```
g++ -o program file1.cpp file2.cpp
```

```
./prog < input.txt // input
./prog > output.txt // output
./prog < input.txt > output.txt // both
```

```
int x = 3, y = 4;
int** arr2D = new int*[x];
for (int i = 0; i < x; ++i) arr2D[i] = new int[y];

arr2D[1][2] = 42;
```

```
const int x = 3; // const not required
const int y = 4;
// Allocate a non-dynamic 2D array
int arr2D[x][y];
```

```
#include <cmath>
```

pow(x, y)	Raising to a power x^y
sqrt(x)	Square root $\sqrt{x}$
log10(x)	Decimal log $\log_{10}(x)$
abs(x)	Absolute value $\lvert x \rvert$
sin(x), cos(x), tan(x)	Sine, cosine, tangent of x (x in radians)

```
// Extracting a substring from index 7 to the end
std::string substring1 = originalString.substr(7);

// Extracting a substring from index 0 to 5
std::string substring2 = originalString.substr(0, 5);

// Extracting a substring from index 7 with length 5
std::string substring3 = originalString.substr(7, 5);
```

```
// Class declaration
class MyClass {
public:
    // Member function declaration
    void printMessage(const std::string& message);
};

// Member function definition outside the class
void MyClass::printMessage(const std::string& message) {
    std::cout << "Message from MyClass: " << message << std::endl;
}
```

```
Developer(string name, string company, int age, string favProgrammingLanguage)
    :Employee(name, company, age)
{
    FavProgrammingLanguage = favProgrammingLanguage;
}
```

```
vector<double> vec = {12.1, 23.4, 234.5, 23.0};
```

```
// Child class constructor
ChildClass() : ParentClass() {
```

obj-> is the same as (*obj). arr[1] is the same as *(arr+1)

```
Include “example.hpp”
```

```
    // Using default constructor
    MyClass obj1;
    // Using overloaded constructor
    MyClass obj2(42);
    // Using new keyword to dynamically allocate an object with overloaded constructor
    MyClass* dynamicObj = new MyClass(99);
```

```
// Initialize a vector
std::vector<int> myVector;
// Add an item at the end
myVector.push_back(42);
// Remove an item at the end
myVector.pop_back();
// Remove an item at a specific index using erase()
// while preserving order in the vector. Say the index
// you want to remove is 2.
myVector.erase(myVector.begin() + 2);
```

```
// Base class
class Animal {
public:
    // Virtual function
    virtual void makeSound() {
        std::cout << "Animal makes a generic sound." << std::endl;
    }
};

// Derived class
class Dog : public Animal {
public:
    // Override function
    void makeSound() override {
        std::cout << "Dog barks." << std::endl;
    }
};
```

```
// Derived class inheriting from both Shape and Color
class ColoredShape : public Shape, public Color {
public: // Shape and Color defintion not shown
    void drawAndColor() {
        // you can call inherited functions like this
        draw();          // Inherited from Shape
        printColor();    // Inherited from Color
```

```
Protected:
Members declared as protected are accessible within the same class and by derived classes.

Private:
Members declared as private are accessible only within the same class.

Public:
Members declared as public are accessible from anywhere in the program
```

```
class Animal {
public:
    void makeSound() const {
        std::cout << "Generic animal sound" << std::endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() const override {
        // Call the makeSound method from the superclass (Animal)
        Animal::makeSound();

        // Add additional behavior specific to Dog
        std::cout << "Dog barks" << std::endl;
    }
};
```

```
// linknking 2 cpp files
g++ -o program Test.cpp Test2.cpp
```

```
// Initializing an object with the default constructor
MyClass obj1;

// Initializing an object with the overloaded constructor
MyClass obj2(42);
```

```
//normal delete
delete obj; obj = nullptr;

//delete dynamic array
delete[] intArray;
intArray = nullptr;

//delete 2d dynamic array
for (int i=0; ( < rows; ++i) {
//deleting subarrays
delete[] twoDArray[i];
twoDArray[i] = nullptr;
}
//delete main array too
delete[] twoDArray; twoDArray nullptr;
```

```
class Rectangle
{
public:
    Rectangle(double l, double w);
    double get_perimeter();
    double get_area();
    void resize(double factor);
private:
    double length;
    double width;
};

Rectangle::Rectangle(double l, double w);
{
    length = l;
    width = w;
}
```

```
double Rectangle::get_perimeter()
{
    return 2 * (length + width);
}

double Rectangle::get_area();
{
    return length * width;
}

void Rectangle::resize(
                    double factor);
{
    width = length * factor;
    width = length * factor;
}
```

```
// abstract class because of pure virtual function
class myAbstractClass{
    // cant be instantiated on it's own
    virtual void exampleFunction() = 0; // must override
};
```

```
class Employee{
// if access modifier not specified
// private by default
    int age;
    string name;
};
```

```
std::swap(obj x, obv y)
```
// swap 2 value of same type

```
std::string longString = "This is a long sentence.";
std::string portion = longString.substr(5, 7); // Extracts "is a lo"
```

```
char source[] = "Hello, ";
char destination[20];

strcpy(destination, source);

char append[] = "World!";
strcat(destination, append);
```

Enumerations, Switch Statement

```
enum Color { RED, GREEN, BLUE };
Color my_color = RED;

switch (my_color) {
  case RED :
    cout << "red";  break;
  case GREEN:
    cout << "green"; break;
  case BLUE :
    cout << "blue";  break;
}
```

String Operations

```
#include <string>
string s = "Hello";
int n = s.length(); // 5
string t = s.substr(1, 3); // "ell"
string c = s.substr(2, 1); // "l"
char ch = s[2]; // 'l'
for (int i = 0; i < s.length(); i++)
{
   string c = s.substr(i, 1);
   or char ch = s[i];
   Process c or ch
}
```

*operator= serves multiple purposes: it's used for multiplication (e.g., a * b), for dereferencing pointers to access the value they point to (e.g., *ptr), and for declaring pointer variables (e.g., int* ptr).

& operator=used for referencing and bitwise operations: it obtains the memory address of a variable (e.g., &var gives the address of var). Additionally, & is used in function signatures to denote reference parameters, allowing functions to modify the original argument. It is also used in binary operations, it performs bitwise AND (e.g., a & b). It can be also used in the logical operator and (&&).

:: operator = known as the scope resolution operator, is used for accessing class members (e.g., ClassName::memberFunction), differentiating members between namespaces (e.g., NamespaceName::FunctionName), and resolving scope ambiguities when identical names exist in different scopes. It's essential for accessing static and enum class members and for clearly specifying which namespace or class scope a function or variable belongs to.

: operator = the colon : is used for defining class inheritance (e.g., class Derived : public Base) and initializing member variables in constructors (e.g., Constructor() : memberVar(value) {}).

-> operator = the -> operator is a shorthand for accessing members of an object through a pointer. It combines dereferencing a pointer and accessing a member, typically used as pointer->member. Essentially, pointer->member is equivalent to (*pointer).member. When pointer is a pointer to a class or struct, pointer->member first dereferences pointer to access the object it points to, and then accesses the member of that object.

New = used for dynamic memory allocation. It allocates memory for objects or arrays at runtime from the heap, and returns a pointer to the beginning of the newly allocated memory. Unlike automatic or static memory allocation, the size and type of objects allocated with new can be determined at runtime, providing flexibility and control over memory management.'

Sizeof = operator in C++ is used to determine the size, in bytes, of a type or object at compile time.

Delete or delete[]= complements new by freeing the dynamically allocated memory. When you allocate memory using new, you're responsible for using delete to release that memory back to the system, preventing memory leaks. delete is used for single objects, and delete[] for arrays. Make sure to set the pointer to nullptr.

Static Variables

```
void increment_print() {
  static int s_value = 0; //static duration
  s_value++;
  cout << s_value << '\n';
} //s_value is not destroyed, but goes out of scope
int main() {
  increment_print(); //1
  increment_print(); //2
}
```

Static Data Members

```
class Item {
private:
   int m_id;
   static int s_id_counter;
public:
   Item() {
      m_id = s_id_counter++;
   }
   int get_id() const {
      return m_id;
   }
};
int Item::s_id_counter = 1;
int main() { //
   Item first;
   Item second;
   cout << first.get_id();  //1
   cout << second.get_id();//2
}
```

Input and Output

```
#include <iostream>
cin >> x; // x can be int, double, string
cout << x;
while (cin >> x) { Process x }
if (cin.fail()) // Previous input failed
#include <fstream>
string filename = ...;
ifstream in(filename);
ofstream out("output.txt");
string line; getline(in, line);
char ch; in.get(ch);
```