OBJECT ORIENTED PROGRAMMING CONCEPTS USING C++

[ET.]

Prepared by: AMITABH SRIVASTAVA

VIRTUAL FUNCTION

- A virtual function is a member function that is always **overridden** in the derived class. This especially applies to cases where a pointer of base class points to an object of a derived class.

- **Example-**

1

```
class Base {
  public:
   void show() {
       cout << "Base Function called!" << endl;
   }
};

class Derived : public Base {
  public:
   void show() {
       cout << "Derived Function called!" << endl;
   }
};
```

2

```
int main() {
    Derived derived1;

    // pointer of Base type
    // that points to derived1
    Base* base1 = &derived1;

    // calls member function
    // of Base class
    base1->show();
}
```

VIRTUAL FUNCTION

- To override the member function of Base class, we need to use the keyword **'virtual'** to make the Base class function as virtual function.

- **Example-**

```
class Base {
   public:
    virtual void show() {
        cout << "Base Function called!" << endl;
    }
};

class Derived : public Base {
   public:
    void show() {
        cout << "Derived Function called!" << endl;
    }
};
```

```
int main() {
    Derived derived1;

    // pointer of Base type
    // that points to derived1
    Base* base1 = &derived1;

    // calls member function
    // of Derived class
    base1->show();
}
```

PURE VIRTUAL FUNCTION

Scenario-

- Sometimes implementation of all functions cannot be provided in a base class because we don't know the exact implementation at that moment. Suppose, we have a class Shape as base class. If we want to calculate the area of this shape, we cannot apply the correct formula because we don't know the exact shape for which we need to calculate the area.
- Therefore, in base class 'Shape', we will create a member function **calculateArea()** without any body and add = 0 at the end. Also, we'll add the keyword **'virtual'** to make it **pure virtual function**.

```
class Shape {
    public:

      // Pure virtual function
      virtual void calculateArea() = 0;
};
```


Class which contains the pure virtual function is known as **Abstract class**.

Actual implementation of pure virtual function is always in derived class.

ABSTRACT CLASS: EXAMPLE

1

```
// Abstract class
class Shape {
   protected:
    double dimension;

   public:
    void getDimension() {
        cin >> dimension;
    }

    // pure virtual Function
    virtual double calculateArea() = 0;
};
```

3

```
int main() {
    Square square;
    Circle circle;

    cout << "Enter Length of square: ";
    square.getDimension();
    cout << "Area of square: " << square.calculateArea() << endl;

    cout << "\nEnter Radius of circle: ";
    circle.getDimension();
    cout << "Area of circle: " << circle.calculateArea() << endl;
}
```

2

```
class Square : public Shape {
   public:
    double calculateArea() {
        return dimension * dimension;
    }
};

class Circle : public Shape {
   public:
    double calculateArea() {
        return 3.14 * dimension * dimension;
    }
};
```

Prepared by: AMITABH SRIVASTAVA

BINDINGS IN C++

- Binding creates **a bridge** between a function call and its corresponding function definition.

- Both the function definition and function calls are stored in the memory at **separate addresses**. Therefore, we need some technique to match the appropriate function call with its function definition.

- The process of matching a specific function call to its respective function definition is known as **binding**.

- There are two types of bindings-
  - Early binding or Static binding
  - Late binding or Dynamic binding

Prepared by: AMITABH SRIVASTAVA

EARLY/STATIC BINDING

- Binding at compile-time is known as **Early/Static** binding.
- Early binding ensures the linking of function call and its function definition at compile-time only.
- Function overloading and Operator overloading falls under the category of Early binding.

LATE/DYNAMIC BINDING

- Binding at run-time is known as **Late/Dynamic** binding.

- There may be a situation in our program when the compiler cannot get all the information at compile-time to resolve a function call. These function calls are linked at runtime.

- Since everything is postponed till runtime, it is also known as Late binding.

- It is implemented using virtual functions.

VIRTUAL DESTRUCTOR

- **Virtual Destructor** is used to release or free up the memory used by the derived class (child class) object when the derived class object is removed from the memory using the pointer object of base class.

- Virtual destructors always execute from derived class to base class.

- The destructor of the parent class uses a **'virtual'** keyword before its name to make itself a virtual destructor to ensure that the destructor of both the base class and derived class should be called at the run time.

  virtual ~Base() { }

- The derived class's destructor is called first, and then the base class releases the memory occupied by both destructors.

- **'delete'** operator/keyword is used to delete the pointer object.

VIRTUAL DESTRUCTOR: EXAMPLE

1

```cpp
class Base {
    public: Base() {
        cout << "Base class constructor!" << endl;
    }
    // Defining virtual destructor.
    virtual ~Base() {
        // At last, it will be printed.
        cout << "Base class destructor!" << endl;
    }
};

class Child: public Base {
    public: Child() {
            cout << "Child class constructor!" << endl;
        }
        ~Child() {
            cout << "Child class destructor!" << endl;
        }
};
```

2

```cpp
int main() {
    // Object refers to the Base class.
    Base *b = new Child;

    // Deleting the pointer object.
    delete b;
}
```

Prepared by: AMITABH SRIVASTAVA

EXCEPTION HANDLING

- When a compiler does not find any errors in a program, it successfully completes the compilation of the program and builds the executable file.

- But there may be a situation where your program doesn't have any syntax errors but during its execution it raises some abnormal conditions like, divide by zero, etc.

- Such errors which arise during the run-time are known as **Exceptions** due to which the program stops its execution or may give some unexpected results.

- **Exceptions** are raised when some internal events occur which changes the normal flow of the program while executing. Or in simple terms you can say when a C++ code comes across a condition it cannot handle at run-time, it produces / raises an exception.

- To handle such exceptions, we have **try** and **catch** blocks in C++.

TRY - CATCH BLOCKS

- When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unexpected things.

- When an error occurs, C++ will normally stops and generates an error message or throws an exception (error).

- Exception handling in C++ consists of three keywords- **try**, **throw** and **catch**.

- **try-** The 'try block' is used to hold the code that is expected to throw some exception.

- **throw-** 'throw' is used to throw exceptions to the exception handler which further communicates the error.

- **catch-** The 'catch block' is used to catch and handle the error(s) thrown from the try block.

SYNTAX

- Following is the syntax to write the exception handling code-

```
try
{
    // code that can raise exception
    throw exception;
}
catch(exception e)
{
    // code for handling exception
}
```

EXAMPLE CODE-1

1

```
int Division(int x, int y)
{
    if(y == 0)
    {
        throw "Division by Zero not allowed!";
    }
    return int(x/y);
}
```

2

```
int main()
{
    int x = 70;
    int y = 0;
    int result;
    try
    {
        result = Division(x, y);
        cout << "Result is: " << result << endl;
    }
    catch(const char *err)
    {
        cout << err << endl;
    }
}
```

Prepared by: AMITABH SRIVASTAVA

EXAMPLE CODE-2

1

```
int Division(int x, int y)
{
    string err = "Division by Zero!!";
    if(y == 0)
    {
        throw err;
    }
    return int(x/y);
}
```

2

```
int main()
{
    int x = 70;
    int y = 10;
    int result;
    try
    {
        result = Division(x, y);
        cout << "Result is: " << result << endl;
    }
    catch(string err)
    {
        cout << err << endl;
    }
}
```

EXAMPLE CODE-3

```
int main()
{
    try
    {
        int age;
        cout << "Enter your age: ";
        cin >> age;
        if (age >= 18)
        {
            cout << "Access granted!";
        }
        else { throw(age); }
    }
    catch (int x)
    {
        cout << "Access denied!, Age is: " << x << endl;
    }
}
```

CATCH ALL BLOCK

```
int main()
{
    try
    {
        int age = 16;
        if (age >= 18)
        {
            cout << "Access granted.";
        }
        else
        {
            // throwing any random value.
            throw 123;
        }
    }
    catch (...)
    {
        cout << "Access denied!" << endl;
    }
}
```

catch(…) block is used to catch all types of exceptions.

THROWING FROM A FUNCTION

Specifying Exception:

```
void myFunc(int a, int b) throw (int, int) // Dynamic Exception specification
{
    if (a <= 0)
        throw(a);
    else if(b > 100)
        throw(b);
    else
        cout << "Sum is: " << a + b << endl;
}

int main()
{
    try {
    myFunc(2,700);
    }
    catch(int x) {
        cout << "Invalid value: " << x;
    }
}
```

RETHROWING EXCEPTION

Section-1

```
void myHandler()
{
    try
    {
        throw "some exception...";
    }
    catch (const char*)
    {
    cout << "Caught exception inside myHandler()!\n";
    throw; //rethrowing exception
    }
}
```

Section-2

```
int main()
{
    cout<< "main() started!\n";
    try
    {
        myHandler();
    }
    catch(const char*)
    {
      cout << "Caught exception inside main()!\n";
    }
       cout << "main() end!";
}
```

PRACTICE CODE

Q. Create an array of type double with length 4 and store any 4 values in it. Ask from the user to enter an index number. If index is invalid, throw an exception "**Error: Array out of bounds!**" otherwise, print all the values from array.

1

```
int main() {

    double numerator, denominator, arr[4] = {1.5, 3.56, 56.7, 87.7};
    int index;

    cout << "Enter array index: ";
    cin >> index;

    try {

        // throw exception if array out of bounds
        if (index >= 4)
            throw "Error: Array out of bounds!";
        else
        {
            for(double values: arr)
            {
                cout << values << " ";
            }
        }
    }
```

2

```
    // catch "Array out of bounds" exception
    catch (const char* msg) {
        cout << msg << endl;
    }

    // catch any other exception
    catch (...) {
        cout << "Unexpected exception!" << endl;
    }
}
```

TEMPLATES

- Templates are used to reduce the code redundancy.

- In templates, data types are passed as a parameter so that we don't need to write the same code for different data types.

- Templates are expanded at compile time.

- Source code contains only single template of a function or class but compiled code may contain multiple copies of the same function/class with different data types of parameters.

- Two types of templates-
    - Function templates
    - Class templates

FUNCTION TEMPLATES

- We can create a single function template to work with different data types.

- **Keywords used-** template and typename

- **Function template syntax-**

```
template <typename T>
T functionName(T param1, T param2, …, T paramN)
{
    // code
}
```

- Here, **T** is a template argument that can accept same set of data types on different calls.

- When the actual arguments passed to the **functionName()**, the compiler generates a version of **functionName()** for those data types passed.

EXAMPLES

Example-1:

```cpp
template <typename T>
T add(T num1, T num2) {
    return (num1 + num2);
}

int main() {
    int result1;
    double result2;

    // calling with int parameters
    result1 = add<int>(2, 3);
    cout << "2 + 3 = " << result1 << endl;

    // calling with double parameters
    result2 = add<double>(2.2, 3.3);
    cout << "2.2 + 3.3 = " << result2 << endl;
}
```

Example-2:

```cpp
template <typename T>
void add(T num1, T num2) {
    cout << "Sum is: " << num1 + num2 << endl;
}

int main()
{

    // calling with int parameters
    add<int>(2, 3);

    // calling with double parameters
    add<double>(2.2, 3.3);
}
```

PRACTICE CODE

Q. Define a function template called "findMax()" to compare two parameters of integer, double and char types?

```
template <typename T>
T findMax(T x, T y)
{
    return (x > y) ? x : y;
}

int main()
{
    // Call findMax for int
    cout << findMax<int>(200, 150) << endl;

    // call findMax for double
    cout << findMax<double>(35.5, 17.5) << endl;

    // call findMax for char
    cout << findMax<char>('c', 'p') << endl;
}
```

TEMPLATE FUNCTION OVERLOADING

Passing parameters of different datatypes-

```
template<typename T>
void myFunc(T x, T y)
{
    cout << "x: " << x << " y: " << y << endl;
}

void myFunc(int w, char z)
{
    cout << "w: " << w << " z: " << z << endl;
}

int main() {
   //calling myFunc for integers
   myFunc(1, 2);

   //calling myFunc for chars
   myFunc('a', 'b');

   //calling myFunc for int and char
   //template function overloading
   myFunc(1, 'b');
}
```

CLASS TEMPLATES

- We can create a single class template to work with different data types.
- **Keywords used-** template and class
- **Function template syntax-**

```
template <class T>
class className {
private:
    T var;
    … …. …..
public:
    T functionName(T arg);
    … …. …..
};
```

- Here, **T** is a template argument which is a placeholder for the data types used.
- Inside class body, a member variable **var** and a member function **functionName()** are both of type T.

EXAMPLE CODE

Section-1:

```
template <class T>
class Number {
private:
    T num;    // Variable of type T
public:
    Number(T n) { num = n; }

    T getNum() {
        return num;
    }
};
```

Section-2:

```
int main() {

    // create object with int type
    Number<int> numberInt(7);

    // create object with double type
    Number<double> numberDouble(7.7);

    // create object with char type
    Number<char> numberChar('5');

    cout << "Number(int): " << numberInt.getNum() << endl;
    cout << "Number(double): " << numberDouble.getNum() << endl;
    cout << "Number(char): " << numberChar.getNum() << endl;
}
```

STANDARD TEMPLATE LIBRARY

- The Standard Template Library (STL) provides a set of programming tools to implement algorithms and data structures like vectors, lists, stacks, queues, maps, etc. to simplify the development of C++ programs.

- STL provides a way to write efficient and reusable code that can be applied to different data types.

- Using STL, we can write our algorithm once and then use it with different data types of data without having to write separate code for each data type.

- STL has three main components-
  - Containers
  - Iterators
  - Algorithms

CONTAINERS

- STL Containers store data and organize them in a specific manner as required.
- For example, Vectors store data of the same type in a sequential order whereas, Maps store data in key-value pairs.
- Containers are categorized in 3 types-

Sequence Containers	Associative Containers	Derived Containers
Vector	Set	Stack
List	Multiset	Queue
Dequeue	Map	Priority queue
	Multimap	

VECTOR

- Vector stores the elements of same data type.

- Size of a vector can grow dynamically. Means, the size can be change at the run-time.

- To use vectors, must include vector header file.

  #include <vector>

- Vector can be initialized by three ways-

  vector<int> vector1 = {1,2,3,4,5};
  vector<int> vector2 {1,2,3,4,5};
  vector<int> vector3(5,10);

VECTOR: EXAMPLE

1

```
int main() {

  // initializing: method-1
  vector<int> vector1 = {1, 2, 3, 4, 5};

  // initializing: method-2
  vector<float> vector2{5.2, 1.25, 82.7, 90.6, 7.4};

  // initializing: method-3
  vector<char> vector3(3, 'C');
```

2

```
  cout << "vector1 = ";
  for (int i : vector1) {
    cout << i << "  ";
  }

  cout << "\nvector2 = ";
  for (float i : vector2) {
    cout << i << "  ";
  }

  cout << "\nvector3 = ";
  for (char i : vector3) {
    cout << i << "  ";
  }
}
```

VECTOR: EXAMPLE CODE

1

```
int main() {

  // initializing
  vector<int> v1 = {1, 2, 3, 4, 5};

  // size before adding
  cout << "Size(before add): " << v1.size() << endl;

  // add an element
  v1.push_back(6);
  v1.push_back(7);

  // size after adding
  cout << "Size(after add): " << v1.size() << endl;

  // edit 3rd element
  v1.at(2) = 300;
```

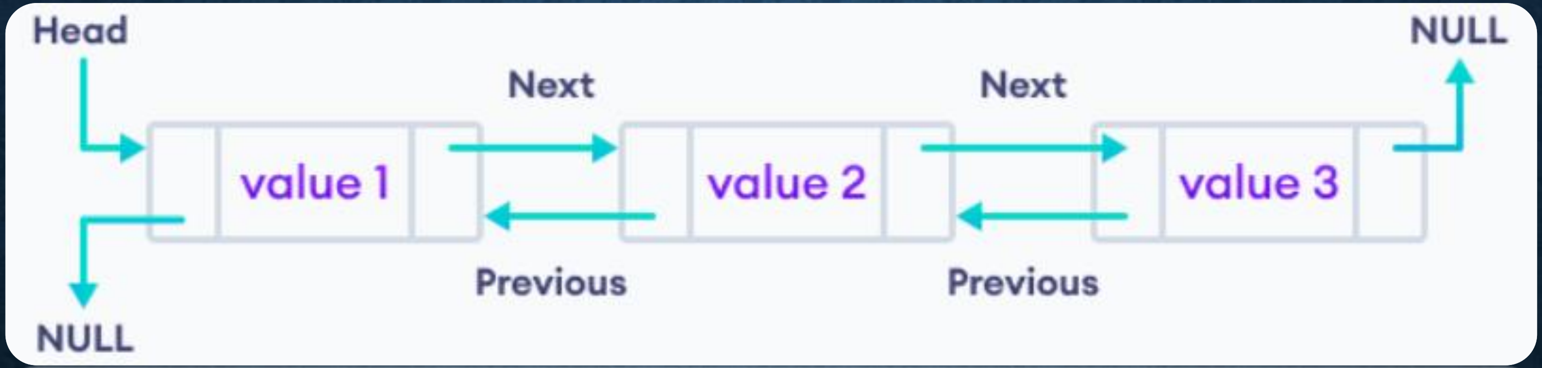
2

```
  // delete last element
  v1.pop_back();

  //returns last element
  cout << "Last element: " << v1.back() << endl;

  // using for loop
  cout << "v1 elements: ";
  for(int i=0;i<=v1.size()-1;i++)
    cout << v1.at(i) << " ";

  //check if empty (returns 1 if empty)
  cout << "\nv1 is empty: " << v1.empty();
}
```

Prepared by: AMITABH SRIVASTAVA

LIST

- List is a STL container that stores elements in random locations. To maintain sequential ordering, every list element includes two links:
  - points to the previous element
  - points to the next element



- List implements the **doubly-linked list data structure,** hence we can navigate both forward and backward.

- To use lists, must include list header file.

  #include <list>

LIST: EXAMPLE

```
int main() {

    // initializing list
    list<int> list1 {1, 2, 3, 4};
    list<double> list2 {25.5, 12.7, 77.7};
    list<char> list3 {'c', 'p', 'a'};

    // display using range for loop
    cout << "list1 Elements: ";
    for(int l1 : list1) {
        cout << l1 << " ";
    }

    cout << "\nlist2 Elements: ";
    for(double l2 : list2) {
        cout << l2 << " ";
    }

    cout << "\nlist3 Elements: ";
    for(char l3 : list3) {
        cout << l3 << " ";
    }
}
```

LIST: FUNCTIONS

- **Adding elements-**
  - push_front() – inserts element to the beginning of the list
  - push_back() – adds an element to the end of the list

- **Accessing elements-**
  - front() – returns the first element
  - back() – returns the last element

- **Deleting elements-**
  - pop_front() – removes the element from the beginning of the list
  - pop_back() – removes the element from the end of the list

- **Other functions-**
  - size() – returns the number of elements in the list
  - empty() – checks whether the list is empty
  - clear() – clears all the values from the list

LIST: EXAMPLE CODE

1

```
int main() {

    // initializing list
    list<int> list1 {1, 2, 3, 4};

    // insert at beginning
    list1.push_front(0);

    // insert at end
    list1.push_back(5);

    // display using range for loop
    cout << "list1 Elements: ";
    for(int l1 : list1) {
        cout << l1 << " ";
    }
```

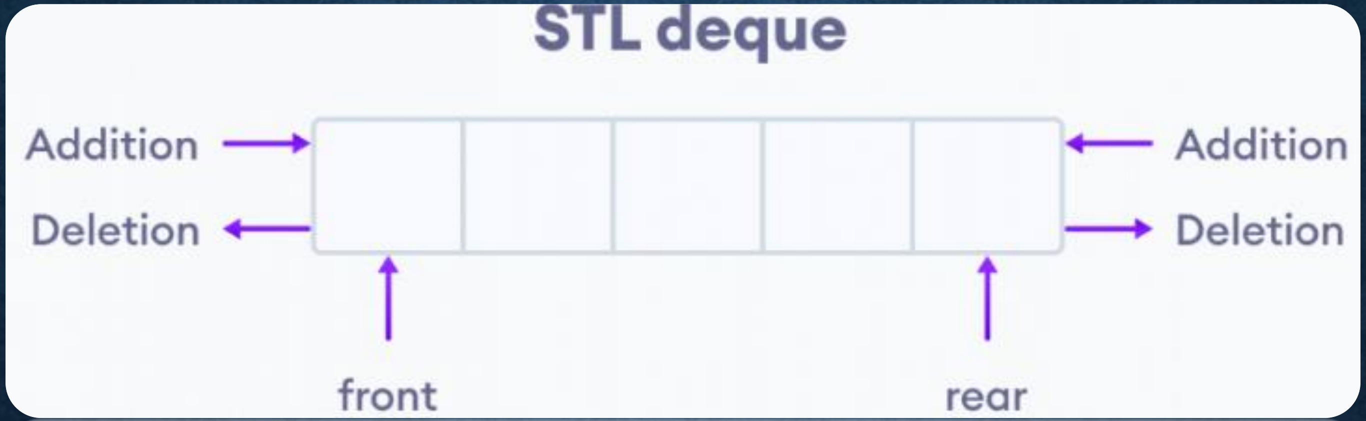
Prepared by: AMITABH SRIVASTAVA

2

```
    cout << "\nFirst element: " << list1.front();
    cout << "\nLast element: " << list1.back();

    // deleting first element
    list1.pop_front();

    // deleting last element
    list1.pop_back();

    cout << "\nFirst element: " << list1.front();
    cout << "\nLast element: " << list1.back();

    // length of list
    cout << "\nTotal no. of elements: " << list1.size();

    // deleting all elements
    list1.clear();
    cout << "\nTotal no. of elements: " << list1.size();
    cout << "\nIs list1 empty? " << list1.empty();
}
```

DEQUEUE

- It is a sequential container that provides the functionality of a double-ended queue data structure.



- In a deque, we can insert and remove elements from both the **front** and **rear**.

- To use deque, must include dequeue header file.
  #include <deque>

DEQUE: FUNCTIONS

- **Adding elements-**
  - push_front() – inserts an element at the beginning of the deque (front)
  - push_back() – inserts an element at the end of the deque (back)

- **Accessing elements-**
  - front() – returns the element at the front
  - back() – returns the element at the back
  - at() – returns the element at specific index position

- **Deleting elements-**
  - pop_front() – removes the element from the front
  - pop_back() – removes the element from the back

- **Other functions-**
  - size() – returns the number of elements in the deque
  - empty() – checks whether the deque is empty
  - clear() – clears all the values from the deque

DEQUE: EXAMPLE CODE

1

```
int main() {

    int pos;
    // initializing deque
    deque<int> dq1 {1, 2, 3, 4};

    // insert at front
    dq1.push_front(0);

    // insert at back
    dq1.push_back(5);
```

Prepared by: AMITABH SRIVASTAVA

2

```
    // display using range for loop
    cout << "dq1 Elements: ";
    for(int d1 : dq1) {
        cout << d1 << " ";
    }

    // deleting front element
    dq1.pop_front();

    cout << "\ndq1 Elements(after deleting): ";
    for(int d1 : dq1) {
        cout << d1 << " ";
    }
}
```

SET/MULTISET

- Sets are a type of associative container which stores only unique elements.
- Multiset can store same values.
- The values are stored in a specific sorted order i.e. either ascending or descending. (hence cannot use indexing to access elements)
- Ascending is the default sort order.
- Set can take any data type depending on the values, e.g. int, char, float, etc.
- To use set/multiset, must include set header file.

  #include <set>

MULTISET/SET: FUNCTIONS

- **Adding elements-**
  - insert() – inserts an element to the set

- **Deleting elements-**
  - erase() – removes the element from the set

- **Other functions-**
  - count() – returns 1 or 0 based on whether the element is present in the set
  - size() – returns the number of elements in the set
  - empty() – checks whether the set is empty
  - clear() – removes all the values from the set

Prepared by: AMITABH SRIVASTAVA

MULTISET/SET: EXAMPLE

1

```cpp
int main()
{

    // initializing set in ascending order
    set<int> s1 = {23, 45, 12, 34, 7, 45};
    cout << "s1 elements: ";
    for (auto i : s1) {
        cout << i << ' ';
    }

    // initializing set in descending order
    multiset <int, greater <int>> s2 = {2, 4, 67, 12, 7, 12};
    cout << "\ns2 elements: ";
    for (auto i : s2) {
        cout << i << ' ';
    }
```


2

```cpp
    // inserting element
    s1.insert(100);
    cout << "\ns1 elements: (after insert)";
    for (auto i : s1) {
        cout << i << ' ';
    }

    // deleting element
    s2.erase(12);
    cout << "\ns2 elements: (after removing)";
    for (auto i : s2) {
        cout << i << ' ';
    }

    // check for an element (0 or 1)
    cout << "\nElement present: " << s1.count(7);
}
```

MAP/MULTIMAP

- Maps are associative containers that store elements in a key-value pair form.
- Each element has a key value and a mapped value.
- All the keys must be unique inside a map.
- To use map/multimap, must include map header file.

  #include <map>

MULTIMAP/MAP: FUNCTIONS

- **Adding elements-**
  - insert() – inserts an element with a particular key in the map

- **Deleting elements-**
  - erase() – removes the element from the map

- **Other functions-**
  - count() – returns no. of matches of an element in the map
  - size() – returns the number of elements in the map
  - empty() –checks whether the set is empty (returns 0 or 1)
  - clear() – removes all the elements from the map

MULTIMAP/MAP: EXAMPLE

1

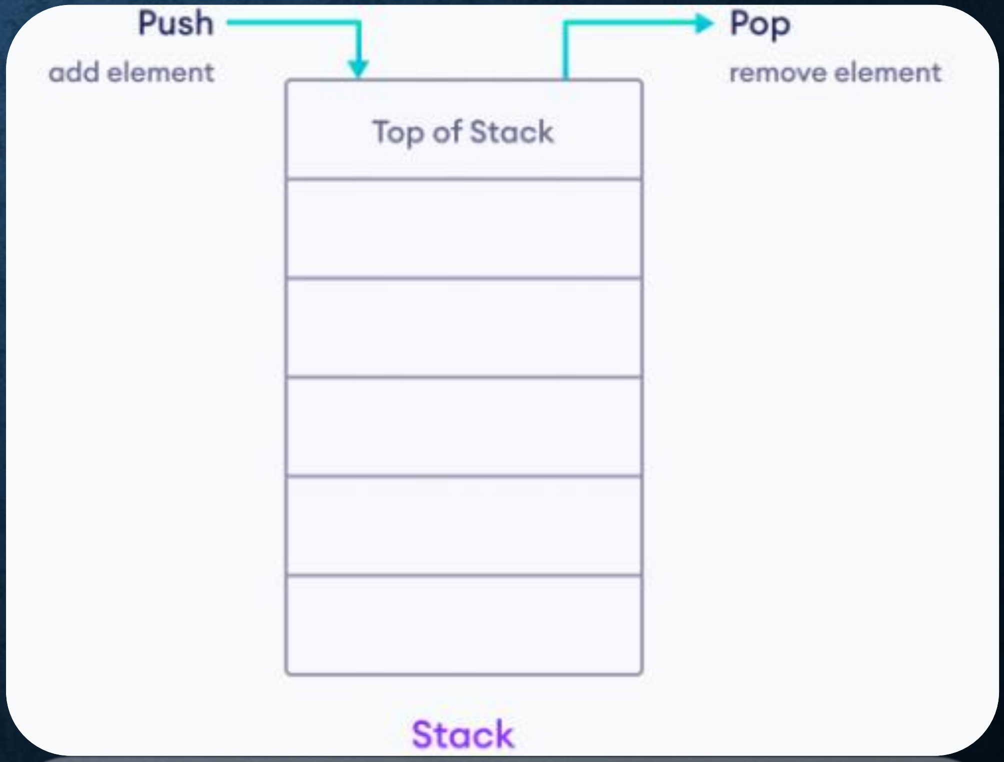
```
int main()
{
    map<string, int> map1;
    map<int, int> map2;
    map<int, string> map3;

    // Inserting values
    map1["one"] = 1;
    map1["two"] = 2;
    map1["three"] = 3;
    map1["four"] = 4;

    map2[1] = 100;
    map2[2] = 200;
    map2[3] = 300;

    map3[1] = "one";
    map3[2] = "two";
    map3[3] = "three";
    map3[2] = "twenty";
```


2

```
    int len3 = map3.size();
    cout << "\nLength of map3: " << len3 << endl;

    // removing an element from map1
    map1.erase("two");

    // Iterator pointing to the first element
    map<string, int>::iterator it = map1.begin();

    cout << "\nmap1 elements: \n";
    // Iterate through the map and print the elements
    while (it != map1.end()) {
        cout << "Key: " << it->first
            << ", Value: " << it->second << endl;
        ++it;
    }

    // check for key "four"
    cout << "\nkey 'four' in map1: " << map1.count("four");
}
```

STACK

- Stacks are a type of derived container that uses LIFO(Last In First Out) type of working.
- Every new element is added at one end (top) and an element is removed from that end only.
- To use stack, must include stack header file.
  #include <stack>



STACK: FUNCTIONS

- **Adding elements-**
  - push() – adds an element into the stack
- **Deleting elements-**
  - pop() – removes the element from the stack
- **Other functions-**
  - top() – returns the element which is at top of the stack
  - size() – returns the number of elements in the stack
  - empty() –checks whether the stack is empty (returns 0 or 1)

STACK: EXAMPLE

```
int main() {

  // create a stack of strings
  stack<string> colors;

  // push elements into the stack
  colors.push("Green");
  colors.push("Yellow");

  // get the stack size
  int size = colors.size();
  cout << "Stack has " << size << " elements." << endl;

  // print elements of stack
  cout << "Stack: ";
   while(!colors.empty()) {
    cout << colors.top() << " ";

    // remove an element
    colors.pop();
  }
}
```

QUEUE

- Queues are a type of derived container that uses FIFO(First In First Out) type of working.
- In Queue linear data structure, elements that are added first will be removed first.
- To use queue, must include queue header file.

  #include <queue>

QUEUE: FUNCTIONS

- **Adding elements-**
  - push() – inserts an element at the back of the queue

- **Deleting elements-**
  - pop() – removes the element from the front of the queue

- **Other functions-**
  - front() – returns the first element of the queue
  - back() – returns the last element of the queue
  - size() – returns the number of elements in the queue
  - empty() –checks whether the queue is empty (returns true if empty)

QUEUE: EXAMPLE

```
int main() {

  // create a queue of string
  queue<string> colors;

  // push elements into the queue
  colors.push("Green");
  colors.push("Yellow");
  colors.push("Red");
  colors.push("Blue");

  // get the size of queue
  int size = colors.size();
  cout << "Queue has " << size << " elements." << endl;

  // print elements of queue
  cout << "Queue: ";
  while(!colors.empty()) {
    // print the element
    cout << colors.front() << " ";
    colors.pop();
  }
}
```

PRIORITY QUEUE

- Priority Queue is a special type of queue in which each element is associated with a **priority value**.

- Elements are served on the basis of their priority. Means, higher priority elements are served first.

- Elements with the same priority served according to their order in the queue.

- To use priority queue, must include queue header file.

```
#include <queue>
```

Prepared by: AMITABH SRIVASTAVA

PRIORITY QUEUE: FUNCTIONS

- **Adding elements-**
  - push() – inserts an element into the priority queue

- **Deleting elements-**
  - pop() – removes the element with the highest priority

- **Other functions-**
  - top() – returns the element with the highest priority
  - size() – returns the number of elements in the priority queue
  - empty() –checks whether it is empty (returns <u>true</u> if empty)

PRIORITY QUEUE: EXAMPLE

```
int main() {

  // create a priority queue of int
  priority_queue<int> ages;

  // add items to priority_queue
  ages.push(16);
  ages.push(25);
  ages.push(7);

  //Check size of priority queue
  cout << "Size: " << ages.size();

  // display all elements of ages
  cout << "\nPriority Queue: ";
  while(!ages.empty()) {
    cout << ages.top() << ", ";
    ages.pop();
  }
  cout << endl;
}
```

ALGORITHM LIBRARY

- STL provides a variety of algorithms that can be implemented on any container.
- We no longer need to develop our own complicated algorithms and can securely rely on the built-in methods supplied by the STL algorithm library.
- Using these algorithms from STL saves time and effort.
- A single algorithm function can be applied on every container type.
- For example, to implement a binary search in C++, we don’t need to write the whole code to achieve binary search. Instead, we can simply use the algorithm library binary_search() function to execute the binary search.
- To use the methods define under algorithm library, we need to import it as:

```
#include <algorithm>
```

ALGORITHM FUNCTIONS EXAMPLE

1

```
int main()
{
    // Initializing vector with int values
    vector<int> vect = {10, 20, 5, 23, 42, 15};

    cout << "Vector is: ";
    for (int i=0; i<vect.size(); i++)
        cout << vect[i] << " ";

    // Sorting the Vector in Ascending order
    sort(vect.begin(), vect.end());

    cout << "\nVector after sorting is: ";
    for (int i=0; i<vect.size(); i++)
    cout << vect[i] << " ";
```

2

```
// Sorting the Vector in Descending order
sort(vect.begin(),vect.end(), greater<int>());

cout << "\nVector in Descending order is: ";
for (int i=0; i<vect.size(); i++)
cout << vect[i] << " ";

// Reversing the Vector
reverse(vect.begin(), vect.end());

cout << "\nVector after reversing is: ";
for (int i=0; i<vect.size(); i++)
    cout << vect[i] << " ";
```

3

```
    cout << "\nMaximum element of vector is: ";
    cout << *max_element(vect.begin(), vect.end());

    cout << "\nMinimum element of vector is: ";
    cout << *min_element(vect.begin(), vect.end());
}
```

Prepared by: AMITABH SRIVASTAVA