Summary made by: Shachar Tsrafati

C++ Summary

Contents

Summary made by: Shachar Tsrafati

Start:

Note: Elina said we are working on C++ version 17, All the codes in this file should work, make sure you test on version 17.

There will be mistakes 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 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)
```

Summary made by: Shachar Tsrafati

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;
}
```

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() {
```

Summary made by: Shachar Tsrafati

```
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
}
```

Summary made by: Shachar Tsrafati

User input:

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

std :: string name;

std :: cout << “Enter name: ”;
std :: cin >> name;

std :: cout << “Hello “ << name;
```

If , Switch, For loop, While loop, break , continue , works like in java.

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:

```
#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);


    // Clean up (not strictly needed here since structs are on
stack, but if dynamically allocated, you'd delete)
    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;
}
//////////////////////////////////
#ifndef 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.

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

class Number {
private:
    double *number;

public:
    // 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
    }


};

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 };
```

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)
```

```
#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.

```
#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;
```

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

```cpp
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;
    }


    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;
}
```

Summary made by: Shachar Tsrafati

Shallow Copy

```cpp
#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.

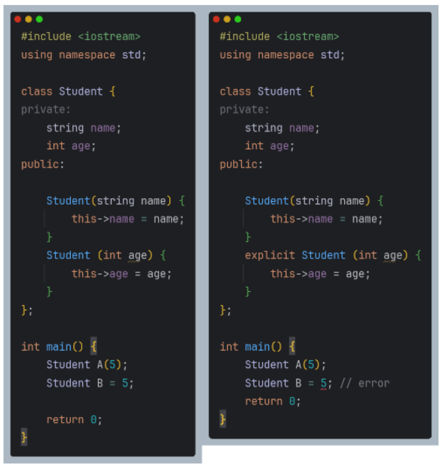
In the **shallow copy**, we use pointers as member variables, so copying copies the addresses (pointers), meaning both objects share the same memory.

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 give an error.



```
#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:
Similar to 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 complier 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:

```
#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
c++
    // 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 (Week 7, Removed)

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

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>
```

Summary made by: Shachar Tsrafati

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 complier error).

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 complier 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");
```

Summary made by: Shachar Tsrafati

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

```
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 tuple to another tuple (We want to use it 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);**

Summary made by: Shachar Tsrafati

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;
```