Lecture 17

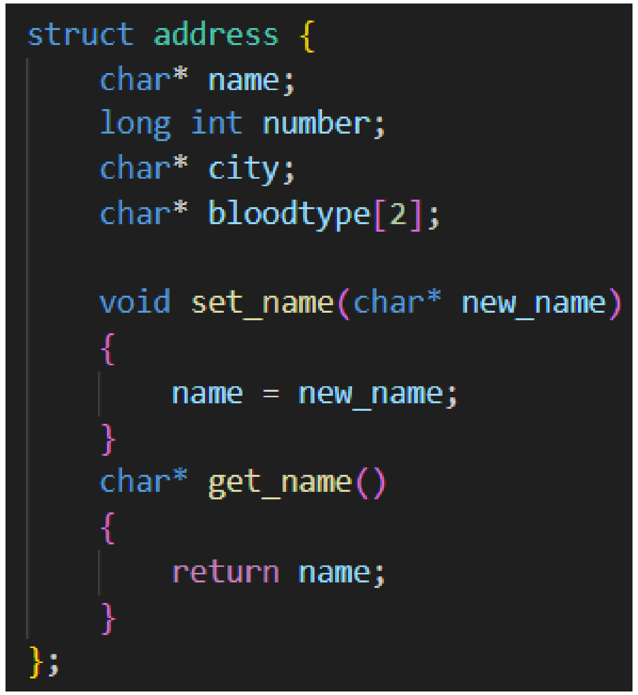
C++ and STL

Dr. Yusuf H. Sahin
Istanbul Technical University

sahinyu@itu.edu.tr

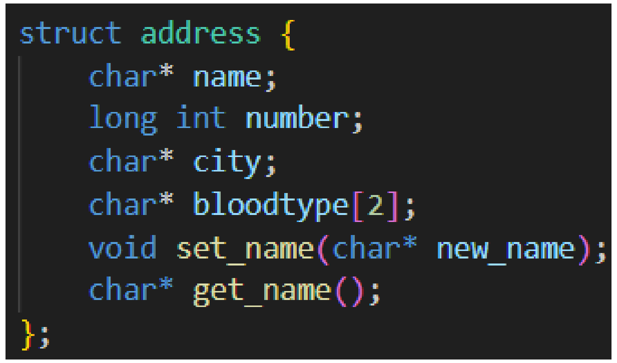
Member Functions

- Functions specified for a given object could be defined/declared within the strcuct block.
    - Especially getters & setters!
- **Function Declaration:** A function declaration, also known as a function prototype, tells the compiler about a function's name, return type, and parameters without the body of the function.

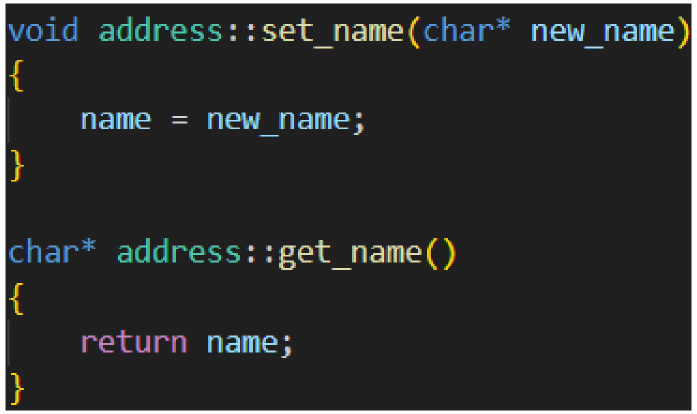


```
struct address {
    char* name;
    long int number;
    char* city;
    char* bloodtype[2];

    void set_name(char* new_name)
    {
        name = new_name;
    }
    char* get_name()
    {
        return name;
    }
};
```

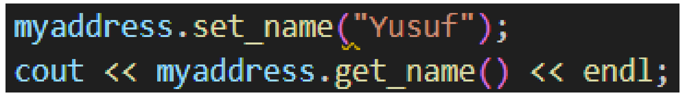
- «::» is used to define a member function of a class outside the class's body.



```
struct address {
    char* name;
    long int number;
    char* city;
    char* bloodtype[2];
    void set_name(char* new_name);
    char* get_name();
};
```



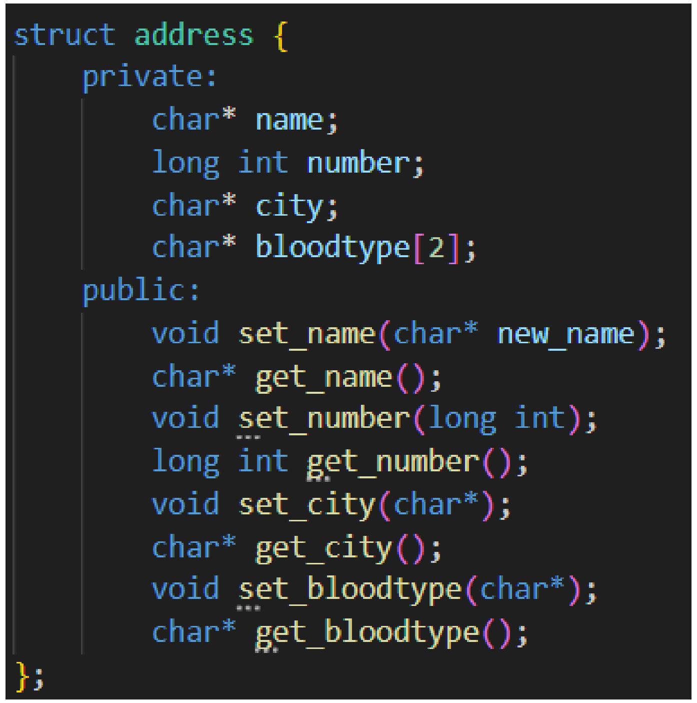
```
void address::set_name(char* new_name)
{
    name = new_name;
}

char* address::get_name()
{
    return name;
}
```



```
myaddress.set_name("Yusuf");
cout << myaddress.get_name() << endl;
```

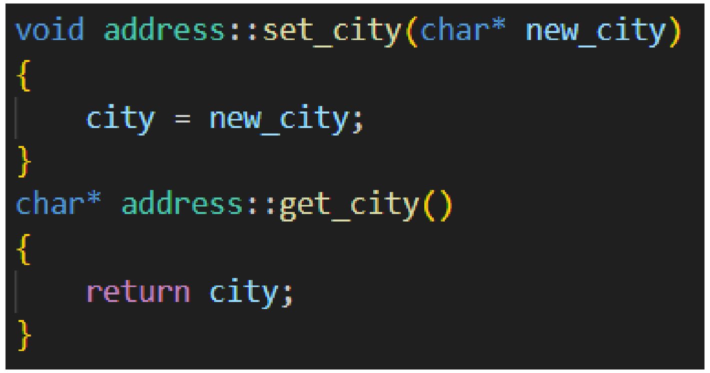
Access Levels

- There are three access specifiers used to set the access level of member functions and variables:
  - **Public:** Default Access level for a struct. Members declared as public are accessible from anywhere in the program where the object of the class is accessible.
  - **Private:** Private members of a struct are accessible only within the struct itself or by its friend classes.
  - **Protected:** Accessible within the struct and by structs or classes that inherit from it. (will be further explained in BLG 252E Object-Oriented Programming)

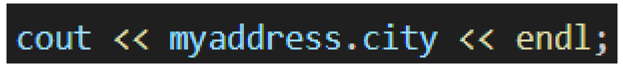


```cpp
struct address {
    private:
        char* name;
        long int number;
        char* city;
        char* bloodtype[2];
    public:
        void set_name(char* new_name);
        char* get_name();
        void set_number(long int);
        long int get_number();
        void set_city(char*);
        char* get_city();
        void set_bloodtype(char*);
        char* get_bloodtype();
};
```

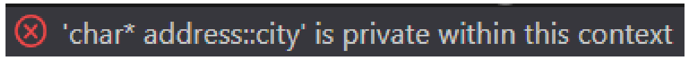
- Getter and setter functions are a common pattern to control access to the private members of a class.



```cpp
void address::set_city(char* new_city)
{
    city = new_city;
}
char* address::get_city()
{
    return city;
}
```



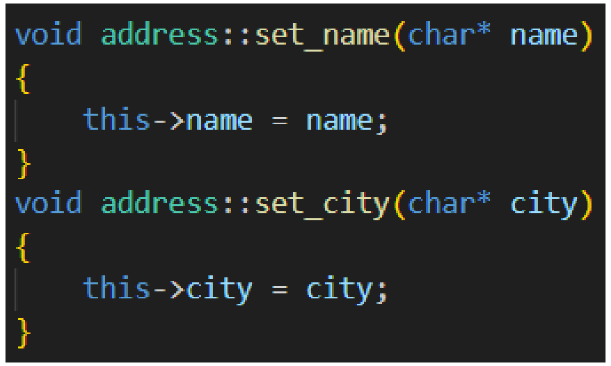
```cpp
cout << myaddress.city << endl;
```



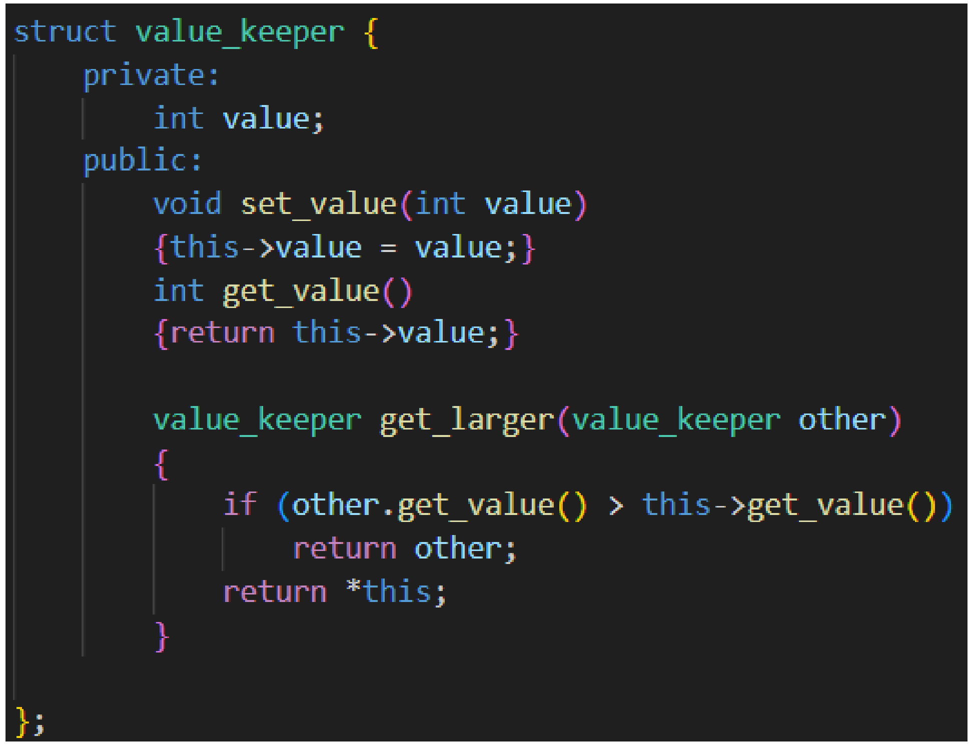
```
'char* address::city' is private within this context
```

Object pointer (This)

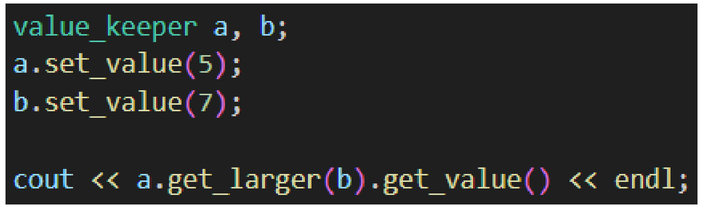
- In C++, the **this** keyword is a pointer which points to the object itself. It is used within a class's member function to refer to the object.



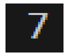
```cpp
void address::set_name(char* name)
{
    this->name = name;
}
void address::set_city(char* city)
{
    this->city = city;
}
```



```cpp
struct value_keeper {
    private:
        int value;
    public:
        void set_value(int value)
        {this->value = value;}
        int get_value()
        {return this->value;}

        value_keeper get_larger(value_keeper other)
        {
            if (other.get_value() > this->get_value())
                return other;
            return *this;
        }

};
```



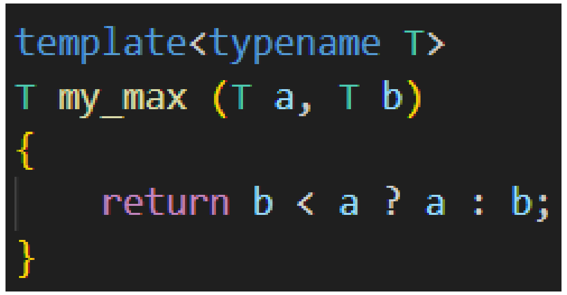
```cpp
value_keeper a, b;
a.set_value(5);
b.set_value(7);

cout << a.get_larger(b).get_value() << endl;
```



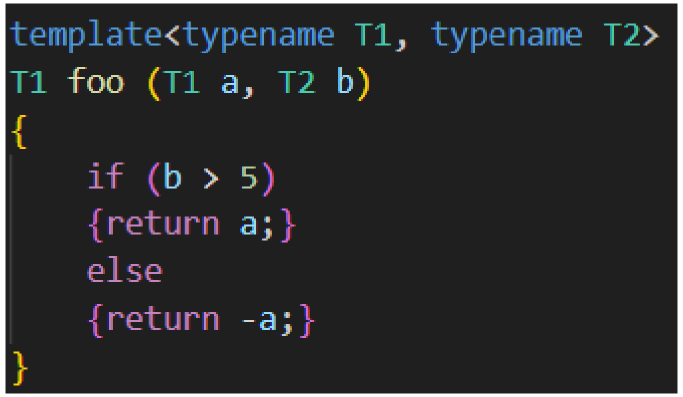
```
7
```

Function Templates

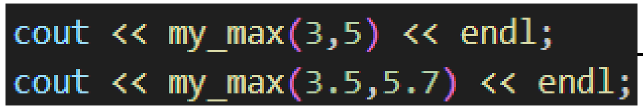
- A function template serves as a blueprint for creating a set of related functions.



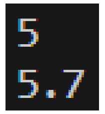
```cpp
template<typename T>
T my_max (T a, T b)
{
    return b < a ? a : b;
}
```



```cpp
template<typename T1, typename T2>
T1 foo (T1 a, T2 b)
{
    if (b > 5)
    {return a;}
    else
    {return -a;}
}
```

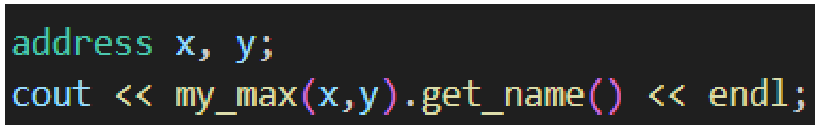


```cpp
cout << my_max(3,5) << endl;
cout << my_max(3.5,5.7) << endl;
```

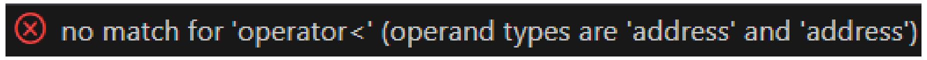


```
5
5.7
```

- The type of these parameters is unspecified and is provided as the template parameter T.
- The types which are not valid for the operations inside the function will raise an error.



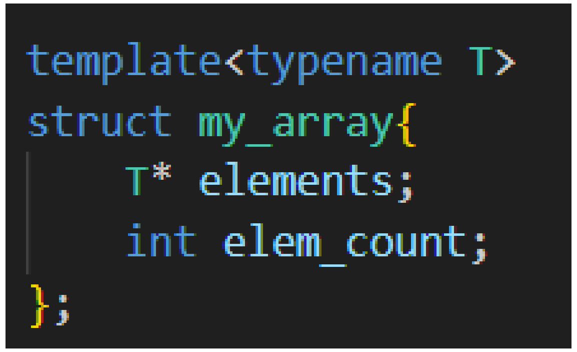
```cpp
address x, y;
cout << my_max(x,y).get_name() << endl;
```



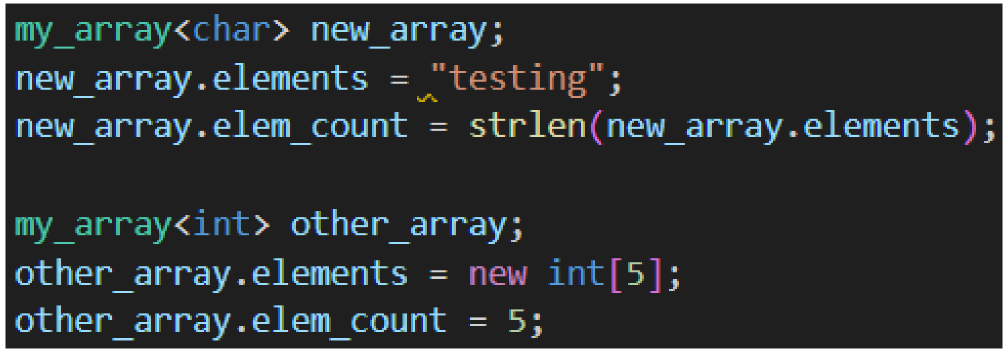
```
no match for 'operator<' (operand types are 'address' and 'address')
```

Struct Templates

- Like functions, structs can be generalized to accept one or more type parameters.

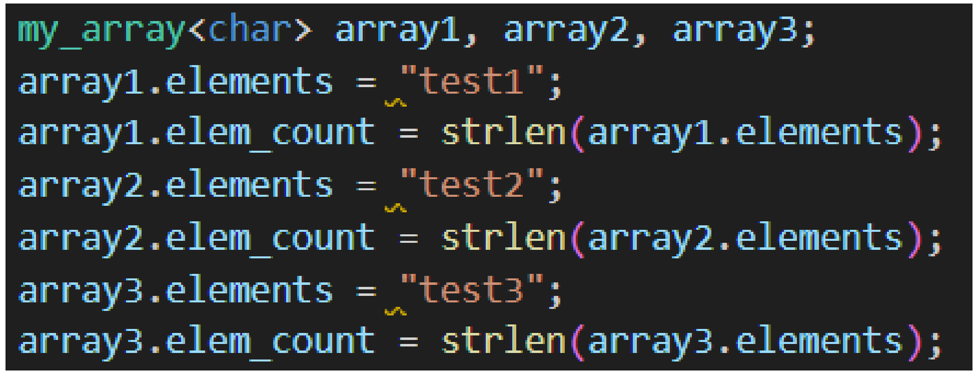


```cpp
template<typename T>
struct my_array{
    T* elements;
    int elem_count;
};
```

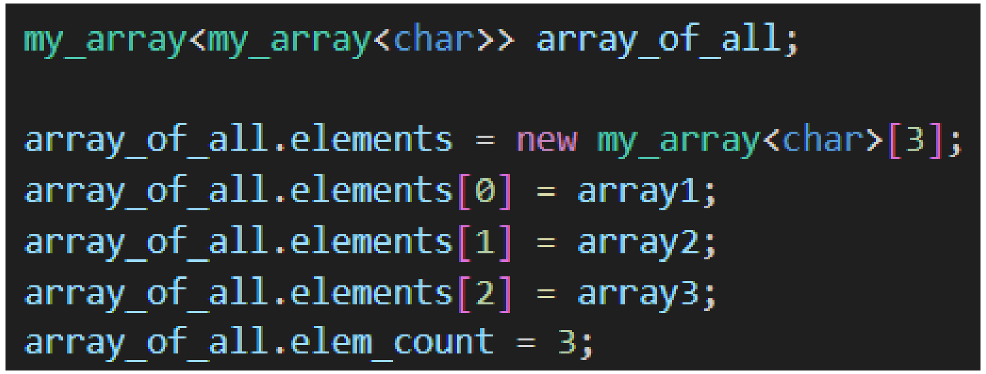


```cpp
my_array<char> new_array;
new_array.elements = "testing";
new_array.elem_count = strlen(new_array.elements);

my_array<int> other_array;
other_array.elements = new int[5];
other_array.elem_count = 5;
```

- Structs of the same type could also be encapsulated using the same template.



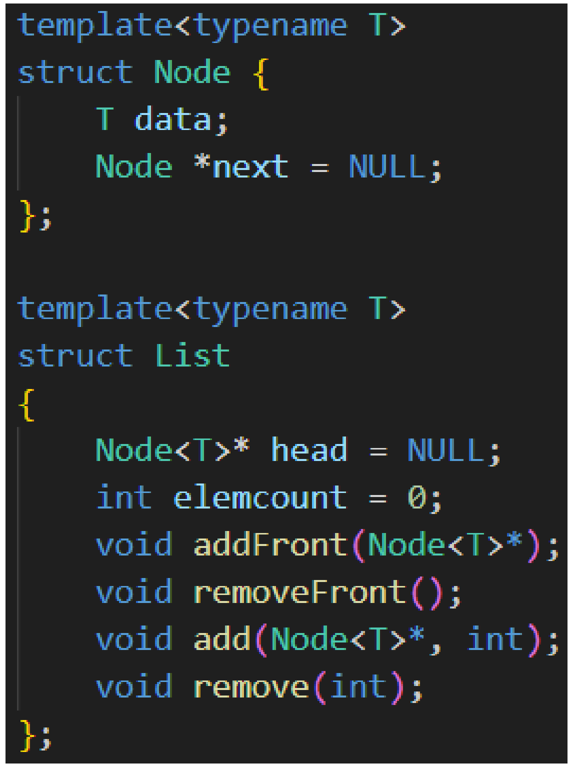
```cpp
my_array<char> array1, array2, array3;
array1.elements = "test1";
array1.elem_count = strlen(array1.elements);
array2.elements = "test2";
array2.elem_count = strlen(array2.elements);
array3.elements = "test3";
array3.elem_count = strlen(array3.elements);
```



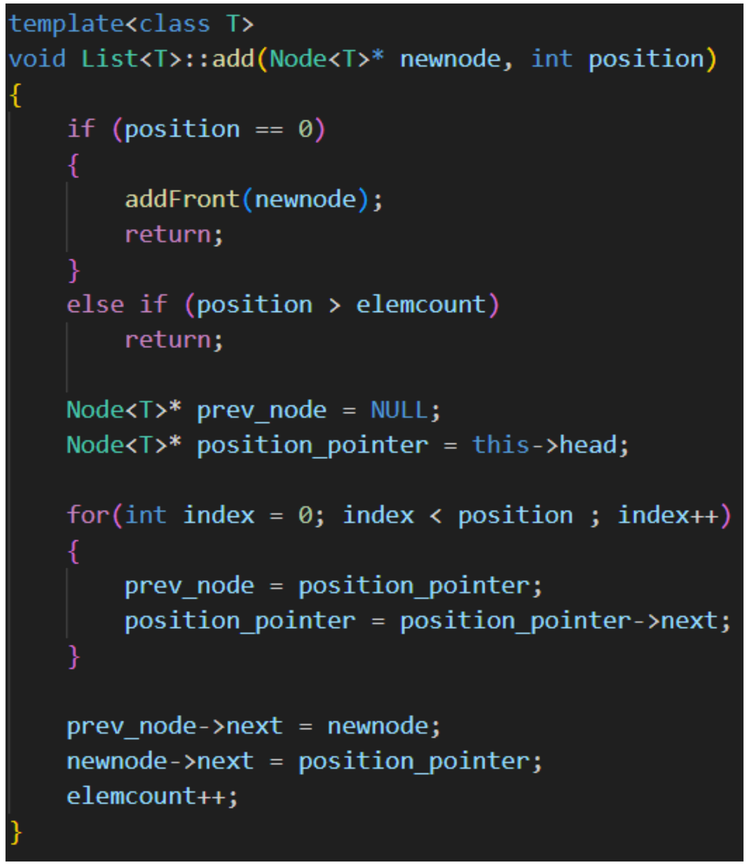
```cpp
my_array<my_array<char>> array_of_all;

array_of_all.elements = new my_array<char>[3];
array_of_all.elements[0] = array1;
array_of_all.elements[1] = array2;
array_of_all.elements[2] = array3;
array_of_all.elem_count = 3;
```

Using templates

- By defining the Node and List structures using templates, we may have the flexibitiy to use different data structures.

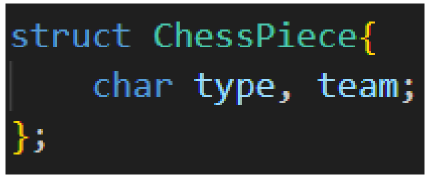


```
template<typename T>
struct Node {
    T data;
    Node *next = NULL;
};

template<typename T>
struct List
{
    Node<T>* head = NULL;
    int elemcount = 0;
    void addFront(Node<T>*);
    void removeFront();
    void add(Node<T>*, int);
    void remove(int);
};
```

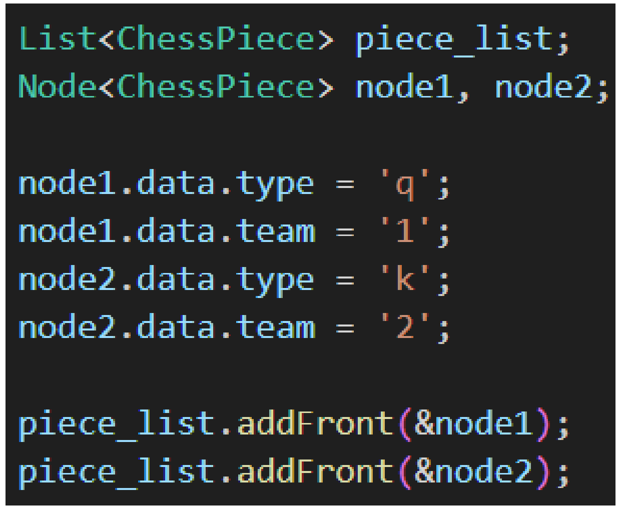


```
template<class T>
void List<T>::add(Node<T>* newnode, int position)
{
    if (position == 0)
    {
        addFront(newnode);
        return;
    }
    else if (position > elemcount)
        return;

    Node<T>* prev_node = NULL;
    Node<T>* position_pointer = this->head;

    for(int index = 0; index < position ; index++)
    {
        prev_node = position_pointer;
        position_pointer = position_pointer->next;
    }

    prev_node->next = newnode;
    newnode->next = position_pointer;
    elemcount++;
}
```

Could be similarly applied to other functions.



```
struct ChessPiece{
    char type, team;
};
```



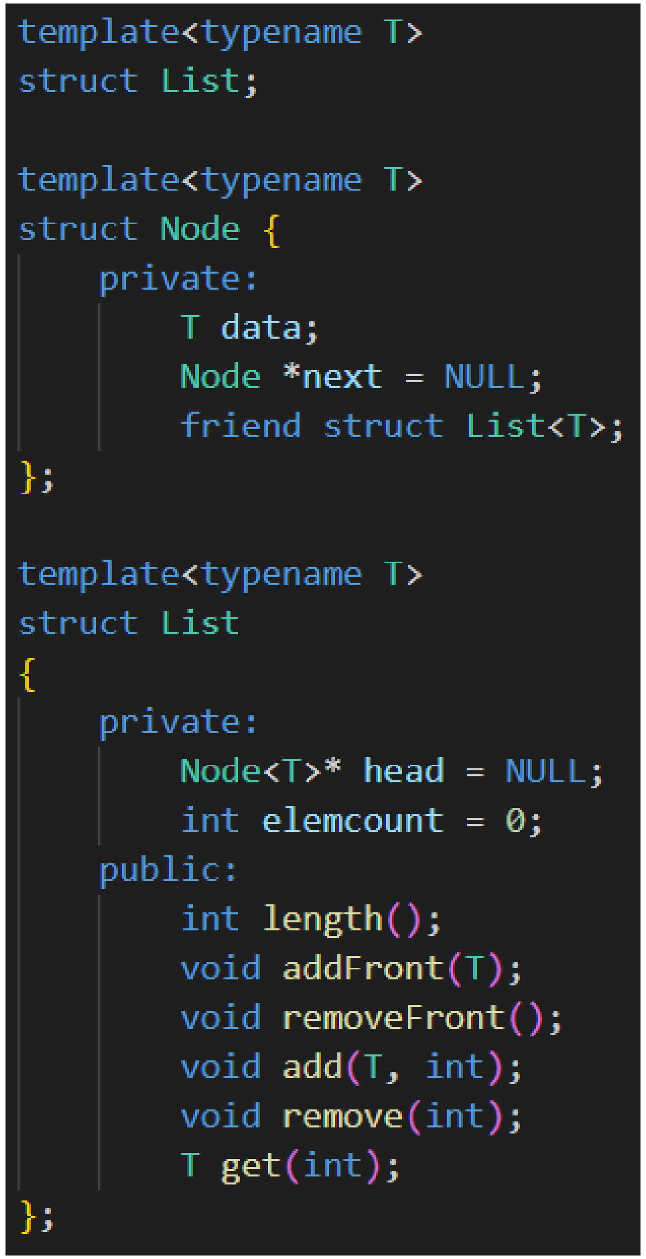
```
List<ChessPiece> piece_list;
Node<ChessPiece> node1, node2;

node1.data.type = 'q';
node1.data.team = '1';
node2.data.type = 'k';
node2.data.team = '2';

piece_list.addFront(&node1);
piece_list.addFront(&node2);
```

Changing access levels

- We may prefer to hide both the Node structure and the underlying pointer mechanism from the users.
- Element count could also be hidden to avoid manipulation.

- A **friend struct** in C++ is given special access privileges to the private and protected members of another class.

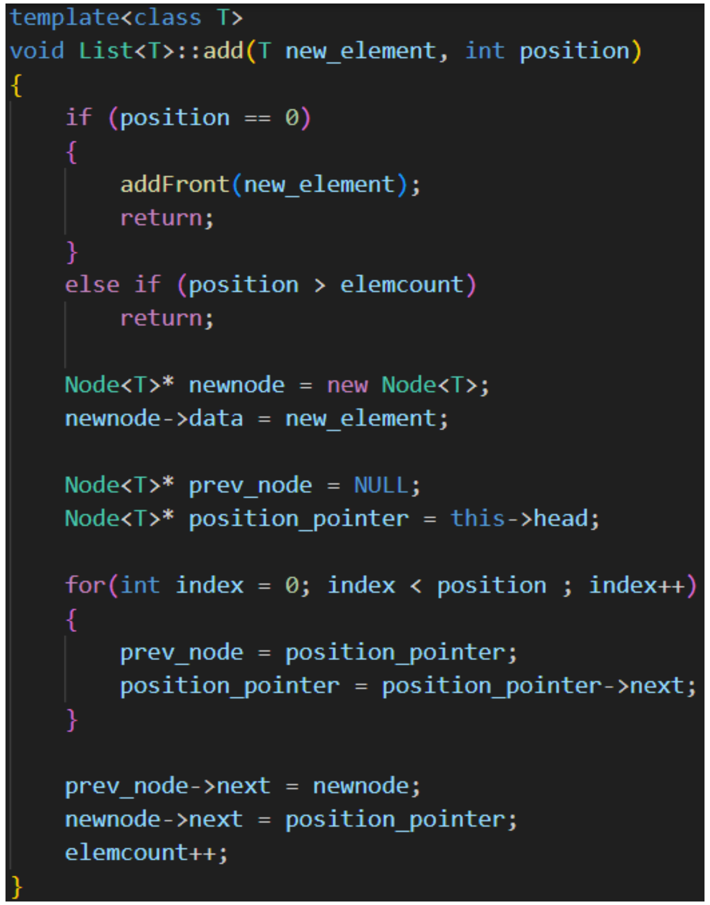
- Instead of creating and using a Node in the main code, we could declare that List struct as a friend of Node struct and thus only List struct could reach the private attributes of the Node.



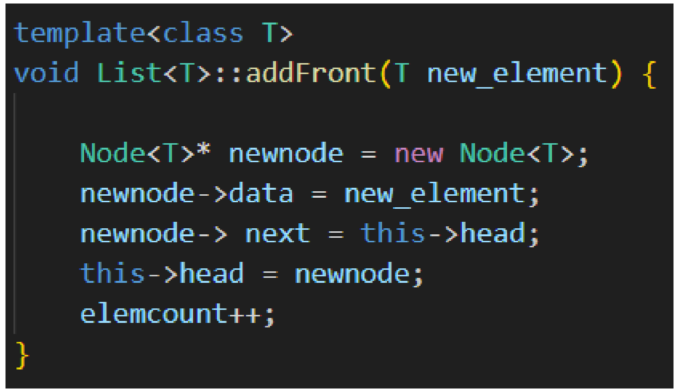
```
template<typename T>
struct List;

template<typename T>
struct Node {
    private:
        T data;
        Node *next = NULL;
        friend struct List<T>;
};

template<typename T>
struct List
{
    private:
        Node<T>* head = NULL;
        int elemcount = 0;
    public:
        int length();
        void addFront(T);
        void removeFront();
        void add(T, int);
        void remove(int);
        T get(int);
};
```

Singly Linked Lists with Templates

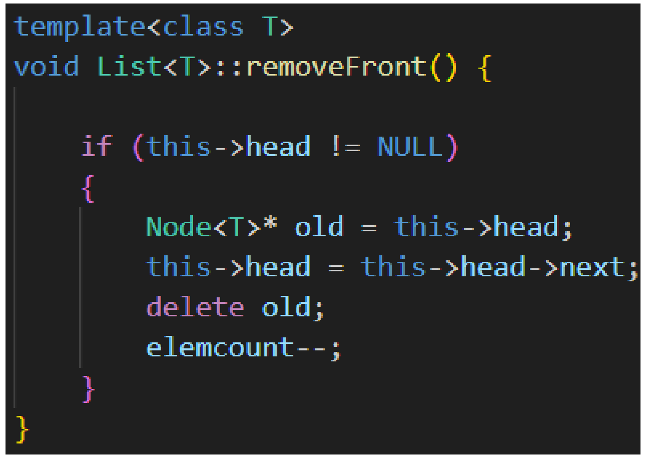
- It is safer to use the Node struct only in the List member functions.



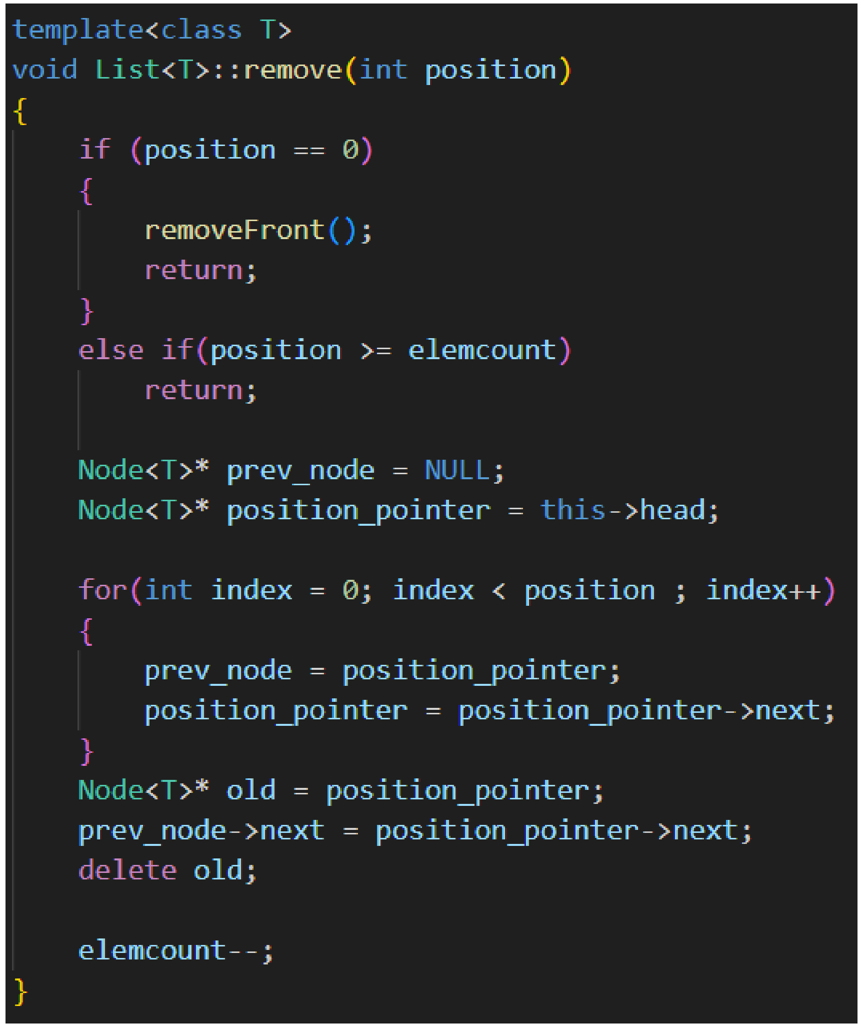
```cpp
template<class T>
void List<T>::add(T new_element, int position)
{
    if (position == 0)
    {
        addFront(new_element);
        return;
    }
    else if (position > elemcount)
        return;

    Node<T>* newnode = new Node<T>;
    newnode->data = new_element;

    Node<T>* prev_node = NULL;
    Node<T>* position_pointer = this->head;

    for(int index = 0; index < position ; index++)
    {
        prev_node = position_pointer;
        position_pointer = position_pointer->next;
    }

    prev_node->next = newnode;
    newnode->next = position_pointer;
    elemcount++;
}
```



```cpp
template<class T>
void List<T>::addFront(T new_element) {

    Node<T>* newnode = new Node<T>;
    newnode->data = new_element;
    newnode-> next = this->head;
    this->head = newnode;
    elemcount++;
}
```

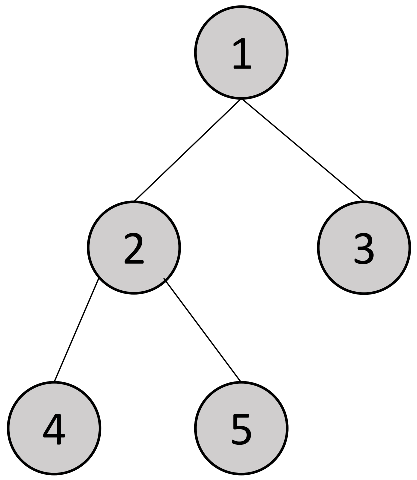


```cpp
template<class T>
void List<T>::removeFront() {

    if (this->head != NULL)
    {
        Node<T>* old = this->head;
        this->head = this->head->next;
        delete old;
        elemcount--;
    }
}
```



```cpp
template<class T>
void List<T>::remove(int position)
{
    if (position == 0)
    {
        removeFront();
        return;
    }
    else if(position >= elemcount)
        return;

    Node<T>* prev_node = NULL;
    Node<T>* position_pointer = this->head;

    for(int index = 0; index < position ; index++)
    {
        prev_node = position_pointer;
        position_pointer = position_pointer->next;
    }
    Node<T>* old = position_pointer;
    prev_node->next = position_pointer->next;
    delete old;

    elemcount--;
}
```

Binary Trees in C++



```
template<typename T>
struct treeNode
{
    T data;
    treeNode* left;
    treeNode* right;
    treeNode* parent;
};
```

```
template<typename T>
treeNode<T>* createNode(T data, treeNode<T>* parent)
{
    treeNode<T>* newtree = new treeNode<T>;
    newtree->data = data;
    newtree->left = NULL;
    newtree->right = NULL;
    newtree->parent = parent;
    return newtree;
}

template<typename T>
treeNode<T>* createNode(T data)
{
    treeNode<T>* newtree = new treeNode<T>;
    newtree->data = data;
    newtree->left = NULL;
    newtree->right = NULL;
    newtree->parent = NULL;
    return newtree;
}
```

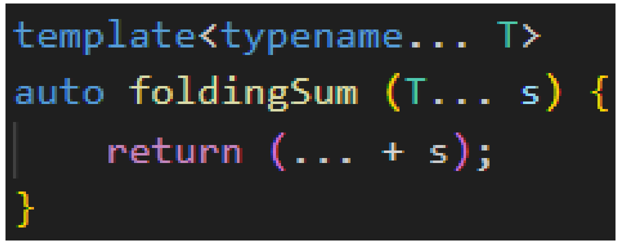
```
treeNode<int>* root = createNode(1);
root->left = createNode(2,root);
root->right = createNode(3,root);
root->left->left = createNode(4,root->left);
root->left->right = createNode(5,root->left);

cout <<root->left->right->parent->parent->data << endl;
```

$\rightarrow$

```
1
```

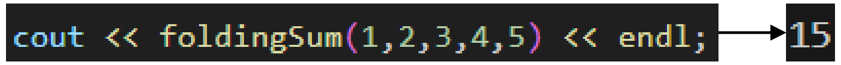
Folding

- Folding is an easy way to apply the same binary operation to many parameters.



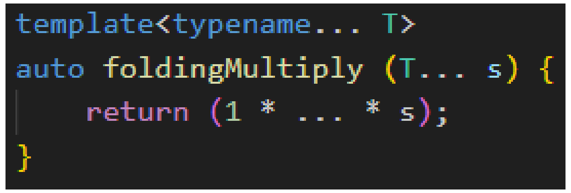
```cpp
template<typename... T>
auto foldingSum (T... s) {
    return (... + s);
}
```

Fold Expression	Evaluation	Example
(... op pack)	(((pack[0] op pack[1]) op pack[2]) ... op pack[N])	return (... + s);
(pack op ...)	(pack[0] op (... (pack[N-1] op pack[N])))	return (s + ...)
(elem op ... op pack)	(((elem op pack[0]) op pack[1]) ... op pack[N])	return (5 + ... + s);
(pack op ... op elem)	(pack[0] op (... (packN op elem)))	return (s + ... + 5);

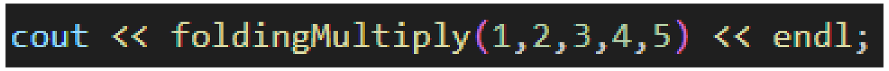


```cpp
cout << foldingSum(1,2,3,4,5) << endl;
```

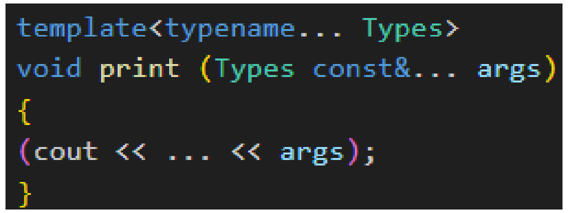
```
15
```



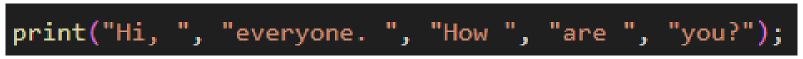
```cpp
template<typename... T>
auto foldingMultiply (T... s) {
    return (1 * ... * s);
}
```



```cpp
cout << foldingMultiply(1,2,3,4,5) << endl;
```



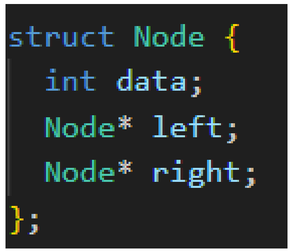
```cpp
template<typename... Types>
void print (Types const&... args)
{
(cout << ... << args);
}
```



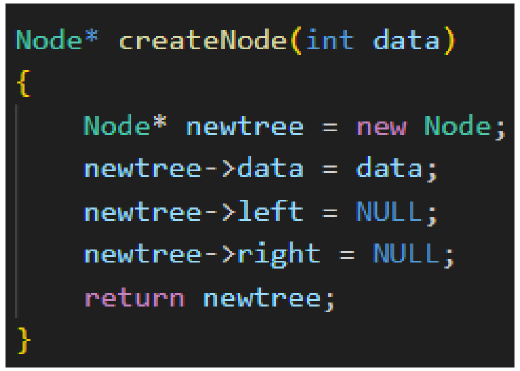
```cpp
print("Hi, ", "everyone. ", "How ", "are ", "you?");
```

Pointer to a data member

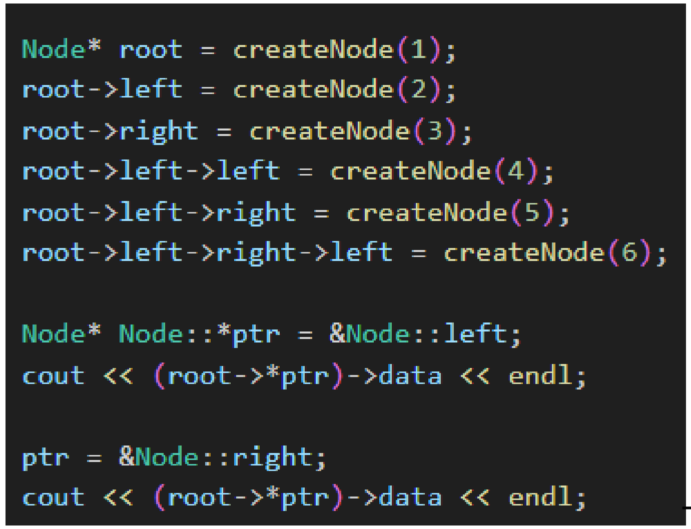
- A pointer to the value of a class member for the object.



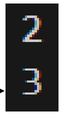
```
struct Node {
    int data;
    Node* left;
    Node* right;
};
```



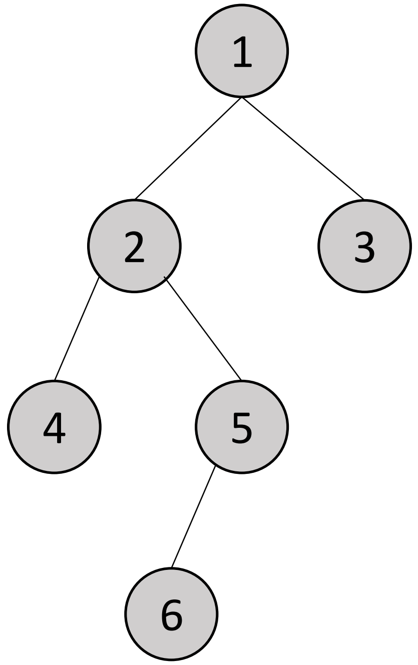
```
Node* createNode(int data)
{
    Node* newtree = new Node;
    newtree->data = data;
    newtree->left = NULL;
    newtree->right = NULL;
    return newtree;
}
```



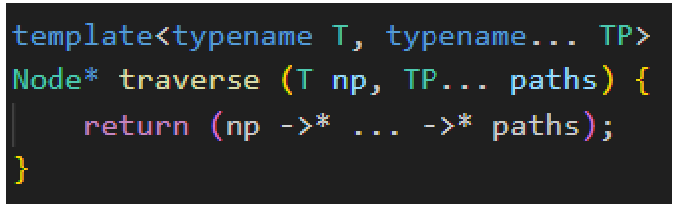
```
Node* root = createNode(1);
root->left = createNode(2);
root->right = createNode(3);
root->left->left = createNode(4);
root->left->right = createNode(5);
root->left->right->left = createNode(6);

Node* Node::*ptr = &Node::left;
cout << (root->*ptr)->data << endl;

ptr = &Node::right;
cout << (root->*ptr)->data << endl;
```



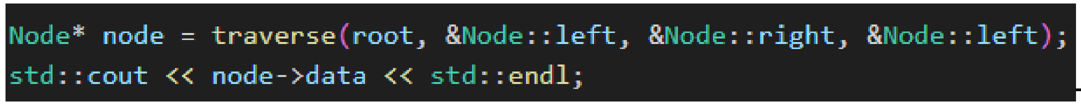
```
2
3
```



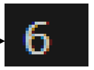
- Folding mechanism could also be applied to tree traversal.



```
template<typename T, typename... TP>
Node* traverse (T np, TP... paths) {
    return (np ->* ... ->* paths);
}
```



```
Node* node = traverse(root, &Node::left, &Node::right, &Node::left);
std::cout << node->data << std::endl;
```

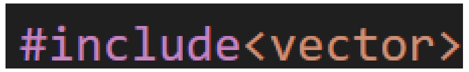


```
6
```

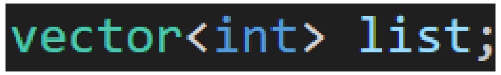
The STL Library

- The STL (Standard Template Library) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates.
  - Vector, queue, stack, map

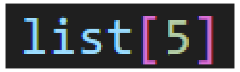
- An STL vector in C++ is a sequence container that encapsulates dynamic size arrays.
  - Unlike static arrays, vectors can automatically resize themselves.
  - Elements in a vector can be accessed using [.].
  - Using «push_back» and «pop_back» operations, it is easy to insert & delete.



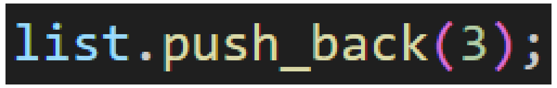
```
#include<vector>
```



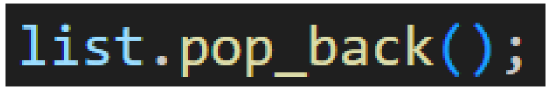
```
vector<int> list;
```



```
list[5]
```

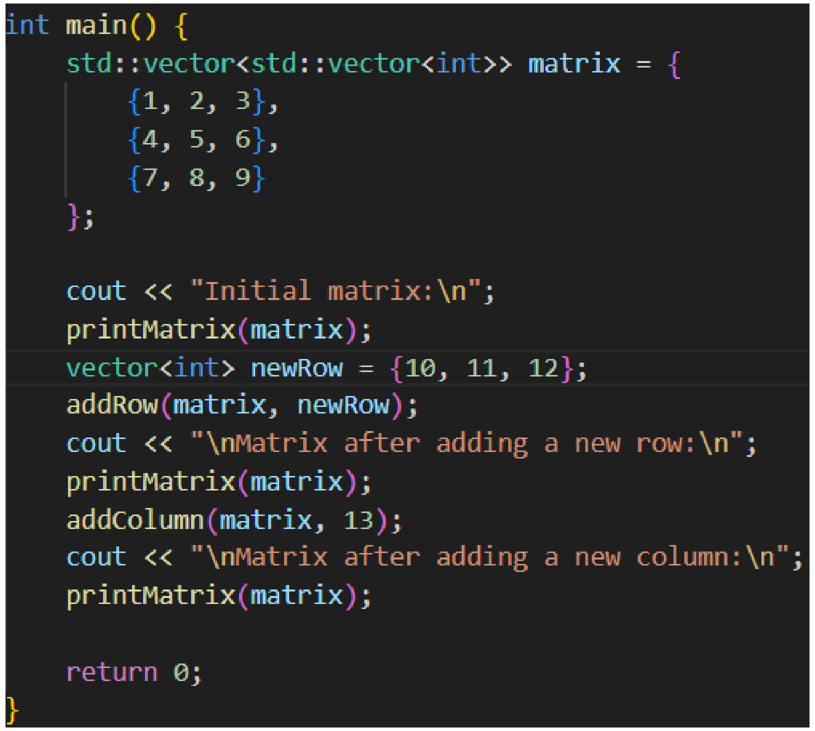


```
list.push_back(3);
```

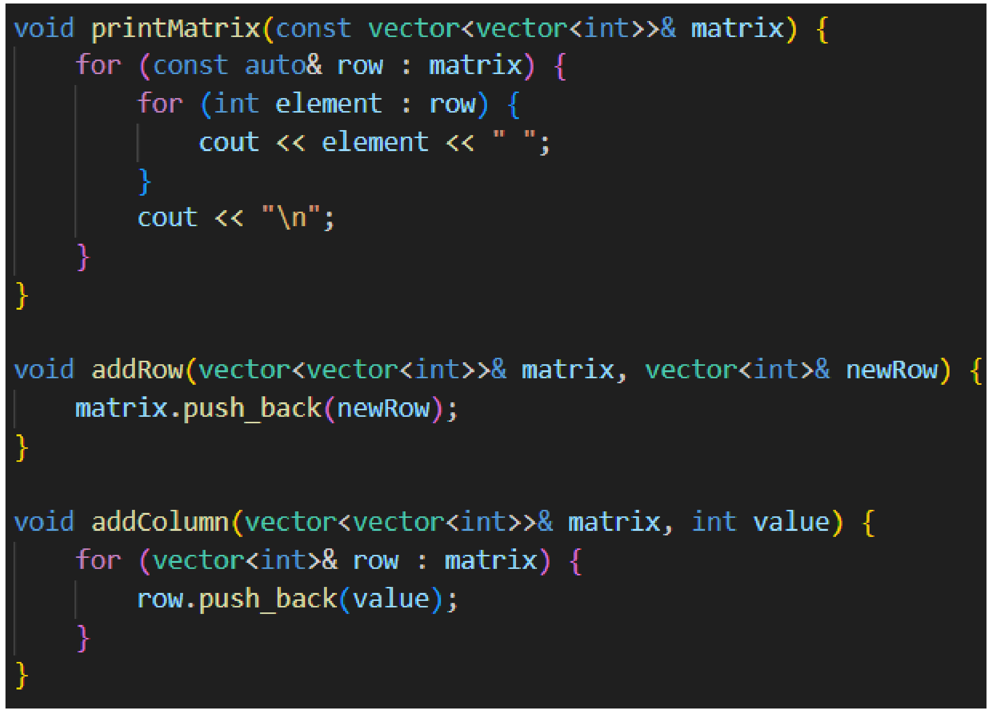


```
list.pop_back();
```

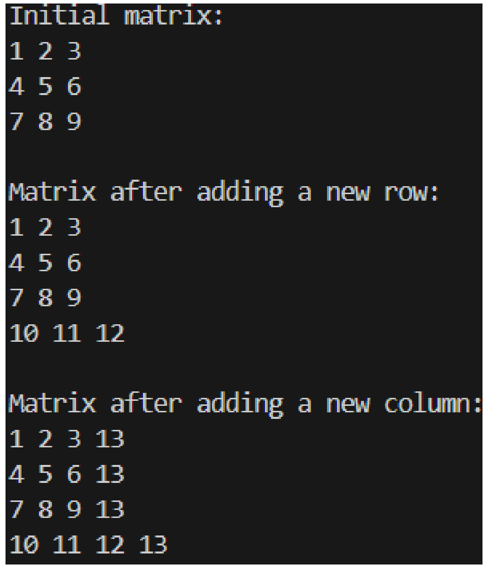
STL Vector



```cpp
int main() {
    std::vector<std::vector<int>> matrix = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    cout << "Initial matrix:\n";
    printMatrix(matrix);
    vector<int> newRow = {10, 11, 12};
    addRow(matrix, newRow);
    cout << "\nMatrix after adding a new row:\n";
    printMatrix(matrix);
    addColumn(matrix, 13);
    cout << "\nMatrix after adding a new column:\n";
    printMatrix(matrix);

    return 0;
}
```



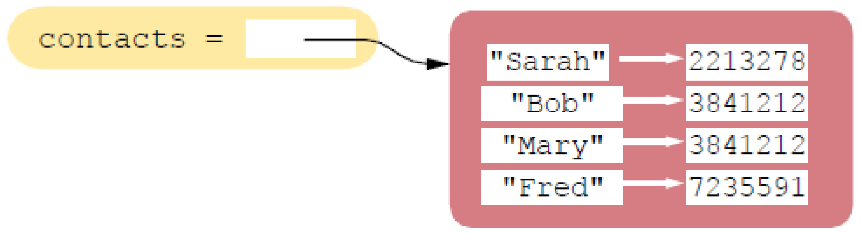
```cpp
void printMatrix(const vector<vector<int>>& matrix) {
    for (const auto& row : matrix) {
        for (int element : row) {
            cout << element << " ";
        }
        cout << "\n";
    }
}

void addRow(vector<vector<int>>& matrix, vector<int>& newRow) {
    matrix.push_back(newRow);
}

void addColumn(vector<vector<int>>& matrix, int value) {
    for (vector<int>& row : matrix) {
        row.push_back(value);
    }
}
```

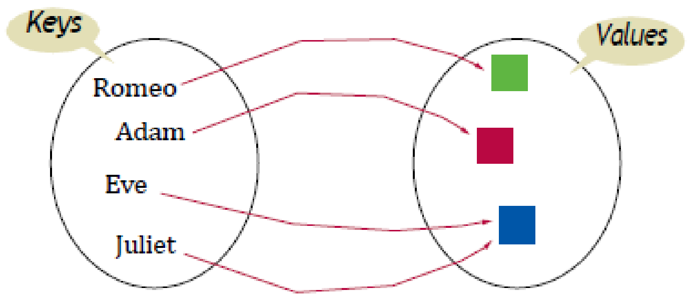


```
Initial matrix:
1 2 3
4 5 6
7 8 9

Matrix after adding a new row:
1 2 3
4 5 6
7 8 9
10 11 12

Matrix after adding a new column:
1 2 3 13
4 5 6 13
7 8 9 13
10 11 12 13
```

Maps

- A map(dictionary) is a structure that stores pairs of keys and values.
- Each key-value pair is called an entry.
- Each key is associated with a specific value, and while each key must be unique within the dictionary, multiple keys can share the same value.
- A map is best used when each key acts as a unique index or address for its corresponding value, essentially serving as a specific location for that value.

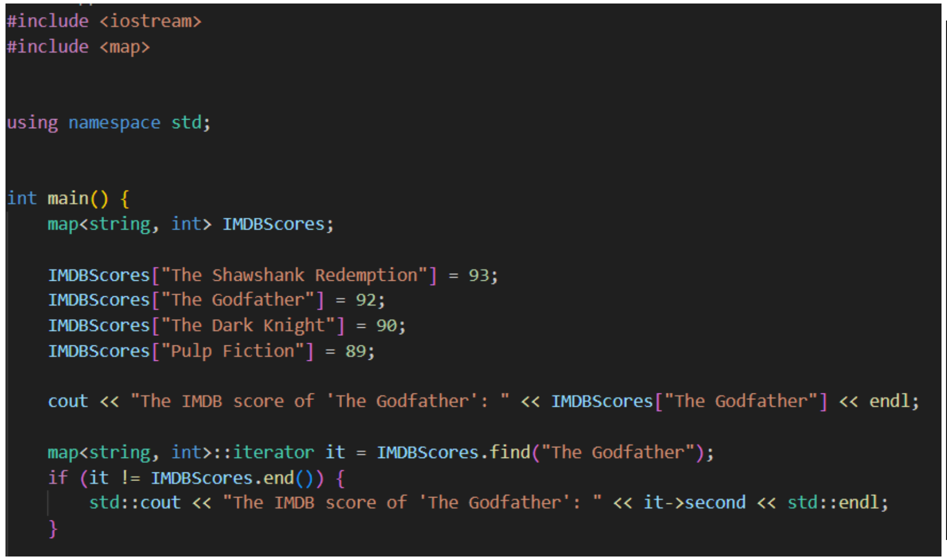




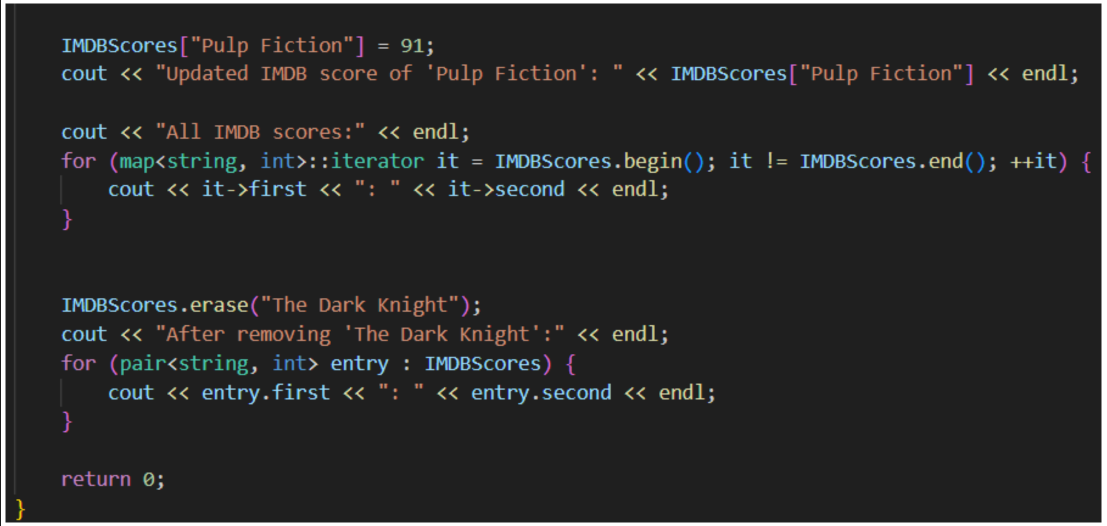
https://horstmann.com/python4everyone/

The STL map Class

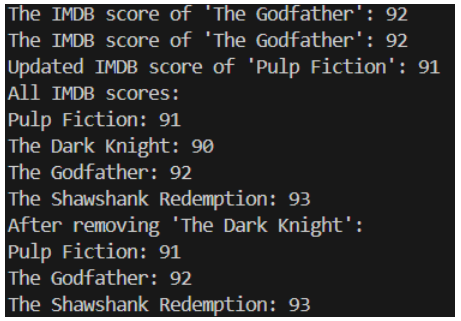
- The C++ Standard Template Library (STL) includes a map implementation, which is referred to simply as map.
- When using an iterator p that points to an element in the map, the key and value of the entry can be accessed with p->first and p->second, respectively.



```cpp
#include <iostream>
#include <map>


using namespace std;


int main() {
    map<string, int> IMDBScores;

    IMDBScores["The Shawshank Redemption"] = 93;
    IMDBScores["The Godfather"] = 92;
    IMDBScores["The Dark Knight"] = 90;
    IMDBScores["Pulp Fiction"] = 89;

    cout << "The IMDB score of 'The Godfather': " << IMDBScores["The Godfather"] << endl;

    map<string, int>::iterator it = IMDBScores.find("The Godfather");
    if (it != IMDBScores.end()) {
        std::cout << "The IMDB score of 'The Godfather': " << it->second << std::endl;
    }
```



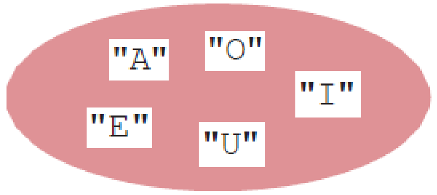
```cpp
    IMDBScores["Pulp Fiction"] = 91;
    cout << "Updated IMDB score of 'Pulp Fiction': " << IMDBScores["Pulp Fiction"] << endl;

    cout << "All IMDB scores:" << endl;
    for (map<string, int>::iterator it = IMDBScores.begin(); it != IMDBScores.end(); ++it) {
        cout << it->first << ": " << it->second << endl;
    }


    IMDBScores.erase("The Dark Knight");
    cout << "After removing 'The Dark Knight':" << endl;
    for (pair<string, int> entry : IMDBScores) {
        cout << entry.first << ": " << entry.second << endl;
    }

    return 0;
}
```

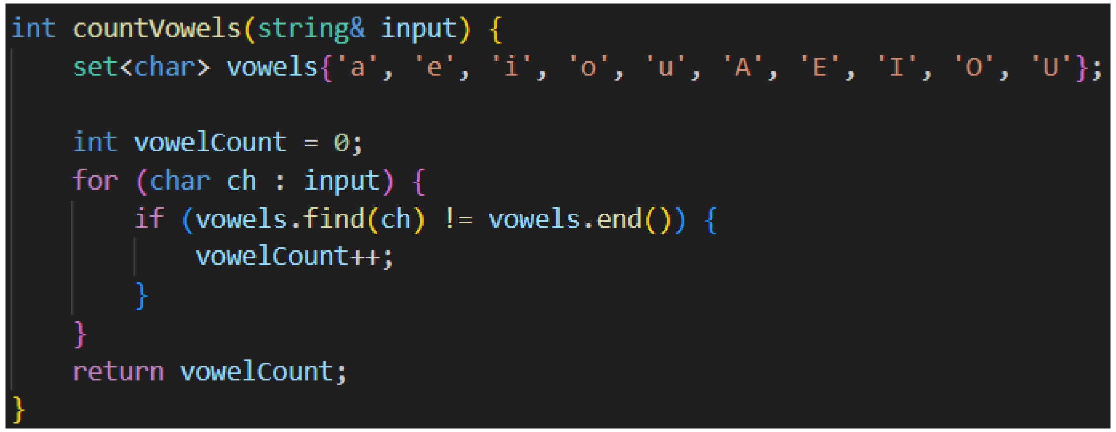


```
The IMDB score of 'The Godfather': 92
The IMDB score of 'The Godfather': 92
Updated IMDB score of 'Pulp Fiction': 91
All IMDB scores:
Pulp Fiction: 91
The Dark Knight: 90
The Godfather: 92
The Shawshank Redemption: 93
After removing 'The Dark Knight':
Pulp Fiction: 91
The Godfather: 92
The Shawshank Redemption: 93
```

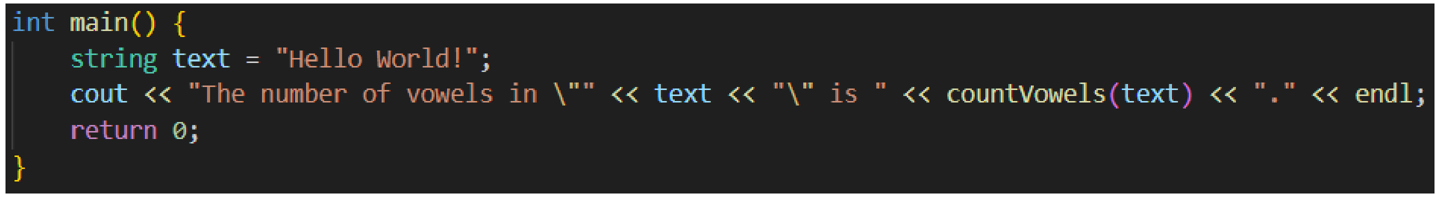
Sets

- A set is a collection of unique elements.
- Sets do not maintain any specific order of elements or associate them with keys.





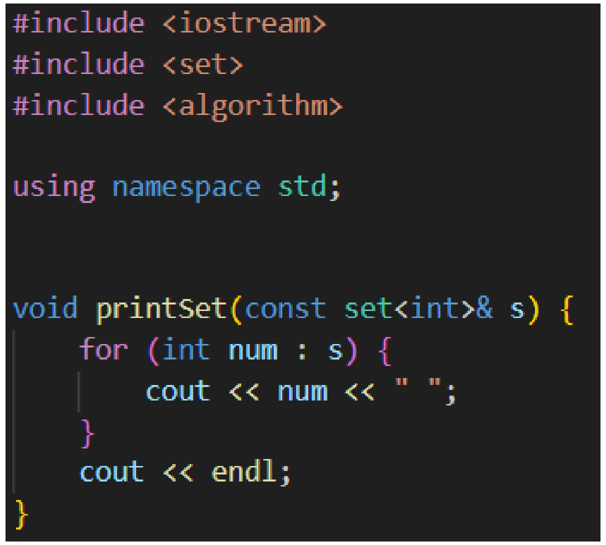
```cpp
int countVowels(string& input) {
    set<char> vowels{'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'};

    int vowelCount = 0;
    for (char ch : input) {
        if (vowels.find(ch) != vowels.end()) {
            vowelCount++;
        }
    }
    return vowelCount;
}
```



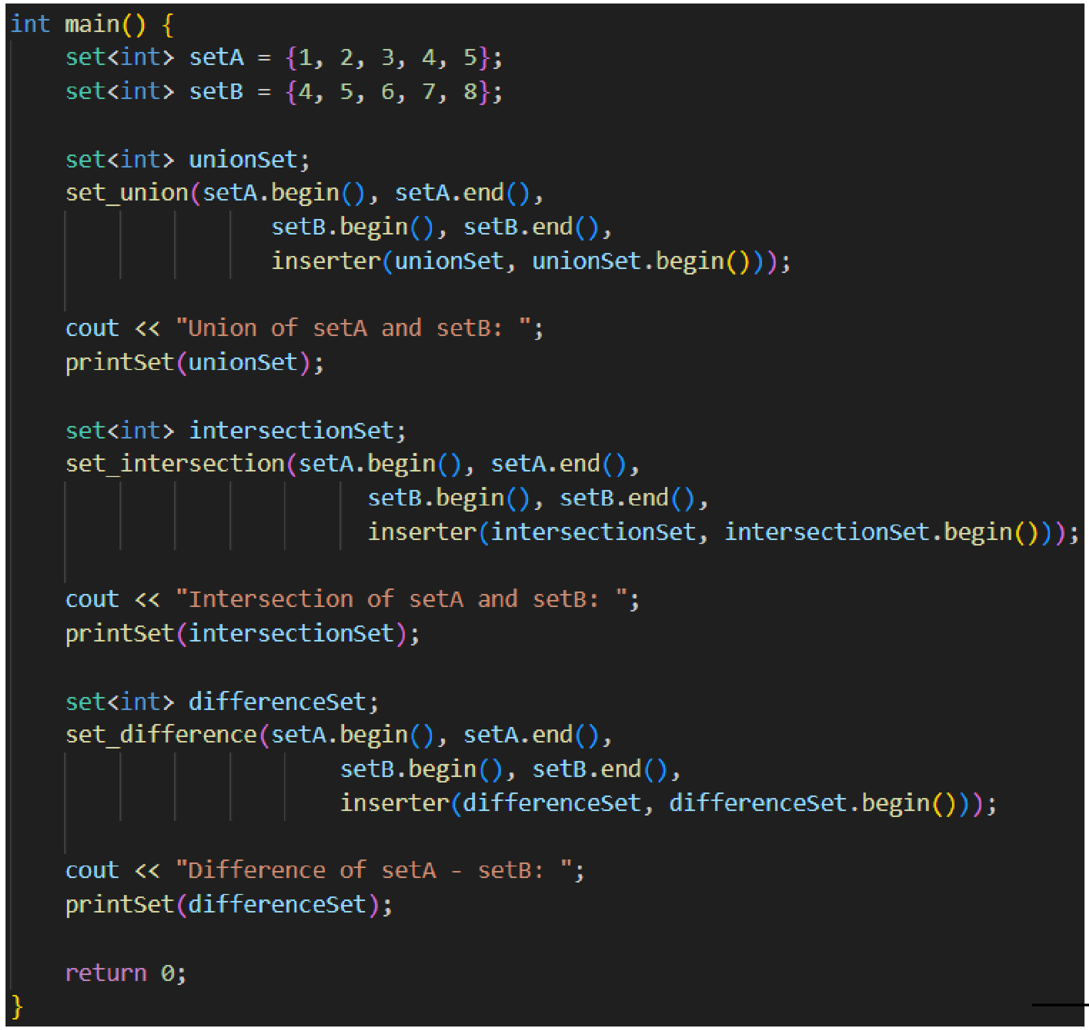
```cpp
int main() {
    string text = "Hello World!";
    cout << "The number of vowels in \"" << text << "\" is " << countVowels(text) << "." << endl;
    return 0;
}
```

Sets

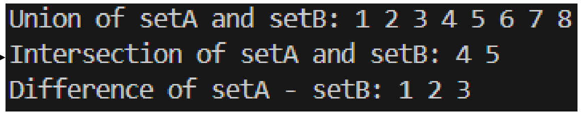
- Union, intersection and subtraction operations could be defined on the set.



```cpp
#include <iostream>
#include <set>
#include <algorithm>

using namespace std;


void printSet(const set<int>& s) {
    for (int num : s) {
        cout << num << " ";
    }
    cout << endl;
}
```



```cpp
int main() {
    set<int> setA = {1, 2, 3, 4, 5};
    set<int> setB = {4, 5, 6, 7, 8};

    set<int> unionSet;
    set_union(setA.begin(), setA.end(),
                      setB.begin(), setB.end(),
                      inserter(unionSet, unionSet.begin()));

    cout << "Union of setA and setB: ";
    printSet(unionSet);

    set<int> intersectionSet;
    set_intersection(setA.begin(), setA.end(),
                              setB.begin(), setB.end(),
                              inserter(intersectionSet, intersectionSet.begin()));

    cout << "Intersection of setA and setB: ";
    printSet(intersectionSet);

    set<int> differenceSet;
    set_difference(setA.begin(), setA.end(),
                            setB.begin(), setB.end(),
                            inserter(differenceSet, differenceSet.begin()));

    cout << "Difference of setA - setB: ";
    printSet(differenceSet);

    return 0;
}
```



```
Union of setA and setB: 1 2 3 4 5 6 7 8
Intersection of setA and setB: 4 5
Difference of setA - setB: 1 2 3
```