**KeyWords in C++**

| **C++ Keyword** | | | |
| --- | --- | --- | --- |
| **asm** | **double** | **new** | [**switch**](https://www.geeksforgeeks.org/switch-statement-cc/) |
| **auto** | **else** | **operator** | **template** |
| **break** | **enum** | **private** | **this** |
| **case** | **extern** | **protected** | **throw** |
| **catch** | **float** | **public** | **try** |
| **char** | **for** | **register** | **typedef** |
| **class** | **friend** | **return** | **union** |
| **const** | **goto** | **short** | **unsigned** |
| **continue** | [**if**](https://www.geeksforgeeks.org/check-if-given-number-is-perfect-square-in-cpp/) | **signed** | **virtual** |
| **default** | **inline** | **sizeof** | **void** |
| **delete** | **int** | **static** | **volatile** |
| **do** | **long** | **struct** | **while** |

**1.Template:**

* Templates enable the creation of functions or classes that can work with different types without having to rewrite the code for each specific type.
* The typename T is a template parameter that represents a generic type(any data type).

Ex:

#include <iostream>

template<typename T>

T maximum(T a, T b) {

return (a > b) ? a : b;

}

int main() {

int maxInt = maximum(10, 5); // maximum<int>(10, 5)

float maxFloat = maximum(3.14f, 2.71f); // maximum<float>(3.14f, 2.71f)

char maxChar = maximum('a', 'b'); // maximum<char>('a', 'b')

std::cout<<maxInt;

}

EX-2:

#include <iostream>

#include <algorithm>

using namespace std;

template<typename T>

void sortArray(T arr[], int size) {

sort(arr, arr + size);

}

int main() {

int intArray[] = { 5, 2, 8, 1, 9 };

double doubleArray[] = { 3.14, 2.71, 1.618, 0.577 };

char charArray[] = { 'd', 'a', 'b', 'c' };

int intSize = sizeof(intArray) / sizeof(int);

int doubleSize = sizeof(doubleArray) / sizeof(double);

int charSize = sizeof(charArray) / sizeof(char);

sortArray(intArray, intSize);

sortArray(doubleArray, doubleSize);

sortArray(charArray, charSize);

// Print sorted arrays

for (int i = 0; i < intSize; i++)

cout << intArray[i] << " ";

cout << endl;

for (int i = 0; i < doubleSize; i++)

cout << doubleArray[i] << " ";

cout << endl;

for (int i = 0; i < charSize; i++)

cout << charArray[i] << " ";

cout << endl;

return 0;

}

**2.try, 3.catch, 4.throw**

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

SYNTAX:

try {

// Block of code to try

throw *exception*; // Throw an exception when a problem arise

}

catch () {

// Block of code to handle errors

}

EX:

#include <iostream>

#include <stdexcept>

using namespace std;

int main() {

bool success = false;

while (!success) {

try {

int numerator, denominator;

cout << "Enter the numerator: ";

cin >> numerator;

cout << "Enter the denominator: ";

cin >> denominator;

if (denominator == 0) {

throw runtime\_error("Division by zero is not allowed.");

}

if (denominator < 0) {

throw invalid\_argument("Denominator must be greater than or equal to zero.");

}

int result = numerator / denominator;

cout << "Result: " << result << endl;

success = true; // Input was valid, exit the loop

} catch (const runtime\_error& e) {

cerr << "Runtime Error: " << e.what() << endl;

} catch (const invalid\_argument& e) {

cerr << "Invalid Argument Error: " << e.what() << endl;

} }

return 0;

}

const exception& e declares a reference variable e of type std::exception (or any derived type) that refers to the caught exception object.

const indicates that the object referred to by e cannot be modified within the catch block.

exception is the type of exception being caught. It is a base class for various exception types in the C++ Standard Library.

& denotes that e is a reference to the caught exception object.

**5.new**

def:new keyword is used for dynamic memory allocation. It allows you to allocate memory dynamically during runtime from the heap (free store) and returns a pointer to the allocated memory.

ex:#include <iostream>

int main() {

int\* dynamicValue = new int;

\*dynamicValue = 10;

std::cout << "Dynamic Value: " << \*dynamicValue << std::endl;

delete dynamicValue;

return 0;

}

EX:2

#include <iostream>

int main() {

int size;

std::cout << "Enter the size of the array: ";

std::cin >> size;

int\* dynamicArray = new int[size]; // Dynamically allocate an array of integers

std::cout << "Enter " << size << " integers: ";

for (int i = 0; i < size; i++) {

std::cin >> dynamicArray[i];

}

std::cout << "Array elements: ";

for (int i = 0; i < size; i++) {

std::cout << dynamicArray[i] << " ";

}

std::cout << std::endl;

delete[] dynamicArray; // Deallocate the dynamically allocated array

return 0;

}

Need:

Object Lifetime Control: Dynamically allocated objects persist until explicitly deallocated using delete, allowing you to control the lifetime of objects. This is especially valuable when dealing with objects that need to exist beyond the scope of a particular function or block.

Variable Size Containers: Dynamic memory allocation enables the creation of containers, such as arrays or linked lists, whose size can be determined at runtime.

Resource Allocation:

**6.Access specifiers →public, 7.private and 8.protected**

**Public:**

class members that are accessible from outside the class

Ex:

#include <iostream>

class MyClass {

public:

int publicVariable;

void publicFunction() {

std::cout << "This is a public function." << std::endl;

}

};

int main() {

MyClass myObject;

myObject.publicVariable = 42; // Accessing public member variable

myObject.publicFunction(); // Accessing public member function

return 0;

}

**Private:**

def:Members declared as private are only accessible within the same class and are not accessible from outside the class, including derived classes.

The significance of the private access specifier is that it provides encapsulation and data hiding. It helps to protect the internal implementation details of a class and ensures that the class's private members are not accessible directly from outside the class

#include <iostream>

#include <string>

using namespace std;

class BankAccount {

private:

string accountNumber;

double balance;

void performInternalTransaction(double amount) {

// Perform internal transaction operations

balance += amount;

}

Public:

//constructor: Constructors are automatically called when an object is created and can be used to set initial values for the object's member variables or perform any necessary setup operations.

BankAccount(const string& accNum, double initialBalance) {

accountNumber = accNum;

balance = initialBalance;

}

void deposit(double amount) {

performInternalTransaction(amount);

cout << "Deposit of $" << amount << " processed successfully." << endl;

}

void withdraw(double amount) {

if (balance >= amount) {

performInternalTransaction(-amount);

cout << "Withdrawal of $" << amount << " processed successfully." << endl;

} else {

cout << "Insufficient funds. Withdrawal failed." << endl;

}

}

void displayBalance() {

cout << "Account Number: " << accountNumber << endl;

cout << "Current Balance: $" << balance << endl;

}

};

int main() {

BankAccount account("123456789", 1000.0);

account.deposit(500.0);

account.withdraw(200.0);

account.displayBalance();

return 0;

}

By making these member variables private, their internal state is encapsulated within the class. This ensures that the internal data is protected and can only be accessed or modified through controlled methods (i.e., public member functions) defined in the class.

These public member functions provide a controlled interface for interacting with the private member variables. They encapsulate the necessary operations and logic related to depositing, withdrawing, and displaying the account balance.

By making these member functions public, external code can use these functions to perform valid operations on the BankAccount object, while still maintaining the encapsulation of the private data.

Accessibility within the class

protected: Members declared as protected are accessible within the class itself and derived classes.

**private:** Members declared as private are accessible only within the class itself.

**protected:**is an access specifier used within a class definition to designate class members that are accessible within the class itself and its derived classes but not accessible from outside the class hierarchy.

EX:

#include <iostream>

using namespace std;

class Animal {

protected:

string name;

public:

Animal(const string& animalName) : name(animalName) {}

void makeSound() {

cout << "Animal sound" << endl;

}

void showName() {

cout << "Animal name: " << name << endl;

}

};

class Dog : public Animal {

public:

Dog(const string& dogName) : Animal(dogName) {}

void makeSound() {

cout << "Woof! Woof!" << endl;

}

void accessBaseClassMembers() {

// We can access the protected member 'name' within the derived class

cout << "Dog's name from the derived class: " << name << endl;

}

};

int main() {

Dog dog("Buddy");

// We cannot access the protected member 'name' from outside the class hierarchy

// dog.name = "Max"; // This will result in a compilation error

dog.showName(); // Output: Animal name: Buddy

dog.makeSound(); // Output: Woof! Woof!

dog.accessBaseClassMembers(); // Output: Dog's name from the derived class: Buddy

return 0;

}

**9.struct and 10.union**

**struct**:struct is a user-defined data type that allows you to group related data members together

#include <iostream>

struct Person {

std::string name;

int age;

double height;

};

int main() {

Person person1;

person1.name = "John Doe";

person1.age = 30;

person1.height = 1.8;

std::cout << "Name: " << person1.name << std::endl;

std::cout << "Age: " << person1.age << std::endl;

std::cout << "Height: " << person1.height << std::endl;

return 0;

}

**Union:**

union is a special type of struct where all its members share the same memory location.

#include <iostream>

union Number {

int intValue;

double doubleValue;

};

int main() {

Number num;

num.intValue = 42;

std::cout << "Integer Value: " << num.intValue << std::endl;

std::cout << "Double Value: " << num.doubleValue << std::endl;

num.doubleValue = 3.14;

std::cout << "Integer Value: " << num.intValue << std::endl;

std::cout << "Double Value: " << num.doubleValue << std::endl;

return 0;

}

**11.signed and 12.unsigned:**

Signed:negative

signed int (int): -2,147,483,648 to 2,147,483,647(32-bit)

Unsigned:positive

unsigned int (unsigned): 0 to 4,294,967,295(32-bit)

#include <iostream>

int main() {

signed int positiveNumber = 10;

signed int negativeNumber = -20;

unsigned int nonNegativeNumber = 30;

std::cout << "Positive Number: " << positiveNumber << std::endl;

std::cout << "Negative Number: " << negativeNumber << std::endl;

std::cout << "Non-Negative Number: " << nonNegativeNumber << std::endl;

return 0;

}

**13.register:**

register keyword is a storage class specifier that suggests to the compiler to store the variable in a processor register for faster access. However, note that the register keyword is just a hint to the compiler and the compiler is free to ignore it

#include <iostream>

int main() {

register int count = 0;

for (int i = 0; i < 100000; ++i) {

count += i;

}

std::cout << "Count: " << count << std::endl;

return 0;

}

**14.asm**: To declare that a block of code is to be passed to the assembler.

**15.this**:A class pointer points to an object or instance of the class.

Using the this keyword helps distinguish between the local function parameters and the class data members when they have the same names. It ensures that the correct data members of the specific object are accessed and modified. Without the this keyword, the compiler would assume that the function parameters are being used, potentially leading to incorrect results.

EX:

#include <iostream>

#include <string>

class Person {

private:

std::string name;

int age;

public:

// Member function to set the name and age

void setNameAge(const std::string& name, int age) {

// 'this' is a pointer to the object on which this function is called

this->name = name;

this->age = age;

}

// Member function to display the name and age

void displayInfo() {

std::cout << "Name: " << name << ", Age: " << age << std::endl;

}

};

int main() {

Person person1;

person1.setNameAge("Alice", 25);

person1.displayInfo(); // Output: Name: Alice, Age: 25

Person person2;

person2.setNameAge("Bob", 30);

person2.displayInfo(); // Output: Name: Bob, Age: 30

return 0;

}

**16:virtual:** A function specifier that declares a member function of a class that will be redefined by a derived class.

→ it enables dynamic dispatch using a virtual table (vtable) mechanism, allowing the correct member function to be called based on the actual object type at runtime, enhancing runtime polymorphism.

→Runtime Polymorphism: Also known as dynamic polymorphism, this is achieved using virtual functions and inheritance. Virtual functions allow a base class to define a function that can be overridden by its derived classes. When a function is declared as virtual, the appropriate version of the function is determined at runtime based on the actual type of the object being referenced.

→virtual functions are invoked through pointers to the base class (Shape\*), and the virtual mechanism (dynamic dispatch) handles the function calls automatically based on the actual object type at runtime.

Ex:

#include <iostream>

class Shape {

public:

virtual void draw() {

std::cout << "Drawing a generic Shape" << std::endl;

}

};

class Circle : public Shape {

public:

void draw() override {

std::cout << "Drawing a Circle" << std::endl;

}

};

class Rectangle : public Shape {

public:

void draw() override {

std::cout << "Drawing a Rectangle" << std::endl;

}

};

int main() {

Shape\* shape1 = new Shape();

Shape\* shape2 = new Circle();

Shape\* shape3 = new Rectangle();

shape1->draw(); // Output: "Drawing a generic Shape"

shape2->draw(); // Output: "Drawing a Circle"

shape3->draw(); // Output: "Drawing a Rectangle"

delete shape1;

delete shape2;

delete shape3;

return 0;

}

#include <iostream>

class Shape {

public:

virtual void draw() {

std::cout << "Drawing a generic Shape" << std::endl;

}

};

Ex-2:

By using the scope resolution operator, you can access the parent's member functions or variables, even if they are overridden in the derived class. This allows you to add additional behavior or functionality in the derived class while still having the option to access the base class's implementation when needed

class Circle : public Shape {

public:

void draw() override {

std::cout << "Drawing a Circle" << std::endl;

// Call the parent's draw() function using the scope resolution operator

Shape::draw(); // Output: "Drawing a generic Shape"

}

};

int main() {

Circle circle;

Shape\* shapePtr = &circle;

shapePtr->draw(); // Output: "Drawing a Circle"

return 0;

}

**17:volatile:**

The volatile keyword cannot remove the memory assignment.

It cannot cache the variables in register.

The value cannot change in order of assignment

The primary significance of using volatile is to ensure that the variable's value is always read from memory and not from a cached copy, especially when the variable can be modified by hardware or other threads outside the normal program flow.

Ex: int num=10

while(num==10){

cout<<”done”;

}

ex:#include <iostream>

// Function to simulate an external interrupt modifying a variable

void simulateExternalInterrupt(volatile int& var) {

var = 100;

}

int main() {

// Regular non-volatile variable

int regularVar = 0;

// Volatile variable

volatile int volatileVar = 0;

// Modify the volatile variable

simulateExternalInterrupt(volatileVar);

// Output the values

std::cout << "Regular variable value: " << regularVar << std::endl;

std::cout << "Volatile variable value: " << volatileVar << std::endl;

return 0;

}

**18.void:**

def:Function which returns nothing means we can print the value from the function but we can’t return the output to others

**19.class**: To declare a user-defined type that encapsulates data members and operations or member functions.

**20.const:** To define objects whose value will not alter throughout the lifetime of program execution.

Ex: #include <iostream>

int main() {

const int constantValue = 10;

// constantValue = 20; // Error: Attempting to modify a const variable

std::cout << "Constant Value: " << constantValue << std::endl;

return 0;

}

**21:continue:**- Transfers control to the start of a loop.(which skip current iteration and continues with next)

ex:#include <iostream>

int main() {

for (int i = 1; i <= 5;i++) {

if (i == 3) {

continue;

}

std::cout << "Iteration " << i << std::endl;

}

return 0;

}

**22.do and 23.while:**

**do**: indicate the start of a do-while statement in which the sub-statement is executed repeatedly until the value of the expression is logical-false.

**While:Start of a while statement and end of a do-while statement. Iterate until the condition is false**

ex:#include <iostream>

int main() {

int i = 1;

// Do-while loop to print numbers from 1 to 5

do {

std::cout << i << " ";

i++;

} while (i <= 5);

std::cout << std::endl;

return 0;

}

**23: switch 24.default:**

allows you to choose one of many alternatives based on the value of an expression. It provides an efficient way to handle multiple cases using a single expression

ex:#include <iostream>

int main() {

int choice;

std::cout << "Enter a number between 1 and 3: ";

std::cin >> choice;

switch (choice) {

case 1:

std::cout << "You chose option 1." << std::endl;

break;

case 2:

std::cout << "You chose option 2." << std::endl;

break;

case 3:

std::cout << "You chose option 3." << std::endl;

break;

default:

std::cout << "Invalid choice. Please enter a number between 1 and 3." << std::endl;

break;

}

return 0;

}

**25.static:** The lifetime of an object-defined static exists throughout the lifetime of program execution.

Ex: #include <iostream>

void function() {

// static local variable

static int count = 0;

count++;

std::cout << "Count: " << count << std::endl;

}

int main() {

for (int i = 0; i < 5; ++i) {

function();

}

return 0;

}

**26:short:** A data type modifier that defines a 16-bit int number.(-32,768 to 32,767)

#include <iostream>

int main() {

short int num1 = 3965;

std::cout << "num1: " << num1 << std::endl;

}

**27: sizeof():** returns the size of type in bytes

ex:#include <iostream>

int main() {

std::cout << "Size of short int: " << sizeof(short int) << " bytes" << std::endl;

return 0;

}

**28.float:** ±1.17549e-38 to ±3.40282e+38.(4bytes)

**29.double**:±2.22507e-308 to ±1.79769e+308.(double)

Ex: #include <iostream>

int main() {

float floatValue = 3.1477890f;

double doubleValue = 2.718281;

std::cout << "Float value: " << floatValue << std::endl;

std::cout << "Double value: " << doubleValue << std::endl;

}

**30.auto**: A storage class specifier that is used to define objects in a block.

#include <iostream>

int main() {

auto x = 42; // The compiler deduces x as int

auto name = "John"; // The compiler deduces name as const char\*

std::cout << "x: " << x << std::endl;

std::cout << "name: " << name << std::endl;

return 0;

}

INTERVIEW QUESTIONS :

1. What is the significance of the const keyword in C++?
2. Explain the differences between const, constexpr, and consteval keywords in C++.
3. What is the difference between new and malloc() in C++? When should you use one over the other?
4. Describe the purpose of the virtual keyword in C++ and its role in polymorphism.
5. What is the use of the static keyword in C++? How is it different when applied to class members and local variables?
6. Explain the role of the auto keyword in C++11 and later versions.
7. Describe the purpose of the typename keyword in C++. When is it used, and why is it necessary?
8. What is the sizeof operator in C++? How does it differ from strlen() when used with strings?
9. What is the friend keyword in C++? How does it affect the access control of class members?
10. Explain the inline keyword in C++. When and why would you use it?
11. Describe the volatile keyword in C++. What does it do, and when is it necessary?
12. What is the use of the default and delete keywords in C++11 and later versions? Provide an example where these keywords are useful.
13. Explain the this pointer in C++. When and how is it used?
14. Describe the purpose of the override and final keywords in C++. How do they relate to virtual functions and inheritance?
15. What is the difference between public, private, and protected access specifiers in C++?
16. What are the keywords try, catch, throw, and exception used for in C++? Provide an example of handling exceptions.
17. Describe the role of the decltype keyword in C++. When and why would you use it?
18. Explain the static\_cast, dynamic\_cast, const\_cast, and reinterpret\_cast operators in C++. How do they differ from each other?