

C++ Summary

Contents

This summary	4
Start.....	4
Variable and basic data types in C++	5
String with using namespace std	5
Const.....	5
Namespace	6
Typedef and type aliases	7
User input.....	8
Control-Flow Statements (selection & iteration)	8
User defined functions explained.....	8
Headers.....	9
Overloading	9
Structs and classes in C++.....	10
Structs in C++	10
Class in C++	11
Constructors	12
Constructors with header and class files:	13
Arrays.....	13
Pointers (Raw Pointers)	14
Static vars and methods	15
Constructing and Destructing.....	16
new keyword	17
Try and catch	19
Composition and initialization.....	20
Example of composition and initialization.....	20
Composition and initialization with basic OOP in C++, using references , const methods, Const & Friend.	21
Friend.....	24
Operator overloading	26
Literals in C++	27
Copy, assign, convert.....	28
Deep Copy	28
Shallow Copy	30

Explicit in C++.....	31
Inheritance in C++.....	32
Override.....	34
Virtual keyword + override keyword.....	35
Type casting Real Time Type Information (less common use in C++)	36
Templates	37
Function Templates.....	37
Class Template.....	39
Template Variations	40
Template Specialization	40
Template Meta-Programming	41
The Standard C++ Library- Looping on containers	41
Tuple.....	41
Concatenating Tuples:	43
Vectors	43
Create a vector:	43
Access a vector:	44
Front and Back:.....	44
Changing an element:.....	44
Adding vector element:	44
Removing an element:	44
Bool Empty:	45
Other methods that are good to know:	45
Sets	46
Adding element to set:	46
Access an element:	46
Finding elements:	46
Update elements:	46
Looping inside Set:.....	46
Deleting elements:	47
Maps	47
Adding element + Looping:	47
Accessing element:	47
Update element:.....	47
Finding element:.....	48
Traversing:	48

Deleting elements	48
<type_traits>	48
Stl-algorithm and iterators	49
Iterators:	49
The “105” Algorithms:	49
Why Learn Them All?	50
Modifying containers:	50
Smart Pointer	51
Unique pointer	51
Unique pointers are not able to be shared	51
Unique pointers can be moved to other unique pointers.....	51
Unique pointers are known for deallocated automatic memory.....	52
Shared pointer	52
Weak pointer	52

This summary

This document was created by Shachar Tsrafati using resources from GeeksforGeeks, official C++ documentation, various YouTube tutorials, and coursework at Ariel University.

Some of the code examples are too long for one page, I recommend copying and running them yourself.

Additional resources:

A companion Makefile summary is also available:

<https://github.com/ShacharTs/C-Plus-Plus-Summary>

Start

Note: This summary is mostly on C++ 17, All the codes in this file should work, make sure you test version C++ 17 and above, lower version may not work.

This summary was made by a student. keep in mind that there will be mistakes and some spelling mistakes, make sure to check and test everything here.

C++ uses files with the .cpp extension. Like C, we need to use a compiler to compile our files. We can use the same compiler and IDE we used for C, such as VSCode, CLion, a text editor, and more.

To start, we must create a new file with a .cpp extension. At the start of the file, we add:

```
#include <iostream> // this lets us access a lot of libraries in C++
```

As always, we need to create a main function. Just like in C, we write:

```
int main() {  
    return 0;  
}
```

return 0; means the program finished successfully without any problems.
Returning 1 or another value can indicate there was an issue in the program.

To print a line in C++, we use:

```
std::cout << "Hello it's me Shachar" << std::endl;  
std::cout << "Hello it's me Shachar" << '\n';
```

Making comments:

```
int x = 5; // typing (makes that part a comment)
```

Variable and basic data types in C++

```
int main(){
int x = 5; // int
double y = 2.5; // double
char grade = 'A' // char
bool pass = true; // Boolean

std:: string name "Shachar"; // string

return 0;
}
```

String with using namespace std

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

int main(){

string name = "Shachar";
cout << name << endl;

return 0;
}
```

Note: From that point , I will mostly use "using namespace std", if you copy the code and get errors, make sure you include <iostream> and "using namespace std".

Const

like in java and C, using const makes that var unchangeable later.

Namespace

let us use the same var NAME in multiple scopes/functions

```
#include <iostream>

namespace first {
    int x = 1;
}

namespace second {
    int x = 2;
}

int main() {
    int x = 0;
    std :: cout << x << std :: endl;
    std :: cout << first :: x << std :: endl;
    std :: cout << second :: x << std :: endl;
    return 0;
}
```

using Namespace without prefix:

```
#include <iostream>

namespace first {
    int x = 1;
}

namespace second {
    int x = 2;
}

int main(){
    using namespace first;
    std :: cout << x << std :: endl; // now this will print x from
    first
    return 0;
}
```

keep in mind that if we are using prefix, we cannot create a var "X" in main(), using namespace save us from typing a prefix every single time, but it won't let us to call multiple vars with the same name normally, so keep that in mind.

A good way to use namespace is to avoid retyping std each time.

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

namespace first {
    int x = 1;
}

namespace second {
    int x = 2;
}

int main(){
    using namespace first;
    cout << x << endl; // now this will print x from first
    return 0;
}
```

Typedef and type aliases

typedef like in c, make us to take a long line of code and makes it to become a one word, so we won't need to type it every single time, we can make a pointer, , list , vars, functions...

```
#include <iostream>
typedef int (*FuncPtr)(int, int);

int add(int a, int b){
    return a + b;
}

int main(){
    FuncPtr ptr = add;
    std::cout << ptr(2, 3) << std::endl; // Prints 5
}
```

NOTE: from c++ version 11, we can use the keyword “using” like typedef, and that word using like in namespace it is better than typedef. So, use that.

```
#include <iostream>
using FuncPtr = int (*)(int, int);

int add(int a, int b){
    return a + b;
}

int main(){
    FuncPtr ptr = add;
    std::cout << ptr(2, 3) << std::endl; // Prints 5
}
```

User input

```
// cout << // output
// cin >> // input

std :: string name;

std :: cout << "Enter name: ";
std :: cin >> name;

std :: cout << "Hello " << name;
```

Control-Flow Statements (selection & iteration)

Credit to YouTube [BroCode](#):

[If](#), [Switch](#), [For loop](#), [While loop](#), [break](#) and [continue](#)

User defined functions explained

Like in C, we want to make some functions, we need to remember like in C when we define functions, we need to make them at the start of the code, or we need to define the keyword of the function at the start of the code.

```
#include <iostream>
void happyBirthday(); // when we define at the start we can call it

int main(){
    happyBirthday();
    happyBirthday();
    happyBirthday();
    return 0;
}

void happyBirthday(){
    std :: cout << "Happy birthday to you" << std :: endl;
}
```


Headers

like in C, we can use header file, that we define all the functions names to a single file. Header file ending like in C, is ".h", this ending support C/C++ files, using ".hpp" only support C++ files.

```
// mainHeader.h
#ifndef MAINHEADER_H
#define MAINHEADER_H

void happyBirthday();

#endif //MAINHEADER_H
////////////////////////////////////
// main.cpp
#include <iostream>
#include "mainHeader.h"

int main() {
    happyBirthday();
    happyBirthday();
    happyBirthday();
    return 0;
}

void happyBirthday() {
    std :: cout << "Happy birthday to you" << std :: endl;
}
```

Overloading

like in java and others, we got Overloading, we can use the same function name for multiple uses.

```
void bakePizza();
void bakePizza(std :: string topping1);
void bakePizza (std :: string topping1, std :: string
topping2);
```

Structs and classes in C++

By default, struct members in C++, are public, and classes members in C++ are private.

Structs in C++

```
#include <iostream>

struct student {
    std::string name;
    double gpa;
    bool inClass;
};

int main() {

    student student1;
    student1.name = "Shachar";
    student1.gpa = 10;
    student1.inClass = true;

    student student2 = {"SpongeBob SquarePants", 20, false}; //
we can use struct like, in one line

    std::cout << student1.name << "," << student1.gpa << ","
<< student1.inClass << std::endl;

    std::cout << student2.name << "," << student2.gpa << ","
<< student2.inClass << std::endl;

}
```

Class in C++

```
#include <iostream>

class Human{
    public: // set this class to be PUBLIC
        std :: string name;
        int age;

        void eat() {
            std :: cout << name << " is eating" << std:: endl;
        }
        void drink() {
            std :: cout << name << " is drinking" << std:: endl;
        }
        void sleep() {
            std :: cout << name << " is sleeping" << std:: endl;
        }
};

int main(){
    Human Shachar = {"Shachar",25};

    std :: cout << Shachar.name << "," << Shachar.age <<
std::endl;
    Shachar.eat();
    Shachar.drink();
    Shachar.sleep();

    return 0;
}
```

Constructors

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

class Human{
public:
    string name;
    int age;

    // CONSTRUCTORS
    Human (string name, int age) {
        // using keyword "this" like in java
        this -> name = name;
        this -> age = age;
    }

};

int main(){
    Human Shachar("Shachar",25);

    cout << "Name: " << Shachar.name << "\nAge: " <<
Shachar.age << endl;

    Shachar.age = 26; // setting new age

    cout << "Name: " << Shachar.name << "\nAge: " <<
Shachar.age << endl;
    return 0;
}
```

Note: Some people use shortcut

C-tor = constructors . **D-tor** = destructor

Constructors with header and class files:

```
// main.cpp
#include <iostream>
#include "Human.cpp" // we are including the human file with the header

int main() {

    Human Shachar("Shachar", 25);
    cout << "Name: " << Shachar.name << "\nAge: " << Shachar.age << endl;

    Shachar.age = 26;
    cout << "Name: " << Shachar.name << "\nAge: " << Shachar.age << endl;

    return 0;
}

// human.cpp file
#include "Human.h"

Human::Human(string name, int age) {
    this->name = name;
    this->age = age;
}

// human.h file
#ifndef HUMAN_H
#define HUMAN_H

#include <string>
using namespace std;

class Human {
public:
    string name;
    int age;

    Human(string name, int age);
};

#endif
```

Arrays

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

int main() {

    string humans[] = {"Spongebob","Patrick","SquidWard"};
    int size = sizeof(humans) / sizeof(humans[0]); // how we getting the
size of the array
    // normal loop
    for (int i = 0; i < size; i++) {
        cout << humans[i] << endl;
    }
    cout << endl; // printing new line

    // for each loop
    for (string human: humans) {
        cout << human << endl;
    }

    return 0;
}
```

Pointers (Raw Pointers)

A raw pointer (T*) simply holds the address of a T object.

If it points to a stack-allocated object (one you declared normally), you must **not** call delete—its lifetime is managed by scope.

If it points to a heap-allocated object (one you created with new or new []), you **must** call delete (or delete[] for arrays) when you're done, otherwise you leak memory.

Forgetting to free heap memory causes **memory leaks**; deleting something you don't own (or deleting it twice) leads to **undefined behavior**.

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

struct human {
    string name;
    int age;
};

void printHumans(human** list, int size) {
    cout << endl;
    for (int i = 0; i < size; i++) {
        auto h = list[i]; // using keyword "auto"
        cout << h -> name << ', ' << h -> age << endl;
    }
    cout << endl;
}

int main() {
    // Basic pointer example
    string name = "Shachar";           // Declare string variable
    string* pName = &name;             // Pointer to address of 'name'

    // Print pointer details
    cout << pName << endl;             // Prints address stored in pName (address of 'name')
    cout << &pName << endl;           // Prints address of pointer variable itself (pName)
    cout << *pName << endl;           // Dereferences pointer, prints value ("Shachar")

    // ---- Struct pointer array section ----

    int size = 5;                      // Size of the list
    human** pListHuman;                 // Declare pointer to array of human pointers
    pListHuman = new human*[size];      // Dynamically allocate array of human pointers

    pListHuman[0] = new human{"Spongebob", 27};
    pListHuman[1] = new human{"Squidward", 31};
    pListHuman[2] = new human{"Patrick", 24};
    pListHuman[3] = new human{"Sandy", 29};
    pListHuman[4] = new human{"Mr.Crab", 69};

    // Yes, they are humans sue me I don't care

    printHumans(pListHuman, size);

    // --- cleanup ---
    for (int i = 0; i < size; ++i) {
        delete pListHuman[i];          // free each human object
    }

    delete[] pListHuman; // Free the array of pointers

    return 0;
}
```

Static vars and methods

```
#include <iostream>
#include "human.h"

using namespace std;

int main() {
    human spongebob = {"spongebob",29};
    cout << spongebob.name << ", " << spongebob.age << ",Human count: "<<human::id << endl;

    human patrick = {"Patrick",24};
    cout << patrick.name << ", " << patrick.age << ",Human count: "<<human::id << endl;

    human squidward = {"squidward",31};
    cout << squidward.name << ", " << squidward.age << ",Human count: "<<human::id << endl;

    human mr_crab = {"mr_crab",69};
    cout << mr_crab.name << ", " << mr_crab.age << ",Human count: "<<human::id << endl;

    spongebob.printHello();

    return 0;
}
////////////////////////////////

#include "human.h"

int human::id = 0;

human::human(string name, int age) {
    this->name = name;
    this->age = age;
    id++; // everytime we create new object, we update the static counter
}

string human::getClassName() {
    return "Human";
}

void human::printHello() {
    cout << "Hello from " << getClassName() << endl;
}
////////////////////////////////
#ifdef HUMAN_H
#define HUMAN_H

#include <string>
#include <iostream>
using namespace std;

class human {
public:
    static int id;
    string name;
    int age;

    human(string name, int age);

    static string getClassName();
    static void printHello();
};
#endif
```

Constructing and Destructing

Destructors in C++ handle cleanup tasks.

For stack objects, destructors are called automatically when the object goes out of scope (function ends, block ends), in reverse order of creation (Last In, First Out).

For heap objects created with new, destructors are called manually using delete.

Destructors free resources like dynamic memory to avoid memory leaks.

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

class Number {
private:
    double *number;

public:
    // Constructor
    Number(double num) {
        number = new double; // allocates memory dynamically
        // call constructor
        *number = num; // define value to address
        cout << "Constructor executing!" << endl;
        cout << "Number: " << *number << endl;
    }

    // Destructor
    ~Number() {
        cout << "Destructor executing!" << endl;
        cout << "Number: " << *number << endl;
        delete (number); // call destructor then free memory
    }
};

void test () {
    Number six (6);
}

// The output will be (5,5,7,6,6,7)
// The output is order when running the code

int main() {
    Number *five = new Number(5);
    delete five;
    Number seven(7);
    test();
    return 0;
}
```


new keyword

Can be used with any data type, (int, double, pointers, struct and more).

```
int *i = new int;  
char **p = new (char *);
```

can be used for classes like in java,

```
class MyClass {  
  
public:  
    MyClass();  
    MyClass( int i );  
    MyClass( double x, double y );  
  
};  
  
MyClass* a;  
a = new MyClass;  
a = new MyClass {5};  
a = new MyClass { 1.0, 0.0 };
```

Summary made by: Shachar Tsrafati

Example of dynamic memory allocation, pointers to objects, manual memory management, constructors & destructors behavior, and accessing object members via pointers.

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

class Number {
public:
    double *number;

    // Constructor
    Number(double num) {
        number = new double; // allocates memory dynamically (heap), call
        constructor
        *number = num; // define value to address
        cout << "Constructor executing!" << endl;
        cout << "Number: " << *number << endl;
    }

    // Destructor
    ~Number() {
        cout << "Destructor executing!" << endl;
        cout << "Number: " << *number << endl;
        delete (number); // call destructor then free memory
    }
};

int main() {
    size_t size = 4;
    Number **arr = new Number *[size];

    for (int i = 0; i < size; i++) {
        arr[i] = new Number(i);
    }

    cout << endl;

    for (int i = 0; i < size; i++) {
        cout << arr[i] << endl; // address of the Number object at arr[i]
        // (heap address)
        cout << &arr[i] << endl; // address of the pointer arr[i] stored in
        // the array
        cout << &arr[i]->number << endl; // address of the value in pointer
        // arr[i]
        cout << *arr[i]->number << endl; // value in pointer arr[i]
        cout << endl;
    }

    for (int i = 0; i < size; i++) {
        delete arr[i];
    }

    delete arr;

    return 0;
}
```

Try and catch

like in java/C. C++ has try and catch for handling errors
to [learn more about try and catch](#)

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

double division(const int a, const int b) {
    try {
        if (b == 0) {
            throw std::string("Cannot divide by zero! ");
        }
    } catch (std::string msg) {
        cout << msg;
        return -1; // Error
    }

    return static_cast<double> (a) / b; // casting double on
"a"
}

int main() {
    double c = division(5, 0);
    cout << c << endl;

    double d = division(5, 2);
    cout << d << endl;

    return 0;
}
```

Composition and initialization

In C++, member objects are actual members, not pointers by default.

In Java, class-type members are references (pointers).

```
class Line {
    Point p1, p2;
};
// C++ -> Members are objects
// Java -> Members are pointers (references)
```

Example of composition and initialization

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

class Student {
public:
    string name;
    int startYear;
    int gradYear;

    Student(string name,int startYear,int gradYear) :
name(name), startYear(startYear) , gradYear(gradYear)
    {
        // NOT NEEDED
        // this->name = name;
        // this->startYear = startYear;
        // this->gradYear == gradYear;

        cout << "Student Object Constructed !" << endl;
    }
};

int main() {
    Student shachar = {"Shachar",2021,2050};

    cout << "Name: " << shachar.name << endl
    << "Start year: " << shachar.startYear << endl
    << "Grad year: " << shachar.gradYear << " :( " << endl;

    return 0;
}
```

Composition and initialization with basic OOP in C++, using references , const methods, Const & Friend.

(See friend topic after this one)

```
#include <iostream>
#include <sstream>
#include <iomanip>
#include <cmath>
using namespace std;

class Point {
private:
    double x;
    double y;

public:
    Point(double x, double y) : x(x), y(y) {
        cout << "Point Constructor executing !" << endl;
    }

    ~Point() {
        cout << "Point Destructor executing!" << endl;
    }

    double getX() const {
        return x;
    }
    double getY() const {
        return y;
    }

    friend void printPoint(const Point& p);

    Point& add(const Point& p) {
        this->x += p.x;
        this->y += p.y;
        return *this;
    }

    string toString() const {
        ostringstream oss;
        oss << fixed << setprecision(2); // show 2 digits after .
        oss << "(" << getX() << ", " << getY() << ")";
        return oss.str();
    }
};

class Line {
private:
    Point p1, p2; // composition: Line has two Points
```

```
public:
    Line(const double x1, const double y1, const double x2, const
double y2) :
    p1(x1,y1),p2(x2,y2) {
        cout << "Line Constructor executing !" << endl;
    }

    ~Line(){
        cout << "Line Destructor executing!" << endl;
    }

    double get_point1_x() const {
        return p1.getX();
    }
    double get_point1_y() const {
        return p1.getY();
    }

    double get_point2_x() const {
        return p2.getX();
    }
    double get_point2_y() const {
        return p2.getY();
    }

    Point& get_point1() {
        return p1;
    }
    Point& get_point2() {
        return p2;
    }

    double distance() const {
        double xDis = pow((p2.getX() - p1.getX()),2);
        double yDis = pow((p2.getY() - p1.getY()),2);
        return sqrt(xDis + yDis);
    }

    void toString() const {
        cout << "Point 1: " << p1.toString() << endl << "Point 2: "
<< p2.toString() << endl;
    }
};

void printPoint(const Point& p) {
    // Directly access private members
    cout << "(" << p.x << ", " << p.y << ")" << endl;
}

int main() {
```

```
// create Line object on stack
Line line(5, 2, 10, 3);

line.toString();
cout << line.get_point1().toString() << endl;
cout << line.get_point2().toString() << endl;

// directly access point members
cout << "Point1 x: " << line.get_point1_x()
    << endl << "Point1 y: " << line.get_point1_y() << endl;

cout << "Point2 x: " << line.get_point2_x()
    << endl << "Point2 y: " << line.get_point2_y() << endl;

cout << line.distance() << endl;

// create Line object dynamically (on heap) using new
auto pLine = new Line(1, 1, 10, 10);

cout << pLine->get_point1().toString() << endl;
cout << pLine->get_point2().toString() << endl;
pLine->toString();

// directly access members of heap-allocated object
cout << "Point1 x: " << pLine->get_point1_x()
    << endl << "Point1 y: " << pLine->get_point1_y() << endl;

cout << "Point2 x: " << pLine->get_point2_x()
    << endl << "Point2 y: " << pLine->get_point2_y() << endl;

cout << pLine->distance() << endl;

// manually delete heap-allocated object
delete pLine;

line.get_point1().add(line.get_point1());
cout << line.get_point1().toString() << endl;

printPoint(line.get_point1()); // using friend function

return 0;
}
```

Friend

Friend in C++ is a keyword that allows certain external functions or classes to access a class's private and protected members.

Sometimes, specific functions or operators (like the << operator for output) are closely related to a class but cannot be defined inside the class itself. In such cases, these functions normally wouldn't have access to the class's private data.

The friend keyword provides a way to give them this special access while keeping the function defined outside the class.

```
#include <iostream>
#include <cmath>

using namespace std;
class Square;
class SquareCals;

class Shapes {
public:
    double findArea(const Square &s) const;
};

class Square {
    double side;
public:
    Square(const double side) : side(side) {
        cout << "Square Constructor executing !" << endl;
    }

    // all friend define inside the class scope

    // friend function from another class (Shapes class's
member function)
    // allows Shapes::findArea to access Square's private
members
    friend double Shapes::findArea(const Square&) const;

    // friend free (non-member) function
    // allows external function updateSide to modify Square's
private members
    friend void updateSide(Square&, double side);

    // friend class
    // grants full access to all private/protected members of
Square to SquareCals class
    friend class SquareCals;
```



```
};

class SquareCals {
public:
    double getPerimeter(const Square&) const;
    double getDiagonal (const Square&) const;
};

double Shapes::findArea(const Square& s) const {
    return s.side * s.side;
}

void updateSide(Square& s,const double side) {
    s.side = side;
}

double SquareCals::getPerimeter (const Square &s) const {
    return 4*s.side;
}

double SquareCals::getDiagonal(const Square &s) const {
    return s.side * sqrt(2);
}

int main() {

    Square sq(5);
    Shapes s;
    SquareCals sc;
    cout << s.findArea(sq) << endl;

    updateSide(sq,7);
    cout << s.findArea(sq) << endl;

    cout << sc.getPerimeter(sq) << endl;

    cout << sc.getDiagonal(sq) << endl;

    return 0;
}
```

Operator overloading

Operators like `+`, `-`, `*`, unlike in Java or Python, are implemented as functions in C++. These operators can be overloaded to define custom behavior for user-defined objects.

Operator overloading allows us to give our objects special functionality — for example, enabling the addition of two objects' values in a simple, intuitive way, which wouldn't be possible with default behavior.

To implement operator overloading, we use the operator **symbol**¹(or similar) syntax inside the class (see footnote).

For more details, see: [Operator Overloading in C++](#)

This is just a small example for operator overloading.

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

class Number {
public:
    double n;
    Number(double set_n) : n(set_n) {
    }

    Number operator+(const Number &numA) const {
        cout << "this -> c.n: " << this->n << endl; // a
        cout << "numA.n: " << numA.n << endl; // b
        return Number(this->n + numA.n); // c
    }

    Number operator-(const Number &numA) const {
        return Number(this->n - numA.n);
    }

    Number operator*(const Number &numA) const {
        return Number(this->n * numA.n);
    }

    Number operator/(const Number &numA) const {
        try {
            if (numA.n != 0) {
                return Number(this->n / numA.n);
            }
            throw runtime_error("Error can not div by 0");
        } catch (const runtime_error& e) {
            cout << e.what() << endl;
        }
        return 0;
    }
}
```

operator+, operator-, operator*, operator/ and more ¹

```
    bool operator==(const Number &numA) const {
        if (this->n == numA.n) {
            return true;
        }
        return false;
    }
};

int main() {
    Number a(5);
    Number b(15);
    Number c = a+b;
    cout << "c.n: " << c.n << endl;
    cout << "a + b: " << (a + b).n << endl;
    cout << "a - b: " << (a - b).n << endl;
    cout << "a * b: " << (a * b).n << endl;
    cout << "a / b: " << (a / b).n << endl;

    return 0;
}
```

Literals in C++

Literals are fundamental elements used to represent fixed values. These values can include numbers, characters, strings, and more. They are generally present as the right operand in the assignment operation.

```
// Integer literals of different bases
int dec= 42;
int oct= 052;
int hex= 0x2A;
int bin= 0b101010;
long int lint= 42L;
unsigned int uint= 42U;
long long int llint= 42LL;

// float literal
float f = 3.14f;

// double literal
double d = 3.14;

// long double literal
long double ld = 3.14L;

// Scientific notation
long double sld = 1.22e11;
```

Copy, assign, convert

Like in other programs language we got “Shallow copy” and “Deep copy”.

Shallow Copy:

Copies the **pointers (addresses)** of the object's members.

Both the original and the copied object **share the same memory** for those members.

Any change to shared data affects both objects.

Deep Copy:

Allocates new memory for the copied object's members. It then copies the actual values (not just addresses) from the original object to the new one.

As a result, both objects have **independent copies of the data**, so any changes made to one object **do not affect the other**.

Copy can come in a few forms:

- 1) Constructing new object from existing
- 2) Passing parameter by value
- 3) Returning by value
- 4) Assigning existing to existing

Note: cast 1-3 handled by **COPY CONSTRUCTOR**, case 4 is handled by assignment operator. BOTH DO SHALLOW COPY.

RULE OF THREE:

When we need to make a deep copy of an object, we need to define all these rules:

- 1) Copy constructor
- 2) Destructor
- 3) Operator =

If a class manages resources (like dynamic memory), and defines one of the following: copy constructor, assignment operator, or destructor, it is recommended to define all three to avoid memory issues (Rule of Three).

Deep Copy

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

class Student {
private:
    string name;
    int age;

public:
    Student(string name, int age) {
        this->name = name;
        this->age = age;
    }

    Student() {
        this->name;
        this->age = 0;
    }
}
```

```
// Copy constructor: Student B = A;
Student (const Student &other) {
    this->name = other.name;
    this->age = other.age;
}

// Assignment operator: B = A; supports chaining
Student& operator=(const Student& other) {
    if (this != &other) {
        name = other.name;
        age = other.age;
    }
    return *this;
}

const string& getName() const {
    return this->name;
}

int getAge() const {
    return this->age;
}

~Student() {
}

};

int main() {
    Student A{"Shachar", 25};
    Student B;

    cout << "A name and age are: " << A.getName() << "," << A.getAge() <<
endl;
    cout << "B name and age are: " << B.getName() << "," << B.getAge() <<
endl;
    cout << "B = A: uses assignment operator" << endl;
    B = A;
    cout << "B name and age are: " << B.getName() << "," << B.getAge() <<
endl;
    cout << "Student C = B: uses copy constructor" << endl;
    Student C = B;
    cout << "C name and age are: " << C.getName() << "," << C.getAge() <<
endl;
    // Check if addresses are different
    cout << "A name address: " << &A.getName() << endl;
    cout << "B name address: " << &B.getName() << endl;
    cout << "C name address: " << &C.getName() << endl;

    return 0;
}
```

Shallow Copy

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

class Student {
private:
    string* name;
    int* age;
public:

    Student(string name, int age) {
        this->name = new string(name);
        this->age = new int (age);
    }

    Student() {
        this->name = new string("Null");
        this->age = new int (0);
    }

    Student(const Student& other) {
        this->name = other.name;
        this->age = other.age;
    }

    Student& operator=(const Student& other) {
        if (this != &other) {
            this->name = other.name;
            this->age = other.age;
        }
        return *this;
    }

    string getName() const {
        return *this->name;
    }

    int getAge() const {
        return *this->age;
    }

    string* getNameAddress() const {
        return this->name;
    }

    int* getAgeAddress() const {
        return this->age;
    }

    ~Student() {
        delete name;
        delete age;
    }
};

int main() {
    Student A{"Shachar", 25};
    Student B;
    cout << A.getName() << ", " << A.getAge() << endl;
    cout << B.getName() << ", " << B.getAge() << endl;
    cout << "B = A  Shallow Copy" << endl;
    B = A;
```

```
cout << A.getName() << ", " << A.getAge() << endl;
cout << B.getName() << ", " << B.getAge() << endl;
cout << "Printing A,B addresses" << endl;
cout << A.getNameAddress() << ", " << A.getAgeAddress() << endl;
cout << B.getNameAddress() << ", " << B.getAgeAddress() << endl;
return 0;
}
```

Both codes are almost the same; the difference is how we handle the constructors and the member variables.

In the **deep copy**, we use value-type members (like string and int), so C++ automatically copies values, giving each object its own data.

In the **shallow copy**, we use pointers as member variables, so copying copies the addresses (pointers), meaning both objects share the same memory.

Explicit in C++

Explicit is a keyword in C++ that **prevents implicit conversions** when calling constructors or conversion operators.

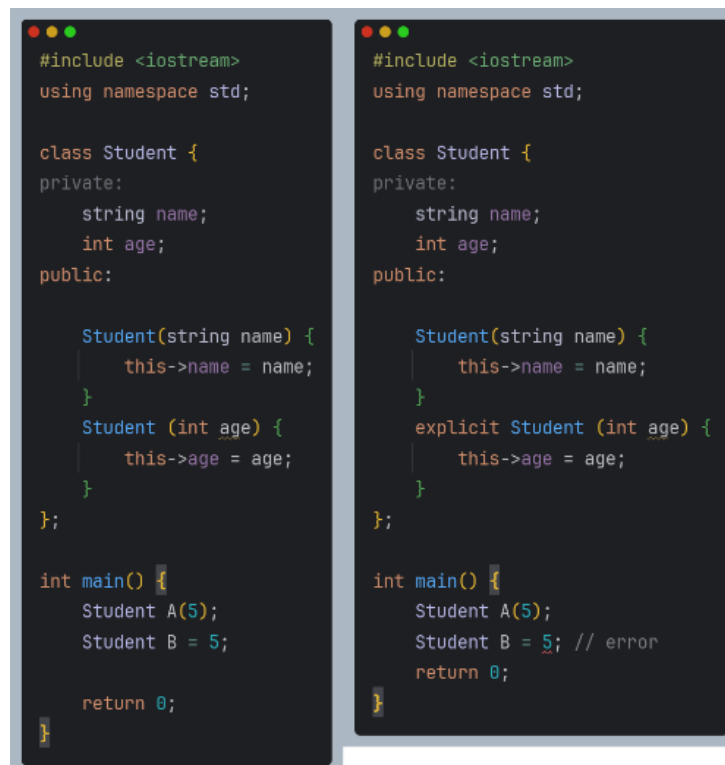
Without the explicit keyword:

C++ allows **automatic (implicit) type conversions**, meaning the compiler will silently convert matching types during object initialization.

With the explicit keyword:

C++ **prevents implicit type conversions**. You must use **direct initialization** (using parentheses '()') otherwise, the compiler will make an error.

Here is an example of how explicit work in C++:



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

class Student {
private:
    string name;
    int age;
public:

    Student(string name) {
        this->name = name;
    }
    Student (int age) {
        this->age = age;
    }
};

int main() {
    Student A(5);
    Student B = 5;

    return 0;
}
```

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

class Student {
private:
    string name;
    int age;
public:

    Student(string name) {
        this->name = name;
    }
    explicit Student (int age) {
        this->age = age;
    }
};

int main() {
    Student A(5);
    Student B = 5; // error

    return 0;
}
```

Inheritance in C++

Unlike C, C++ supports **inheritance** and even allows **multiple inheritance** (unlike Java).

In Java, we use the keyword `extends` to inherit from a class.

In C++, inheritance is done using the `:` symbol.

For example:

```
class Animal
public:
    void speak() {
        cout << "Animal speaks!" << endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
        cout << "Dog barks!" << endl;
    }
};
```

Access Specifier:

When a class inherits another class, we must define the **access level**:

- `public` → Keeps inherited members' access the same (most common).
- `private` → Makes inherited public/protected members private in the derived class.
- `protected` → Keeps inherited members protected or private.

Private/Protected access modifier :

In java inherit always public, in C++ we can choose to make inherit private or protected.

Private inherit:

Let us make sure we can only call inherited methods **ONLY from that class**.

If we create an object and want to call a method from the parent class, if inherit was private, we cannot call it.

Protected inherit:

Like protected in Java, it does the same thing if and only if we are inside the derived class or its child classes. It's not accessible outside.

Examples:

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

class Tf2 {
public:
    void spawn() const{
        cout << "User spawn back to the game" << endl;
    }
protected:
    void kill() const {
        cout << "Killing so much blood" << endl;
    }

    void talk() const {
        cout << "Talking bla bla " << endl;
    }
};

class Scout : public Tf2 { // public
public:
    void mvp() const{
        talk();
    }
    void domination() const{
        kill();
    }
};

int main() {
    Scout scout;
    scout.domination();
    scout.mvp();
    scout.spawn();
    scout.kill(); // Member is inaccessible
    return 0;
}
```

To not repeat the same code to show one change, here is the outcome when we change access modifier:

“ class Scout : protected Tf2 “ → scout.spawn(); // Member is inaccessible

“ class Scout : private Tf2 “ → Same outcome as protected: scout.spawn(); → inaccessible outside.

Note: the logic for members in inheritance works the same way as inheritance for methods.

Override

In C++ there are 2 ways to override.

Function hiding :

Just write the same signature name and type what you want.

Problem with function hiding is using that method not polymorphism and the compiler won't warn you if we mismatch signature, this is risky and can cause runtime problems.

Virtual keyword + override keyword :

Virtual keywords are used for base class to enable runtime polymorphism.

Override keyword is used for the compiler to check if the signatures are match correctly.

Function hiding example working.

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

class Base {
public:
    void print() const{
        cout << "Base print" << endl;
    }
};

class Derived : public Base {
public:
    void print() const{
        cout << "Derived print" << endl;
    }
};

int main() {
    Base base;
    base.print();
    Derived derived;
    derived.print();
    return 0;
}
```

This code will work fine until we use upcasting or downcasting:

```
int main() {
    Base* base = new Derived(); // Upcasting
    base->print();
    return 0;
}
```

Output will be : " Base print".

Virtual keyword + override keyword

A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method

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

class Base {
public:
    virtual void print() {
        cout << "Base print" << endl;
    }
};

class Derived : public Base {
public:
    void print() override{
        cout << "Derived print" << endl;
    }
};

int main() {
    Base* base = new Derived(); // Upcasting
    base->print();
    // Ignore downcast syntax i searched,how we downcast in
    // WARNING: Unsafe downcasting. May cause undefined
    // behavior!
    Derived* derived = static_cast<Derived*>(new Base()); //
    down casting
    derived->print();

    // Safe down casting using dynamic_cast
    Base* base3 = new Derived(); // Actually a Derived object
    Derived* derivedSafe = dynamic_cast<Derived*>(base3);
    if (derivedSafe) {
        derivedSafe->print(); // Output: Derived print (safe
        downcast)
    } else {
        cout << "Downcast failed!" << endl;
    }
    delete base;
    delete derived;
    delete base3;
    return 0;
}
```

This example will work with up/down casting and override correctly.

Type casting Real Time Type Information (less common use in C++)

C++ has 3 styles of casting

- 1) `reinterpret_cast(expression)`
- 2) `static_cast(expression)`
- 3) `dynamic_cast(expression)`

reinterpret_cast operator:

Used for casting **POINTER TYPE** to another **POINTER TYPE**

basic example, we want to cast an `int*` pointer to `double*` pointer.

This type of casting is very dangerous and used on rare occasions only.

Example for using that casting is for writing image files.

static_cast operator:

Used for casting between types when the conversion is known at compile time.

Examples include primitive type conversions like `double → int → double`.

It can also be used for upcasting (converting Derived → Base class pointers).

In C++, upcasting is always safe because it is an implicit conversion handled by the compiler.

Conversion constructor / Conversion operator:

In classes, we can define:

- Conversion constructor: Enables converting from another type to the class type.
- Conversion operator: Enables converting from the class type to another type.

These allow us to control how objects of a class can be converted to or from other types.

The `static_cast` operator can be used alongside these to explicitly perform such conversions when needed.

dynamic_cast operator:

Used when the dynamic type of the object is not known at compile time.

It is typically used for **safe downcasting** in polymorphic class hierarchies (**classes with at least one virtual function**).

If the object **does not actually match the target type**, `dynamic_cast` returns:

- A **`nullptr`** when casting pointers.
- Throws a **`bad_cast`** exception when casting references.

This allows you to check whether the cast is valid at runtime, preventing undefined behavior.

	Compile-time	Run-time
<code>reinterpret_cast</code>	Checks if source & target are pointers or references	No run-time checks at all (dangerous)
<code>static_cast</code>	Checks if conversion exists (upcasting, primitive)	Applies fixed conversion (no run-time check)
<code>dynamic_cast</code>	Checks if class is polymorphic (has virtual functions)	Expensive run-time type check (safe downcasting)

Templates

Function Templates

When we define a function we usually pass some variables (int, double, members, etc...), but when we define a function it allow only that type of variables we defined, for example we want to make a swap function for int, our function will take 2 int variables and swap between them, but lets say we want to swap double, now we need to define a new function that take a double variables, so we define the same function twice.

Function templates allow us to define a generic function that can take and return unknown types. This helps us avoid writing multiple overloaded functions manually, as the compiler automatically generates the correct version based on the argument types.

To make a function template first we need to define the template type, at the start of the code.

Template functions are declaration, we will define in header file.

```
template <typename T>
```

Now we can make the swap function with T type.

```
void temp_swap(T& x, T& y) {  
    T temp = x;  
    x = y;  
    y = temp;  
}
```

We can use T type to return too.

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

If we use the same typename for multiple functions, we need to redefine that typename before each function (if we won't define, we will have a compiler error).

Summary made by: Shachar Tsrafati

We can use multiple typename in one method and use the keyword auto, to automatically return the correct type we want, for example.

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

In this example when we use the keyword auto, the compiler knows which type of var to return, let's say X is an int and Y is a double, if X is bigger than Y, then it will return int type, else it will return a double type.

Function templates can be used for classes too.

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

class A {
public:
    int a;

    A(int a) {
        this->a = a;
    }
};

template <class Banana> // temp name for class
void classSwap (Banana& a, Banana&b) {
    Banana c = a;
    a = b;
    b = c;
}

int main() {
    A a = {5};
    A b = {10};
    cout << "a.a = " <<a.a << endl;
    cout << "b.a = " <<b.a << endl;
    classSwap(a,b);
    cout << "a.a = " <<a.a << endl;
    cout << "b.a = " <<b.a << endl;
    return 0;
}
```

Keep in mind that if we got the same function with the same template, we would have compilation error.

Class Template

Like function templates, C++ provides **Class Templates**.

Class templates are useful when a class works with different data types, but the logic is the same regardless of type. They help avoid writing the same class multiple times for different types. This is especially useful in data structures like LinkedList, Stack, Queue, etc.

Key Points:

- Class templates allow us to define a class with **unknown types** (generic types).
- You can define **one or more type parameters**.
- They provide flexibility, eliminating the need to duplicate code for different types.
- You can combine **class templates, function templates, and overloading** together.

Example for class template with multiple parameters combined with function template for flexible and reusable behavior.

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

template <typename T, typename U>
class Box {
    T valueT;
    U valueU;
public:
    Box(T valueT, U valueU) {
        this->valueT = valueT;
        this->valueU = valueU;
    }

    void show() {
        cout << "ValueT: " << this->valueT << endl << "ValueU: " << this->valueU << endl;
    }
};

template <typename T, typename U, typename W, typename Z>
void show(Box<T,U> T_U, Box<W,Z> W_Z) {
    cout << "Show class template with T_U and W_Z" << endl;
    cout << "Show class T_U" << endl;
    T_U.show();
    cout << "Show class W_Z" << endl;
    W_Z.show();
}

int main() {
    Box intChar (10, 'a');
    Box doubleString(5.5, "Hello world");
    intChar.show();
    doubleString.show();

    show(intChar, doubleString);

    return 0;
}
```

Template Variations

A template can receive several arguments, look at the example above.

Template Specialization

Template class specialization allows targeting specific types to define custom behavior.

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

template <typename T, int N>
class Array {
private:
    T (&array) [N];
public:
    Array(T (&array) [N]) : array(array) {
        cout << "General Specialized" << endl;
    }
    // code here
};

template <int N>
class Array < double ,N> {
private:
    double (&array) [N];
public:
    Array(double (&array) [N]) : array(array) {
        cout << "Double Specialized" << endl;
    }
    // code here
};

int main() {
    int arrInt[] = {1,2,4,5,8,10};
    double arrDouble[] = {10.5,3.7,4.0,9.7,8.1};
    float arrFloat[] = {1.21423,2.2234};
    Array intArray (arrInt);
    Array doubleArray(arrDouble);
    Array floatArray(arrFloat);
    return 0;
}
```


Template Meta-Programming

Template Meta-Programming is used to pre-calculate algorithms at compile time, reducing runtime overhead and improving execution speed.

```
// primary template computes 3 to the Nth
template<int N> class Pow3 { public:
    static const int result =
        3*Pow3<N-1>::result;
};
// full specialization to end recursion
template<> class Pow3<0> { public:
    static const int result = 1;
};
int main() {
    cout << Pow3<1>::result<<"\n"; //3
    cout << Pow3<5>::result<<"\n"; //243
    return 0;
}
```

The Standard C++ Library- Looping on containers

Tuple

Tuples are collection in C++, To define an array with multiple different variables with a fixed size.

Defining tuple: To define tuple in C++ we need to include <tuple>.

Tuple is a class template word in C++ to create a tuple we need to write.

```
tuple <var,var,var,var...> t1;
tuple <int,char,string> t1; // for example t1 =
make_tuple(5,'a',"Hello World");
```

Note: using the word "make_tuple" is convenient because it automatically deduces the types.

```
//tuple <int,char,string> t1;
auto t1 = make_tuple(5,'a',"Hello World");
```

We can combine auto + make_tuple to define in one

line. Get and Set:

To get a variable from a tuple we can get from few ways

- 1) If the tuple does contain one of each type of variable (NO DUPES) we can use a quick way to get that variable

```
auto t1 = make_tuple(5,'a',"Hello World");
cout << get<int>(t1) <<endl;
```

- 2) If the tuple does contain more than one of each type of variable (DUPES) we need to define which index, we want to get

Summary made by: Shachar Tsrafati

```
auto t1 = make_tuple(5,7,"Hello World");  
cout << get<1>(t1) <<endl; // this will print 7
```

Note: we can use get with index in both methods.

Set:

If we want to change a variable, we simply use get = the variable we want to change

```
auto t1 = make_tuple(5,7,"Hello World");  
cout << get<1>(t1) <<endl; // this will print 7  
get<1>(t1) = 100;  
cout << get<1>(t1) <<endl; // this will print 100
```

Copy tuple to another tuple:

We can copy tuples to another tuple (We want to use them when both tuples have the same type, we can copy from double to int, but it is not recommended)

```
tuple<int,bool> t1;  
tuple<int,bool> t2;  
t1 = make_tuple(5,true);  
t2 = t1;  
auto t3(t2);  
cout << std::get<0>(t1) <<endl;  
cout << std::get<0>(t2) <<endl;  
cout << std::get<0>(t3) <<endl;
```

Swapping tuples:

we can use the swap method to swap 2 tuples, t2.swap(t1);

Unpacking tuples:

We can unpack a tuple using the keyword "tie"

```
tuple<int,bool> t1;  
t1 = make_tuple(5,true);  
int x;  
bool y;  
tie(x,y) = t1;  
cout << x <<endl;  
cout << y <<endl;
```

Concatenating Tuples:

In C++ we can concatenate tuples to one big tuple; to concatenate we need to use the method "tuple_cat".

Before we can concatenate multiply tuples to one big tuple, we need to define the tuple and the order we want to concatenate them, without defining it won't work.

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

tuple <int, char> t1 (23, 'H');
tuple <char, string> t2 ('B', "Hello World");
// you must define t3 first, before using tuple_cat
tuple <int, char, char, string> t3 = tuple_cat(t1, t2);

int main() {
    cout << get<0> (t3) << endl;
    cout << get<1> (t3) << endl;
    cout << get<2> (t3) << endl;
    cout << get<3> (t3) << endl;

    return 0;
}
```

Vectors

Vector in C++ is dynamic array that we can change its size.

Like an array, vector has the same data structure used to store multiple elements of the same data **type**.

To use a vector, we need to include it. #include <vector>

Create a vector:

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

int main() {
    vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
    for (const string& car: cars) {
        cout << car << "\n";
    }

    return 0;
}
```

Access a vector:

To access an element inside the vector it is the same as accessing an element inside an array, using operator [].

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};  
cout << cars[0] << endl;
```

We can use the method "at" to access an element inside the vector, it is the same as using operator[].

Front and Back:

There are 2 methods inside the vector data structure for quick access, "front, back".

Front → first element

Back → last element

```
cout << cars.front();  
cout << cars.back();
```

Changing an element:

Like an array, we can change an element inside the vector after we define the vector.

We can use the operator[] or use the method "at"

```
cars.at(0) = "TANK";
```

This will change the value inside index 0 to "TANK".

Adding vector element:

As we said, the advantage of using vector is to have the ability to add new elements after we initialize.

To add elements, we will use the method "push.back()".

```
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};  
cars.push_back("Tesla");
```

We can push as many elements as possible. Keep in mind that vector acts like an array it means you can have the same element multiply times inside.

Removing an element:

Like adding elements, we can remove elements using the method "pop_back()".

The problem with removing elements using the method "pop_back()", this will always remove the last element inside the vector.

To remove an element from the middle we need to use the method "erase()" with some "tricks".

To remove an element from the middle the method erase will get inside it "vector.begin() + index_Number_we_want_to_remove).

```
cars.erase(cars.begin() + 3);
```

We can not just type 0 + 3, or just 3, it won't compile.

Bool Empty:

We can use method `empty()`, to test if vector is empty or not.

Output = 1 → Empty

Output = 0 → Not Empty

Other methods that are good to know:

`Size()` → return the size of the vector (numbers of elements inside (including nulls)).

`Capacity()` → return the amount of storage allocated (in elements).

`Insert(index, element)` → allow us to insert new element before that index.

`Clear()` → clear the vector (removing all the elements inside of the vector).

`Resize(number)` → Grow OR Shrink a vector (If growing it will add empty elements) if shrink it will remove the elements from end to start.

`Reserve(number)` → Pre-allocate capacity

`T* pointer = vector.data()` → allow us to use pointers to access the vector

`Emplace_back()` → allow us to insert element at the end of the vector. It is more efficient as it avoids unnecessary copying or moving.

`Shrink_to_fit()` → change the vector capacity to match the number of elements inside the vector.

`Swap(vectorA,vectorB)` → swapping 2 vectors **with the same data structure**.

Sets

Sets in C++ are data structures like arrays, but by default, their elements are **automatically sorted** and **unique** (no duplicates).

To use sets, we need to include the header:

```
#include <set>
```

To create a set, the syntax is similar to vectors:

```
set<int> s = {1, 4, 2};
```

Adding element to set:

To add elements to a set, we use the methods `insert()` or `emplace()`:

```
s1.insert(8);  
s.emplace(10);
```

Access an element:

Unlike vectors or arrays, we **cannot** use the `[]` operator or the `at()` method, because a set is **not index-based**, and its elements are automatically sorted.

To access a specific position (like the second or third element), use **iterators**. You can advance an iterator using the `next()` function from `<iterator>`:

```
auto it = next(s.begin(), 2);
```

Finding elements:

To search for an element inside the set, use the `find()` method. If the element exists, it returns an iterator to it; otherwise, it returns `s.end()`:

```
set<int> s = {1, 4, 2, 3, 5};  
auto it = s.find(3);  
if (it != s.end()) {  
    cout << *it;  
}else {  
    cout << "Element not found !" <<endl;  
}
```

Update elements:

You cannot directly modify elements in a set, because modifying them could violate the sorted order or uniqueness. If you want to update a value, remove it and insert a new one.

Looping inside Set:

Looping inside set is easy just like any other container. Simple for loop.

Deleting elements:

To delete an element from a set, use the `erase()` method. You can erase by value or by iterator.

```
s.erase(5); // by value
s.erase(s.begin()); // by index
```

Maps

Maps are containers that store data in the form of **key-value pairs**, sorted based on the keys. No two elements in a map can have the same key. By default, the elements are stored in **ascending order of the keys**, but this can be changed if needed.

```
map<int, string> mp1 = {
    {1, "Hello"}, {2, "World"}
};
```

Adding element + Looping:

You can add elements using the `insert()` method or by using the bracket `[]` operator:

```
map<int, string> mp1 = {
    {1, "Hello"}, {2, "World"}
};
mp1.insert({3, "You"});
mp1[4] = "Me";

for (const auto &t : mp1) {
    cout << "Key (int): " << t.first << " String: " <<
t.second << endl;
}
```

Accessing element:

Map elements can be accessed by using the operator `[]` or using the method `"at"`.

```
cout << mp1[1] << endl;
cout << mp1.at(2) << endl;
```

Update element:

In maps we can change elements, but we cannot change the keys, to update an element can be done with the operator `[]` or with the method `"at"`.

```
mp1[1] = "No";
mp1.at(2) = "Way";
```

Finding element:

Map provides fast element search by key using the **find()** member function. This function returns iterator the element if found, otherwise returns end() iterator.

```
auto it = mp1.find(2);
if (it != mp1.end()) {
    cout << "Key: " << it->first << ", String: " << it->second
<< endl;;
} else {
    cout << "Key not found";
}
```

Traversing:

Maps can be easily traversed by using either **range based for loop** or using **begin()** and **end()** iterator with traditional loops:

```
for (const auto &it: mp1) {
    cout << "Key: " << it.first << ", String: " << it.second
<< endl;
}
```

Deleting elements:

Map elements can be deleted from a map using **erase()** method by passing the key or an iterator.

```
mp1.erase(2);
for (const auto &it: mp1) {
    cout << it.first << ", " << it.second << endl;
}
```

<type_traits>

<type_traits> contains helper classes for:

- Creating compile-time constants
- Querying type characteristics as compile-time values
- Transforming types by applying specific compile-time modifications

To learn more about <type_traits>, [consult the C++ standard documentation](#). Most uses of <type_traits> involve template metaprogramming and specialization to impose constraints or implement type-dependent algorithms.

[Geeks for geeks](#)

Stl-algorithm and iterators

- **STL = Standard Template Library**
- A set of **container** and **container-adapter** classes (e.g. vector, list, stack, queue, set, map) plus a suite of **non-member algorithms** and **iterator** types.

Iterators:

Iterators are the bridge between containers and algorithms. Every STL algorithm works on an iterator range [first, last) so you only write the algorithm once, and it works for any container (or even raw arrays, streams, or custom ranges) that exposes the proper iterator type.

Category	Operations	Example iterator
Input	*it (read), ++it (single-pass)	std::istream_iterator<T>
Output	*it = value, ++it (single-pass)	std::ostream_iterator<T>
Forward	*it (read/write), ++it (multi-pass)	std::forward_list<T>::iterator
Bidirectional	above + --it	std::list<T>::iterator, std::map<K,V>::iterator
Random-Access	above + pointer-arithmetic (it+n, it[-k])	std::vector<T>::iterator, raw pointers

The “105” Algorithms:

Family	What it does	Key examples
Queries	inspect without changing	find, count, binary_search, accumulate
Permutations	reorder within range	next_permutation, prev_permutation
Set ops	on sorted ranges	set_union, set_intersection, set_diff
Movers	copy/move between ranges	copy, copy_if, move, swap_ranges
Value modifiers	change elements in place	fill, generate, replace, transform
Structure changers	squeeze/reorder without resize	remove, remove_if, unique, partition
Raw-memory	uninitialized-memory ops	uninitialized_copy, uninitialized_fill

Note: Container member functions (push_back, pop, enqueue, dequeue) are not counted among the 105—they live in the container’s own API.

Why Learn Them All?

- Reliability: battle-tested, no surprises.
- Performance: optimal asymptotic complexity.
- Expressiveness: one call replaces dozens of handwritten loops.
- Maintainability: everyone “speaks the same language.”

Modifying containers:

Container-modifying member calls:

```
c.insert(it, x);           // insert single element before it
c.insert(it, first, last); // insert range [first,last)
                           // before it
c.emplace(it, args...);    // construct in-place before it
it = c.erase(it);          // erase element at it, returns
                           // next valid iterator
c.erase(it, jt);           // erase [it,jt), returns jt
```

Iterator invalidation rules:

- **list, set, map:** only the erased iterator is invalidated; all others remain valid.
- **vector, deque:** inserting or erasing at position it invalidates all iterators at or after it (elements shift).

Safe-erase idiom:

```
for (auto it = c.begin(); it != c.end(); /*no ++it*/) {
    if (should_remove(*it))
        it = c.erase(it); // use returned iterator
    else
        ++it;
}
```

Smart Pointer

Smart pointers are template classes (in <memory>) that wrap a raw pointer to automatically manage dynamic memory.

There are 3 types of smart pointers:

- 1) Unique pointer
- 2) Shared pointer
- 3) Weak pointer

Unique pointer

```
unique_ptr<int> unPtr1 = make_unique<int>(25);  
cout << &unPtr1 << endl; // Address  
cout << *unPtr1 << endl; // Dereference
```

Unique pointers are not able to be shared

```
unique_ptr<int> unPtr1 = make_unique<int>(25);  
unique_ptr<int> unPtr2 = unPtr1; // WILL NOT WORK
```

ERROR:

Attempt to use deleted constructor

```
std::unique_ptr<int>::unique_ptr(const unique_ptr<int> &)
```

Unique pointers can be moved to other unique pointers

```
unique_ptr<int> unPtr1 = make_unique<int>(25);  
unique_ptr<int> unPtr2 = move(unPtr1);  
  
cout << *unPtr2 << endl; // will print 25
```

The moment you move a unique pointer the **pervious unique pointer becomes a null pointer**, if you try to access the previous pointer then you will get a null pointer exception.

```
unique_ptr<int> unPtr1 = make_unique<int>(25);  
unique_ptr<int> unPtr2 = move(unPtr1);  
  
cout << *unPtr2 << endl;  
cout << *unPtr1 << endl;
```

Unique pointers are known for deallocated automatic memory.

```
#include <iostream>
#include <memory>
using namespace std;
class MyClass {
public:
    MyClass() { cout << "Constructor invoked\n"; }
    ~MyClass() { cout << "Destructor invoked\n"; }
};

int main() {
    unique_ptr<MyClass> unPtr1 =make_unique<MyClass>();

    cout << "Bla bla bla\n";

    return 0;
}
```

Shared pointer

shared pointers are pointers that can be shared by multiple owners, we can know how many pointers are been shared by using the method “use_count()”.

```
#include <iostream>
#include <memory>
using namespace std;
int main() {
    shared_ptr<MyClass> shared_ptr1 = make_shared<MyClass>();
    cout << "Share count: " << shared_ptr1.use_count() << '\n';

    shared_ptr<MyClass> shared_ptr2 = shared_ptr1;
    cout << "Share count: " << shared_ptr2.use_count() << '\n';
}
```

If we apply pointers inside of a scope, after the program exits that scope, it will destroy all the shared pointers that inside scope.

Weak pointer

The main difference between weak pointer to shared pointer when you assign a specific memory location to a shared pointer there is increase the number of owners of that

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

int main() {
    shared_ptr<int> shared_ptr1 = make_shared<int>(25);
    cout << "Share count: " << shared_ptr1.use_count() << '\n'; {
        shared_ptr<int> shared_ptr2 = shared_ptr1;
        cout << "Share count: " << shared_ptr1.use_count() << '\n';
    }
    cout << "Share count: " << shared_ptr1.use_count() << '\n';
}
```

memory location. But if we assign that same memory location to a weak pointer that will not increase the share owner count.

We use weak pointers to observe objects in memory but will not keep that object alive if nothing else needed.