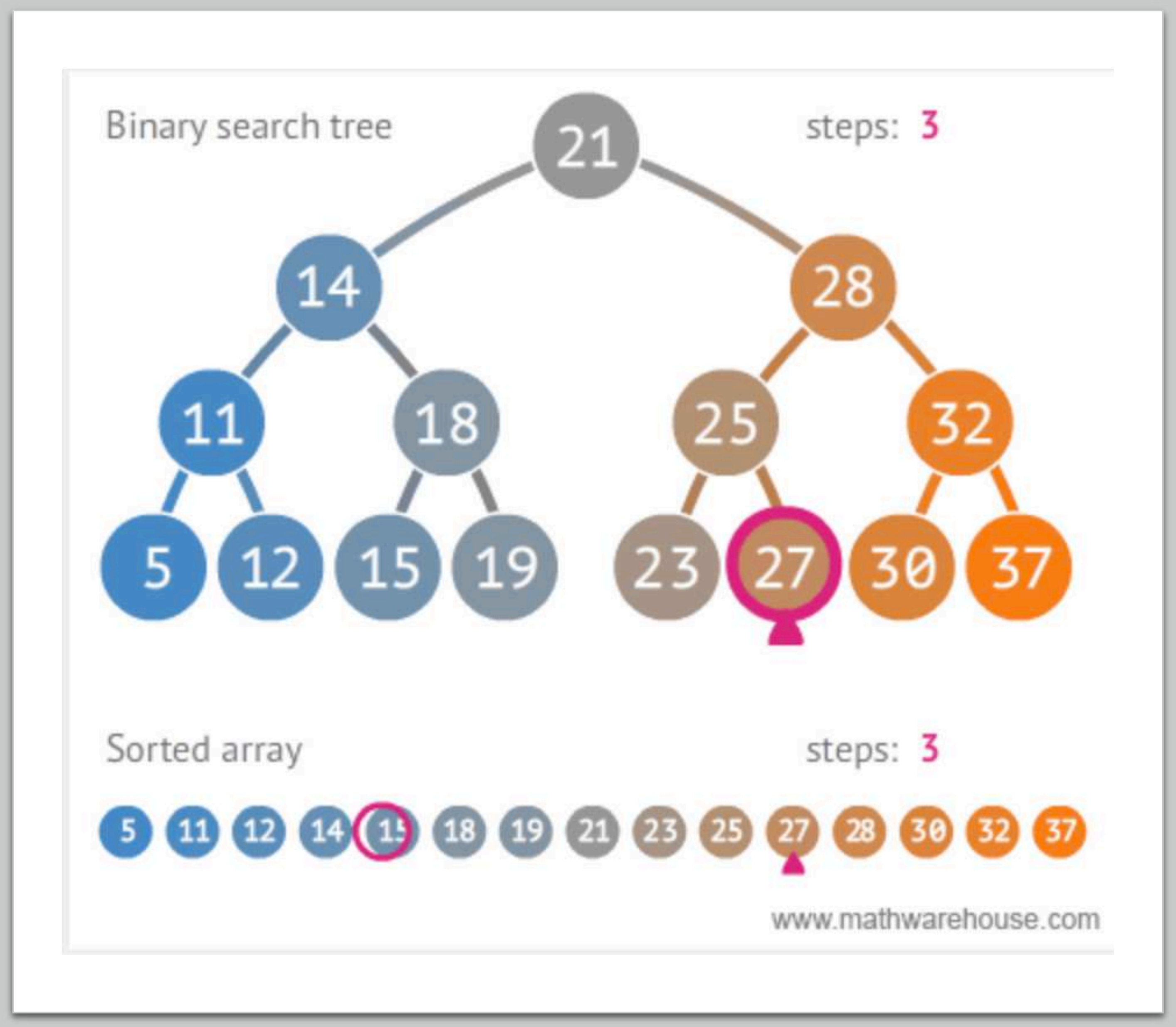
Source : https://ee.usc.edu/~redekopp/cs104/slides/L09_STL.pdf

- Tree where all nodes meet the property that:

– All descendants on the left are less than the parent’s value

– All descendants on the right are greater than the parent’s value

- Can find value (or determine it doesn’t exit) in log2n time by doing binary search



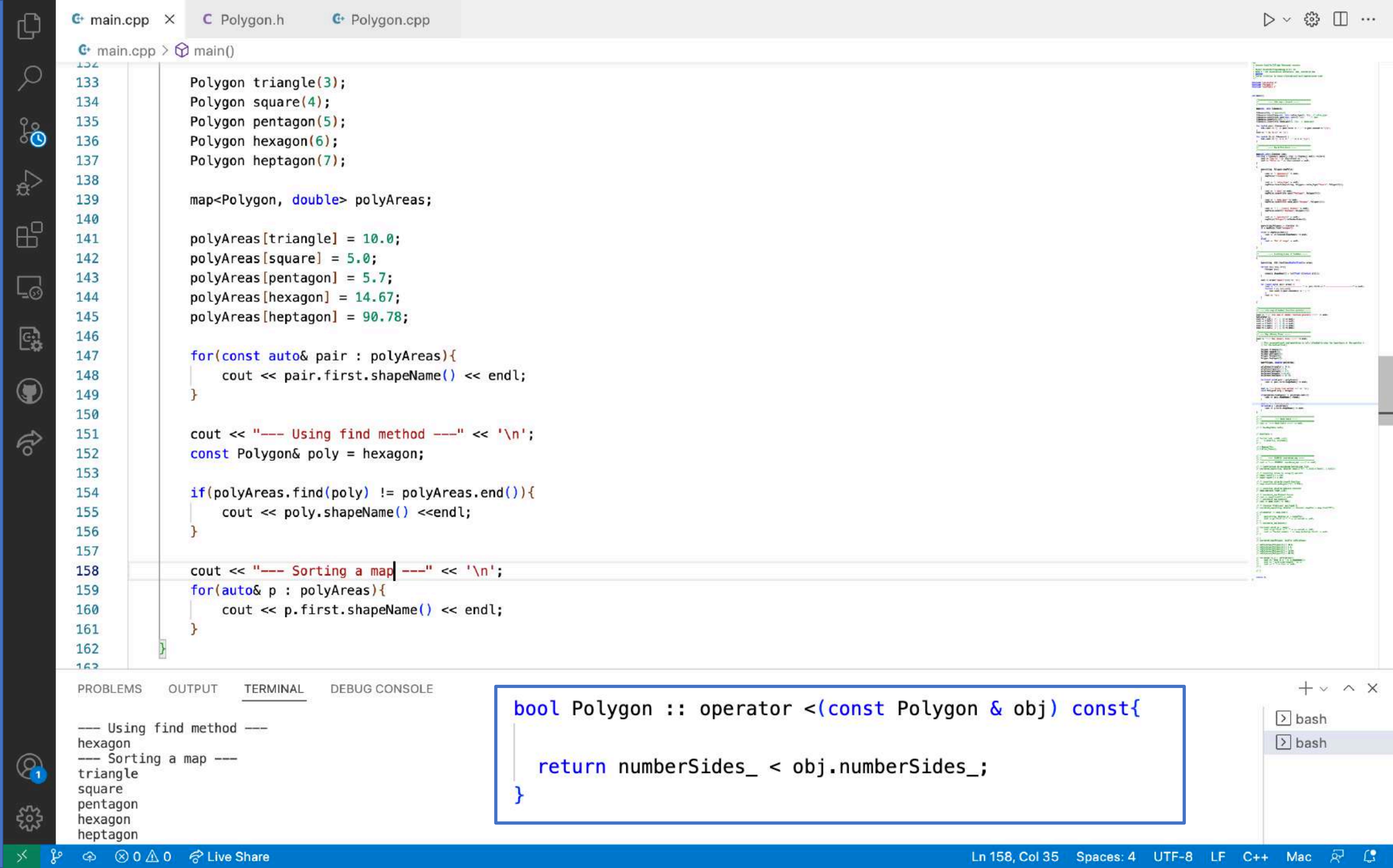
Trees & Maps

Maps and sets use binary trees internally to store the keys

This allows logarithmic find/membership test time

This is why the less-than (<) operator needs to be defined for the data type of the key

Operator <



```cpp
    Polygon triangle(3);
    Polygon square(4);
    Polygon pentagon(5);
    Polygon hexagon(6);
    Polygon heptagon(7);

    map<Polygon, double> polyAreas;

    polyAreas[triangle] = 10.0;
    polyAreas[square] = 5.0;
    polyAreas[pentagon] = 5.7;
    polyAreas[hexagon] = 14.67;
    polyAreas[heptagon] = 90.78;

    for(const auto& pair : polyAreas){
        cout << pair.first.shapeName() << endl;
    }

    cout << "--- Using find method ---" << '\n';
    const Polygon& poly = hexagon;

    if(polyAreas.find(poly) != polyAreas.end()){
        cout << poly.shapeName() <<endl;
    }

    cout << "--- Sorting a map ---" << '\n';
    for(auto& p : polyAreas){
        cout << p.first.shapeName() << endl;
    }
}
```

```
--- Using find method ---
hexagon
--- Sorting a map ---
triangle
square
pentagon
hexagon
heptagon
```

```cpp
bool Polygon :: operator <(const Polygon & obj) const{

  return numberSides_ < obj.numberSides_;
}
```

Unordered map

Unordered map is an associative container that contains key-value pairs with unique keys. Search, insertion, and removal of elements have average constant-time complexity.

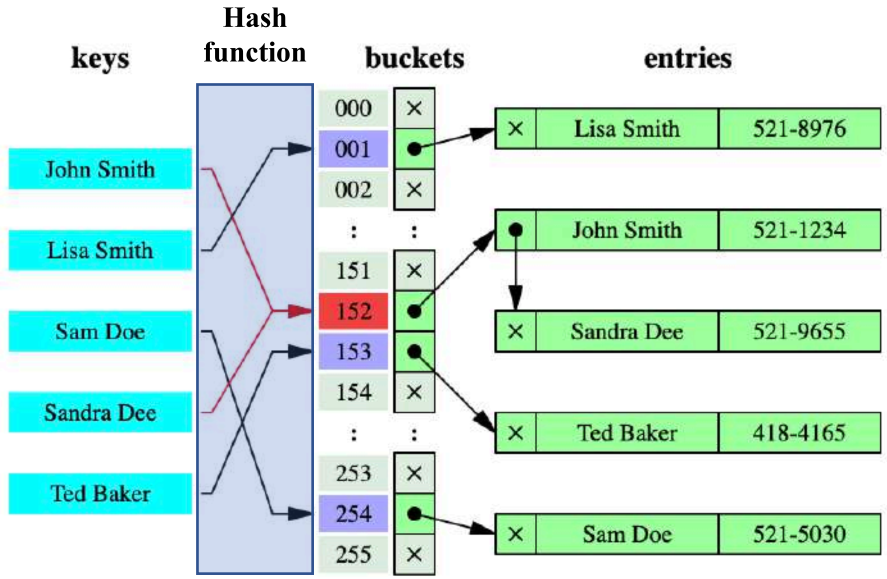
Internally, the elements are not sorted in any particular order, but organized into buckets. Which bucket an element is placed into depends entirely on the hash of its key. Keys with the same hash code appear in the same bucket.

unordered_map containers are faster than map containers to access individual elements by their *key*, although they are generally less efficient for range iteration through a subset of their elements.

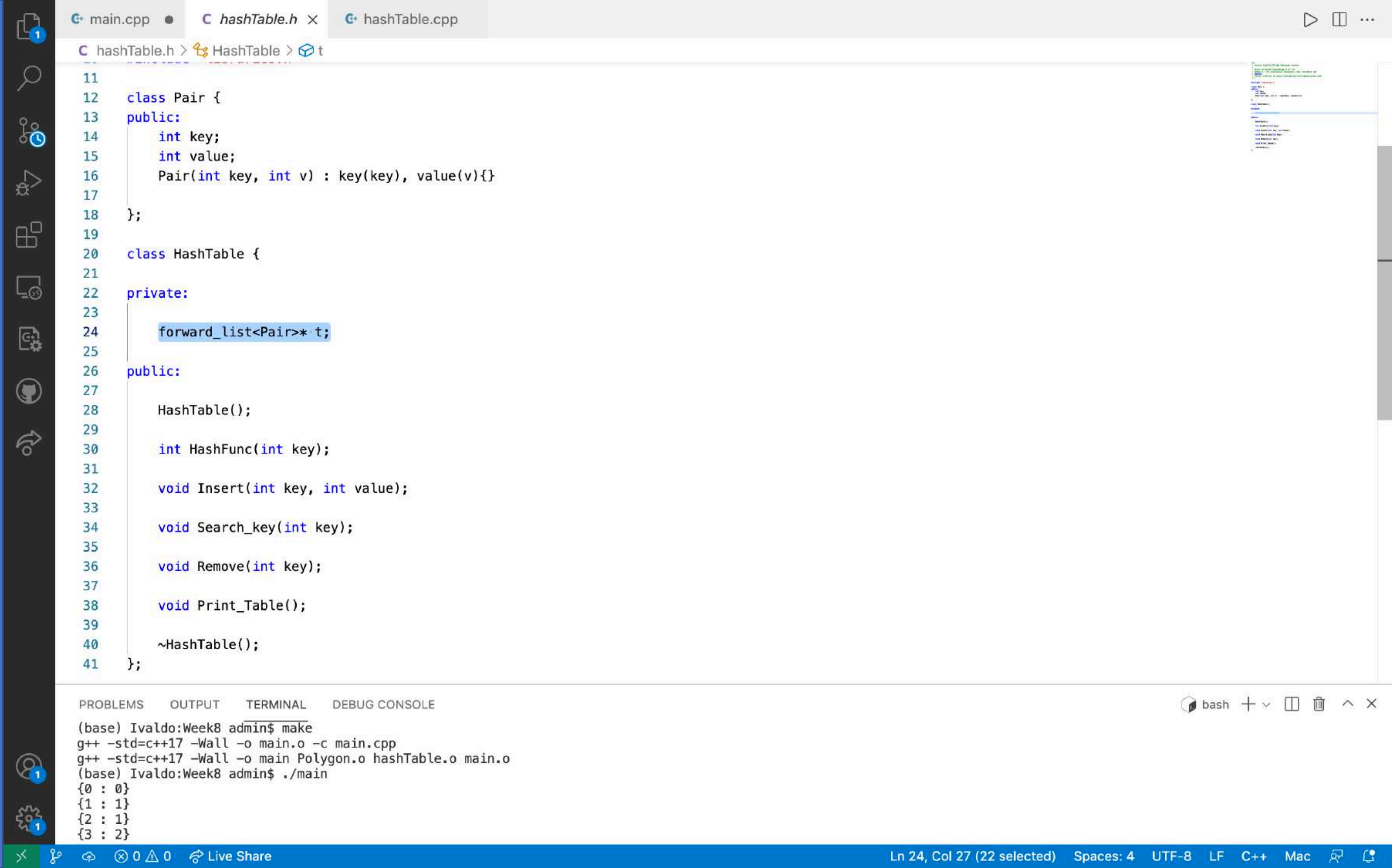
Hash table

- A hash table is a data structure which is used to store key-value pairs. Hash function is used by hash table to compute an index into an array in which an element will be inserted or searched.[9]

- A hash table is traditionally implemented with an array of linked lists. When we want to insert a key/Value pair, we map the key to an index in the array using the hash function. The value is then inserted into the linked list at that position.



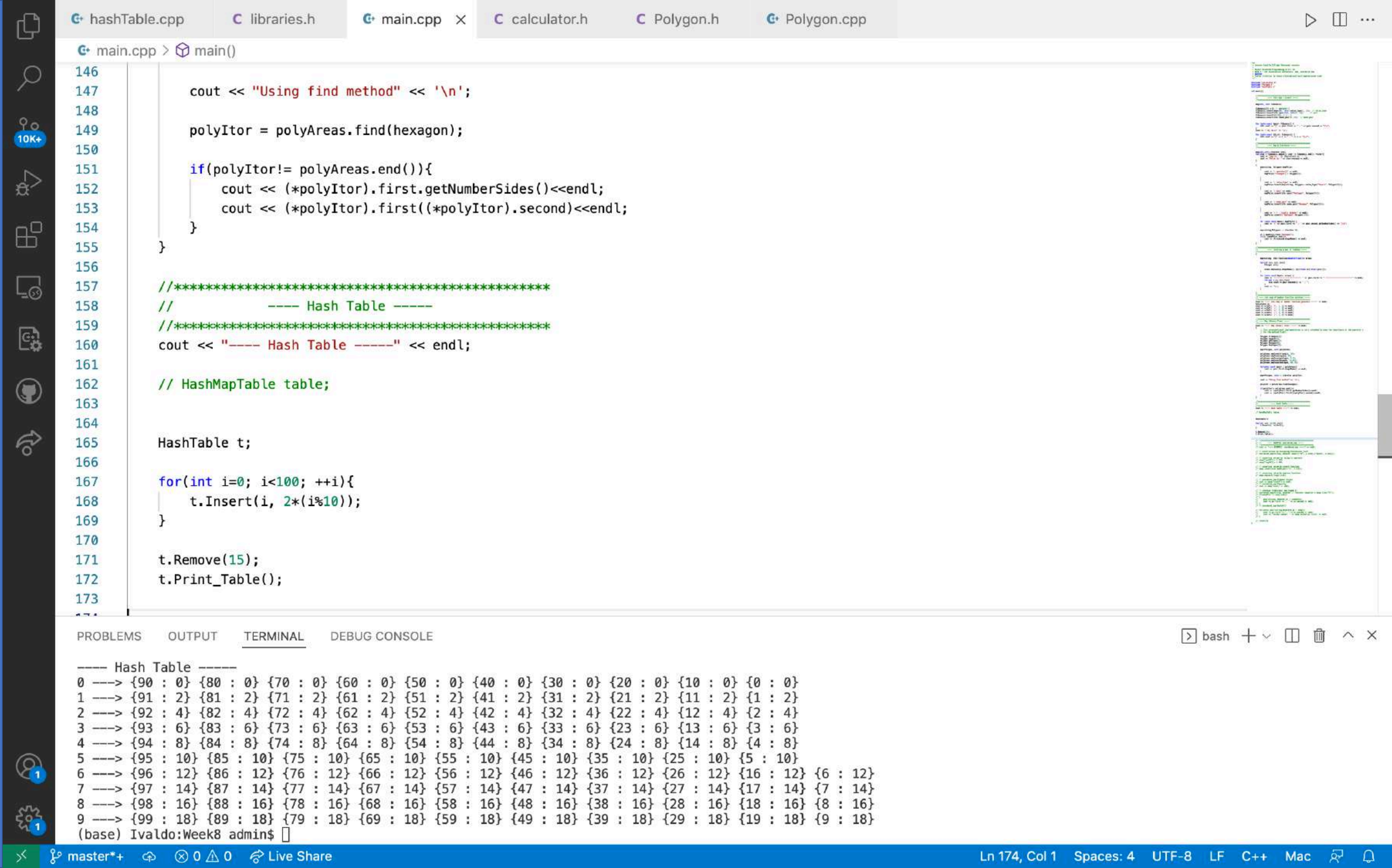
Hash Table



```cpp

class Pair {
public:
    int key;
    int value;
    Pair(int key, int v) : key(key), value(v){}

};

class HashTable {

private:

    forward_list<Pair>* t;

public:

    HashTable();

    int HashFunc(int key);

    void Insert(int key, int value);

    void Search_key(int key);

    void Remove(int key);

    void Print_Table();

    ~HashTable();
};
```

```
(base) Ivaldo:Week8 admin$ make
g++ -std=c++17 -Wall -o main.o -c main.cpp
g++ -std=c++17 -Wall -o main Polygon.o hashTable.o main.o
(base) Ivaldo:Week8 admin$ ./main
{0 : 0}
{1 : 1}
{2 : 1}
{3 : 2}
```

Hash Table



```cpp

        cout << "Using find method" << '\n';

        polyItor = polyAreas.find(hexagon);

        if(polyItor!= polyAreas.end()){
            cout << (*polyItor).first.getNumberSides()<<endl;
            cout << (*polyItor).first((*polyItor).second)<<endl;
        }
    }

    //*************************************************
    //            ---- Hash Table -----
    //*************************************************
    cout << "---- Hash Table -----" << endl;

    // HashMapTable table;


    HashTable t;

    for(int i=0; i<100; ++i){
        t.Insert(i, 2*(i%10));
    }

    t.Remove(15);
    t.Print_Table();

```

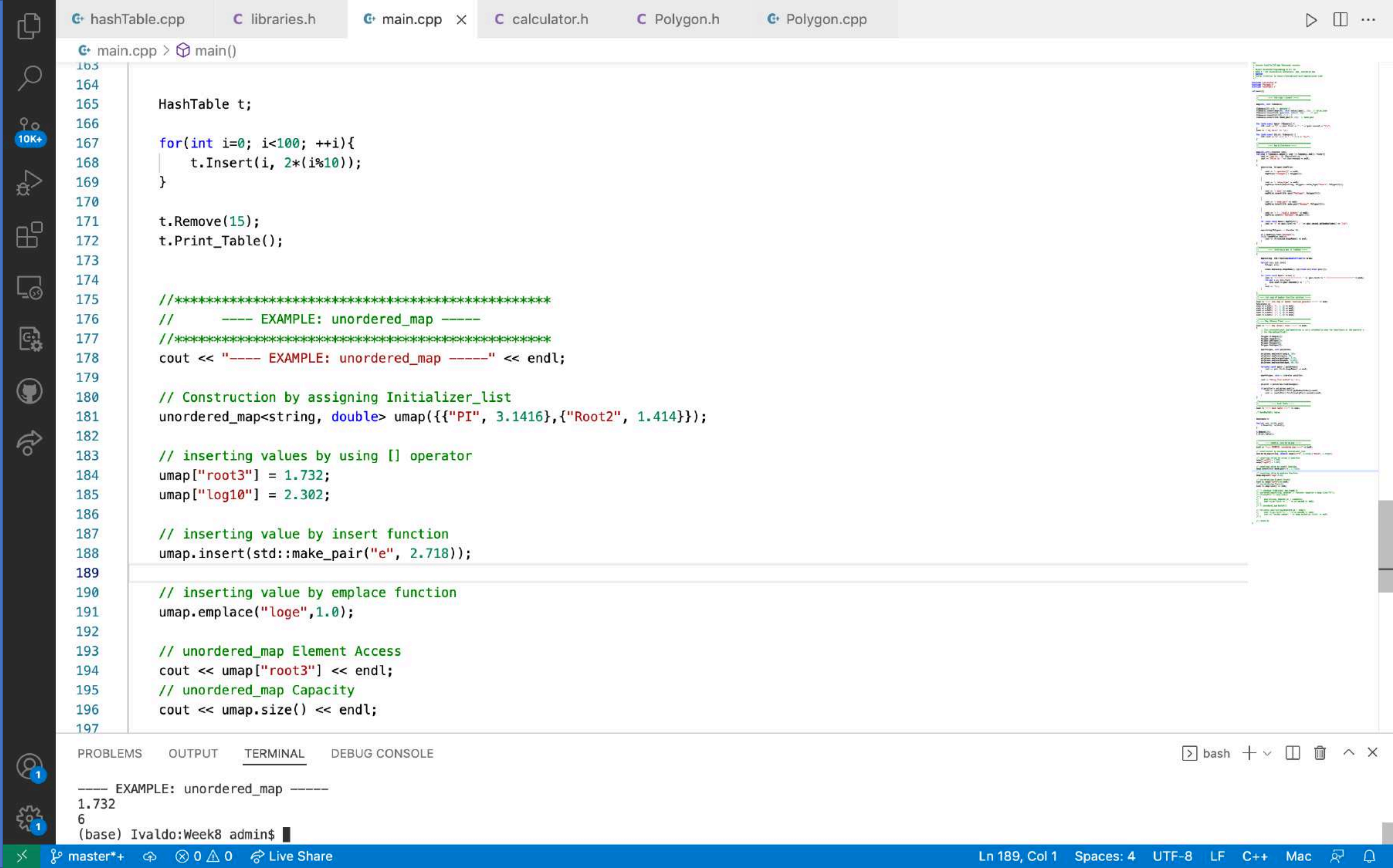
```
---- Hash Table -----
0 ---> {90 : 0} {80 : 0} {70 : 0} {60 : 0} {50 : 0} {40 : 0} {30 : 0} {20 : 0} {10 : 0} {0 : 0}
1 ---> {91 : 2} {81 : 2} {71 : 2} {61 : 2} {51 : 2} {41 : 2} {31 : 2} {21 : 2} {11 : 2} {1 : 2}
2 ---> {92 : 4} {82 : 4} {72 : 4} {62 : 4} {52 : 4} {42 : 4} {32 : 4} {22 : 4} {12 : 4} {2 : 4}
3 ---> {93 : 6} {83 : 6} {73 : 6} {63 : 6} {53 : 6} {43 : 6} {33 : 6} {23 : 6} {13 : 6} {3 : 6}
4 ---> {94 : 8} {84 : 8} {74 : 8} {64 : 8} {54 : 8} {44 : 8} {34 : 8} {24 : 8} {14 : 8} {4 : 8}
5 ---> {95 : 10} {85 : 10} {75 : 10} {65 : 10} {55 : 10} {45 : 10} {35 : 10} {25 : 10} {5 : 10}
6 ---> {96 : 12} {86 : 12} {76 : 12} {66 : 12} {56 : 12} {46 : 12} {36 : 12} {26 : 12} {16 : 12} {6 : 12}
7 ---> {97 : 14} {87 : 14} {77 : 14} {67 : 14} {57 : 14} {47 : 14} {37 : 14} {27 : 14} {17 : 14} {7 : 14}
8 ---> {98 : 16} {88 : 16} {78 : 16} {68 : 16} {58 : 16} {48 : 16} {38 : 16} {28 : 16} {18 : 16} {8 : 16}
9 ---> {99 : 18} {89 : 18} {79 : 18} {69 : 18} {59 : 18} {49 : 18} {39 : 18} {29 : 18} {19 : 18} {9 : 18}
(base) Ivaldo:Week8 admin$ 
```

Methods on unordered_map

A lot of functions are available which work on unordered_map. most useful of them are: operator =, operator [], empty and size for capacity, begin and end for the iterator, find and count for lookup, insert and erase for modification.

The C++11 library also provides functions to see internally used bucket count, bucket size, and also used hash function and various hash policies but they are less useful in real applications. [10]

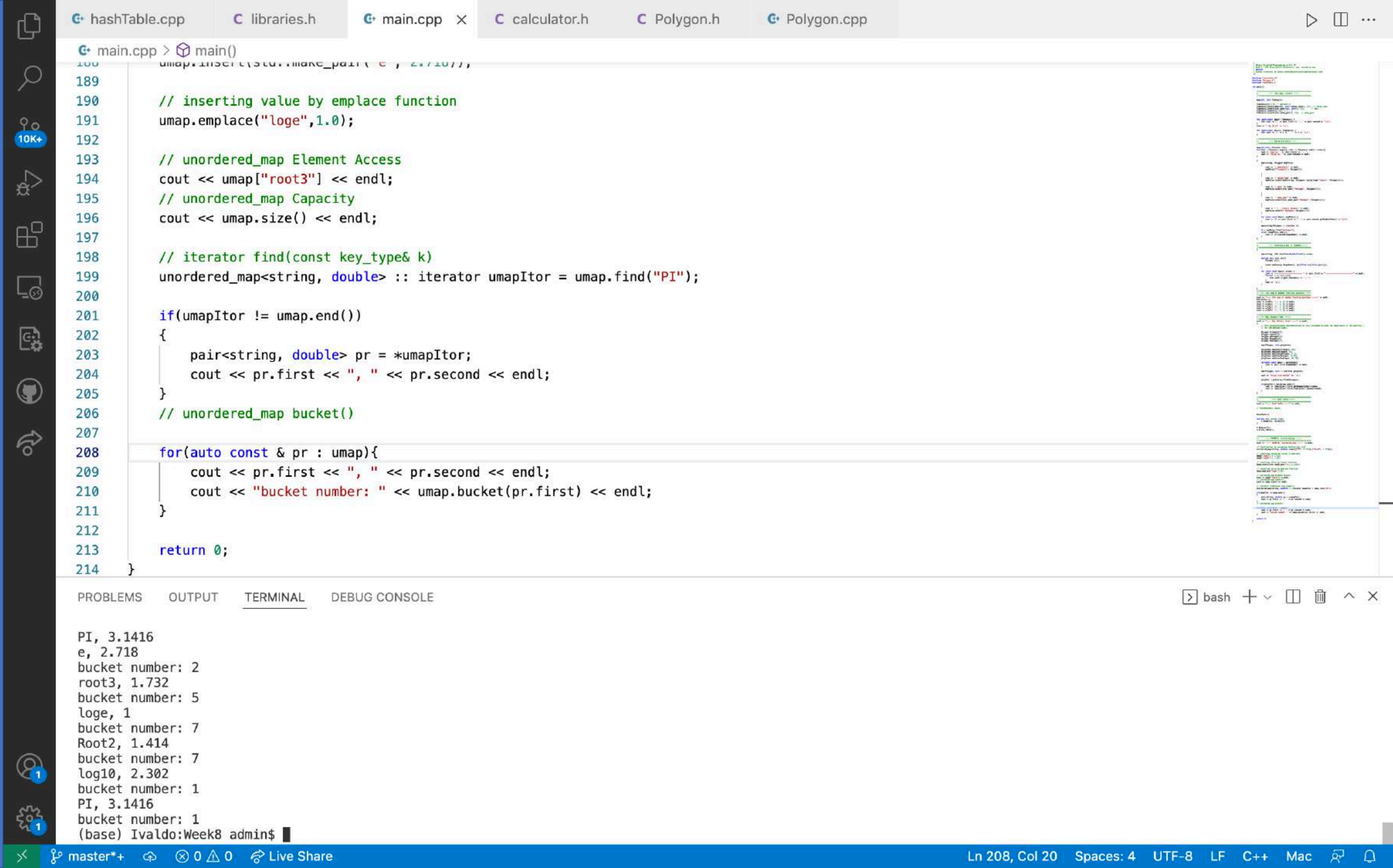
Methods



```cpp

    HashTable t;

    for(int i=0; i<100; ++i){
        t.Insert(i, 2*(i%10));
    }

    t.Remove(15);
    t.Print_Table();


    //************************************************
    //      ---- EXAMPLE: unordered_map -----
    //************************************************
    cout << "---- EXAMPLE: unordered_map -----" << endl;

    // Construction by assigning Initializer_list
    unordered_map<string, double> umap({{"PI", 3.1416},{"Root2", 1.414}});

    // inserting values by using [] operator
    umap["root3"] = 1.732;
    umap["log10"] = 2.302;

    // inserting value by insert function
    umap.insert(std::make_pair("e", 2.718));

    // inserting value by emplace function
    umap.emplace("loge",1.0);

    // unordered_map Element Access
    cout << umap["root3"] << endl;
    // unordered_map Capacity
    cout << umap.size() << endl;
```

```
---- EXAMPLE: unordered_map -----
1.732
6
(base) Ivaldo:Week8 admin$ 
```

Methods



```cpp
    umap.insert(std::make_pair("e", 2.718));

    // inserting value by emplace function
    umap.emplace("loge",1.0);

    // unordered_map Element Access
    cout << umap["root3"] << endl;
    // unordered_map Capacity
    cout << umap.size() << endl;

    // iterator find(const key_type& k)
    unordered_map<string, double> :: iterator umapItor = umap.find("PI");

    if(umapItor != umap.end())
    {
        pair<string, double> pr = *umapItor;
        cout << pr.first << ", " << pr.second << endl;
    }
    // unordered_map bucket()

    for(auto const & pr : umap){
        cout << pr.first << ", " << pr.second << endl;
        cout << "bucket number: " << umap.bucket(pr.first) << endl;
    }

    return 0;
}
```

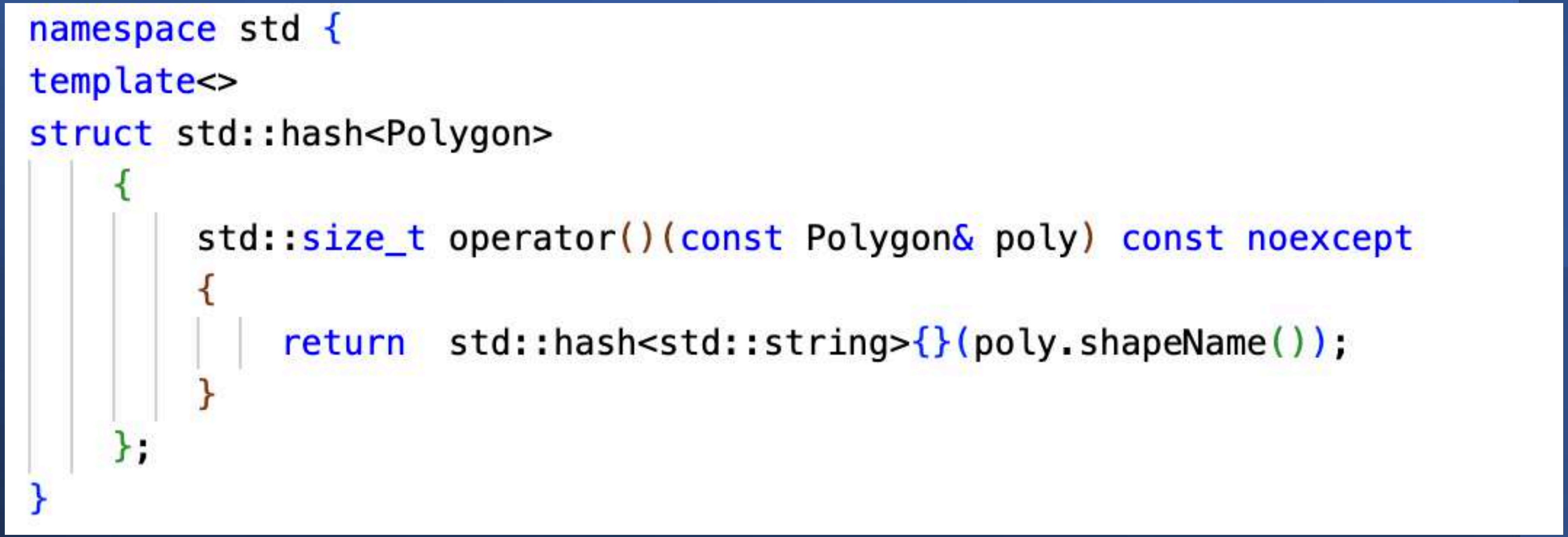
```
PI, 3.1416
e, 2.718
bucket number: 2
root3, 1.732
bucket number: 5
loge, 1
bucket number: 7
Root2, 1.414
bucket number: 7
log10, 2.302
bucket number: 1
PI, 3.1416
bucket number: 1
(base) Ivaldo:Week8 admin$
```

Implementing a hash function.

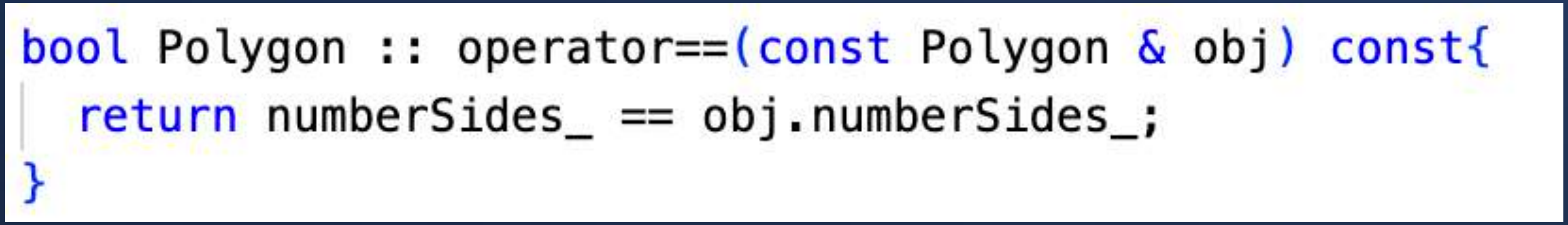
We need to define two things:

- A **hash function**; this must be a class that overrides operator() and calculates the hash value given an object of the key-type. One particularly straight-forward way of doing this is to specialize the std::hash template for your key-type.

- A **comparison function for equality**; this is required because the hash cannot rely on the fact that the hash function will always provide a unique hash value for every distinct key by overloading operator==() for your key.

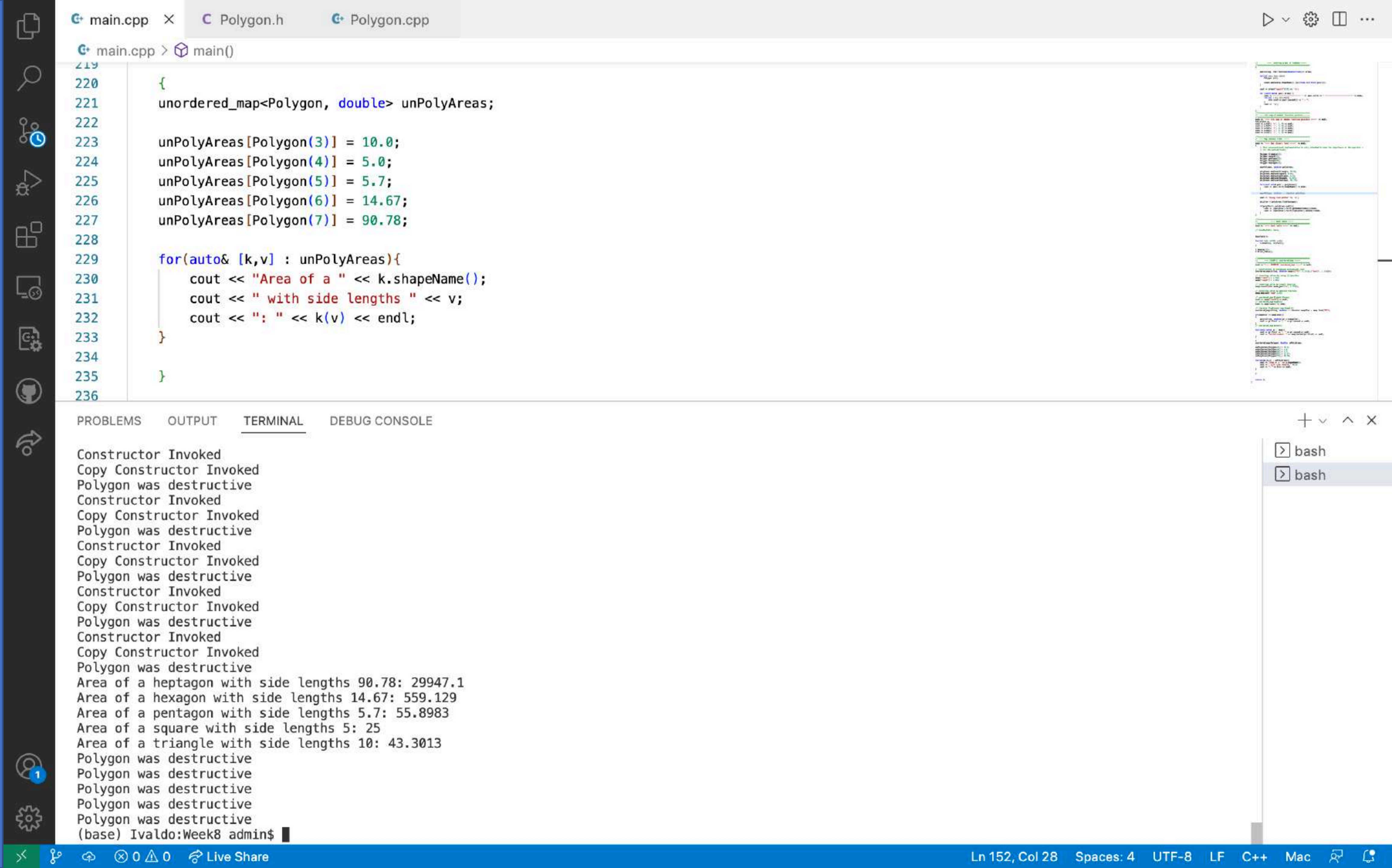


```cpp
namespace std {
template<>
struct std::hash<Polygon>
    {
        std::size_t operator()(const Polygon& poly) const noexcept
        {
            return  std::hash<std::string>{}(poly.shapeName());
        }
    };
}
```



```cpp
bool Polygon :: operator==(const Polygon & obj) const{
  return numberSides_ == obj.numberSides_;
}
```

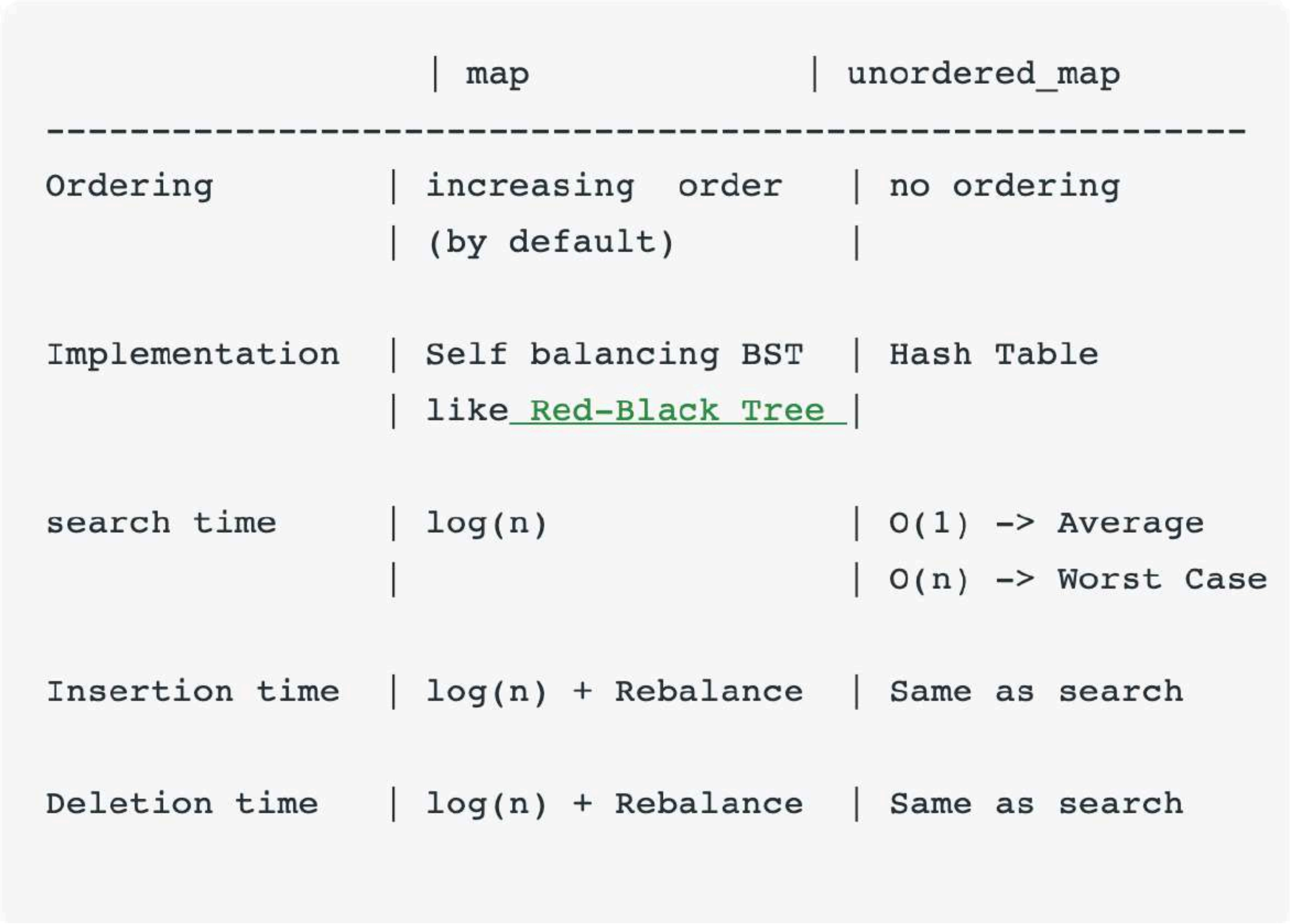
std::hash



```cpp
    {
    unordered_map<Polygon, double> unPolyAreas;

    unPolyAreas[Polygon(3)] = 10.0;
    unPolyAreas[Polygon(4)] = 5.0;
    unPolyAreas[Polygon(5)] = 5.7;
    unPolyAreas[Polygon(6)] = 14.67;
    unPolyAreas[Polygon(7)] = 90.78;

    for(auto& [k,v] : unPolyAreas){
        cout << "Area of a " << k.shapeName();
        cout << " with side lengths " << v;
        cout << ": " << k(v) << endl;
    }

    }
```

```
Constructor Invoked
Copy Constructor Invoked
Polygon was destructive
Constructor Invoked
Copy Constructor Invoked
Polygon was destructive
Constructor Invoked
Copy Constructor Invoked
Polygon was destructive
Constructor Invoked
Copy Constructor Invoked
Polygon was destructive
Constructor Invoked
Copy Constructor Invoked
Polygon was destructive
Area of a heptagon with side lengths 90.78: 29947.1
Area of a hexagon with side lengths 14.67: 559.129
Area of a pentagon with side lengths 5.7: 55.8983
Area of a square with side lengths 5: 25
Area of a triangle with side lengths 10: 43.3013
Polygon was destructive
Polygon was destructive
Polygon was destructive
Polygon was destructive
Polygon was destructive
(base) Ivaldo:Week8 admin$ 
```

map vs unordered_map in C++



	map	unordered_map
Ordering	increasing order (by default)	no ordering
Implementation	Self balancing BST like Red-Black Tree	Hash Table
search time	log(n)	O(1) -> Average O(n) -> Worst Case
Insertion time	log(n) + Rebalance	Same as search
Deletion time	log(n) + Rebalance	Same as search

Figure from : https://www.geeksforgeeks.org/map-vs-unordered_map-c/

Bibliography

1. Umich (2022/02/17) Using Pointers to Member Functions, http://websites.umich.edu/~eecs381/handouts/Pointers_to_memberfuncs.pdf
2. Embedded Artistry(2022/02/17) An Overview of C++ STL Containers, https://embeddedartistry.com/blog/2017/08/02/an-overview-of-c-stl-containers/
3. USC (2022/02/17) C++ STL; Iterators, Maps, Sets, https://ee.usc.edu/~redekopp/cs104/slides/L09_STL.pdf
4. Geeksforgeeks (2022/02/17) Map in C++ Standard Template Library (STL), https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl
5. Learn C++ (2022/02/17) Static member functions, https://www.learncpp.com/cpp-tutorial/static-member-functions/
6. C++ (2022/02/17) unordered_map, https://www.cplusplus.com/reference/unordered_map/unordered_map/
7. C++ preference (2022/02/17) unordered_map, https://en.cppreference.com/w/cpp/container/unordered_map
8. Tutorials Point (2022/02/17) C++ Program to Implement Hash Tables, https://www.tutorialspoint.com/cplusplus-program-to-implement-hash-tables
9. Geeksforgeeks (2022/02/17) unordered_map in C++ STL, https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/
10. Stackoverflow (2022/05/20) C++ unordered_map using a custom class type as the key, https://stackoverflow.com/questions/17016175/c-unordered-map-using-a-custom-class-type-as-the-key
11. cppreference.com (2022/05/20) std::hash, https://en.cppreference.com/w/cpp/utility/hash

Appendix

- Libraries.h
- calculator.h
- hashTable.h
- hashTable.cpp
- Polygon.h
- Polygon.cpp
- main.cpp