* + **static**: Class-level or function-level variables/methods.
  + **virtual**: Enables runtime polymorphism.
  + **abstract**: Marks classes with pure virtual functions (in C++ use = 0).
  + **final**: Prevents inheritance or function overriding.
  + **explicit**: Prevents implicit type conversions.
  + **this**: Pointer to the current object.
  + **new**: Allocates memory dynamically.
  + **const**: Defines immutability.
  + **super**: Conceptually used to refer to the base class (in C++ use the base class name directly).
* --------------------------------  
  @static
  + **Purpose**: Defines class-level or function-level variables or methods that are shared among all instances of a class or persist across function calls.
  + **Usage**:
    - **Class Members**: Shared by all instances of the class. It belongs to the class, not any specific instance.
    - **Function Variables**: Preserves the variable’s state between function calls.
* class MyClass {  
  public:  
   static int count; // Static data member  
   void incrementCount() {  
   count++;  
   }  
  };  
  int MyClass::count = 0; // Definition and initialization of static member  
  int main() {  
   MyClass obj1, obj2;  
   obj1.incrementCount();  
   obj2.incrementCount();  
   std::cout << "Count: " << MyClass::count << std::endl; // Output: Count: 2  
   return 0;  
  }  
     
  ------------------------------------------------------------------------------------------------  
  @virtual  
  Purpose: Allows a member function to be overridden in derived classes. It supports dynamic (runtime) polymorphism.  
  Usage: Used in a base class to indicate that a function can be overridden in derived classes.  
  Example:  
  cpp  
  Copy code  
  class Base {  
  public:  
   virtual void show() { // Virtual function  
   std::cout << "Base class show()" << std::endl;  
   }  
  };  
  class Derived : public Base {  
  public:  
   void show() override { // Overriding the base class function  
   std::cout << "Derived class show()" << std::endl;  
   }  
  };  
  int main() {  
   Base\* obj = new Derived();  
   obj->show(); // Output: Derived class show()  
   delete obj;  
   return 0;  
  }  
  ------------------------------------------------------------------------------------------------  
  @abstract  
  Purpose: Marks a class as abstract, meaning it cannot be instantiated and may contain at least one pure virtual function.  
  Usage: In C++, the abstract keyword does not exist; instead, you use pure virtual functions (declared with = 0).  
  Example:  
  cpp  
  Copy code  
  class AbstractBase {  
  public:  
   virtual void pureVirtualFunction() = 0; // Pure virtual function  
  };  
  class ConcreteDerived : public AbstractBase {  
  public:  
   void pureVirtualFunction() override { // Implementing pure virtual function  
   std::cout << "ConcreteDerived implementation" << std::endl;  
   }  
  };  
  int main() {  
   ConcreteDerived obj;  
   obj.pureVirtualFunction(); // Output: ConcreteDerived implementation  
   return 0;  
  }  
  -----------------------------------------------  
     
  @final  
  Purpose: Prevents further inheritance of a class or overriding of a virtual function.  
  Usage:  
  For Classes: Used to prevent the class from being inherited.  
  For Functions: Prevents derived classes from overriding the function.  
  Example:  
  cpp  
  Copy code  
  class Base {  
  public:  
   virtual void show() final { // Function cannot be overridden  
   std::cout << "Base show()" << std::endl;  
   }  
  };  
  class Derived : public Base {  
  // This will cause a compile-time error  
  // void show() override { std::cout << "Derived show()" << std::endl; }  
  };  
  int main() {  
   Base obj;  
   obj.show(); // Output: Base show()  
   return 0;  
  }  
  ------------------------------------------------------------------------------------------------  
     
  @explict  
  Purpose: Prevents implicit conversions and copy-initialization when usingconstructors and conversion operators.  
  Usage: Used with constructors to avoid automatic type conversion.  
  Example:  
  cpp  
  Copy code  
  class MyClass {  
  public:  
   explicit MyClass(int x) { // Constructor with explicit keyword  
   std::cout << "MyClass constructor with value: " << x << std::endl;  
   }  
  };  
  int main() {  
   MyClass obj1 = 10; // Error: No implicit conversion allowed  
   MyClass obj2(10); // OK: Direct initialization  
   return 0;  
  }  
  ------------------------------------------------------------------------------------------------  
  @this  
  Purpose: A pointer to the current object instance.  
  Usage: Used within member functions to refer to the instance of the object on which the function is called.  
  @this  
  This->prop is same as \*(this).prop  
     
  Example:  
  cpp  
  Copy code  
  class MyClass {  
  public:  
   int value;  
   void setValue(int value) {  
   this->value = value; // Refers to the member variable  
   }  
  };  
  int main() {  
   MyClass obj;  
   obj.setValue(5);  
   std::cout << "Value: " << obj.value << std::endl; // Output: Value: 5  
   return 0;  
  }  
  **this Pointer**   
  this pointer holds the address of the current object. In simple words, you can say that this pointer points to the current object of the class.   
     
  There can be three main usages of this keyword in C++.
  + It can be used to refer to a current class instance variable.
  + It can be used to pass the current object as a parameter to another method.
  + It can be used to declare indexers.

Let’s take an example to understand this concept.  
   
#include <bits/stdc++.h>  
using namespace std;  
class mobile  
{  
 string model;  
 int year\_of\_manufacture;  
   
public:  
 void set\_details(string model, int year\_of\_manufacture)  
 {  
 this->model = model;  
 this->year\_of\_manufacture = year\_of\_manufacture;  
 }  
   
 void print()  
 {  
 cout << this->model << endl;  
 cout << this->year\_of\_manufacture << endl;  
 }  
};  
int main()  
{  
 mobile redmi;  
 redmi.set\_details("Note 7 Pro", 2019);  
 redmi.print();  
}  
   
Output:  
Note 7 Pro2019  
   
Here you can see that we have two data members model and year\_of\_manufacture. In member function set\_details(), we have two local variables with the same name as the data members’ names. Suppose you want to assign the local variable value to the data members. In that case, you won’t be able to do until unless you use this pointer because the compiler won’t know that you are referring to the object’s data members unless you use this pointer. This is one of example where you must use this pointer.  
   
------------------------------------------------------------------------------------------------  
@new  
Purpose: Allocates memory dynamically for objects or arrays.  
Usage: Creates instances of classes or arrays at runtime.  
Example:  
cpp  
Copy code  
class MyClass {  
public:  
 int data;  
};  
int main() {  
 MyClass\* obj = new MyClass(); // Dynamically allocated object  
 obj->data = 10;  
 std::cout << "Data: " << obj->data << std::endl;  
 delete obj; // Frees allocated memory  
 return 0;  
}  
------------------------------------------------------------------------------------------------  
   
@const  
Purpose: Defines constants, immutable data, and member functions that donot modify the object.  
Usage: Used with variables, pointers, and member functions.  
Example:  
cpp  
Copy code  
class MyClass {  
public:  
 const int constantValue = 10; // Constant data member  
 void display() const { // Const member function  
 std::cout << "Constant value: " << constantValue << std::endl;  
 }  
};  
int main() {  
 MyClass obj;  
 obj.display(); // Output: Constant value: 10  
 return 0;  
}  
------------------------------------------------------------------------------------------------  
   
@super  
Purpose: In C++, super does not exist. Instead, you use the base classname to access base class members.  
Usage: Refers to the base class from within a derived class (conceptually similar to super in other languages like Java).  
Example:  
cpp  
Copy code  
class Base {  
public:  
 void show() {  
 std::cout << "Base class show()" << std::endl;  
 }  
};  
class Derived : public Base {  
public:  
 void show() {  
 Base::show(); // Calling base class method  
 std::cout << "Derived class show()" << std::endl;  
 }  
};  
int main() {  
 Derived obj;  
 obj.show(); // Output: Base class show()  
 // Derived class show()  
 return 0;  
}