

```

class base
{
int val;
Public:
    base(int a): val(a){}
    base(base& obj)
    {
        val=obj.val;
        cout<<"From copy constructor"<<endl;
    }
    base operator = (base& obj)
    {
        val=obj.val;
        cout<<"From copy assignment operator"<<endl;
        return *this;
    }
};

int main()
{
    base b1(10);
    base b2(20);
    b1=b2;
    return 0;
}

```

If we remove the & from the return type in the copy assignment operator, it calls the copy constructor again, resulting in the output:

"From copy assignment operator"
 "From copy constructor"

ChatGPT said:

The issue is that your copy assignment operator is returning **by value** instead of by reference. This leads to an extra call to the **copy constructor**.

Understanding the Problem

Your copy assignment operator is currently:

base operator=(base obj) // ✗ Returning by value (incorrect)

Since it returns **by value**, a **temporary object** is created as the return value.
 This temporary object is **copied** from *this, which calls the **copy constructor**.

Step-by-Step Execution

b1 = b2;

1. b1.operator=(b2) is called.
2. Inside operator=, obj (a copy of b2) is created due to **pass-by-value**, triggering the **copy constructor**.
3. val = obj.val; assigns the value.
4. Since operator= **returns base by value**, a **new temporary object** is created and returned.
5. The temporary object is then copied using the **copy constructor** before being discarded.

Result

From copy assignment operator
 From copy constructor

You said: If I remove the & in front of base in the move assignment operator's return type, in move assignment operator why after calling the move assignment operator then why is not calling the move constructor(compared with copy constructor)??

```
class base {
int val;
public:
base(int a): val(a){}
base(base&& obj)
{
val=obj.val;
cout<<"From move constructor"<<endl;
}
base operator = (base&& obj)
{
val=obj.val;
cout<<"From move assignment operator"<<endl;
return *this;
}
};
```

ChatGPT said:

If you remove the & in front of base in the move assignment operator's return type, it will return **by value**, just like in the copy assignment case. However, unlike the previous scenario with the copy assignment operator, the move constructor is **not necessarily called**. Let's analyze why.

Your Code with Move Semantics

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

class base {
    int val;
public:
    base(int a) : val(a) {}

    base(base&& obj) { // Move Constructor
        val = obj.val;
        cout << "From move constructor" << endl;
    }

    base operator=(base&& obj) { // Move Assignment Operator (Returning by Value)
        val = obj.val;
        cout << "From move assignment operator" << endl;
        return *this; // Returning by value
    }
};

int main() {
    base b1(10);
    base b2(20);
    b1 = move(b2); // Calls Move Assignment Operator
    return 0;
}
```

```
}
```

Expected Output

From move assignment operator

You would expect **both the move assignment operator and move constructor** to be called, but the move constructor is **not called after move assignment**.

Why Doesn't the Move Constructor Get Called?

When returning `*this` **by value** in the move assignment operator:

```
base operator=(base&& obj) { // ❌ Returning by value
```

```
    val = obj.val;
```

```
    cout << "From move assignment operator" << endl;
```

```
    return *this; // This creates a copy instead of moving
```

```
}
```

Step-by-step Execution

1. `b1 = move(b2);` calls the **move assignment operator**.
2. Inside the move assignment operator:
 - o `obj` is an **rvalue reference** to `b2`.
 - o `val = obj.val;` transfers ownership.
 - o `return *this;` **returns by value**, meaning a copy of `*this` is created.
3. **Since `*this` is an lvalue, the copy constructor is used instead of the move constructor.**
4. However, in this case, there's no explicit copy constructor in your class, and since `*this` is returned by value, **compiler optimizations like Return Value Optimization (RVO) can eliminate the call to the copy or move constructor.**

You said:

In my compiler, removing the `&` from the return type of the move assignment operator results in an error. However, if I explicitly define the copy constructor, it compiles successfully and produces the output: 'Move assignment operator, Copy constructor.'

ChatGPT said:

That behavior makes sense! Let's analyze it carefully and understand why removing `&` from the move assignment operator causes an **error**, and why adding the copy constructor makes it work.

Why Does Removing `&` from the Move Assignment Operator Cause an Error?

When you declare:

```
base operator=(base&& obj)
```

- It **returns by value**, meaning it creates a temporary object.
- That temporary object needs to be **copied or moved** to the return value.

Now, your class **only has a move constructor but does not have a copy constructor**.

If the compiler cannot use the move constructor, it needs the copy constructor.

Since you haven't explicitly defined a copy constructor, the compiler **deletes the default copy constructor** because a move constructor is present.

Effect of Removing `&` in Move Assignment

1. `b1 = move(b2);` calls the **move assignment operator**.
2. `return *this;` tries to return by **value**, meaning it creates a temporary object.
3. The compiler must **copy or move** `*this` to create the temporary object.
4. **Move constructor should be called**, but due to compiler behavior (optimizations or specific rules), it attempts to use the copy constructor.
5. **Since no copy constructor exists, compilation fails.**

Fix: Add Copy Constructor

When you explicitly add a copy constructor:

```
cpp
CopyEdit
base(const base& obj) {
    val = obj.val;
    cout << "From copy constructor" << endl;
}
```

- Now the compiler has a **valid copy constructor** to use when returning `*this` by value.
- So, instead of moving, it **copies** `*this`, resulting in **both the move assignment operator and copy constructor being called**.

Example Code and Output

Your Modified Code (with Copy Constructor)

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

class base {
    int val;
public:
    base(int a) : val(a) {}

    base(const base& obj) { // Explicit Copy Constructor
        val = obj.val;
        cout << "From copy constructor" << endl;
    }
    base(base&& obj) { // Move Constructor
        val = obj.val;
        cout << "From move constructor" << endl;
    }
    base operator=(base&& obj) { // Move Assignment Operator, but returning by value
        val = obj.val;
        cout << "From move assignment operator" << endl;
        return *this; // ❌ Returning by value causes an extra copy constructor call
    }
};

int main() {
    base b1(10);
    base b2(20);
    b1 = move(b2); // Calls Move Assignment Operator
    return 0;
}
```

Output

```
From move assignment operator
From copy constructor
```

- The **move assignment operator** is called first.
- The **copy constructor** is called because `return *this;` returns **by value**, requiring a copy.

You said:

In this code, where the move constructor, copy constructor, and move assignment operator are present, why does removing & from the return type of the move assignment operator cause the copy constructor to be called instead of the move constructor?

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

class base {
    int val;

public:
    base(int a) : val(a) {}

    base(base&& obj) { // Move Constructor
        val = obj.val;
        cout << "From move constructor" << endl;
    }

    base(const base& obj) { // Copy Constructor
        val = obj.val;
        cout << "From copy constructor" << endl;
    }

    base operator=(base&& obj) { // Move Assignment Operator (returns by value)
        val = obj.val;
        cout << "From move assignment operator" << endl;
        return *this; // ✗ Returning by value causes unexpected behavior
    }
};

int main() {
    base b1(10);
    base b2(20);
    b1 = move(b2);
    return 0;
}
```

ChatGPT said:

Let's analyze your code carefully and understand why removing & from the move assignment operator causes the **copy constructor** to be called instead of the move constructor.

Understanding Your Code

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

class base {
    int val;

public:
    base(int a) : val(a) {}

    base(base&& obj) { // Move Constructor
```

```

    val = obj.val;
    cout << "From move constructor" << endl;
}

base(const base& obj) { // Copy Constructor
    val = obj.val;
    cout << "From copy constructor" << endl;
}

base operator=(base&& obj) { // Move Assignment Operator (returns by value)
    val = obj.val;
    cout << "From move assignment operator" << endl;
    return *this; // ❌ Returning by value causes unexpected behavior
}

};

int main() {
    base b1(10);
    base b2(20);
    b1 = move(b2);
    return 0;
}

```

Expected vs. Actual Output

If the move assignment operator returned **by reference (base&)**, we would expect:
 From move assignment operator

However, since you removed &, returning **by value** causes an **unexpected call to the copy constructor**:
 From move assignment operator
 From copy constructor

Why is the Copy Constructor Called Instead of the Move Constructor?

The problem is the **return type** of the move assignment operator:

cpp

CopyEdit

```

base operator=(base&& obj) { // ❌ Returning by value
    val = obj.val;
    cout << "From move assignment operator" << endl;
    return *this; // ❌ This creates a copy instead of moving
}

```

Step-by-Step Execution

1. `b1 = move(b2);` calls the **move assignment operator**.
2. Inside the move assignment operator:
 - `obj` is an **rvalue reference** to `b2`.
 - `val = obj.val;` moves the value.
 - **But then, return *this; returns *this by value.**
3. **Returning by value requires a copy or move:**
 - Ideally, the move constructor should be used.
 - **But *this is an lvalue, not an rvalue!**
 - Since an lvalue is returned, **the copy constructor is used instead of the move constructor.**

Key Rule

 **Returning an object by value always prefers the copy constructor if the returned object is an lvalue.**
 Since `*this` is an **lvalue**, it is copied.

You said:

If the copy constructor is explicitly defined, the compiler does not generate a move constructor. Similarly, if the move constructor is explicitly defined, the compiler does not generate a copy constructor. Is this correct?

ChatGPT said:

Yes, your understanding is **partially correct**, but there are some important details to consider. Let's break it down properly.

❏ If You Explicitly Declare a Copy Constructor

👉 The compiler will **NOT** generate a move constructor.

Example

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

class base {
    int val;
public:
    base(int a) : val(a) {}

    base(const base& obj) { // Explicit Copy Constructor
        val = obj.val;
        cout << "From copy constructor" << endl;
    }
};

int main() {
    base b1(10);
    base b2 = move(b1); // ❌ No move constructor → falls back to copy constructor
    return 0;
}
```

◆ Output:

From copy constructor

◆ Why?

- Since we explicitly **defined a copy constructor**, the compiler **does NOT generate a move constructor**.
- When we try `move(b1)`, it falls back to the **copy constructor** instead of moving.

❏ If You Explicitly Declare a Move Constructor

👉 The compiler will **DELETE** the copy constructor if no copy constructor exists.

Example

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

class base {
    int val;
public:
    base(int a) : val(a) {}
```

```

base(base&& obj) { // Explicit Move Constructor
    val = obj.val;
    cout << "From move constructor" << endl;
}
};

int main() {
    base b1(10);
    base b2 = b1; // ❌ ERROR: Copy constructor is deleted
    return 0;
}

```

◆ **Compilation Error:**

```

lua
CopyEdit
error: use of deleted function 'base::base(const base&)'

```

◆ **Why?**

- When you **explicitly declare a move constructor**, the compiler **DELETES the default copy constructor**.
- So base b2 = b1; fails because there's **no copy constructor available**.

❗ If You Explicitly Declare Both Copy and Move Constructors

👉 Both exist, and the compiler does not generate any additional ones.

Example

```

cpp
CopyEdit
#include <iostream>
using namespace std;

class base {
    int val;
public:
    base(int a) : val(a) {}

    base(const base& obj) { // Copy Constructor
        val = obj.val;
        cout << "From copy constructor" << endl;
    }

    base(base&& obj) { // Move Constructor
        val = obj.val;
        cout << "From move constructor" << endl;
    }
};

int main() {
    base b1(10);
    base b2 = b1;    // Calls copy constructor
    base b3 = move(b1); // Calls move constructor
    return 0;
}

```


◆ Output:

pgsql
CopyEdit
From copy constructor
From move constructor

◆ Why?

- The compiler does **not** generate any default constructors since we explicitly provided both.

Summary Table

Case	Copy Constructor Generated?	Move Constructor Generated?	Behavior
No constructors written	✓ Yes	✓ Yes	Compiler generates both copy & move constructors.
Only copy constructor written	✓ Yes (explicitly)	✗ No	Move operations fall back to copy.
Only move constructor written	✗ No (deleted)	✓ Yes (explicitly)	Copy operations cause a compilation error.
Both copy & move constructors written	✓ Yes (explicitly)	✓ Yes (explicitly)	Compiler does nothing extra.

Key Takeaways

1. Declaring a copy constructor prevents the compiler from generating a move constructor.
2. Declaring a move constructor deletes the compiler-generated copy constructor.
3. If you want both, you must explicitly define both.
4. **Best Practice:** If you define one of them, it's usually a good idea to define both to avoid surprises.

You said:

what it means of delete copy constructor?

ChatGPT said:

Great question! Let's break it down in simple terms.

What Does "Deleted Copy Constructor" Mean?

When the compiler **deletes** the copy constructor, it means you **cannot use copy operations** on that class. If you try to copy an object, the compiler will give a **compilation error**.

How Does It Happen?

The copy constructor can be explicitly deleted like this:

```
class Base {  
public:  
    Base() = default;  
    Base(const Base&) = delete; // ✗ Copy constructor is deleted  
};
```

Now, if you try:

```
Base b1;
```

```
Base b2 = b1; // ❌ Compilation error!
```

◆ Error Message (Typical)

error: use of deleted function 'Base::Base(const Base&).'

When Does the Compiler Automatically Delete the Copy Constructor?

The compiler **automatically** deletes the copy constructor if:

1. You explicitly define a move constructor (`Base(Base&&)`) but don't provide a copy constructor.
2. The class has non-copyable members (e.g., `unique_ptr`).

Example: Move Constructor Deletes Copy Constructor

```
#include <iostream>
```

```
using namespace std;
```

```
class Base {
public:
    Base() = default;

    Base(Base&& obj) { // Move Constructor
        cout << "Move Constructor Called" << endl;
    }
};
```

```
int main() {
    Base b1;
    Base b2 = b1; // ❌ ERROR: Copy constructor is deleted
    return 0;
}
```

◆ Error Message:

error: use of deleted function 'Base::Base(const Base&).'

Why Does the Compiler Delete It?

When a move constructor exists **without** a copy constructor, the compiler assumes **the object should not be copied, only moved**.

So it **automatically deletes the copy constructor** to prevent accidental copying.

When Should You Delete the Copy Constructor?

✅ If you want to **prevent copying** but allow moving, explicitly delete the copy constructor:

```
cpp
```

```
CopyEdit
```

```
class NonCopyable {
```

```
public:
```

```
    NonCopyable() = default;
```

```
    NonCopyable(const NonCopyable&) = delete; // Prevent copying
```

```
    NonCopyable& operator=(const NonCopyable&) = delete; // Prevent copy assignment
```

```
};
```

Now, these operations are **forbidden**:

```
cpp
CopyEdit
NonCopyable obj1;
NonCopyable obj2 = obj1; // ❌ ERROR: Copy constructor is deleted
obj2 = obj1;           // ❌ ERROR: Copy assignment is deleted
```

Summary

Case	Can Copy?	Can Move?	Explanation
No constructors provided	✅ Yes	✅ Yes	Compiler generates both copy & move constructors.
Copy constructor explicitly deleted (= delete)	❌ No	✅ Yes	Copying is forbidden, but moving is allowed.
Move constructor provided, but no copy constructor	❌ No (Deleted)	✅ Yes	Compiler deletes the copy constructor.
Both copy & move constructors provided	✅ Yes	✅ Yes	No compiler-generated constructors; user-defined behavior.

Key Takeaways

- A **deleted copy constructor** means copying is **forbidden**.
- If you write a **move constructor but no copy constructor**, the **copy constructor is deleted automatically**.
- If you want a **non-copyable class**, explicitly delete the copy constructor.
- Deleting constructors helps enforce object **ownership** and **avoid accidental copies**.

Let me know if you have any more questions! 😊🚀

You said:

ok but if you created copy constructor then move constructor not created by compiler but why here not deleting move constructor?

ChatGPT said:

That's a great question! Let's clarify the **difference between "not created" and "deleted"** when it comes to the move constructor.

❏ If You Declare a Copy Constructor, Why Isn't the Move Constructor Deleted?

When you **explicitly declare a copy constructor**, the compiler **does not delete the move constructor**—it just **does not generate it automatically**.

This means:

- The move constructor **does not exist** unless you explicitly define it.
- It is **not deleted**, just **not auto-generated**.

Example: Copy Constructor Blocks Automatic Move Constructor

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

```

class Base {
public:
    Base() = default;

    Base(const Base&) { // Explicitly defined Copy Constructor
        cout << "From Copy Constructor" << endl;
    }
};

int main() {
    Base b1;
    Base b2 = move(b1); // ❌ No Move Constructor (but not deleted)
    return 0;
}

```

◆ Error Message (Typical)

error: no matching function for call to 'Base::Base(Base&&)'

💡 What happened?

- The move constructor is **not deleted** but simply **not generated**.
- Since **no move constructor exists**, the compiler tries **copying instead**.

🔗 When Does the Move Constructor Get Deleted?

A move constructor gets **explicitly deleted by the compiler** if:

1. The class contains **non-movable** members (like const or unique_ptr).
2. You explicitly delete the move constructor.
3. The class has a **deleted copy constructor**.

Example: Deleted Copy Constructor Also Deletes Move Constructor

```

#include <iostream>
using namespace std;

class Base {
public:
    Base() = default;

    Base(const Base&) = delete; // ❌ Copy Constructor Deleted
};

int main() {
    Base b1;
    Base b2 = move(b1); // ❌ ERROR: Move constructor is also deleted!
    return 0;
}

```

◆ Error Message

error: use of deleted function 'Base::Base(Base&&)'

💡 What happened?

- Since the copy constructor is **explicitly deleted**, the compiler **automatically deletes the move constructor**.
- Now, neither copy nor move is allowed.

🔑 Key Differences: "Not Created" vs. "Deleted"

Case	Copy Constructor	Move Constructor	Behavior
No constructors written	✓ Created	✓ Created	Compiler generates both.
Only copy constructor written	✓ Created	✗ Not Created (But not deleted)	Move operations fallback to copy.
Only move constructor written	✗ Deleted	✓ Created	Copy operations are forbidden.
Copy constructor explicitly deleted	✗ Deleted	✗ Deleted	Neither copy nor move allowed.

Summary

- If you **explicitly write a copy constructor**, the compiler **does not delete** the move constructor—it just **doesn't generate one**.
- If the copy constructor is **deleted**, then the move constructor **is also deleted**.
- If you want a move constructor **when a copy constructor is present**, you must **explicitly define it**.