Move Constructors in C++ with Examples

Difficulty Level : Medium • Last Updated : 01 Sep, 2021

Prerequisites: l-value and r-value references in C++ , Copy Constructor in C++.

What is a Move Constructor?

The copy constructors in C++ work with the l-value references and copy semantics(copy semantics means copying the actual data of the object to another object rather than making another object to point the already existing object in the heap). While move constructors work on the r-value references and move semantics(move semantics involves pointing to the already existing object in the memory).

On declaring the new object and assigning it with the r-value, firstly a temporary object is created, and then that temporary object is used to assign the values to the object. Due to this the copy constructor is called several times and increases the overhead and decreases the computational power of the code. To avoid this overhead and make the code more efficient we use move constructors.

Why Move Constructors are used?

Move constructor moves the resources in the heap, i.e., unlike copy constructors which copy the data of the existing object and assigning it to the new object move constructor just makes the pointer of the declared object to point to the data of temporary object and nulls out the pointer of the temporary objects. Thus, move constructor prevents unnecessarily copying data in the memory.

Work of move constructor looks a bit like default member-wise copy constructor but in this case, it nulls out the pointer of the temporary object preventing more than one object to point to same memory location.

Below is the program without declaring the move constructor:

C++

```
// C++ program without declaring the
// move constructor
#include <iostream>
#include <vector>
using namespace std;

// Move Class
class Move {
private:
    // Declaring the raw pointer as
    // the data member of the class
    int* data;

public:
    // Constructor
    Move(int d)
```

```
    {
        // Declare object in the heap
        data = new int;
        *data = d;

        cout << "Constructor is called for "
             << d << endl;
    };

    // Copy Constructor to delegated
    // Copy constructor
    Move(const Move& source)
        : Move{ *source.data }
    {

        // Copying constructor copying
        // the data by making deep copy
        cout << "Copy Constructor is called - "
             << "Deep copy for "
             << *source.data
             << endl;
    }

    // Destructor
    ~Move()
    {
        if (data != nullptr)

            // If the pointer is not
            // pointing to nullptr
            cout << "Destructor is called for "
                 << *data << endl;
        else

            // If the pointer is
            // pointing to nullptr
            cout << "Destructor is called"
                 << " for nullptr"
                 << endl;

        // Free the memory assigned to
        // data member of the object
        delete data;
    }
};

// Driver Code
int main()
{
    // Create vector of Move Class
    vector<Move> vec;
```

```
    // Inserting object of Move class
    vec.push_back(Move{ 10 });
    vec.push_back(Move{ 20 });
    return 0;
}
```

Output:

```
Constructor is called for 10
Constructor is called for 10
Copy Constructor is called - Deep copy for 10
Destructor is called for 10
Constructor is called for 20
Constructor is called for 20
Copy Constructor is called - Deep copy for 20
Constructor is called for 10
Copy Constructor is called - Deep copy for 10
Destructor is called for 10
Destructor is called for 20
Destructor is called for 10
Destructor is called for 20
```

Explanation:
The above program shows the unnecessarily calling copy constructor and inefficiently using the memory by copying the same data several times as it new object upon each call to copy constructor.

Syntax of the Move Constructor:

```
Object_name(Object_name&& obj)
   : data{ obj.data }
{
   // Nulling out the pointer to the temporary data
   obj.data = nullptr;
}
```

This unnecessary use of the memory can be avoided by using move constructor. Below is the program declaring the move constructor:

C++

```cpp
// C++ program with declaring the
// move constructor
#include <iostream>
#include <vector>
using namespace std;

// Move Class
class Move {
private:
    // Declare the raw pointer as
    // the data member of class
    int* data;

public:

    // Constructor
    Move(int d)
    {
        // Declare object in the heap
        data = new int;
        *data = d;
        cout << "Constructor is called for "
             << d << endl;
    };

    // Copy Constructor
    Move(const Move& source)
        : Move{ *source.data }
    {

        // Copying the data by making
        // deep copy
        cout << "Copy Constructor is called -"
             << "Deep copy for "
             << *source.data
             << endl;
    }

    // Move Constructor
    Move(Move&& source)
        : data{ source.data }
    {

        cout << "Move Constructor for "
             << *source.data << endl;
        source.data = nullptr;
    }

    // Destructor
    ~Move()
    {
        if (data != nullptr)
```

```
            // If pointer is not pointing
            // to nullptr
            cout << "Destructor is called for "
                 << *data << endl;
        else

            // If pointer is pointing
            // to nullptr
            cout << "Destructor is called"
                 << " for nullptr "
                 << endl;

        // Free up the memory assigned to
        // The data member of the object
        delete data;
    }
};

// Driver Code
int main()
{
    // Vector of Move Class
    vector<Move> vec;

    // Inserting Object of Move Class
    vec.push_back(Move{ 10 });
    vec.push_back(Move{ 20 });
    return 0;
}
```

Output:

```
Constructor is called for 10
Move Constructor for 10
Destructor is called for nullptr
Constructor is called for 20
Move Constructor for 20
Constructor is called for 10
Copy Constructor is called -Deep copy for 10
Destructor is called for 10
Destructor is called for nullptr
Destructor is called for 10
Destructor is called for 20
```

Explanation:

The unnecessary call to the copy constructor is avoided by making the call to the move constructor. Thus making code more memory efficient and

decreasing the overhead of calling the move constructor.

23

Related Articles

1. std::move in Utility in C++ | Move Semantics, Move Constructors and Move Assignment Operators
2. When Does Compiler Create Default and Copy Constructors in C++?
3. C++ | Constructors | Question 2
4. C++ | Constructors | Question 3
5. C++ | Constructors | Question 4
6. C++ | Constructors | Question 5
7. C++ | Constructors | Question 6
8. C++ | Constructors | Question 7
9. C++ | Constructors | Question 8
10. C++ | Constructors | Question 9

Previous

List of Stacks in C++ STL

Next

Represent Tree using graphics in C/C++

▲