Constructor with program in C++

In C++, a constructor is a special member function of a class that is automatically called when an object of that class is created. It is used to initialize the object. A constructor has the same name as the class and does not have a return type.

- **Constructor** ki madad se jab object banaya jaata hai, uske saare members ko initial values mil jaati hain bina kisi extra step ke.

- Agar aap setter functions ka use karte hain, toh aapko manually har object ka initialization karna padta hai, jo bad mein galtiyan karne ka chance badha deta hai.

```cpp
#include<iostream>
using namespace std;
class Rectangle {
private:
    int width, height;
public:
    // Default constructor
    Rectangle() {
        width = 0;
        height = 0;}
    // Parameterized constructor
    Rectangle(int w, int h) {
        width = w;
        height = h;}

    void display() {
        cout << "Width: " << width << ", Height: " << height << endl
            ;
    }
};
int main() {
    Rectangle rect1;  // Calls default constructor
    Rectangle rect2(5, 10);  // Calls parameterized constructor
    rect1.display();
    rect2.display();
}
```

```
Width: 0, Height: 0
Width: 5, Height: 10


=== Code Execution Suc
```

Overriding - Overloaded – Overloading

→

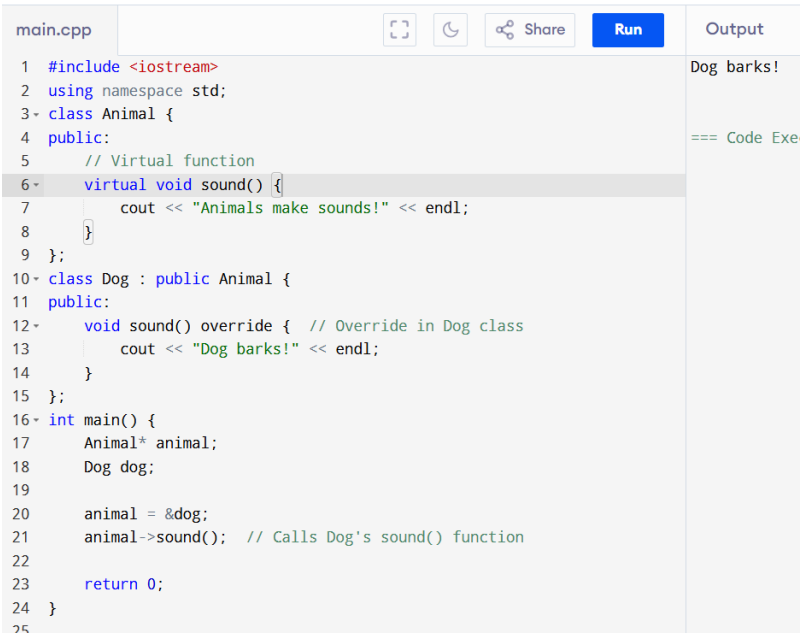
Overloading (Function Overloading) (overloaded function)

main.cpp Share Run Output

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

class Calculator {
public:
    // Function to add two integers
    int add(int a, int b) {
        return a + b;    }
    // Function to add three integers (overloaded function)
    int add(int a, int b, int c) {
        return a + b + c;    }
    // Function to add two double values (overloaded function)
    double add(double a, double b) {
        return a + b;  }
};
int main() {
    Calculator calc;

    // Calling overloaded functions
    cout << "Sum of 2 and 3: " << calc.add(2, 3) << endl;
    cout << "Sum of 1, 2, and 3: " << calc.add(1, 2, 3) << endl;
    cout << "Sum of 2.5 and 3.7: " << calc.add(2.5, 3.7) << endl;
    return 0;
}

```

```
Sum of 2 and 3: 5
Sum of 1, 2, and 3: 6
Sum of 2.5 and 3.7: 6.2


=== Code Execution Success
```

Feature	Function Overloading	Method Overriding
Definition	Ek hi function naam ke saath multiple functions create karna, jisme parameters alag ho.	Parent class ke method ko child class mein redefine karna.
Purpose	Ek function ko multiple ways mein use karne ke liye.	Parent class ke method ka specific behavior child class mein dena.
Function Signature	Overloaded functions mein signature (function name + parameter types) alag hote hain.	Overridden method ka signature parent class ke method ke signature ke saath bilkul same hona chahiye.
Usage	Ek hi function ko alag data types ya number of parameters ke saath call karne ke liye.	Inheritance ka use karte hue parent class ka function child class mein modify karne ke liye.
Compile-time / Run-time	**Compile-time polymorphism** (Function Resolution during compilation)	**Run-time polymorphism** (Function resolution during execution)



Virtual Function:

Virtual Function ek function hota hai jo **parent class** mein declare kiya jaata hai aur jab yeh function **child class** mein override hota hai, tab runtime pe decide hota hai ki kis class ka function call hoga (yani, child class ya parent class ka).

Virtual function ka use C++ mein run-time polymorphism achieve karne ke liye hota hai. Iska matlab yeh hai ki aap **base class pointer** ya **reference** ka use karte hue **derived class** ke function ko call kar sakte hain.

Structure and Struct

Structure aur **`struct`** dono ka concept ek hi hai, lekin C++ mein in dono ka use kaafi similar hota hai. **`struct`** C++ mein ek keyword hai jo structure define karne ke liye use hota hai, aur **structure** ek data type hota hai. Dono ka basic idea ek hi hai, lekin unka usage aur context thoda different ho sakta hai.

1. Default Access Specifiers:

- **C** mein structure ke members ka default access **`public`** hota hai.
- **C++** mein bhi structure ke members ka default access **`public`** hota hai, lekin **class** ka default access **`private`** hota hai.

Difference between `struct` in C and `struct` in C++

Feature	`struct` in C	`struct` in C++
Access Modifier	Default is `public`	Default is `public`
Object-Oriented Support	No support for OOP (Object-Oriented Programming)	Supports inheritance, polymorphism, etc. (like classes)
Functions Inside struct	No functions are typically used inside a struct	Can define member functions inside a struct
Data Encapsulation	Limited, no access control mechanisms	Supports data encapsulation and OOP concepts
Use	Grouping related data in a procedural way	Grouping data and using OOP features

main.cpp Share Run Output

```c
#include <stdio.h>

// C structure definition
struct Person {
    char name[50];   // public by default in C
    int age;         // public by default in C
};

int main() {
    struct Person person1;  // Creating structure variable
    person1.age = 25;  // Accessing members (public by default)
    printf("Age: %d\n", person1.age);
    return 0;
}

```

```
Age: 25


=== Code
```

With oops method →

main.cpp Share Run Output

```cpp
#include <iostream>
using namespace std;
struct Student {
    string name; int age; float marks;
    void display() {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
        cout << "Marks: " << marks << endl;  }
};
// Inheriting the Student structure
struct GraduateStudent : public Student {
    string degree;
    void display() {
        // Calling the base class display function
        Student::display();
        cout << "Degree: " << degree << endl;  } };
int main() {
    // Create an object of GraduateStudent
    GraduateStudent grad1;
    grad1.name = "John";
    grad1.age = 25;
    grad1.marks = 90.5;
    grad1.degree = "Masters in Computer Science";
    // Displaying student information
    grad1.display();
    return 0;   }
```

```
Name: John
Age: 25
Marks: 90.5
Degree: Masters in Computer Science


=== Code Execution Successful ===
```

DYNAMIC MEMORY ALLOCATION :

Dynamic memory allocation ka use tab kiya jata hai jab aapko runtime pe memory allocate karni hoti hai, jo size compile time pe decide nahi hota. C++ mein dynamic memory allocation ke liye **`new`** aur **`delete`** keywords ka use hota hai.

main.cpp Share Run Output

```cpp
#include <iostream>
using namespace std;
int main() {
    // Dynamically allocate memory for a single integer
    int* ptr = new int;  // Allocating memory for one integer
    *ptr = 100;

    // Accessing the value stored in the dynamically allocated memory
    cout << "The value of ptr is: " << *ptr << endl;

    // Deallocating the memory
    delete ptr;

    return 0;
}
```

```
The value of ptr is: 100


=== Code Execution Successf
```

`new int`: Yeh memory allocate karta hai jo ek integer store kar sake.

`delete` Operator:

- **`delete`** operator dynamically allocated memory ko free karne ke liye use hota hai.
- Jab aap `new` ka use karte hain, tab memory ko **manually** deallocate karna zaroori hota hai.
- **`delete`** ka use single variable ke liye hota hai aur **`delete[]`** ka use array ke liye hota hai.

Use of Virtual Keyword & Program

C++ mein **`virtual`** keyword ka use inheritance aur polymorphism ke concept ko implement karne ke liye hota hai. Jab hum **virtual function** define karte hain, toh C++ ko yeh bata sakte hain ki run-time polymorphism ko support karna hai. **Virtual functions** ka use hum **base class** mein karte hain, jisse **derived class** mein function ko **override** kiya ja sakta hai.

main.cpp Share Run Output

```cpp
#include <iostream>
using namespace std;
class Animal {
public:
    virtual void sound() {
        cout << "This is an animal sound" << endl;  }
    virtual ~Animal() {}    };
class Dog : public Animal {
public:
    void sound() override {
        cout << "Woof woof!" << endl; }  };
class Cat : public Animal {
public:
    void sound() override {
        cout << "Meow meow!" << endl;  }  };
int main() {
    Animal* animalPtr;
    Dog dog;
    Cat cat;
    animalPtr = &dog;
    animalPtr->sound();
    animalPtr = &cat;
    animalPtr->sound();
    return 0;
}
```

```
Woof woof!
Meow meow!


=== Code Ex
```

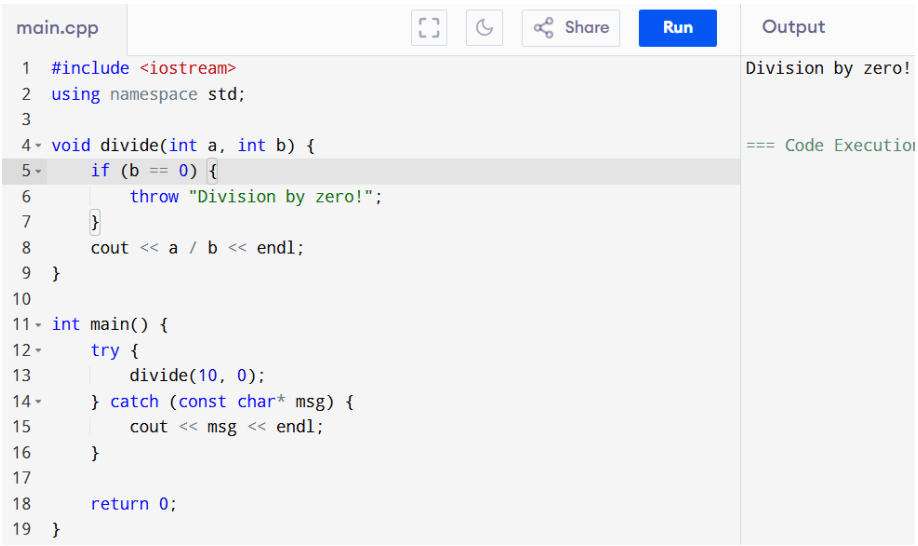
Exception handling →

C++ mein exception handling ek mechanism hai jo program ko runtime errors (exceptions) se bachane ke liye use hota hai. Jab koi error occur hota hai, toh program crash nahi hota, balki uss error ko handle karke program ko aage continue karne ka moka milta hai.

C++ mein exception handling ko `try`, `throw`, aur `catch` ke through implement kiya jata hai.

C++ mein Exception Handling ke 3 main parts hote hain:

1. **try block**: Yahan pe wo code likhte hain jahan error aa sakta hai.
2. **throw statement**: Agar error aata hai, toh yeh error ko throw (raise) karta hai.
3. **catch block**: Yahan pe hum error ko handle karte hain.



- I removed the unnecessary variables `x` and `y` and directly called `divide` with the values `10` and `0`.

- The `divide` function directly handles the exception check and prints the result if no exception occurs.

- `try` **block**: Is block mein woh code hota hai jo exception throw kar sakta hai. Aapke case mein, yeh `divide` function ko call karta hai `10` aur `0` ke arguments ke sath. Kyunki `0` se divide karna allowed nahi hai, yeh exception throw karega.
- `throw` **statement**: `divide` function ke andar, agar `b` `0` hai, toh `throw` statement execute hota hai, jo ek exception (is case mein, string "Division by zero!") ko `catch` block ke paas bhej deta hai.

- **`catch` block**: Yeh block exception ko catch karta hai jo `try` block mein throw hoti hai. Yeh ek parameter (is case mein, `const char* msg`) leta hai, jo exception hai. Yeh block phir us exception ko handle karta hai, jaise error message print karna.

Agar kuch galat hota hai, `catch` block ensure karta hai ki program crash na ho aur gracefully error ko handle kar sake

Heap Memory kya hai?

Heap memory runtime par dynamically allocate hoti hai. Matlab, jab aapko program ke dauraan memory chahiye hoti hai tab aap usse le sakte ho aur jab kaam ho jaye toh free kar sakte ho.

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

int main() {
    int* ptr = new int;  // Ek integer ke liye memory allocate kar rahe hain heap par
    *ptr = 100;  // Us memory mein value 100 rakh rahe hain

    cout << "Value: " << *ptr << endl;

    delete ptr;  // Memory ko free kar rahe hain

    return 0;
}
```

```
Value: 100


=== Code Exec
```

Difference Heap & Main Memory

Speed	Fast hoti hai kyunki automatically handle hoti hai.	Relatively slow kyunki manually handle karni padti hai.
Memory Size	Limited size hoti hai.	Zyada size hoti hai.
	Zyada memory allocate karne par stack overflow error ho sakti hai.	Large amount of memory allocate kar sakte hain.
Usage	Local variables, function parameters, aur return addresses ke liye use hoti hai.	Dynamic memory allocation ke liye use hoti hai, jaise linked lists, trees, aur large objects.

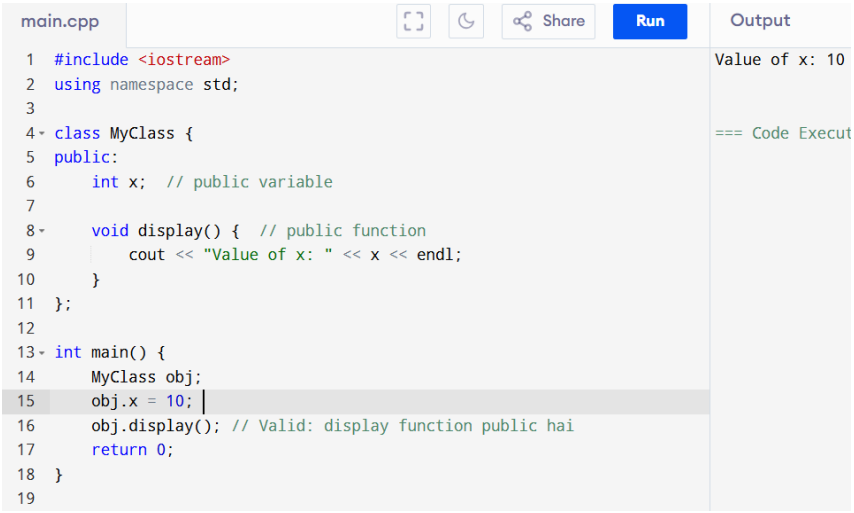
Aspect	Stack Memory	Heap Memory
Lifetime (Duration)	Function scope tak rahti hai.	Manually control kar sakte hain.
	Function end hone par automatically free ho jati hai.	Jab tak explicitly free nahi karte, tab tak rehti hai.
Allocation & Deallocation	Automatically handle hoti hai.	Manually handle karni padti hai using `new` aur `delete`.

C++ mein **access modifiers** ka use class members (variables aur functions) ke access ko control karne ke liye kiya jata hai. C++ mein **3 main access modifiers** hote hain:

- **public**
- **private**
- **protected**

1. public Access Modifier

- **Public members** ko class ke **bahar** se **access** kiya ja sakta hai.
- Yeh generally wo functions ya variables hote hain jo users ko interact karne ke liye available hote hain.



2. private Access Modifier

- **Private members** sirf class ke **andar** hi access kiye ja sakte hain.
- Class ke **bahar** se private members ko directly access nahi kiya ja sakta.
- Yeh usually sensitive data ko hide karne aur encapsulation achieve karne ke liye use hota hai.

main.cpp

Share Run Output

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

class MyClass {
private:
    int x;
public:
    void setX(int val) {  // public function to set x
        x = val;
    }
    int getX() {  // public function to get x
        return x;  }  };
int main() {
    MyClass obj;
    obj.setX(10);
    cout << "Value of x: " << obj.getX() << endl;
    return 0;
}

```

```
Value of x: 10


=== Code Executio
```

3. protected Access Modifier

- **Protected members** ko class ke **andar** aur class ke **derived (child) classes** ke **andar** access kiya ja sakta hai.
- **Class ke bahar** se protected members ko directly access nahi kiya ja sakta.
- Yeh **inheritance** ke time pe kaam aata hai, jab aap derived class ke andar parent class ke protected members ko access karna chahte hain.

main.cpp

Share Run Output

```cpp
#include <iostream>
using namespace std;
class Base {
protected:
    int x;  // protected variable
public:
    void setX(int val) {  // public function to set x
        x = val;
    }
};
class Derived : public Base {
public:
    void show() {
        cout << "Vaue of x in Derived class: " << x << endl;
        }
};
int main() {
    Derived obj;
    obj.setX(10);  // Valid: setX public hai
    obj.show();    // Valid: show function is accessing protected
        member x
    // obj.x = 20;  // Invalid: x protected hai, isliye directly
        access nahi kar sakte
    return 0;
}

```

```
Vaue of x in Derived class: 10


=== Code Execution Successful ===
```

Difference →

Feature	`private`	`protected`
Access in same class	**Accessible** – Class ke andar directly access kiya ja sakta hai.	**Accessible** – Class ke andar directly access kiya ja sakta hai.
Access in derived class (Inheritance)	**Not accessible** – Derived class ke andar directly access nahi kiya ja sakta.	**Accessible** – Derived class ke andar directly access kiya ja sakta hai.
Access outside class	**Not accessible** – Class ke bahar se directly access nahi kiya ja sakta.	**Not accessible** – Class ke bahar se directly access nahi kiya ja sakta.
Purpose	Sensitive data ko **protect** karna.	Derived class ko base class ke members access karne ki **permission** dena.
Example	`private` members ko sirf class ke andar use kiya ja sakta hai.	`protected` members ko derived class ke andar access kiya ja sakta hai.

STATIC KEYWORD and Without Static →

Without :

main.cpp — Share — Run — Output

```cpp
#include <iostream>
using namespace std;
class MyClass {
public:
    int count;
    MyClass() {
        count = 0;    }
    void increment() {
        count++;  }
    void display() {
        cout << "Count: " << count << endl;  }  };
int main() {
    MyClass obj1;
    MyClass obj2;
    obj1.increment();
    obj1.display(); // Outputs: Count: 1

    obj2.increment();
    obj2.display(); // Outputs: Count: 1

    return 0;
}

```

```
Count: 1
Count: 1


=== Code Exe
```

- `count` non-static hai, isliye har object apni alag copy rakhta hai.

- `obj1` aur `obj2` ke `count` variables alag-alag hain, isliye dono objects ke `increment` aur `display` calls alag-alag results dete hain.

With Static

main.cpp Share Run Output

```cpp
#include <iostream>
using namespace std;
class MyClass {
public:
    static int count;
    MyClass() {
        count = 0;
    }
    void increment() {
        count++;
    }
    void display() {
        cout << "Count: " << count << endl;  }  };
// Static variable ko class ke bahar initialize karte hain
int MyClass::count = 0;
int main() {
    MyClass obj1;
    MyClass obj2;
    obj1.increment();
    obj1.display(); // Outputs: Count: 1
    obj2.increment();
    obj2.display(); // Outputs: Count: 2

    return 0;
}

```

```
Count: 1
Count: 2


=== Code Exe
```

- `count` static hai, isliye yeh class ke sabhi objects ke beech share hota hai.

- `obj1` aur `obj2` ka `count` variable common hai, isliye `increment` aur `display` calls dono objects ke liye same result dete hain.

Destructor kya hota hai?

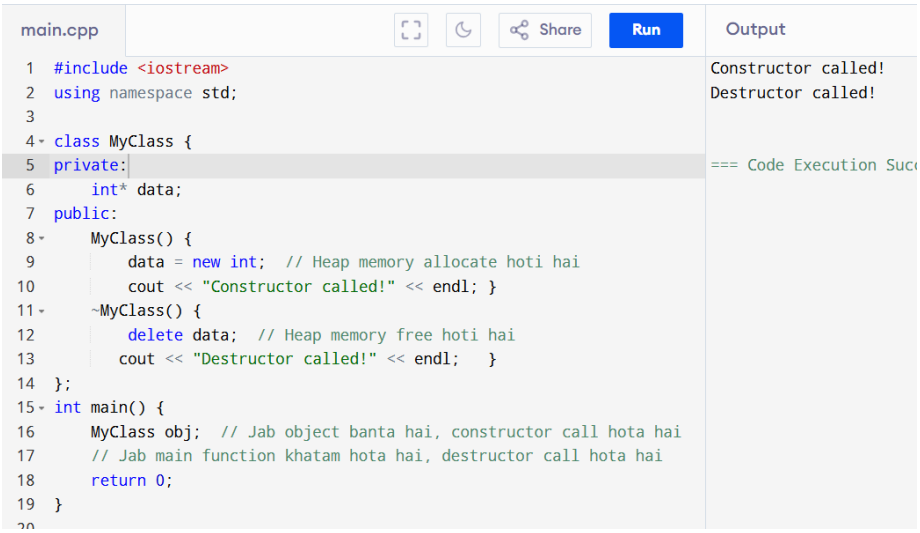
Destructor ek special function hota hai jo tab call hota hai jab object destroy hota hai, matlab jab object ka lifecycle khatam hota hai ya object delete hota hai. Destructor ka kaam resources ko release karna hota hai jo object ne use kiye hain, jaise dynamically allocated memory.

Destructor ka naam class ke naam ke aage tilde (~) laga ke banaya jaata hai aur iske koi parameters nahi hote.

```cpp
class MyClass {
public:
    ~MyClass() {
        // Destructor body
    }
};
```

Dynamic Memory Example:

Jab object dynamically memory allocate karta hai **(heap memory),** to destructor memory ko free karne ke liye use hota hai.



Template Argument kya hota hai?

Templates C++ mein ek tareeka hain flexible aur reusable code likhne ka. Template arguments basically woh specific types ya values hain jo aap templates ko dete ho taaki woh unke sath kaam kar sakein.

Types of Template Arguments:

1. **Type Argument:** Yeh data type specify karta hai, jaise `int`, `double`, etc.
2. **Non-Type Argument:** Yeh ek constant value specify karta hai, jaise `int`, `char`, etc.

main.cpp Share Run Output

```cpp
#include <iostream>
using namespace std;
template <typename T>
class MyClass {
public:
    T data;
    MyClass(T data) : data(data) {}

    void display() {
        cout << "Data: " << data << endl;   }
};
int main() {
    MyClass<int> obj1(100);       // T yahan 'int' hai
    MyClass<double> obj2(3.14);   // T yahan 'double' hai
    MyClass<string> obj3("Hello"); // T yahan 'string' hai

    obj1.display();  // Outputs: Data: 100
    obj2.display();  // Outputs: Data: 3.14
    obj3.display();  // Outputs: Data: Hello

    return 0;
}
```

```
Data: 100
Data: 3.14
Data: Hello


=== Code Execut
```

NON- type

main.cpp Share Run Output

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

template <int N>
class MyArray {
public:
    int arr[N];
    void fill() {
        for (int i = 0; i < N; ++i) {
            arr[i] = i * 10;   }
    }
    void display() {
        for (int i = 0; i < N; ++i) {
            cout << arr[i] << " ";
        }
        cout << endl;   }
};
int main() {
    MyArray<5> obj;  // N yahan 5 hai
    obj.fill();
    obj.display();  // Outputs: 0 10 20 30 40

    return 0;
}
```

```
0 10 20 30 40


=== Code Execut
```

. (DOT) operator →

The . (dot) operator in C++ is used to access members (variables and functions) of an object. It's a way to call methods or access properties of a particular instance of a class. Here's a simple explanation and example:

main.cpp Share Run Output

```cpp
#include <iostream>
using namespace std;
class MyClass {
public:
    int value;
    void display() {
        cout << "Value: " << value << endl;
    }
};

int main() {
    MyClass obj;     // Create an object of MyClass
    obj.value = 10;    // Use . operator to set the value of the
        member variable
    obj.display();     // Use . operator to call the member function

    return 0;
}
```

```
Value: 10


=== Code Exec
```

Casting in C++:

Casting ka matlab hai ek type ke variable ko doosre type ke variable mein badalna (convert karna)

C-Style Cast:

- Sabse simple aur purani tareeka.
- Syntax: `(type) variable`

main.cpp Share Run Output

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

int main() {
    float a = 10.3;             // Integer variable
    int b = (int) a;     // C-Style cast to convert int to double

    cout << "Integer value: " << a << endl;    // Outputs: Integer
        value: 10
    cout << "Double value: " << b << endl;     // Outputs: Double
        value: 10.0

    return 0;
}
```

```
Integer value: 10.3
Double value: 10


=== Code Execution Succes
```

C++ mein references ka use variables ko ek alias (nickname) dene ke liye hota hai. References pointers jaise hi hote hain, lekin unhe initialize karna aur use karna zyada simple hota hai. Aayein, simple shabdon mein samjhte hain references ko aur ek example dekhte hain:

- **Declaration:** Reference ko declare karte samay usse initialize karna zaroori hota hai.
- **Syntax:** `type &ref = original_variable;`

- **Alias:** Reference kisi original variable ka alias banta hai. Matlab, koi bhi operation jo reference par kiya jata hai, woh original variable par apply hota hai.

main.cpp

Share Run

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

int main() {
    int a = 10;    // Original variable
    int &ref = a;  // Reference `ref` ko `a` ka alias banate hain

    cout << "Original value (a): " << a << endl;   // Outputs: 10
    cout << "Reference value (ref): " << ref << endl; // Outputs: 10

    ref = 20;  // Reference ko update karte hain

    cout << "Updated original value (a): " << a << endl; // Outputs:
        20
    cout << "Updated reference value (ref): " << ref << endl; //
        Outputs: 20

    return 0;
}
```

Output

```
Original value (a): 10
Reference value (ref): 10
Updated original value (a): 20
Updated reference value (ref): 20


=== Code Execution Successful ===
```

Vectors kya hain?

- Vectors C++ mein ek type ka dynamic array hota hai.
- Vectors ka size badhaya ya ghataaya ja sakta hai jab zaroorat ho.

main.cpp

Share Run

```cpp
#include <iostream>
#include <vector>  // Vector library ko include karte hain
using namespace std;

int main() {
    vector<int> vec;  // Ek integer type ka vector declare karte hain

    // Vector mein elements add karte hain
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);

    // Elements ko access karte hain using index
    cout << "Element at index 0: " << vec[0] << endl; // Outputs: 10
    cout << "Element at index 1: " << vec[1] << endl; // Outputs: 20
    cout << "Element at index 2: " << vec[2] << endl; // Outputs: 30

    return 0;
}
```

Output

```
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30


=== Code Execution Success
```

Find size →

main.cpp Share Run Output

```cpp
#include <iostream>
#include <vector>  // Vector library ko include karte hain
using namespace std;

int main() {
    vector<int> vec;  // Ek integer type ka vector declare karte hain

    // Vector mein elements add karte hain
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);

    // Elements ko access karte hain using index
    cout << "Element at index 0: " << vec[0] << endl; // Outputs: 10
    cout << "Element at index 1: " << vec[1] << endl; // Outputs: 20
    cout << "Element at index 2: " << vec[2] << endl; // Outputs: 30

    cout << "Size of vector: " << vec.size() << endl; // Outputs: 3

    return 0;
}
```

```
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Size of vector: 3


=== Code Execution Succ
```

Type Def :

Agar aap directly data types use karte hain, toh functionality par koi farak nahi padta. Lekin typedef ka fayda ye hai ki aapke code ko zyada readable aur maintainable banata hai, khas kar jab complex data types ya long declarations hote hain.

Without typedef :

main.cpp Share Run Output

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

struct Student {
    string name;
    int age;
};

int main() {
    struct Student s1;
    s1.name = "John";
    s1.age = 20;

    cout << "Name: " << s1.name << ", Age: " << s1.age << endl;

    return 0;
}
```

```
Name: John, Age: 20


=== Code Execution Suc
```

With :

Advantages:

- **Readability:** Code zyada readable aur understandable ho jata hai.
- **Maintenance:** Code ko maintain karna easy ho jata hai.
- **Clarity:** Complex types ko simplify karke clarity badhata hai

::

Recursive function ek aisi function hoti hai jo apne aap ko call karti hai. Yeh useful hoti hai jab aapko kisi problem ko smaller subproblems mein tod ke solve karna hota hai. Recursive functions ko use karte waqt base case aur recursive case ka dhyaan rakhna padta hai.

Key Components of Recursive Function:

1. **Base Case:** Yeh condition hoti hai jo recursion ko end karti hai. Without base case, recursive function infinite loop mein fas sakti hai.
2. **Recursive Case:** Yeh part function ko call karta hai apne aap ko with different arguments.

main.cpp

Share | Run | Output

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

// Recursive function to calculate factorial
int factorial(int n) {
    if (n == 0) {
        return 1;  // Base case: factorial of 0 is 1
    } else {
        return n * factorial(n - 1);  // Recursive case
    }
}
int main() {
    int number = 5;
    cout << "Factorial of " << number << " is " << factorial(number)
        << endl;  // Outputs: 120
    return 0;
}
```

```
Factorial of 5 is 120

=== Code Execution Suc
```

- **Base Case:** `if (n == 0) return 1;` yeh condition recursion ko end karti hai.
- **Recursive Case:** `return n * factorial(n - 1);` yeh part function ko call karta hai with `n-1`.

Example with Steps:

For `factorial(5)`:

1. `factorial(5)` calls `factorial(4)`
2. `factorial(4)` calls `factorial(3)`
3. `factorial(3)` calls `factorial(2)`
4. `factorial(2)` calls `factorial(1)`
5. `factorial(1)` calls `factorial(0)`
6. `factorial(0)` returns `1`
7. Then the functions return values step by step:
   - `factorial(1)` returns `1 * 1 = 1`
   - `factorial(2)` returns `2 * 1 = 2`
   - `factorial(3)` returns `3 * 2 = 6`
   - `factorial(4)` returns `4 * 6 = 24`
   - `factorial(5)` returns `5 * 24 = 120`

ARGUMENT AND WITHOUT ARGUMENT IN C++

main.cpp Share Run Output

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

// Function with arguments
void greet(string name) {
    cout << "Hello, " << name << "!" << endl;
}

int main() {
    greet("Alice");  // Outputs: Hello, Alice!
    greet("Bob");    // Outputs: Hello, Bob!
    return 0;
}
```

```
Hello, Alice!
Hello, Bob!


=== Code Execut
```

- `greet(string name)` ek function hai jo ek argument `name` leta hai.

- `greet("Alice")` aur `greet("Bob")` function calls hain jo `name` ko "Alice" aur "Bob" set karte hain.

Without Arguments →

main.cpp Share Run Output

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

// Function without arguments
void greet() {
    cout << "Hello, World!" << endl;
}

int main() {
    greet();  // Outputs: Hello, World!
    return 0;
}
```

```
Hello, World!


=== Code Executi
```