Copy Constructor

There is a default copy constructor auto generated by compiler

called when?

- Create object from existing object
- Pass object to function
- Return object from function by value

User defined copy constructor

Ability to create object from another created object
The first constructor is a default constructor. The default constructor is a constructor which can be called with no arguments.

The second constructor is a copy constructor. The copy constructor is a constructor which initializes an object using another object of the same class. In this case, the `Player` copy constructor takes a reference to another `Player` object as its argument.

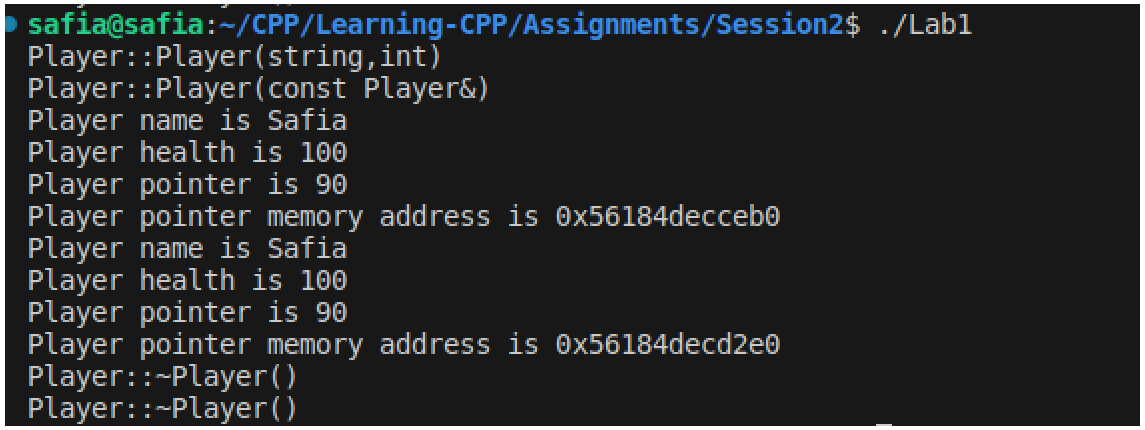
```
class Player {
    // Class members and methods go here
    std::string name;
    int health;
    int *ptr;
    public:
// default constructor , delgate to another constructor
    Player():Player("none",0,0){
        std::cout << "Player::Player()" << std::endl;
```

```cpp
    }
    Player(int source) : Player("none",source,0){
        std::cout << "Player::Player(int)" << std::endl;
    }
    Player (std::string source) : Player(source,0,0){
        std::cout << "Player::Player(string)" << std::endl;
    }
    // parameterized constructor
    Player(std::string source1,int source2,int source3) : name{source1}
                                                         ,health{source2}
                                                         ,ptr{new int(source3)}{
        std::cout << "Player::Player(string,int)" << std::endl;
    }
    // user defined copy constructor
    Player(const Player &source){
        std::cout << "Player::Player(const Player&)" << std::endl;
        name = source.name;
        health= source.health;
        ptr = new int;
        *ptr = *(source.ptr);
    }
    // display info
    void display_info(void)
    {
        std::cout << "Player name is " << name << std::endl;
        std::cout << "Player health is " << health << std::endl;
        // Print the memory address of the pointer
        std::cout << "Player pointer memory address is " << ptr << std::endl;
    }
    // Destructor
    ~Player(){
        std::cout << "Player::~Player()" << std::endl;
        delete ptr;
    }
};
```

Test

```
int main()
{
    //Parameterized constructor
    Player p1{"Safia",100,90};
    Player p2=p1;       //This will call the copy constructor
    p1.display_info();
    p2.display_info(); //Address of the pointer will be different in both objects
}
```

Output



```
safia@safia:~/CPP/Learning-CPP/Assignments/Session2$ ./Lab1
Player::Player(string,int)
Player::Player(const Player&)
Player name is Safia
Player health is 100
Player pointer is 90
Player pointer memory address is 0x56184decceb0
Player name is Safia
Player health is 100
Player pointer is 90
Player pointer memory address is 0x56184decd2e0
Player::~Player()
Player::~Player()
```

Why copy constructor needed?

```
    // parameterized constructor
    /* Ptr is a pointer to an integer and it is dynamically allocated,
    in case of default copy constructor it will be shallow copy and
    it will cause a problem as it will point to the same memory location
    so we need to define a copy constructor to make a deep copy of the pointer
    */
    Player(std::string source1,int source2,int source3) : name{source1}
                                                         ,health{source2}
                                                         ,ptr{new int(source3)}{
        std::cout << "Player::Player(string,int)" << std::endl;
    }
```

1. **Deep Copy**: If an object has pointers or dynamic memory allocations, a simple bit-wise copy (default behavior) would just copy the addresses, leading to multiple objects pointing to the same memory location. This can cause issues when one object modifies the memory or deallocates it. A copy constructor allows for deep copying, where new memory is allocated for the new object, and the contents are copied.
2. **Copy of Objects with Non-Trivial Construction**: If an object needs specific actions to be taken during construction (like opening a file, acquiring a resource), the copy constructor ensures these actions are performed correctly when a new object is created as a copy.
3. **Control Over Copying**: Sometimes, you might want to keep track of how many copies of an object exist, or you might want to prevent copying altogether. Having a copy constructor gives you control over these aspects.
4. **Passing Objects by Value**: When an object is passed by value to a function, the copy constructor is called to create the copy. Similarly, when an object is returned by value from a function, the copy constructor is called.
5. **Initializing Objects**: The copy constructor is used to initialize one object from another of the same type. For example, `MyClass obj1(obj2);` Here, `obj1` is initialized with `obj2` using the copy constructor.

Move Constructor

A move constructor in C++ is a special type of constructor that is designed to transfer resources from a temporary object (an object that is about to be destroyed) to a new object. It's used to optimize performance by avoiding unnecessary copy operations.

The move constructor is used in several scenarios:

1. **Returning objects from functions**: When a function returns an object by value, the compiler can use the move constructor instead of the copy constructor to create the returned object, avoiding a potentially expensive copy operation.
2. **Inserting objects into containers**: When you insert an object into a C++ container (like `std::vector`), the container may use the move constructor to transfer the object into its internal storage, again avoiding a copy.
3. **Sorting and rearranging containers**: When you sort a container or otherwise rearrange its elements, the container may use the move constructor to efficiently move objects around.
4. **Explicitly requested by `std::move`**: you can explicitly request to use the move constructor by using `std::move`.

```
    Player(Player &&source){
        std::cout << "Player::Player(Player&&)" << std::endl;
        source.name = name;
        source.health = health;
        source.ptr = nullptr;
    }
```

Note

In C++, lvalue references and rvalue references are used to categorize expressions based on their values and behaviors.

1. **lvalue reference**: An lvalue reference is a reference that binds to an lvalue (an object that has a specific memory location). You can think of it as an alias for the object it refers to. `int &l_ref = x;` creates an lvalue reference `l_ref` that refers to `x`. Any changes made to `l_ref` will affect `x`, and vice versa, because they both refer to the same memory location.
2. **rvalue reference**: An rvalue reference is a reference that binds to an rvalue (a temporary object or a value not associated with a specific memory location). Rvalue references are used to implement move semantics and perfect forwarding, which can improve the performance of your programs. `int &&Ref = 3;` creates an rvalue reference `Ref` that refers to the temporary value `3`. This allows you to modify the temporary value through `Ref`, as shown by `Ref = 5;`.

Remember, lvalue references can bind to lvalues, but not to rvalues. On the other hand, rvalue references can bind to rvalues, but not to lvalues. This is why you can't do something like `int &l_ref = 3;` or `int &&Ref = x;` in C++.